IoT-Based Indoor Air Quality Monitoring System

Andrea Gurioli, Mario Sessa
Project of Internet of Things , A.A. 2021/22
andrea.gurioli2@studio.unibo.it, mario.sessa@studio.unibo.it
0000984711 - 0000983529

Abstract: This report shows the infrastructure, business logic, and components of an Indoor Air Quality Monitoring System based on ESP-32 devices and DHT-22 and MQ-2 sensors. Specifically, we introduce considerations and analysis of the final product according to the followed development pattern. We divided this document into the following structure: in section I, we introduce some supported contexts where we can use the system; in sections II and III, we discuss the project's architecture and implementation from an analytical point of view. In section IV, we analyze results on performances and tests related to the monitoring system.

1. Introduction

Internet of Things is a significant field of Computer Science applied in many environments: Agriculture, Urban, and Smart Homes. Generally, with the improvement of the quality of life and the continuous increment in the domestic time living, the quality of our home environment has become very important. Furthermore, the gas leaks in the house could be hazardous to personal safety. The proposed project has as its central idea to build a system to monitor the presence of extended gas concentration, indoor temperature, and humidity. These parameters are given by two types of sensors: DHT-22 and MQ-2; these sensors are connected to one of the ESP-32 in the environment and communicate in an interactive system with the capability to control each sensor remotely, calculate additional information (AQI, air quality indicator and RSS, received signal strength) and modify monitoring setup (minimum and maximum gas concentration related to modifying the AQI parameter and sample frequency related to the interval of state updating on sensors values connected to one of the indoor devices). The direct communication of sensors is related to the integration with a proxy server that can forward communication between monitoring dashboards, databases, forecasting servers, and devices. The proxy server controls state sessions composed of business logic values, metadata, general attributes, and monitored measurement states. This server provides a service to manage the data management with Influx Database; it maintains a connection with time series of monitored features and updates them with the new sensor's measurements and forecasting values. Influx Database has a series of collections formed by time series of measurements, filtered by GPS and device identification, to maintain a global or relative view about value changing over time. The Influx database was connected to a Grafana Dashboard with some links on the Monitoring Dashboard in the front-end and plots to display forecasting data given by the forecasting server and real measurements filtered by devices for every data sent from sensors. Finally, a complete front-end monitoring dashboard aims to display sensors on a map, visualize Grafana plots, and set up the device's hyper parameters or execute some tests on the communication. In addition, we built a Telegram Bot service as a notification service in case of a trigger on Grafana Alert given by a constant value of AQI above the standard value, which we will discuss more accurately in the next section.

2. Project's Architecture

2.1. Sensors

The core of data sensing is based on an Arduino-like standard for micro-controllers. For instance, the development board used in the development phase was an ESP32 board. The sensors needed for our aims are the temperature sensor and a gas sensor, respectively, a DHT22 and an MQ-2 sensor. The overall architecture is designed in order to communicate with the proxy server exploiting two different protocols:

- **MQTT**: It is a lightweight, publish-subscribe, machine-to-machine network protocol. It is designed for connections with remote locations that have devices with resource constraints or limited network bandwidth. It must run over a transport protocol that provides ordered, lossless, bi-directional connections—typically, TCP/IP.
- **CoAP**: It enables those constrained devices to communicate with the wider Internet using similar protocols. CoAP is designed for use between devices on the same constrained network (e.g., low-power, lossy networks), between devices and general nodes on the Internet, and between devices on different constrained networks, both joined by an internet. CoAP is also being used via other mechanisms, such as SMS on mobile communication networks.

As we transfer data by two different protocols, we have to handle the connectivity for two different communication standards; when it comes to the MQTT architecture, the communication is grounded on the TCP network protocol. As we follow the MQTT standard, the sensor must publish on a broker server its data. The cornerstone of this protocol, together with the lightweight packets, is the publish-subscribe messaging mechanism which allows us to uncouple the sensor, which is working as a client, and the proxy server, which binds our data sensing part with our data storing module. The MQTT protocol is exploited as a data transmission protocol in regards to our sensors' data and meta-data, which involves:

- **Board Identification:** It is hard-coded in the firmware and defines the device uniquely in the monitoring network.
- **GPS:** It is latitude and longitude values related to the geographical position of the sensors. It is hard-coded in the firmware.
- **RSS:** It is the Received Signal Strength defined by the Wi-Fi module of the micro-controller.
- **Temperature:** Temperature is given by the DHT-22 sensor in Celsius.
- **humidity:** It is the percentage of humidity in the air, the DHT-22 sensor provides this value.
- **Gas Concentration:** It is defined by the MQ-2 sensor, which measures the concentration of gas like smoke or alcohol in the air.
- **AQI:** Air quality indicator is defined below and indicates an evaluation metric for the air type in the indoor environment.
- **Sample Frequency:** The time in milliseconds represents the interval between 2 measurements on the DHT-22 and MQ-2 sensors.
- **IP**: When the micro-controller is connected to the Wi-Fi, it receives a public IP making him visible for the internet communication.

