DAIS: a Domestic Automatic Irrigation System for Next-Generation Houses

Emanuele Cozza
Alma Mater Studiorum Bologna
Bologna, Italy
emanuele.cozza@studio.unibo.it

***Abstract*—Watering plants has always been a tedious task that everyone has had to handle at least once in their lifetime. This task is typically performed by non-experts and often results in inefficient water usage. Due to climate change, water waste has become a major global concern. This paper presents an irrigation system that leverages the functionalities of the Internet of Things to make it autonomous, user-friendly, and intelligent. The system can predict water requirements over short periods, making it suitable for integration into next-generation homes to promote responsible water usage.**

I. Introduction

In Italy, an average citizen consumes between 130 and 140 liters of water per day (mainly potable in Emilia-Romagna) for various activities, including food preparation, personal hygiene, and the use of appliances such as washing machines and dishwashers. A significant portion of this consumption is dedicated to the irrigation of domestic green spaces.

Italy, like much of Europe, has been experiencing abnormal weather conditions during recent summer periods. Severe droughts are rarely interrupted by storms that bring heavy rainfall, which often flows away without being utilized for the aforementioned purposes. As these phenomena are likely correlated with climate change, next-generation houses must adapt by being equipped with systems for rainwater collection and storage. Additionally, such houses must be able to manage this water responsibly and forecast water needs. This is crucial because the implementation of these systems does not eliminate the possibility that water may need to be produced through sea-water desalination processes or imported.

DAIS, the solution proposed in this paper, is a system for managing water storage at a domestic level, featuring two different subsystems for soil moisture and water availability monitoring, along with a module for predicting future water requirements. The key aspects of this application are ease of use, reduced costs, and system scalability, enhanced by the connectivity features of the implemented devices.

The article is organized as follows: Section II describes the architecture of the entire system. Section III covers the technical aspects, including the communication protocols adopted and the algorithms used for forecasting. Section IV presents the numerical results, detailing the system's performance.

II. System's Architecture

The architecture of DAIS features different elements. Fig. 1 illustrates the main relationships between these modules and how tasks are distributed between the in-field components and the Cloud in a real-world application. Hereafter each element will be discussed from the point of view of the system architecture, with a brief explanation of the modifications needed due to the hardware resource limitations in the experimental setup.

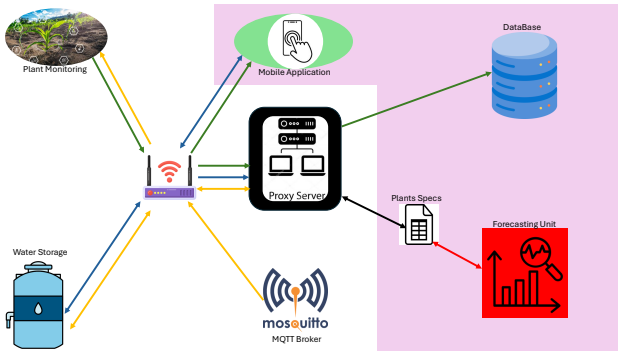

Fig. 1. DAIS' general architecture. The pink side represents the components on the Cloud.

A. Data Proxy

The *Data Proxy* represents the core of the entire system, designed to execute multiple tasks simultaneously. Its main functions are:

- enabling communication with the sensor nodes for configuration purposes and acquiring and sharing data with them;
- transmitting some of the acquired data to the Cloud for storage;
- activating on a daily basis irrigation routines customized for each plant;
- sending notifications to users' mobile phone;

This module has been implemented using a Python script running on a general-purpose personal computer due to limited hardware resources. In a real-world application, this program would need to be run on more suitable hardware, such as a Raspberry Pi module.

