IoT-Based Indoor Air Quality Monitoring System

A project report for Internet of Things course at Univerity of Bologna.

Zhanel Zhexenova[1], Aleksi Edvard Zubkovski[1]

June, 2022

Abstract

With rise of popularity in various internet of things deployments a curiosity emerged on ways of rapid construction of small-scale, unindustrial IoT services. In this paper, serving as a technical report of a project done for Internet of Things -course at University of Bologna Alma Mater Studiorum, a full stack IoT service deployment is described. The project included programming an end-device using ESP32 microconroller embedded with sensors, implementing multiple communication protocols, a data proxy controlling data flows connected to a database, data analytics and visualisation service and control of the service implemented using bot running on a popular messenger service. The deployed service had been tested and its performance has been analysed.

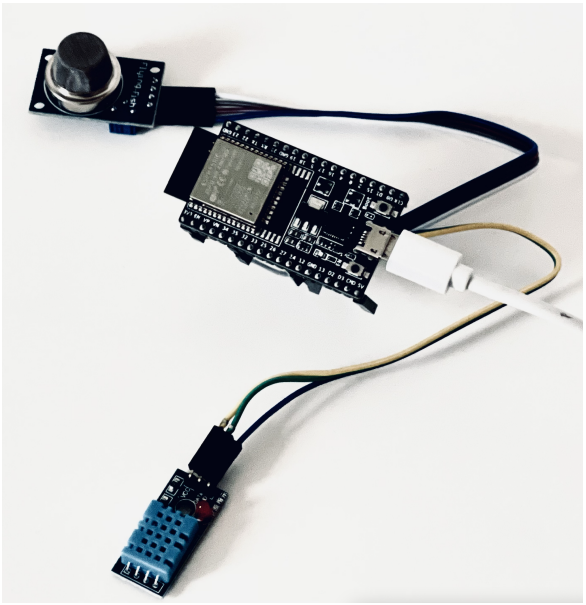

Figure 1: The End-Device in Development

1. Introduction

The idea for the project was to create an easily replicable and expandable IoT service with a single end device providing minimal functionality of monitoring indoor air quality and micro climate, its prediction and forecasting.

Additionally service management such as distant control over data flow, connection quality monitoring and control of end-device behaviour were to be added.

2. Project Design and Architecture

The Architecture is constituted of three parts, an end-device, server and database. The end-device is essentially a microcontroller embedded with required sensors and WiFi connectivity. It reads sensors, pre-processes data and handles MQTT and http connectivity. Data can be sent using both protocols, while instructions to the end-device such as frequency of collecting samples and adjusting ways of pre processing of the data, are sent only via MQTT.

On the server side the dataflow coming from the end device goes to data proxy, essentially, a Node.js runtime environment, through which server side scripting and dataflows are managed. It also handles internally the http server, connects to MQTT broker running on the server, gets temperature and humidity readings from OpenWeatherMap API [1] and handles how messenger bot operates through Telegram API. The purpose of the messenger bot is to provide an easy and accessible way of controlling the end-device from any mobile or stationary device. The data proxy also routes collected data to cloud instance of Influx time series database. The collected data is then displayed alongside with predicted values on Grafana visualisation platform where alarm rules were set as well. Predictions are done runtime using Facebook Prophet and Influx Python API.