Except for the AQI value, all the variables are sensed or hard coded in the firmware. The AQI variable is an integer variable that spans between 0 and 2, computed by calculating the mean gas value on five measurements moving average window. The AQI value is computed as follows:

$$AQI = \begin{cases} 2 & \text{if avg} \leq MAX\_GAS\_VALUE \\ 1 & \text{if MIN\_GAS\_VALUE} \leq avg < MAX\_GAS\_VALUE \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

These variables are sent with the COAP and MQTT protocol. However, the MQTT protocol is employed not only for sending data but also for receiving meta parameters for data acquisition and filtering, such as the SAMPLE_FREQUENCY, MIN_GAS_VALUE and the MAX_GAS_VALUE. The MQTT subscribe phase in the board is also used to start the testing phase of the MQTT protocol, to receive the ping signal for the testing MQTT protocol, and to switch from one protocol to another. The COAP protocol is the second protocol implemented in our architecture. The main problem that occurs with the COAP protocol is given by the URI consciousness needed by our client. In order to solve this problem, the MQTT is settled as a starting protocol; the IP variable transmitted is then stored and used later for the COAP transmission. As the protocol changes, the overall architecture needs to be redesigned; the sensor switched in COAP mode is now settled as a Server and receives requests from the proxy client for data sensing and protocol testing.

The architecture is developed as a multi-sensor paradigm; all the phases described above, except for the soap protocol, exploit the sensors' ids for a double check in data transmission and receiving. Parameters such as sample frequency need to be sent, keeping the focus on each sensor at a time. As the MQTT protocol 'send' mechanism works by sending to all the 'topic' subscribers the data going through the broker, a filter is needed.

For the sensor testing phase, the aim is to check the overall delay (RTT) for the packet transmission and the packet delivery ratio, which is addressed to the proxy server in both protocols. The board switched in COAP protocol works as a server, given that the COAP testing phase is addressed to the proxy server while the board computes the MQTT test.

In the MQTT testing phase, the board works as a publisher and receiver, being able to compute the results all by itself; given that, the MQTT test can be triggered both by the board with the serial input and by the backend server with a topic subscribing. The overall result is given by the mean of five RTT resulting times. The proxy server addresses the package delivery ratio in both cases and is described later.

2.2. *Proxy Server*

Proxy server represents one of the central component of the overall infrastructure. It maintains the business objects related to the connected micro-controllers in the monitoring network and forwards communications and services between components. Specifically, it retrieves measurements from connected and authorized devices in two different ways:

- **MQTT Broker:** in the startup phase, proxy subscribe itself to some topics on a fixed MQTT Broker. These subscriptions channel let the proxy server to stay in listening of new states or data provided by sensors devices. Furthermore, we will use this broker as a notification channel in a bidirectional way; in other words, we publish some messages on ad-hoc topics to send requirements for a communication test, switching the protocol for devices communications to a specific host or changing metadata like sample frequency or gas bounds for AQI definition.
- **CoAP Messages:** when a micro-controller changes its protocol to CoAP, it became automatically a server and, synchronously with the device sample frequency, the proxy server sends CoAP requests to retrieves sensors measurements in response.

During the first communication, this component adds a session business object related to the current state of the connected micro-controllers, and it maintains information about:

1. **Board Identification:** it represents the unique key used to manage directly related host data within a unique reference to the session object. It is also used in the monitoring dashboard to retrieve communication, protocol switching, and metadata setup information.
2. **Board IP:** we maintain the public IP of connected ESP-32 for the CoAP server discovery service. This service allows the proxy server to maintain a table of possible endpoints requiring data directly within a request. This service is able only if the micro-controller has CoAP as the current runtime protocol.
3. **protocol:** it represents the current protocol used in the communication. We use this to display information on the dashboard, set up switching services, and perform communications settings with the device.
4. **Sample Frequency:** it is used to maintain a consistent monitoring flow with the CoAP requests and communication testing with simulations within accurate context metadata.
5. **Testing Metrics:** a device can switch to the testing mode. If the proxy server triggers this event from the dashboard, setup the communication in CoAP as a standard communication, but, during the message analysis, it can retrieve additional information about packages times and timestamps, which are helpful for the runtime measurements related to the percentage of package loss and communication latency.
6. **Status:** it is a measurement changed at runtime about the maximum delay. The maximum delay is related to the sampling frequency of the devices multiplied by five packages. If, in this interval, the proxy server does not receive data from the device, it will be considered disconnected. Otherwise, it is connected. A device can reconnect quickly with its session object sending easily new data to the proxy server.
7. **Mode:** it indicates if the current state of a device is related to the testing task on its current protocol or in a normal mode.
8. **GPS:** it is used to insert the device's geographical location into the dashboard map but also as tags for the Influx filtering system on the monitored data.

In addition, session objects can have temporal metadata related to a specific task like the list of measurements for the latency calculus or the last partial latency average in the communication during the testing mode. We will discuss more in detail in the next paragraphs.

Sensors Registration. When a new device connects to the proxy server, it does not accept direct data. This behavior is related to the authorization mechanism related to the administration dashboard. A device must be registered in the proxy server if it wants to connect and interact with the monitoring network. The dashboard has a form where the administrator can insert the device's identifications and let them join the network. This feature is useful for malicious sensors who want to inject malicious or noisy data into the Influx database.

Hyperparameter Setup. The monitoring dashboard can send some information to change metadata related to the communication features, AQI definition, and even the prediction length. The proxy server accepts these data and controls if the input identification is present and connected to the monitoring network. Then, using MQTT communication with quality of service set to 2 sends a message to a specific topic with the identification of the changed micro-controller. Devices read these messages and accept the setup changing only if the specified identifier is their own. A particular parameter is the prediction length which is in common for every device's measurement predictions.

