Immagine che contiene testo, Carattere, logo, simbolo

Descrizione generata automaticamente

**Software Engineering for the Internet of Things**

**Prof. Davide Di Ruscio**

**A.Y.: 2024/2025**

**A monitoring sytem for an Hydroelectric Power plant**

Students:

* Marco Giarrusso, id: 231133, e-mail: marco.giarrusso@student.univaq.it
* Fabio Paolantonio, id: 293028, e-mail: fabio.paolantonio@student.univaq.it

Github project link:

https://github.com/oobooee/MySe4IoT\_Public.git

**Introduction: Benefits of Hydroelectric Power**

Green hydroelectric power is a renewable and sustainable energy source that leverages the natural water cycle to generate electricity. It provides a clean alternative to fossil fuels, producing no direct greenhouse gas emissions, thus helping to combat climate change. Hydroelectric plants offer high energy efficiency, with conversion rates exceeding 90%, making them one of the most effective forms of renewable energy.

They also contribute to energy security by utilizing local water resources, reducing dependency on imported fuels. Hydroelectric systems can be highly flexible, quickly adjusting to demand fluctuations and serving as a backup during peak periods, enhancing grid stability.

Additionally, reservoirs created by hydroelectric dams support water storage, irrigation, and flood control, promoting agricultural productivity and disaster mitigation.   
Hydroelectric plants have long lifespans, offering cost-effective energy generation over decades, and their modular designs allow for scalability and integration with modern technologies, ensuring adaptability to future energy needs.

1. **The SE4IoT Hydropower System Project**

The purpose of this project is to monitor, analyze, and represent data collected from remote IoT devices installed in a simulated hydroelectric power plant. Typically, a hydroelectric power plant consists of two main components: a storage reservoir and a downstream power station that harnesses the water's flow from the reservoir to generate electricity through the operation of a turbine.

In this specific simulation context, the data is generated randomly using Python scripts and functions. However, the model has been carefully designed to ensure consistency and that the simulated values remain reasonably close to real-world data.  
To further enhance the realism of the simulation, factors such as seasonality, resilience to critical situations related to reservoir limits, and approximate modeling of real-world methodologies for electricity production (e.g., water head height, fluid dynamics, and mass flow) have been incorporated.

The main goals of this project are to provide an overview of all components, enabling the end user to:

* Monitor the infrastructure and energy production in real time.
* Access historical data and summary reports.
* Detect anomalies and receive alarm notifications.

Immagine che contiene schermata, testo, Software per la grafica, Software multimediale

Descrizione generata automaticamente

1. **Functional Requirements:**

* Monitor the IoT devices serving the reservoir and its tributaries, in particular:
  + The lake level
  + The estimated quantity in cubic meters of water
  + The inflow water flow rate
  + The outflow water flow rate
* Monitor the devices of the plant with the readings of:
  + The total historical energy produced
  + The instantaneous energy produced
  + The energy required by the grid
* Monitor the turbine and its components in particular:
  + The operating power, current, voltage, and its efficiency
* Record the historical values of the components for the purpose of:
  + Optimizing production, taking into account periods of drought or unforeseen events and the trend of energy required by the grid
  + Optimizing maintenance intervals
  + Preventing failures or incidents
  + Providing data to third parties for in-depth analysis
* Reporting anomalies and sending synthetic reports of data and alarms.