B. Sensor Node for Plant Monitoring

This end-device features an ESP32 development board as a micro-controller equipped with a capacitive soil moisture sensor to monitor the water needs of the plant. Due to its

field of application, the sensor node is supposed to be battery-powered and connected to the domestic WiFi to communicate with the Data Proxy. In particular, it performs the following operations:

- sensing the soil moisture;
- transmitting data to the Proxy if available, otherwise directly to the Cloud for storage;
- sharing moisture data with other Cloud facilities to enable data visualization on mobile applications;

In a real-world application, one sensor node of this kind is required for each plant. However, due to resource limitations, only one plant was considered, and consequently only one sensor node was deployed. In addition, instead of being battery-powered, this sensor has been powered by the household electrical outlet.

C. Sensor Node for Water Management

This subsystem comprises an ESP32 development board, a distance sensor, and a water pump. It is connected to the domestic WiFi network to communicate with the Data Proxy and it is powered by the household electrical outlet due to the large power requirements for water pumping operations. The operations this node has to perform are:

- monitoring the water level within the water tank;
- sharing this data with Cloud facilities to enable data visualization and control on mobile applications;
- activating the irrigation procedure when triggered by the Data Proxy;

In a real-world application, where multiple plants need to be watered, each plant would require its own pump or at least a valve to adjust the amount of delivered water to ensure responsible irrigation. In large-scale scenarios, where distances are too great for cables or the number of plants is very large, additional micro-controllers should be implemented to control the irrigation routines for each plant while maintaining a single device for water availability monitoring.

In this context, since a single plant was considered, only one pump was used, and due to limited hardware resources, its management and the water monitoring task have been performed by the same micro-controller.

D. Forecast Module

This module fetches the history of moisture data for each plant from the implemented database. Based on this data, statistical methods are employed to forecast the humidity trend for the next few days.

By obtaining weather forecast data available on the Cloud, it estimates the amount of water the plants will receive from natural precipitation and the amount of water that will be collected in storage. According to the plants' specifications, the predicted soil moisture values allow for an estimation of water needs for each plant. This enables end-users to determine how much water they need to request from the public water supply system a few days in advance.

The plant specifications are collected in an Excel file, organized as a table with as many rows as there are plants involved in the system. The main fields for each plant in this table are:

- the identifier of the sensor associated with the plant;
- minimum and maximum moisture levels;
- root volume mean, representing the volume including the roots of the plant;
- the date of the last irrigation, along with the amount of water used, divided into water from personal storage and water acquired from the public supply system;
- the next irrigation date based on the predictions of the forecasting module, along with the amount of water required by each plant, divided as explained in the previous point;
- the ID of the pump associated with the given plant.

As shown in Fig. 1, this file is accessed by both the Data Proxy and the Forecasting Module and its content is manged with a `pandas` dataframe. For simplicity, it has been created as a file in the same working environment as these subsystems, since both are launched on the same personal computer. In a real-world application, while the Forecasting Module would run on Cloud facilities, the Proxy would execute on a different physical device, such as a Raspberry Pi module. This setup would introduce a clear separation between these elements, complicating the implementation of the file in the current manner. However, an Excel file was chosen for several reasons. First, the simplicity of the working environment allows the user to adjust its parameters easily, as most people are familiar with this type of file. Second, this environment can easily be replaced by Google Sheets in Google Drive, making possible to add connectivity features to the Proxy and the Forecasting Unit to access this file in such a scenario.

III. System's Implementation

The functionalities offered by DAIS rely on the implementation of different communication protocols in the field of the Internet of Things, such as the *HyperText Transfer Protocol* (HTTP), the *Message Queuing Telemetry Transport* (MQTT) protocol and the *Constrained Application Protocol* (CoAP). Due to the limited amount of hardware resources, a personal computer was used to launch both the Data Proxy and the MQTT broker. The broker capabilities were obtained by mean of a `mosquitto` broker, whose functionalities must be customized. Specifically, it has been configured in order to listen to a specific IP address, to refuse the connection to all those clients having an empty client-ID and a combination of username and password has been defined for all the modules that connect them self to the broker, in order to provide a proper level of network security.