Alive Service. In a real Internet of Things environment, there is the probability of losing the connection with one of the connected sensors' endpoints. According to this observation, we implemented a continuous evaluation system related to the current sample frequency's micro-controllers which can detect if one of the connected endpoints does not send correct data for a time interval equal to $I = SF * DT$ where DT is the fixed delay value and SF is the last updated sample frequency related to the monitored device. In the current prototype, DT is set to 10.

Switching Service. Proxy server processes the dashboard setup data and can switch the protocol of a connected endpoint from MQTT to CoAP or vice versa. In case of a switching event, the proxy server writes in a specific MQTT data related to the identification of the interested device and the new protocol with a quality of service set to 1; this choice is given by the evidence that, generally, if a device is in a specific protocol and receives a switching message with the same communication paradigm, there is no effect on the performances or in the current runtime settings.

Testing Service. Our architecture maintains the last values related to protocol testing in the session object. However, the dashboard can perform only a testing phase on the current protocol because endpoints cannot communicate data in two different ways at the exact times. For the same reasons, they cannot perform testing with different protocols simultaneously. We adopted two different approaches for testing in CoAP and MQTT according to the communication mechanisms:

1. **CoAP Testing:** During the normal proxy requests on one of the CoAP endpoints, we retrieve some additional operations to measure the current latency and package loss according to the sample frequency values. The current evaluation works following this pattern:
   (a) Proxy server receives a testing mode flag from the dashboard and sets its session mode attribute to 1.
   (b) During request sending, we trigger the session model as a particular condition that starts a subroutine where it calculates the current interval between request and response. Suppose this interval is greater than the sampling frequency. In that case, the package is considered lost, and we increment a temporal attribute related to the percentage of losing package during the testing transmission. We transmit the package to re-evaluate the latency. This operation is done several times until the proxy does not go above a time testing limit or receives five responses correctly.
   (c) Finally, the proxy obtains data to perform the formula for metrics evaluation described below.
2. **MQTT Testing:** When the proxy receives a request for a test from the dashboard to an MQTT endpoint, it sends a message on a specific MQTT channel for the notification of testing mode on monitoring network nodes, and, if the device identification is equaled to the value defined in the message, it starts the testing mode. The micro-controller sends and receives some in an embedded channel some messages and evaluates the latency between sending and receiving. This mechanism is possible because devices can filter each message by the identification provided inside the payload, and endpoints can perform the testing procedure and send data simultaneously in one iteration of the loop clause.

Above, we defined procedures for metadata retrieval according to testing metrics; endpoints and proxy obtain an initial time T_{start_0}, than there is a continuous sending messages with timestamp $T_{start_i} \forall i \in [2..5]$ every F_d milliseconds and, finally, when proxy or endpoints receive 5 responses, each of them at time $T_{stop_j} \forall j \in [1..5]$ and define the timestamp of the fifth packages T_{stop_5}, these information can defines the package loss P_{loss} and mean latency L of a device d according to its sample frequency F_d as follow:

$$\begin{aligned} P_{loss} &= 100 - \left(\frac{5}{N_{req}}\right) \times 100 \\ L_d &= \frac{\sum_{i=0}^{5}(T_{stop_i} - T_{start_i})}{5} \end{aligned} \tag{2}$$

T_{start_i} and T_{stop_i} defines intermediate intervals in relation to the response i received by the proxy or the endpoints, N_{req} is the number of request sen during the test until receiving five responses. According to the previous procedures, endpoints calculates MQTT tests internally, meanwhile CoAP tests are always defined by the proxy server.

Data Storaging. Proxy receives every measurements from sensors internally; it uses also a Influx Manager internal component to store a sampling in the Influx time series according to sensors identifications, GPS location and type of attribute (bucket). Every information is given by the session object saved in the business logic session.

Forecasting Requests. proxy can request the Forecasting server calling embedded API. When a device sends data to the proxy server, it uses its device identification and the current prediction length to call a function on the Forecasting Server via API, which returns three different collections of predictions for gas, temperature, and humidity measurements. The size of each list is equal to the prediction length.

Outdoor Monitoring. proxy can call API to communicate GPS location to an external server which measures the external temperature of a specific location. In the response, the proxy processes the mean temperature according to latitude and longitude defined by an endpoint during the communication. This information will be stored in Influx Database with device data, measurements, and predictions.

2.3. Influx Database

Influx Database works like the storage layer in the overall architecture. This service provides to save data in time series. So, we can visualize and manage data and query over time and retrieve necessary information for tasks like forecasting or analysis of variable changes over time. Specifically, we built an Influx Database to store sensors and endpoints information: indoor and outdoor temperature, indoor humidity, gas concentration, AQI, and even RSS. Each data type mentioned above is defined in an Influx bucket, filtering by the internal tag system to distinguish endpoints using device identification, latitude, and longitude.

2.4. Grafana Dashboard

Grafana Dashboard is a view component directly connected to the monitoring dashboard. It consists of an interface with multiple panels which display time series of stored information related to their predictions (forecasting on gas, humidity, and indoor temperature). It is also the component that triggers an alert. Alert is a particular service provided by Grafana that can notify an external component of an event related to variable changing. In our case, we used it for the AQI alerting system. When the AQI is above 1, the alert system goes in pending mode until a fixed interval. If the value continues to be 2, an alert will be emitted to an external Telegram channel connected to an embedded bot.