1. **Non-Functional Requirements:**

* Portability
  + The architecture supports modularity with Dockerized containers for each component, allowing independent scaling and deployment of services such as MQTT, RabbitMQ, Telegraf, and databases, facilitating horizontal and vertical scalability.
* Scalability
  + The system dynamically adapts to changing conditions (e.g., seasonal variations, fluctuating inflow and outflow) and allows customization of dam parameters (e.g., max height, initial efficiency) through external configuration files instead of hardcoding values.
  + Changing the .env variables and run the clients code let the system to be ready to manage secondary (and more) dam, maintaining on the databases both the data in the relative path.
  + More dashboard can be created to aggregate values or more data can be extracted doing specific queries over the multiple dam subfolders.
  + With ad hoc queries over influxdb the user can compare, for example, the TurbinEfficiency value between two or more dam systems in different remote sites.
* Resilience
  + Telegraf is configured with dual agents (both running MQTT and RabbitMQ) to ensure redundancy and continuous data acquisition, even in the case of partial system failures.
  + The system uses RabbitMQ and MQTT for message persistence and delivery. RabbitMQ ensures reliable message queuing with durable queues to prevent data loss, while MQTT supports lightweight, real-time communication between sensors and the system, making it suitable for IoT environments. Both the protocols are deployed with the appropriate callbacks to ensure reconnections over network or system failures.
  + The project employs two distinct databases—InfluxDB for time-series data and MySQL for relational data storage enabling data separation and failover mechanisms for reliability and data integrity.
  + The use of redoundant containers as for example nodered that chan be completely shutted off without no significant effects on the final visualization dashboard on Grafana, or the failover telegraf agend, let the system to be reconfigured,
* User friendly design
  + The project integrates Node-RED and Grafana to create visual dashboards for monitoring sensor data, system status, and alerts in real time.
  + The system check the threshold values and dynamically change the UI with colors and scale to give the the user a better user experience
  + The code, nodes and the database structure are self-explicative to give to the user the benefits of better understandings what each component does.
* Security
  + The system includes authentication mechanisms for MQTT and RabbitMQ using username-password pairs to restrict unauthorized access.
  + Communication channels are secured with credentials stored in environment variables or configuration files.
  + Database credentials are configured through environment variables to avoid exposing sensitive data in code.
  + The use of Telegraf agents provides isolation of data flows, reducing the risk of data manipulation or unauthorized access.

1. **System architecture**

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

After developing a prototype using the Wowki platform—described in detail below but limited by the lack of AMQP protocol support on ESP32 devices—the system was migrated to a Docker container. This migration overcame the limitations of the online platform and enabled further development using the Pika library for Python.

The system described above collects data from sensors, aggregates it into JSON format, and forwards it to an MQTT broker and RabbitMQ. The code implements reconnection mechanisms to handle faults.

A separate thread for MQTT and AMQP communication ensures robustness and reliability in the simulation script.

Using the Node-RED middleware, it was possible to develop all components and later integrate or replace Node-RED with Telegraf agents.

The rationale behind this change lies in the fact that Node-RED would act as a single point of failure in the infrastructure, as it was responsible for acting as both an MQTT and AMQP client, managing the middleware, and writing data to databases.

Node-RED was initially used to create a prototype of the entire infrastructure, which was later expanded with additional features.

Two Docker containers were subsequently developed, each running Telegraf agents. These agents, duplicated in another container, replaced the Node-RED subscribers. They subscribed to the Mosquitto broker and consumed messages from the RabbitMQ queue.

Duplicating the container was a simple process and provided the infrastructure with the quality attribute of resilience.

Later, the same agents were equipped with an output script in the configuration file, allowing them to write data directly to InfluxDB. The data was organized, for consistency and readability, in the same structure as the defined topics.

The incremental development approach did not involve full duplication of all messages. Instead, data related to the reservoir and its tributaries is published to Mosquitto, while data related to the turbine is published to an AMQP queue.

Expanding both systems dynamically and making the structure even more resilient would be straightforward.

Although the Node-RED container is no longer critical for the system’s survival, it still performs the following functions

* Monitoring the status of its nodes and reporting any malfunctions
* Subscribing to the RabbitMQ queue and writing the data to InfluxDB
* Writing hourly readings to MySQL
* Reading the data daily and generating a summary report via email
* Displaying the data on its internal UI for immediate verification

Thanks to MQTT message persistence and RabbitMQ queue consumption, the Node-RED and Telegraf containers can run simultaneously without introducing significant overhead to the infrastructure. At the same time, they ensure high fault tolerance and resilience.

1. **Wowki Platform for Simulation, Testing, and Verification of the Simulated Model, Threshold Parameters, and Alarm Validation**

Unlike other IoT systems, which typically monitor parameters such as temperature and humidity, in this specific context, the values and readings are closely interrelated. For example, the inflow rate affects both the water height and volume, while also influencing evaluation parameters to determine actions, such as opening hypothetical safety gates or favoring water accumulation. For this reason, the simulation model needs to be as accurate as possible.

In this particular context, simple regulation functions have been introduced to prevent overflows or to meet the network’s demand for water flow toward the turbines. However, the main focus has been on data acquisition, verification of consistency, and storage for consultation, evaluation, and analysis purposes.  
Initially, the Wowki platform was used to create real IoT sensors that simulated flow meters, actuators, potentiometers, ammeters, and wattmeters. The primary objective, besides providing immediate feedback on readings, was to test the system under critical conditions—for example, lake height variations, turbine faults, and in general, to validate thresholds and alarms.  
The system sends MQTT readings to a public broker. The sensors related to the turbine are entirely simulated and are not physically represented on the board or connected to the ESP32.