For Cloud data storage, instances of InfluxDB databases have been used to collect data during the DAIS' lifetime. Although the water level in the tank has been periodically monitored, there was no interest in retaining the history of such data since no data analysis was intended for them. Therefore, only a single instance of InfluxDB has been used to collect moisture data and the corresponding predictions, with a data

retention policy of 30 days, the maximum duration allowed for a free account. Grafana's functionalities were utilized to visualize the data time series.

The ESP32 boards deployed within the sensor nodes have been programmed in Arduino's Integrated Development Environment (IDE), while the Python scripts for the Data Proxy and the Forecasting Unit have been developed in a Jupyter Notebook.

In the following, the main elements of the previously described system's architecture are described form a technical point of view.

A. Sensor Node for Plant Monitoring

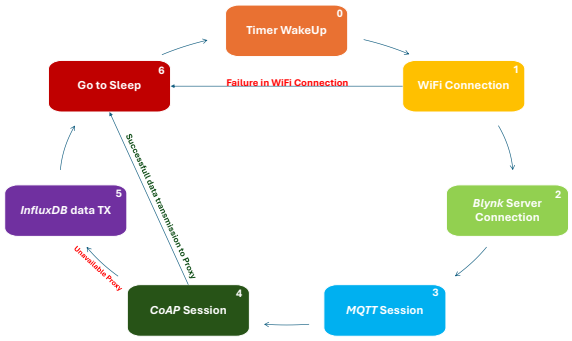

Fig. 2. Plant Monitoring Time-Multiplexing scheme.

As mentioned in Section II-B, in a real-world application, this sensor node is designed to be battery-powered. Consequently, the power consumption of the micro-controller represented a major concern in designing this module. To reduce power consumption as much as possible, the ESP32 board has been programmed to duty cycle its operation, spending most of its time in deep sleep mode. According to [1], the current consumption in this configuration ranges between $10\mu A$ and $150\mu A$, corresponding approximately to the 0.05% of the current required to receive and transmit data via WiFi. Recovery from deep sleep is triggered by different types of interrupts. Given the slowly-varying nature of the monitored phenomenon and the implemented configuration, the soil moisture sampling period was made to coincide with the sleep mode duration. Therefore, a RTC Timer was adopted as the only interrupt source to wake up the processor. The default sleep time was set to 30 minutes. Upon waking, the system attempts to establish a connection to an MQTT broker and subscribes to the topic `<sensID>/sleep-time` to receive possible updates regarding this parameter.

The plant-monitoring sensor node is designed to execute various tasks, such as the aforementioned communication with a MQTT broker, soil sensing or data transmission to the Proxy. Therefore, as illustrated in Fig. 2, a time-multiplexing approach was adopted to allocate a limited amount of time to each task and execute them one at a time.

The system's primary goal—monitoring the soil moisture of the associated plant—is achieved during *phase 4*, using a capacitive analog sensor dedicated to this task. This sensor provided $12-bits$ values ranging from 0 to 4096 representing the soil moisture (M) defined as:

$$M = \frac{\text{Water Volume}}{\text{Soil Volume}} \tag{1}$$

To generate a consistent measure, 25 samples are acquired in a 1 second time-window and averaged. This datum is then forwarded to the Data Proxy. Due to the resource-limited nature of the application, the CoAP protocol was implemented for this task. Because of the sensor node's unpredictable availability due to sleep mode, the Data Proxy was configured as a CoAP server able to handle PUT requests coming from the sensor nodes (CoAP clients) at the resources `<sensID>/<resource-name>`. In the scope of the considered experiment the resource was the soil moisture only. The `simple-coap` library for ESP32 was used to transmit the message from the micro-controller. Specifically, once the moisture datum is available, the packet is created as follows:

- The message ID (MID) is defined as a 16-bit random integer.
- A token is created as an array of four random 8-bit integers.
- The payload (moisture data in percentage units) consists of four characters: 2-3 characters for the integer part, one for the decimal point, and one for the fractional part.