2.5. Telegram Bot

During the AQI analysis, we have to trigger the event of variable changing if the AQI value reaches a value above 1. The Telegram bot is a service provided by Bot Father which can be used to notify some events to the end-user. In our case, we use it for:

1. Showing means query about the last values of one of the data stored in an Influx bucket.
2. Lists possible service commands as a manual of use.
3. Provides a listener service to trigger an alert in Grafana and notify with a message the end user.

The bot is connected to a specific public channel on Telegram; every user who is a member of the channel can receive the notification.

2.6. Forecasting Experimentation

Before entering the designing process of the forecasting server, an experimental phase to choose the forecasting algorithm has been done. All the experimental processes were made by exploiting the Google Colab platform, working with Jupiter notebooks as a standard for data science processes. The algorithms tested for the data forecasting are:

- ARIMA: It is a statistical analysis model that uses time-series data to understand better the data set or predict future trends. It can predict future values based on past values and uses lagged moving averages to smooth time series data. They are widely used in technical analysis to forecast future security prices.

- Recurrent Neural Network: We use an LSTM to define the memorization of the past series as a condition to establish the future prediction. It is based on a particular version of the Recurrent Neural Network widely used for complex and irregular time series with high dependency between their values over time.

ARIMA model The ARIMA model is an auto-regressive moving average model which works by combining values, exploiting

- **p** is auto-regressive terms
- **d** is the number of nonseasonal differences needed for stationarity
- **q** is the number of lagged forecast errors in the prediction equation.

This algorithm works only on stationary data, so, as a first step, a Dickey-Fuller test has to be done to check the data congruence with the requested data shape. The Dickey-Fuller is a hypothesis test where the non-stationary series give the null hypothesis. By obtaining a p-value less than our predefined threshold, which is 0.05, we can say that the null hypothesis is rejected, given that the data is stationary. Suppose the resulting p-value is greater than 0.05. In that case, the null hypothesis is accepted, and the data has to be transformed to a stationary shape in order to proceed with further investigation. Given the stationary data, the next step is to understand how to settle the ARIMA model parameters. For the p and q estimation, an auto-correlation and partial auto-correlation test must be done.

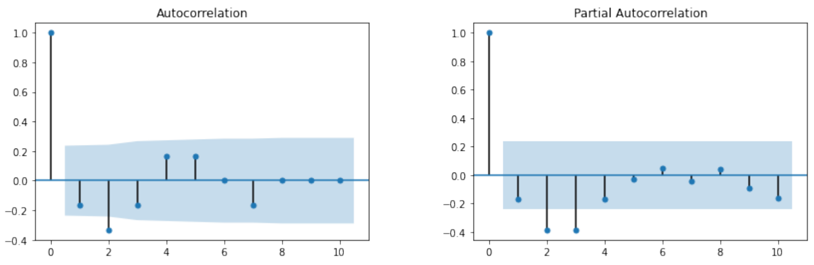

Fig. 1. Auto-correlation and partial auto-correlation tests

In these examples, the p variable obtained by exploiting the auto-correlation function is 1, and the q value obtained by looking at the partial autocorrelation function is one too. Given a low level of auto-correlation and non-stationary data, a differentiation order of one is the best choice. If the series has positive auto-correlations out to a high number of lags, then it probably needs a higher order of differencing.

Recurrent Neural Network For the recurrent neural network model, the first step to tackle in order to obtain a valuable input is to generate data for the day and year seasonality. The timestamp obtained by our time-series database is used to project day and year in a sine and cosine function. The projection is computed by calculating the number of seconds in a day and in a year in order to compute

$$day_s = 60 \times 60 \times 24$$

$$year_s = (365.2425) \times day_s$$

$$Day_s = \sin(Timestamp_s \times (2\pi/day_s))$$

$$Day_c = \cos(Timestamp_s \times (2\pi/day_s))$$

$$Year_s = \sin(Timestamp_s \times (2\pi/year_s))$$

$$Year_c = \cos(Timestamp_s \times (2\pi/year_s))$$

The models will make predictions based on a window of consecutive samples from the data. The main features of the input windows are:

- The width (number of time steps) of the input and label windows.
- The time offset between them.
- Which features are used as inputs, labels, or both.

The label window is the one that has to be predicted; The architecture model is based on an LSTM module for the RNN phase, followed by a dense layer with one neuron for the data forecasting. As a loss function, it has been used the mean squared error function with an Adam optimizer.

2.7. Forecasting Server

Forecasting Server provides some API to invoke according to different tasks:

1. **Sample Frequency Management:** It is an API called by the Proxy Server to notify the Forecasting Server about changing the sample frequency value related to one of the connected endpoints. This information is important to synchronize the model training with the forecasting invocation.
2. **Forecasting:** Proxy server can provide some information to carry out a forecasting invocation on this server. It calls a specific GET request with some parameters:
   (a) **Prediction Length:** Defined by the administrator of the monitoring dashboard. Its default value is equal to 4. It provides the number of forecasting values to predict by the final model related to a specific bucket of one endpoint.
   (b) **Device Identification:** We built a prediction model for every endpoint, so we need to refer to a specific time series related to an Influx query filtered by device identification.
   (c) **Bucket Type:** Due to the different training data, each endpoint has three different models stored in the local file system related to forecasting humidity, temperature and gas concentration. So, the Forecasting Server needs to distinguish which model to call for the predictions.
   (d) **Sample Frequency:** During the forecasting invocation, we need to update models with the last data provided by the related endpoint and stored on Influx. We use the sampling frequency to start or modify a thread that starts a new training phase with updated Influx data provided by a query, parallel to the forecast procedure. So, we can train and use a model with updated data at the same time.
3. **Training Model:** Training phase provides to update a model or create a new one and stores it in the file system of the host where the Forecasting Server is deployed. At the end of the training phase, the model overwrites the model file synchronously with the forecasting usage to avoid conflicts in the usage of the shared resource.