[1] https://openweathermap.org/

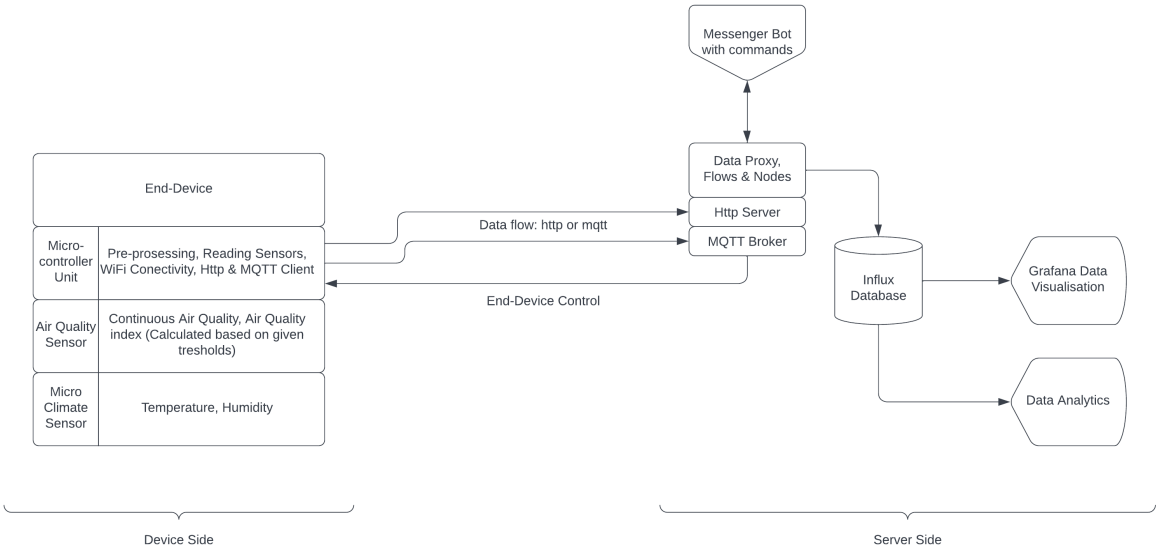

Figure 2: Project Architecture

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

Figure 3: Calculation of Air Quality Index

3. Project Implementation

3.1. End-Device

For the end-device a ESP32 is used for its low-cost and low-power and connectivity capabilities. It is a bit excessive for such project as it is, but considering expandability it fits well as a very versatile development board. MQ2 air quality sensor and DHT11 were chosen, as they are widely used, having become an industry standard, while providing well enough resolution and accuracy. The ESP32 was programmed utilising Arduino libraries for their ease of use.

From DHT11 sesor ESP32 reads room temperature and humidity through special one wire protocol. From MQ2 the ESP32 reads analog value representing air quality reading. This value is retransmitted to http or MQTT client, and additionally an air quality index is calculated to then add it to data pool. For each air quality reading we store five moving average values: adding new value read over data reading frequency and removing the last one. Then the average of recorded values is calculated.

According to Figure 3, the air quality index, AQI, is then computed. MAX.GAS.VALUE and MIN.GAS.VALUE can be modified from server side, thus changing the thresholds of acceptable air quality levels.

Another value added to the data pool sent to the server is WiFi signal's RSSI (received signal strength index), which can help with debugging networking problems on runtime and help with end-device deployment & placement.

3.2. Communication

Both MQTT and http protocols are used for exchanging data between server and the end-device. A possibility to change between protocols was added and is triggered from the server side. The ESP32 constantly runs a http client for http server on server side. The end device is also constantly connected to the MQTT broker running on the server. Since the sensor is supposed to operate indoors, all the connectivity is implemented on local network.

3.3. Data Proxy

Data proxy is implemented with Node.js using a higher level application Node-Red [2]. It drives dataflows with nodes, and runs scripts for handling data. For each dataflow there is a receiving node: e.g. node connected to MQTT broker's topic, when data is received, it is processed using script and sent to database as you can see from Fig 4. Node-red is capable of hosting http server on its own simply by adding the nodes required by the protocol. Also, it accesses current weather data from openweathermap.com via http and sends it to Influx.