Setting the message ID and the token is necessary to ensure the Data Proxy can send the proper response to the correct sensor node. Since these parameters cannot be set using the `.pub()` function presented in the `simple-coap` library, the `.send()` method was implemented. The data to be exchanged are in the `text/plain` format. In this way the payload contains only a string representing a number with no further information. This choice may affect the interoperability of the system, but it allows for a reduction of the payload size, which enhances the scalability of the application. In fact, with the described structure, each CoAP packet requires 13 bytes, assuming the `options` field is empty.

The monitoring of soil moisture was meant to be a continuous and periodic task. However, since the Data Proxy was launched on a personal computer, it could not be kept always on. Therefore, in case of unavailable Proxy, the moisture datum is sent directly to the Cloud, specifically to an instance of an InfluxDB database, using the `InfluxDbClient` library for ESP32. This approach increases the system's reliability by reducing the number of missed transactions, but it comes at a higher energy cost because it exploits the HTTP protocol for transmission.

Finally, the *Blynk* mobile application was used to allow the last recorded moisture level to be visible on the mobile phone of the end-users. To do so, the moisture datum is sent to the Blynk sever with the aid of the `BlynkSimpleEsp32` library, leveraging the HTTP protocol for data communications. Due to the discontinuous activity of this sensor node, and the possibility of missed transactions, a 3 hours data invalidation policy was adopted in the Blynk app configuration.

B. Sensor Node for Water Management

This module executes two main tasks: monitoring the water level inside the dedicated storage and driving the water pump.

For water level monitoring, the HC-SR04 ultrasonic distance sensor was used. Specifically, the dimensions of the water storage in terms of cross-section (CS) and depth (D) were uploaded to the micro-controller, and the amount of available water (W_a) was estimated as follows:

$$W_a = CS \cdot (D - W_r) \qquad [cm^3] \tag{2}$$

where W_r represents the distance measured by the sensor positioned at the top of the tank. To ensure a consistent measurement, 10 samples are acquired and averaged.

The default sampling period is set to 5 seconds, but the sensor node is connected to the MQTT broker to receive updates concerning this metric on the `pump1/sampling` topic. Additionally, this MQTT connection is used to publish an alert message on the `pump1/alert` topic.

Water level measurement is crucial for resource availability during the irrigation routine scheduling. Therefore, this data must be accessible to the Data Proxy when needed. For this reason, the sensor node was programmed to act as an HTTP server that can be queried by the Data Proxy (HTTP client). The HTTP protocol was chosen for its higher level of security and the absence of power constraints. Security is essential because the pump directly accesses the water reservoir, a valuable resource. Specifically, this HTTP server can handle GET requests to provide water availability data to clients, as well as PUT requests, which the Data Proxy uses to transmit the amount of water (W_p) to be delivered to the plant associated with the specified pump ID.

Upon connection, the sensor node is automatically assigned an IP address on the local WiFi network. However, this address is not accessible to clients unless the server shares it on the network. To address this issue, the `ESPmDNS` ESP32 library was used to make the sensor node visible on the local network with an address of the form `<pumpID>.local`.

The pump control exploits a relay module. By modulating the time the relay provides power to the pump, the amount of water delivered to the plant can be tuned. The pump specifications in terms of water flow (Φ) have been uploaded to the micro-controller, so that the time (T) during which the pump must stay active can be evaluated as:

$$T = \frac{W_p}{\Phi \cdot \eta} \qquad [s] \tag{3}$$

where $\eta = 0.95$ is the pump efficiency, a parameter empirically estimated.

Since this module is not energy constrained and due to the limited network traffic deriving from these operations, messages have been exchanged using the `json` format to enhance system interoperability.

The Blynk mobile application was used to visualize the water level inside the tank, along with the capability to activate the pump from the mobile application.