Thread management and multiple calls support API functions let available forecasting on multiple endpoints at the same time.

2.8. Monitoring Dashboard

Monitoring Dashboard represents the front-end of the entire application. It is composed of three different pages:

1. **Setup Tools:** It provides some forms and tools to manage the setup of an endpoint. Specifically, it has a service to register new devices provided by an identification code, a table with general information about connected devices like signal metrics (package loss and latency), and some functions related to switching and testing services. Furthermore, we can manage the setting of sensors metadata related to sample frequency and gas bounds for the air quality indicator (AQI). Finally, we also have a form to update the prediction length in common for every forecasting model of the system.

2. **Map Visualization:** It is a page with a map and markers to visualize the current position of each sensor connected to the monitoring network. If we click on one of the markers, we can see their GPS coordinates provided by the endpoint during the communication with the proxy.

3. **Grafana Dashboard:** This page is provided by the Grafana service; it is a link to the Grafana host related to the dashboard previously defined. We can filter panel information by a variable related to the identification of the endpoint and go back to the homepage of the Monitoring Dashboard by clicking on a link button.

Using the Monitoring Dashboard, an administrator is able to manage every feature of a connected device, even its authorization.

2.9. Monitoring Network Architecture

In this subsection, we provide an abstraction of the deployment diagram of the system for a better understanding of the component functionality.

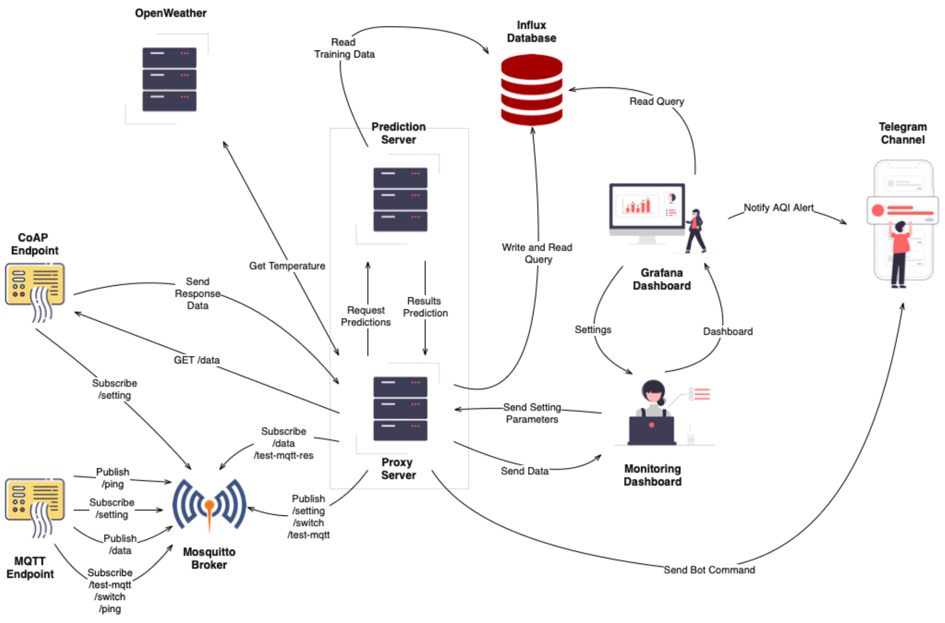

Fig. 2. Figure 1. shows the architecture of the Monitoring Systems provided by different component intercommunication. On the left, we can see endpoints communication using different protocols (CoAP and MQTT) with the ability to switch them in runtime. In the center, we can see the Proxy Server and Forecasting Server, which have the role of Gateway in the IoT environment. They provide some important management on business logic objects management and forward data, information, and API invocation between other components. On top, we can see the Influx Database for storing data and, finally, on the right, the front-end dashboards and their interactions with also Telegram notification channel.

3. Implementation

3.1. Sensors

One of the most important components of the overall project is the microcontroller and sensors. In this subsection, we will describe how we implemented the main functionality described in the architecture section, with particular attention to the library's usage. Micro-controllers are ESP-32 connected to two sensors: DHT-22 and MQ-2, which, respectively, retrieve data for temperature, humidity, and gas. Data processing uses *dht*22.*h* library to retrieve data from the DHT22 sensor; meanwhile, it uses an analogical reading directly from the data pin of the MQ-2 to retrieve the voltage related to the gas concentration. ESP-32 uses two libraries for the CoAP, and MQTT supports *Thing.CoAP.h* and *PubSubClient.h* respectively to make visible CoAP APIs for clients and adapt a communication pattern to the Publish-Subscriber design and connect the client object to an MQTT broker provided by Mosquitto. Data management procedures use JSON objects provided by the *ArduinoJson.h*, ESP-32 uses these objects to transmit data from different communication channels and merge data in a unique well-formatted structure. Internet connection to communicate with brokers and expose APIs for CoAP clients is defined by the classic *WiFi.h* library; it is also used to retrieve information like IP and RSS. In the next subsection, we will describe a more accurate description of the protocol implementation in ESP-32 devices.