The final version is available at this link:

https://wokwi.com/projects/414848196626808833

The system consists of:

* Three actuators simulating the percentage of inflow and outflow and the power demand from the grid.
* A sensor simulating the lake’s height and its approximate volume.

The height and volume values are generated through appropriate calculations, although these values are treated as if they were sensor readings rather than being calculated.  
All these values can be modified in the global variables of the MicroPython script.

Random micro-faults are generated in the turbine values, and once detected by the system, they reduce overall efficiency.

Setting potentiometers to zero will make the model completely random. By adjusting them from 0+1 to the maximum value, it is possible to manually control the described inputs.  
For example, to quickly increase the lake’s height, simply maximize the inflow rate and minimize the outflow rate.  
The model simulates inflow by considering seasonality, outflow rates, and other dynamics.   
In addition to publishing specific topics for each IoT subsystem, it also publishes a summary JSON containing all readings from the entire system.

Immagine che contiene testo, schermo, schermata, diagramma

Descrizione generata automaticamente

Enforce 0 to 100% fixed sensors data for testing purposes

Local feedback

Height

Power request

Water out

Water in

Fig. 1: The Wowki simulation environment

1. **Transfer of MicroPython Source Code to Python Code within a Dedicated Docker Container**

After conducting the simulation through the online platform and recognizing the need to introduce the AMQP protocol, which is not compatible with MicroPython used online, the code was redesigned and restructured within a Docker container, with several additional features implemented.   
Specifically, the code is now responsible for simulating:

* A data aggregation gateway that collects data from remote sensors to generate a coherent output in a standard format (in this project: JSON)
* Importing global parameters such as height, flow rate, volume, initial pergentage for testing purposes and the unique ID to identify the plant in the database with the name:  
  Immagine che contiene testo, schermata, Carattere, numero

  Descrizione generata automaticamente
* Dedicated threads, functions, and callbacks for MQTT and RabbitMQ (connection, disconnection, reconnection, publish operations).
* Simulation functions for sensors (inflow, outflow, lake height, volume, energy demanded by the grid, turbine efficiency).
* Simulate and dynamically calculate the power sensor (in Megawatts) using the formula:  
  wher P = power produced (in watts, W), η = system efficiency, ρ = water density,   
  Q = water flow rate (in m³/s), g = gravitational acceleration, h = height of the water drop or head (in meters)
* Introduce variations to evaluate the generated values, making the user experience realistic, along with micro-faults that affect efficiency and production capacity.
* Publish data to MQTT topics (flow values, production, and threshold parameters) and RabbitMQ (turbine values), simulating a data aggregation gateway.

1. **The MQTT and RabbitMQ containers**

Below are the Docker Compose configurations for the RabbitMQ and Mosquitto containers.  
After configuring Mosquitto through its configuration file, including a password file with security credentials for the clients, the RabbitMQ configuration is also shown with access credentials to meet the project requirements.  
The Healthcheck ensures that dependent services are started and ready, preventing inconsistencies or errors.

|  |  |
| --- | --- |
| Immagine che contiene testo, schermata, Carattere  Descrizione generata automaticamente | Immagine che contiene testo, Carattere, schermata  Descrizione generata automaticamente |

Fig. 2: Docker compose - mqtt and rabbitmq

Examples illustrating the operational status of the services are shown below.

|  |  |
| --- | --- |
| Fig. 3: rabbitmq exchange, queue and binding status | Fig. : mqtt published values |

1. **Nodered container and configuration**

The use of Node-RED ensures extreme flexibility throughout all phases of the project development. It allows the creation of client consumers, the ability to write and read data from the database, parse messages, and interface with external email delivery mechanisms.  
One of the first implemented features is shown in Figure 6: Node-RED, through the status node, can check the condition of another node and make decisions accordingly.  
In this case, whenever the services are either successfully started or become unavailable—for example, due to a fault—a notification email is sent to the administrator with a rate limit to avoid excessive alerts.

|  |
| --- |
|  |

Fig. 5: Node to check the element status

For subscribing to MQTT topics, an alternative MQTT library, node-red-contrib-mqtt-dynamictopic, was used. This library allowed for real-time topic customization by reading values from an .env file loaded by the Docker container:



This tool, with and input available, makes it possible to dynamically create the topic (msg.topic) to witch to subscribe to. However, its functionality is not perfect, as it subscribes to the broker but returns data as strings. These strings must then be manipulated again, which makes the library less than ideal.