C. Data Proxy

The Data Proxy leverages the capabilities offered by the `asyncio` Python library to perform several functions simultaneously. Specifically, it performs three tasks:

- Monitoring data traffic on an MQTT network
- Acting as a CoAP server
- Activating the irrigation procedure on a daily basis

The MQTT functionalities offered to the Data Proxy were implemented with the aid of the `paho.mqtt.client` Python library. In particular, such functionalities consist in the connection establishment between the Proxy and the broker and the definition of two callback functions to handle connection attempts to the broker and message publications. Upon a successful connection, the Proxy subscribes to the following topics:

- `pump1/alert`
- `pump1/sampling`
- `<sensID>/sleep-time` for all the sensors listed in the plants specification dataframe.

The publication of alert messages triggers a specific part of the corresponding callback function, which sends a Telegram notification to the end-user about the encountered issue. In this context, the only alert source considered is the fullness of the water reservoir. Subscriptions to the remaining topics are solely for monitoring purposes. A dedicated program was developed specifically to publish data to the desired MQTT topic, distinct from the server. This decision was made because the server, as currently programmed, would necessitate stopping the application, manually inserting the desired data, and then relaunching it, which is not user-friendly. Future enhancements for this module could include implementing KeyBoard interrupts for insertion of new values.

The CoAP server was defined by leveraging the `aiocoap` Python library, and instantiating a Python class containing the desired callback functions handling CoAP requests. Since the plant-monitoring sensor nodes are supposed to only forward data to the server, only the function handling PUT requests was defined. This function extracts the payload from the CoAP packet and retrieves the sensor ID and the resource name from the topic where the datum was published. Once the datum is retrieved, it is sent to an InfluxDB database instance, specifying the field as resource name retrieved from the topic and adding the obtained sensor ID as a tag. This operation was made possible by the implementation of the `influxdbclient` Python library. Two response messages in the `text/plain` format, `OK` and `NO`, are defined to notify the CoAP client about the transmission outcome. These two messages require 2 bytes each to minimize network traffic.

Finally, to activate irrigation on a daily basis, two functions are defined. The first function, for which a task has been dedicated alongside the CoAP server and the MQTT subscriber, checks the time every 3 hours. When a specific user-defined time is reached, the irrigation procedure, described by the second function, is activated. This procedure starts with a GET request to the sensor node monitoring the tank to obtain the

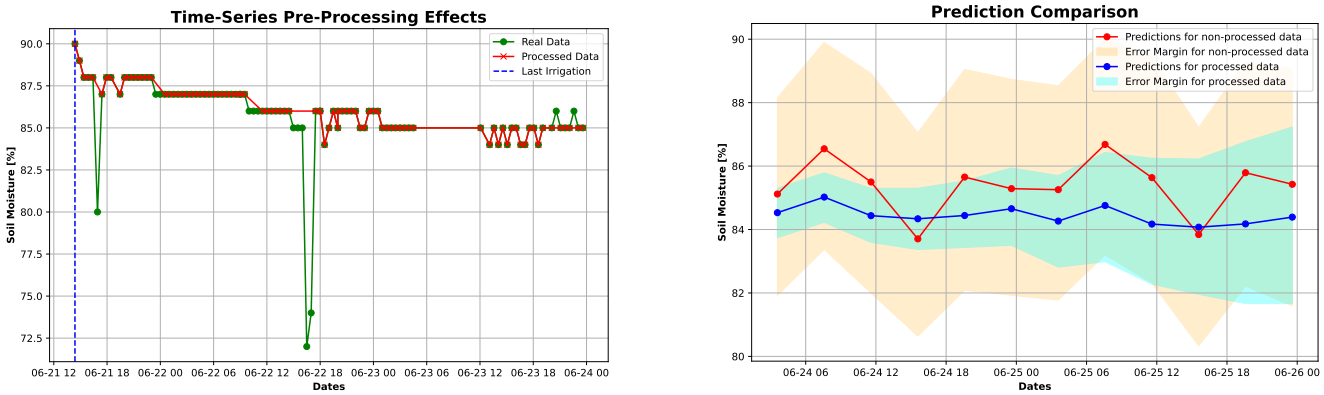

Fig. 3. Impact of Spike-Removal pre-processing on time-series and prediction accuracy.