3.1.1. CoAP Implementation

CoAP server listening is activated by the protocol mode variable during the loop checking. It provides only one API callable from the Proxy Server: */data*. The micro-controller makes measurements on sensors and defines a data package in JSON format, and then sends it in the CoAP message as a response.

3.1.2. MQTT Implementation

ESP-32 provides some subscriptions on MQTT topics about testing, data transmission, and connectivity checking. We can see in detail these topics as follow:

1. **Setup Topic:** It is used to retrieve possible metadata management for the micro-controller itself. We can receive gas bound for AQI evaluation and sample frequency modification.
2. **Switch Topic:** ESP-32 receives from this topic a new protocol choice related to the current configuration and the new one defined in the MQTT payload.
3. **Testing Topic:** When a message is triggered from this topic, the micro-controller starts a testing method related to a series of ping on the broker. It provides a subscription and a publishing session on the same channel to read and write from an embedded ping topic and measure the time interval between the sending and receiving the same message, and it also counts if there are lost messages. The mechanism of evaluation is defined in the architecture section.

In relation to these channels, we always have a notification topic to inform the Proxy Server of the correct state of the procedures related to the triggered message. In addition, the micro-controller publishes data on the */data* topic in a time interval defined by the sampling frequency.

3.2. Proxy Server

Proxy Servers use *express.js* to build the basic infrastructure for the HTTP intercommunication with the Monitoring Dashboards. It implements MQTT channels for MQTT micro-controller communication support using the official *MQTT* library. Otherwise, it uses *coapClientJS* library for CoAP support. We use the client packages because the Proxy server is a client actor that asks micro-controllers to retrieve sensor data in response. About the internal communication with the Forecasting Server, we use the *request* library to perform GET requests for predictions on a particular bucket for one of the connected micro-controller data forecasting. In this section, we will define only how we implemented the main functions related to the operations of the micro-controller, separating the proxy behavior and environments into three sections, each of them referring to one of the supported proxy protocols: MQTT, CoAP, and HTTP. Every function related to MQTT, CoAP, and even HTTP is defined in the *protocol.js* module.

3.2.1. HTTP Implementation

The following APIs focuses the implementation in *express.js* library, every functionality is implemented in *protocol.js* module and exported to *app.js* :

1. **Sensor Data Retrieval:** It is an API requested from the Monitoring Dashboard to push from Proxy Server to Monitoring Dashboard connected micro-controller data.

2. **Update Prediction Length:** It is an API function called to update the global prediction length on the proxy server. It will be used to retrieve the collection of forecasting predictions from the embedded Forecasting Server.

3. **Model Registration:** When an administrator from the Monitoring Dashboard calls it, the proxy server forwards the micro-controller identification to the Forecasting Server to be aware of the micro-controller registration.

4. **Node Registration:** It is a simpler function compared to the previous one, where the micro-controller device is registered only on the Proxy Server.

5. **Update Setup:** This is a function to update the session object on the Proxy Server and, furthermore, publish on an MQTT embedded topic the new setup for the forwarding on the referred micro-controller.

6. **Testing MQTT:** This API is used to invoke a testing mode on a registered micro-controller, and it takes track of the lost packages. It returns metrics related to the MQTT test to the Monitoring Dashboard.

7. **Testing CoAP:** This function is similar to the previous one. However, the testing procedure is different: Proxy Server does not send a message to the micro-controller for testing starts, but it continues to send CoAP messages to the micro-controller server and, meanwhile, monitors statistics about package loss and communication latency when CoAP obtains five responses, the function return to the Monitored Dashboard the required metrics.

8. **Switching Mode:** This API notifies the Proxy Server to update the protocol for a connected device. The proxy server publishes on an MQTT topic the forwarded information to update the micro-controller and change the protocol on a related session object.

Every API above are related to HTTP communication using *express.js*; the first is available in the GET method. Meanwhile, the others are POST API. The component that uses it actively is only the Monitoring Dashboard but, according to the introduction to this section, we have also another way to communicate data between components via HTTP: using the *request* library. So, Proxy Server is even a client for the Forecasting Server. It performs the following GET requests to the Forecasting Server:

1. **Change Frequency:** It is an API to change the sampling frequency of the Forecasting Server used to update the predictions model related to a specifically connected micro-controller.

2. **Forecasting:** It is a function that requires a prediction length, a board identification, a bucket, and the sampling frequency related to the micro-controller to optimize the re-training phase of the related model.

The proxy model implicitly controls the updating of prediction models giving the sampling frequency that defines the interval between two consecutive procedure invocations.

3.2.2. MQTT Implementation

Proxy Server uses MQTT to publish, set changes, and communicate on embedded channels with a micro-controller, the broadcasting nature of the publishing-subscribe pattern is filtered by the device identification. In other terms, the Proxy Server writes messages on topics but only the interesting device processes information inside. This filtering system is defined by the correct device identification matching. Furthermore, data transmitted in these channels are not private, so we do not have any possible vulnerability related to data leaks. Specifically, Proxy Server publishes data in the following MQTT topics:

1. **Test MQTT:** Proxy Server sends information about the interesting device by its own identification.

2. **Switch Request:** It sends the device identification and the new protocol to update. If the protocol is equal to the current one used to the micro-controller, updating is simply ignored.