Nevertheless, for our testing and prototyping purposes, it it’s enough. Here how it works: Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Another feature developed for the project was the ability to display all values generated by the simulated sensors through Node-RED's native UI, providing immediate feedback on their reliability.  
To make the system flexible and easily scalable, bounds, such as maximum height or maximum flow rate, are calculated dynamically and sets in the user interface.

This approach ensures a more consistent user experience and scalability of the infrastructure.  
For example, when using a gauge, the outflow rate must refer to the maximum value to provide the user with immediate and visually satisfying feedback. Figures 7 and 8 illustrate this concept.  
Additionally, the ui.control of Node-RED can be redefined so that the gauge values have a minimum and maximum range that is not hardcoded, but instead based on the values read.

|  |
| --- |
|  |

Fig. 6: Subscribe and parse data for the UI visualization and bounds set

|  |
| --- |
| dynamic range  Fig. 7: dynamic value and dynamic range limits |

The following figure shows the Node-RED consumers for MQTT and RabbitMQ, which are listening on their respective topic and queue. The acquired data is sent to the InfluxDB and MySQL databases:

* In InfluxDB, all data is stored.
* In MySQL, only a subset of data is stored, and at hourly intervals.

This approach ensures data persistence in a database (MySQL) that is not specifically designed to handle time series data.   
To keep the project simpler, the RabbitMQ queue is used only to handle turbine data, while MQTT publishes both turbine data and sensor readings from all devices.

|  |
| --- |
|  |
|  |

Fig 9: The data flow : a consumer amqp and a subscriber mqtt. Data are stored over influxdb, with extra failover on turbin data. Also, summary data are collected hourly and stored on mysql

The final result of the customized user interface is shown below. On the left, it displays the lake height value, in the center, the total energy produced (all-time), the instantaneous energy, and the energy demanded by the grid. On the right, it shows the sensors representing the water inflow and outflow rates, and finally, the turbine operating parameters, including voltage, current, power, and efficiency.

Immagine che contiene testo, schermata, Software multimediale, software

Descrizione generata automaticamente

Fig 4: The nodered embedded UI

1. **Telegraf containers and configuration files**

We now present the configuration of the Telegraf instances. The configuration files are identical, except for the MQTT client ID, which must be defined as unique. The two independent Docker instances provide a reliable solution. If one agent fails, the other can continue to operate autonomously, effectively replacing the first.  
Each agent defines a consumer for MQTT and another for the AMQP protocol. Each consumer is customized with specific settings to ensure that the data inserted into InfluxDB remains consistent under all circumstances. This guarantees that both Telegraf agents and Node-RED nodes write to the same table.  
The third component of the agent is the output, which specifies where Telegraf will store the acquired information. In this case, the Docker container URL and the authorization token must be defined. Finally, Telegraf telemetry is enabled to monitor the status of the agents directly in InfluxDB.

|  |  |
| --- | --- |
| Immagine che contiene testo, schermata, Carattere, Elementi grafici  Descrizione generata automaticamente  Fig. 8:Identical docker structure for telegraf    Fig. 9: telegraf images for both agents containers (primary and high availability), with healthy dependancies | Fig. 10: agent mqtt consumer configuration. The only manadatory attribute that differs is the mqtt client id.    Fig. 11: agent amqp consumer config |
| Fig. : output over influxdb    Fig. 13: Collect and store telegraf agents statistics | |

1. **Influxdb and mysql**

The databases used to store all sensor data are InfluxDB and MySQL. As mentioned earlier, InfluxDB will receive the complete dataset, while MySQL will store only a selected portion of the data.  
Below, we present the Docker Compose configuration for both databases.

|  |
| --- |
|  |
|  |

This solution meets a non-functional requirement for resilience and availability. In the event of a fault, the availability of information is still ensured through an alternative database.  
The configuration of InfluxDB includes the definition of the bucket and the creation of an authorization token. This token is used in Node-RED, Grafana, and the two Telegraf agents.

|  |  |
| --- | --- |
| Immagine che contiene testo, schermata, Carattere, software  Descrizione generata automaticamente | Immagine che contiene testo, schermata, software  Descrizione generata automaticamente |