amount of available water. The plants specification dataframe is accessed, and those requiring watering are identified by accessing a specific field within such a data structure. For each plant that needs watering, its last recorded moisture datum (M_l) is fetched from the InfluxDB instance. Information concerning the target moisture level (M_t) and the root volume (V_r) is retrieved from the dataframe as well. Based on these data, the water required by each plant (W_p) is estimated as follows:

$$W_p = V_r \cdot \frac{(M_t - M_l)}{100} \qquad [cm^3] \tag{4}$$

If the water in storage is sufficient, a PUT request is sent to `<pumpID>.local`, containing the amount of water to be delivered to the corresponding plant. Otherwise, the required water must come from the public water system. The amount of water used from both sources is recorded after estimation, and the dataframe is updated to temporarily keep track of these information.

All operations involving connectivity may give rise to errors. To account for these events, error messages are defined. At the end of the entire irrigation procedure, a Telegram notification is sent to the end-user containing a text file reporting its outcome, including any encountered error message and the amount of water used, divided into components from personal storage and the public water system.

D. Forecasting Module

The forecasting unit is based on a Python script leveraging the capabilities of Facebook's `Prophet` library to predict the trend of the plants' soil moisture. The primary phenomenon to be monitored and forecasted is the decay of soil humidity, which is affected by both a decreasing and seasonal trend. `Prophet` was chosen for its automatic nature, which eliminates the need for parameter tuning, and its strong performance in predicting time-series with significant seasonal effects on a yearly, weekly, and daily basis.

Upon launching, the program first accesses the plants' specification dataframe to retrieve necessary information for each plant, such as root volume and the date of the last irrigation. It then obtains the amount of water in the personal storage via a GET request to the water management sensor node. Additionally, the script leverages the OpenWeatherMap API to obtain weather forecasts through an HTTP GET request, evaluating upcoming rainfall in millimeters per unit area. This operation allows for estimating the water that will be collected in the storage, based on the collecting area (e.g., house roof or cross-section of an open cistern), and the amount of water the plants will acquire autonomously, considering the root area specified in the plants' specification dataframe.

The program fetches the moisture data history from the InfluxDB instance and manages the data locally using a `pandas` dataframe.

As illustrated in Fig. 3, the moisture time-series sometimes exhibits undesired and unjustifiable spikes, given their magnitude and relatively short duration. Therefore, the dataframe containing moisture data undergoes pre-processing before being fed into the `Prophet` model. Dedicated functions are designed to filter the time series properly and remove such spikes.

Irrigation activation alters the environment being monitored, introducing potential autocorrelation in the time series. To address this, the first function filters out all data points before the last irrigation date, preparing the dataframe for `Prophet`. The second function evaluates the mean value and standard deviation of the filtered time-series over a 6-hour moving window, removing any data points whose deviation from the mean exceeds $1.5\times$ the standard deviation. The phenomenon is assumed to be statistically stationary within this time window. Fig. 3 illustrates the impact of the final processing step on the time-series, highlighting its significant benefits for forecasting. In fact, the figure demonstrates a substantial reduction in prediction error margins following the removal of outliers.

Although this step could be implemented directly on the micro-controller to off-load the Python application, the micro-controller periodically sleeps and may encounter faults causing hours of delay. The Forecasting unit can process data with a reliable time window, whereas the micro-controller would only rely on individual data points without ensuring the length of the time window, potentially invalidating the stationarity assumption.