3. **setup:** the proxy server sends information about sample frequency, minimum gas, and maximum gas or some of this information within the device identification on a specific MQTT channel to update the configuration data of a specific micro-controller. Proxy controls in the back-end if the minimum and maximum gas value or sample frequency are equal to acceptable values.

The $MQTT$ library also provides the functionality to subscription, independently from the node types. So, we used this feature to subscribe the proxy server to specific topics with the functions to receive additional notifications or communication from the microcontroller:

1. **Testing MQTT Response:** This topic is used to notify the Proxy Server at the end of the testing phase for the MQTT protocol on one of the connected microcontrollers. With the device identification, the message contains also results in metrics from the test.

2. **Switch Response:** This topic is used to notify the Proxy Server of a correct protocol shift after receiving a message from the previous Switch Request topic. This topic is useless in terms of confirmation of changing events because we also have the Quality of Service set to 1, but it is useful for the postponed updating on the business logic object related to the micro-controller.

3.2.3. CoAP Implementation

Proxy Server uses CoAP with *coapClientJs* library only in case a micro-controller switches to this protocol. During the configuration setting, the Proxy Server stops to receive possible MQTT messages from the micro-controller and starts to send CoAP messages. The IP is provided by the micro-controller registration from the previous communication. In fact, the micro-controller also sends its personal IP with data during the sensor's measurements communications. Proxy Server sends a request to the default CoAP port on the specified IP maintained in the session object related to the micro-controller on the URL $/data$. Requests are sent in an interval equal to the sampling frequency. This approach lets us know that a possible response has always updated data, reducing overhead and obtaining new measurements at each call. The testing phase uses a different approach from the MQTT method. Proxy Server measures the time interval using the timestamp provided by $Date$ javascript class and implements equations (2) previously described in section 2.2.

3.3. Influx Database

Influx Database is deployed locally in the same host of the Proxy and Forecasting Server. It uses $flux$ language as a query language for the best integration of the Javascript management functions defined in the $InfluxManager$. The $InfluxManager$ is a Javascript class defined in the $protocol.js$ and $bot.js$ to let available the interaction between Proxy Server, Telegram Bot, and Influx Database. Query management and write operations are defined internally of the $InfluxManager.js$ as methods. We use $InfluxDBClient$ library to perform different procedures:

1. **Writing on InfluxDB:** This function takes as input the micro-controller identification, GPS coordinates, bucket, and value to store and make a *Point* object. Using the $WriteApi$ object provided by $InfluxDBClient$ library, we are able to define tags related to the point (host, latitude, longitude, and prediction), which defines the micro-controller time series in the database uniquely even from its own predictions (prediction tag should be *no* in case of real measurements and *yes* in case of forecasting points. We used two strings to define these parameters because the library does not support false boolean values. This method returns false in the case of error detection.

2. **Query on InfluxDB:** This method is used to print values on the database from a micro-controller data correctly registered in Influx Database, filtering host, latitude, and longitude on real measurements. The submitted query was written in $flux$ language, and we use the *queryApi* object given by the $InfluxDBClient$ library. The method receives in input the query and print results.

3. **Query for Mean Value:** This function is defined for the Telegram Bot. It receives in input the bucket and the device identification and returns the current mean of the last values in the past 10 minutes. The query is defined internally in $flux$ language.

4. **Write Forecasting Points on InfluxDB:** This method is different from the simple write method because we need to store in the Influx database different points in specific timestamps. This method has as input the micro-controller identification, GPS location, collections of predictions, referred bucket, and the sampling frequency of the related micro-controller. The interval of each of the n written point is equal to $I_i = i \times SF; \forall i \in [1..n]$ where I_i is the interval of the point i, i is the position inside the collection parameter and SF is the sample frequency given in input.

3.4. Forecasting Server

Forecasting Server is implemented in the $app.py$ module on the $forecasting$ folder. It is simple. $Flask$ application which provides some function in the $RouteDecorator$ callable by GET requests. We use $pickle$ Python library to save and load forecasting models related to a specific bucket and host. During the training invocation defined by the internal function $update$ we use $pandas$ with the $influxdb\_client$ libraries to perform a query and retrieve Influx rows for training data on a specified model. Every trained model focuses ARIMA learning method provided by $statsmodels$ library. Depending on the bucket type, ARIMA will perform the training phase on different parameters according to experiments done in the Google Colab notebook. Additionally, every training invocation is defined in an interval and continuously executed over time on a specific thread. The mechanisms of parallel training are defined by the class $RepeatedTimer.js$, which defines a thread for each invocation, a time interval in seconds, and a function to call.

3.5. Monitoring Dashboard

Monitoring Dashboard focuses on the classic implementation pattern. The layout is defined in $HTML$, styles in $Boostrap$ and logic management in $PureJavascript$. Invocation of HTTP Proxy APIs focuses $AJAX$ structure. Furthermore, we provide a front-end notification system defined by $Swal$ library to provide some additional information related to the procedure called during the dashboard usage. Finally, we got the monitoring panels directly linked to $Grafana$ tools and a map view implemented with $Overleaf$ to show the current geographical locations of the connected micro-controller in real-time.

3.6. Deployment Diagram

About the component organization during the implementation, we configured Influx Database and Grafana locally; meanwhile, MQTT broker focuses on the academic Mosquitto broker available for students for the Internet of Things course, Bot and Telegram Channel are tested on a normal mobile device with internet connection and Telegram app installed, dashboards are available only in the local network as well as sensors transmission via CoAP protocol and, finally, we have micro-controller with its personal public IP available after the WiFi connection.