Fig. 14: Retention period config and r/w api key for telegraf and nodered

In the following figure, we observe the structure of the bucket, which is divided into two sections for consistency—the first related to lake parameters and generated power, and the second dedicated to specific turbine telemetry.  
Note that two different systems exists, with different names. The .env value (eg:   
On the right side of the same image, the Docker configuration file for MySQL is shown. In this context, a simplified table is created, containing only the relevant information acquired from the sensors, which is necessary for generating summary reports.

|  |  |
| --- | --- |
|  |  |

Fig. 16: influxdb bucket with dam data, turbin status data, for each hydroelectric dam and mysql table to store syntetic data for report

1. **Grafana**

The use of Grafana is necessary to display historical data and trends, which is not possible with Node-RED. A custom dashboard has been created to visualize all the values acquired from the sensors.

|  |
| --- |
|  |

To retrieve values from InfluxDB, Grafana allows executing direct queries on the previously defined bucket.  
Below, we present an example where the average values within the range defined in the dashboard are retrieved for the power value. The same query will be used to retrieve the energy demand value from the grid.   
Some queries fetch the latest value, representing the instantaneous value, while others retrieve historical values for trend analysis.

|  |  |
| --- | --- |
|  |  |

In Grafana, the minimum and maximum thresholds for parameters are dynamically utilized to enhance the user experience during data visualization.  
One of the most interesting features of Grafana is its ability to override threshold parameters, similar to what was done in Node-RED.  
In this image, where the threshold parameters have been temporarily disabled, it is evident how the instantaneous power value is displayed on a gauge, scaled proportionally between the minimum and maximum values.  
The minimum and maximum values are retrieved through two separate queries.

|  |
| --- |
|  |
|  |

Another feature that has been implemented is the ability to leverage the structure of the InfluxDB, specifically designed to allow switching between readings from different dams in Grafana. In the following example, we see how the system dynamically reads the data stored in the database and automatically provides the option to switch from one dashboard to another.

|  |
| --- |
| Immagine che contiene testo, schermata, Software multimediale, software  Descrizione generata automaticamente |

The final result of the dashboard is shown in the following figure. The overview provides the end user with immediate and consistent visual feedback on all values acquired, both for the most recent moments and for the previous hours, related to the IoT sensors.  
  
  
Immagine che contiene schermata, Software multimediale, Modellazione 3D, Software per la grafica

Descrizione generata automaticamente

1. **Alerts and reports**

One of the project requirements is to generate alerts that are triggered under specific conditions.  
In Node-RED, we have already seen how to generate a report on the operational status of one of the services, which sends an email notification when the MQTT broker is not operational.

|  |
| --- |
|  |

Let’s now look at another email report that is generated daily at a preset time after reading the latest entries from the MySQL database. The report formats the data into text for the email and sends it to a mailing list using the email node, which operates via SMTP.  
An alert within the UI, displayed through the LED node, notifies the user of a critical situation if the lake level exceeds the critical threshold.  
This value could also have been sent via email, but for simplicity, it was not included in the email notifications.

|  |
| --- |
|  |

|  |
| --- |
|  |

A similar alerting mechanism can also be implemented using the SMTP configuration options within the Docker container to create a contact point.  
Subsequently, two demo alert rules were created, as shown below.  
The first rule is triggered when the lake height exceeds a preset threshold, while the second rule monitors the turbine efficiency and is activated if it falls below 60%.

|  |
| --- |
|  |

|  |
| --- |
|  |
|  |

When the system detects a parameter falling below a certain threshold within the preset alert time window, an alarm is triggered, and an email is sent, detailing the threshold values exceeded.

|  |
| --- |
|  |
|  |

1. **Future Developments and Improvements**

Many of the features introduced could be further improved. For example, the topic structure could be better redefined, and failover mechanisms could be added for the MQTT and RabbitMQ brokers. Additionally, the middleware logic built on the gateway could be entirely migrated to Node-RED.  
Using two InfluxDB databases instead of two different technologies could simplify data storage and improve consistency. Moreover, it would be possible to duplicate turbine and power plant messages on MQTT and vice versa, ensuring redundancy and data synchronization.  
To test this project, besides downloading the code from GitHub, it is necessary to reconfigure contact email addresses in Node-RED and Grafana to enable the reporting and alert functionalities via email.  
The learning process throughout this project has been progressive, as many concepts were new to us. For this reason, we apologize for any inaccuracies or errors in the code or procedures.