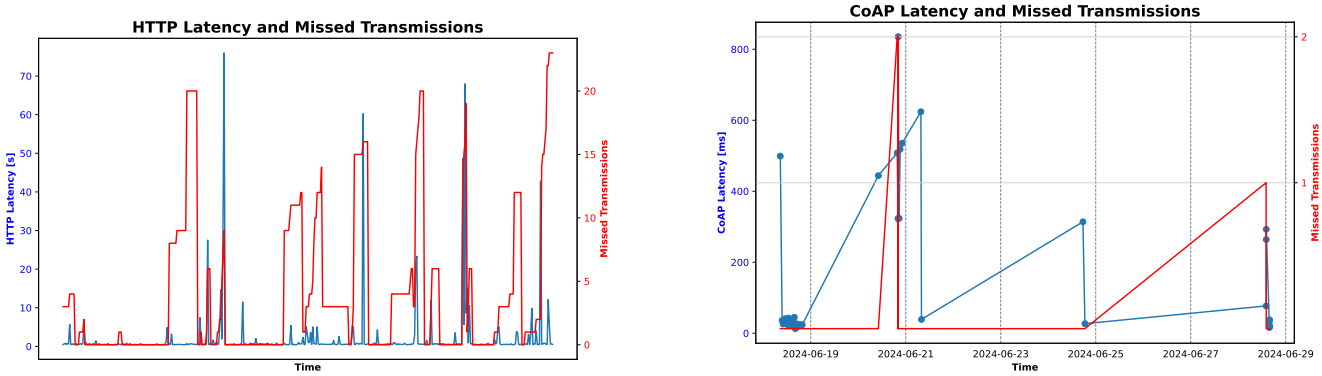

Fig. 4. Latency and Packet Loss measures for HTTP (left) and CoAP (right) protocols.

Once cleaned from outliers, the time-series is fed into the `Prophet` model, predicting 12 samples at 4-hour intervals to cover 2 days. The predicted values are transmitted to the same InfluxDB instance as the real data. Based on the predicted moisture data and water availability, the amount of water to be delivered to each plant is estimated as described in Section III-C, specifying the components from the personal water storage and those from the public water supply system.

This iterative process repeats for all plants in the dataframe, with all evaluations recorded in this data structure for each plant until the next prediction.

IV. Results

The evaluation of DAIS performance included assessing the following metrics:

- mean latency in data transmission to the Proxy or to the Cloud
- packet loss rate
- Mean Square Error (MSE) in the moisture data forecast

To evaluate both the CoAP and HTTP connection latencies to the Data Proxy and the InfluxDB instance respectively, this metric was measured as the time elapsed from the packet's transmission by the sensor node to the receipt of the server response. These measurements were conducted directly on the ESP32 board, which then forwarded the results to a dedicated InfluxDB instance for storage. Concurrently, the ESP32 board recorded missed transactions by incrementing a dedicated variable every time a WiFi connection or data transmission failure occurred. This variable was also made available on InfluxDB.

Visual representations of these evaluations are presented in Fig. 4 for HTTP and CoAP protocols, respectively. The plant-monitoring node transmitted a total of 998 samples directly to InfluxDB using the HTTP connection, featuring an average delay of $2080.04ms$ and a packet loss rate of around 20%. In contrast, the CoAP protocol sent 40 data points to the Data Proxy, exhibiting an average delay of $181.8ms$ and a packet loss rate of 7.5%. This lower rate for CoAP compared to HTTP was unexpected, as HTTP was anticipated to perform better. This anomaly can be attributed to the significant difference between the two sample sets. While the HTTP transmissions spanned approximately 30 days of continuous operation, CoAP transmissions covered less than 2 days in total. Additionally, the local WiFi connection between the sensor node, the router, and the Data Proxy likely contributed to the superior CoAP results. In contrast, the HTTP connection had to forward the data to an external server, possibly much further away in terms of physical distance. Furthermore, when evaluating mean transmission times, both successful transmissions and instances of missed transmissions were considered. HTTP connections experienced idle intervals of up to 75 seconds, while CoAP transmissions adhered to the time-multiplexing approach in Fig. 2, which limited idle time to a maximum of 5 seconds.