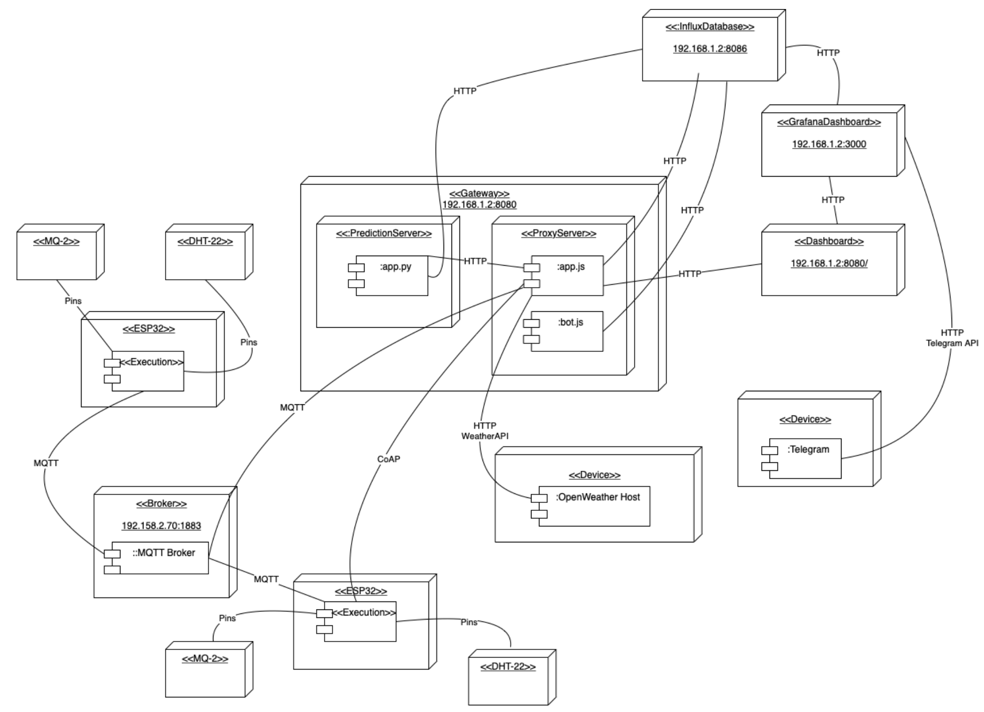

Fig. 3. Figure 2 shows how components are interconnected, protocol usages and application in the runtime environment during a demo applied after the implementation phase.

4. Results

4.1. Monitoring Metrics

In this subsection, we will analyze and discuss results about metrics on micro-controller connection, latency, and package loss percentage. Our experiment about the evaluation is defined by a study on a single micro-controller for both protocols. We will also display the mean value μ and related standard deviation σ; these values are related only to the latency.

Protocol	Latency	Package Loss
MQTT	41.30 ms ($\pm$ 1.63)	0%
CoAP	146.08 ms ($\pm$ 31.44)	17%

Table 1. Metrics are done on one of the micro-controller defined in the experiment. Micro-controller was connected at the same Wi-Fi of the servers

The evaluation considers a series of 14 testing with a temporal distance of 2 minutes where every component are connected to the same LAN, and the mean received signal strength is equal to -75dBm.

4.2. Forecasting Models Evaluations

In this subsection, we evaluate a standard model on the history of 8 hours. During this period, we let the original values for temperature and humidity and add an artificial gas generation in the air around the MQ-2 sensor to support possible variation in the final model. We used a Google Colab notebook to find the best ARIMA parameters related to our data. This approach lets us evaluate on a base model the forecasting behavior in a real context for a daily period. Table 2 shows how the gas concentration is unpredictable according to the confidence interval

Model Type	RMSE	Prediction Mean	Interval of Confidence
Temperature	0.019	29.91	[29.83, 29.99]
Humidity	0.118	40.52	[36.01, 45.03]
Gas	180.939	4095	[3567.68, 4622.31]

Table 2. Forecasting Metrics related to the Root Mean Square Error (RMSE) and the interval of confidence for the estimation of values on the model predictions.

and a real variation given by gas presence. We think that gas presence changes the value of the curve drastically, and the prediction model interprets it as outlier detection. According to the adaptation on the prediction with the current substitution of the actual value when it occurs in the Influx Database, the outliers model behavior carries out only at the first variation but, due to the sensitive impact of outliers on RMSE and MSE, presence of these value decrease drastically RMSE.

5. Conclusions

In conclusion, we can define some additional considerations for the overall project. Internet of Things patterns has additional problems compared to normal distributed or network architectures. First of all, the low-energy and low-computational power of the micro-controller introduces many constraints during the implementation phase. Gateways also have high complexity due to the runtime management of many flows of data. Forecasting methods were the most difficult part of the entire project; constraints related to the quality of data but even the parallel management of the continuous training on different models are a critical part during the design and implementation phase. Switching protocols in the runtime phase was one of the best features to adapt the proxy on multiple protocols support but even to manage data in a different way according to the best metrics related to performances on each protocol. Finally, we can say that the final result is adaptable for a deployment phase and run in a real context easily. This is useful for building a real air quality monitoring system in an indoor environment, even for sensor integration inside a smart home.