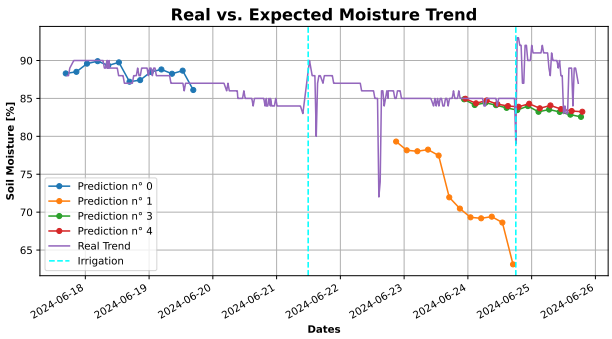

Fig. 5. Section of the recorded soil moisture trend for comparison with forecast data.

Regarding the Forecasting Unit, Fig. 5 shows a section of the recorded soil moisture trend along with some predictions. As mentioned in Section III-D, the presence of undesired spikes significantly influences the quality of the predictions. Three different predictions were made after the occurrence of the last spike. While *prediction 3* and *4* approximately follow the trend, *prediction 1* is divergent. This discrepancy can be attributed to different factors. First, the prediction was made immediately after the spike sample, which led to `Prophet` failing to recognize it as a glitch. Second, the spike-removal preprocessing step described in Section III-D had not been

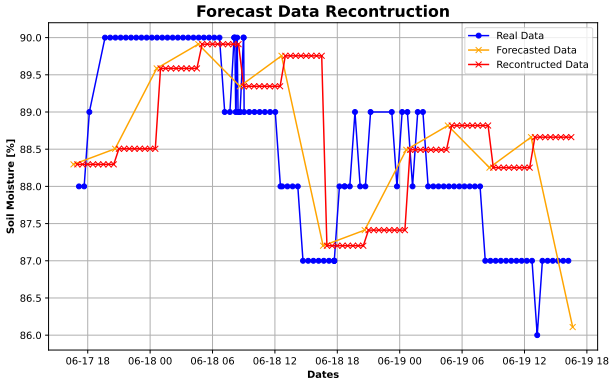

Fig. 6. Forecast data reconstruction method to enable the MSE evaluation.

implemented yet.

For performance evaluation, only *prediction 0* was considered. The other predictions were excluded either because they were too inaccurate to provide relevant results or, as in the case of *prediction 3* and *4*, the predicted values triggered the irrigation procedure and altered the environmental conditions being monitored. The performance assessment involved calculating the MSE between the original and forecasted data. Due to the difference in the number of samples in the series, the predicted sequence underwent a resampling technique described in Fig. 6. Specifically, each of the 12 predicted samples was replicated every 30 minutes, covering a 4-hour interval until the next forecasted data point. In this context, the MSE was found to be 1.12014. The low magnitude of the MSE can be ascribed to the fact that the sequence to be forecasted is a series of relative data ranging from 0% to 100%. This results in a contained standard deviation, even in the presence of undesired spikes, favoring good predictions. On the other hand, the error is low considering that the set of points on which the `Prophet` model was trained was reduced by the filtering step mentioned in Section III-D, which deleted all the data history prior to the last recorded irrigation, significantly reducing the set of training points.

Thus, this analysis highlights the importance of preprocessing steps like spike removal to ensure the accuracy and reliability of predictions made by the Forecasting Unit.

V. Conclusion

This paper presented a smart irrigation system designed for next-generation houses. After describing the system architecture, significant space was dedicated to explaining its implementation and performance. The analysis highlighted the system's adaptability to larger-scale scenarios compared to the experimental setup. Moreover, the fact that soil moisture predictions can be affected by a limited amount of historical data suggests that this system can also be implemented to monitor and predict other phenomena, such as water usage by house tenants during different daily activities. In fact, the metric to be measured would not trigger any countermeasure altering the environmental data, as was experienced in the irrigation procedure. Consequently, DAIS would not need to delete the data history prior to a given event, thereby increasing the quality of predictions. Thus, DAIS may be scaled not only to larger irrigation scenarios but also to entirely different situations where water usage could be monitored for future forecasts.

References

[1] Espressif Systems, "ESP32 Series Datasheet", version 4.5, pp. 29, 2024.