[2] https://nodered.org/

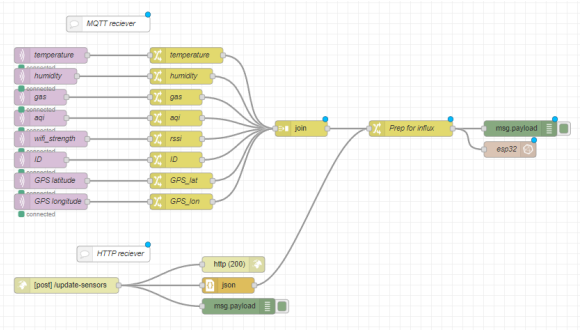

Figure 4: MQTT and HTTP flow setup on Node-Red

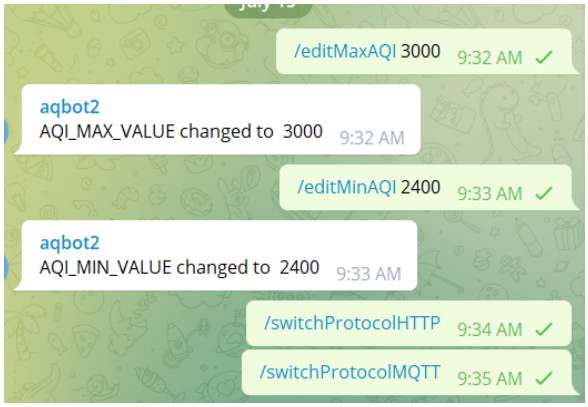

Figure 5: Example of Telegram bot usage

3.4. Service Management

The service is managed through a Telegram bot. Telegram messenger service provides an API for communicating with bots. The bot's functionality is implemented in Node-red using a library provided by Telegram. With the Bot it is possible to set sampling frequency, switch communication protocols (MQTT or http) and to adjust threshold values for calculating air quality level on the end device (Fig 6, 5). The command coming from the bot are received by the data proxy and routed to specific MQTT topics, to which the end-device constantly listens and reacts to received instructions instantly. It is also possible to request quick statistics of the measurements from the Telegram bot (7).

3.5. Database

For storing data an Influx Database was chosen for its capabilities in storing sensor data especially time-connected data. The database receives data straight from MQTT and http nodes which are converted to a JSON object there and then sent to the database, again through a Node-red node using a special library from InfluxDB. On the Node-red service some scripts are running capable of writing an InfluxDB query to the database at a command

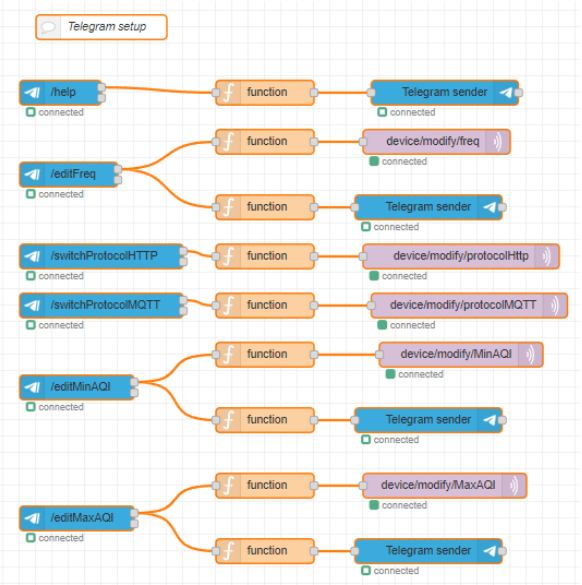

Figure 6: Setup of the Telegram bot's flow

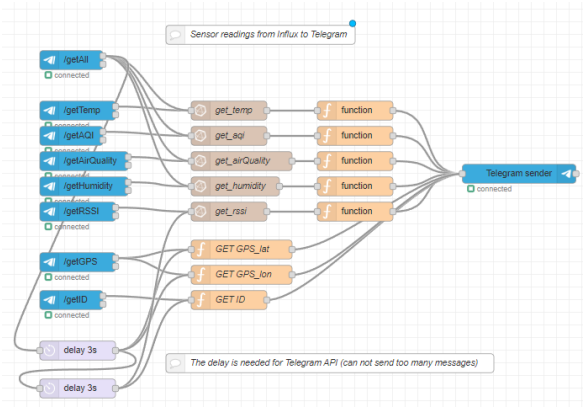

Figure 7: Setup of the Telegram bot's queries from Influx

coming from the Telegram bot and then sending received data back.

The data is managed through queries and sent to data visualisation service locally.

3.6. Data Visualisation

A popular visualisation service, Grafana, is used for visualising the data stored on the database, running on the server side (8).

For forecasting data an intelligent service based on capable of working with time series data, Facebook Prophet, is running as a Python script on the server. Having access to the database it provides a quite decent forecasting capabilities on the go adding new values to the training set every time.

4. Results

4.1. Network

In order to calculate time delay between sending and receiving a data package an experiment was conducted. Network time protocol library for ESP32 was used to timestamp upcoming and incoming messages. The placement of the end device in the building was altered in order to test the performance with bad RSSI values as well. Messages were logged for half an hour with good end-device placement (hightest available RSSI reading in the building) collected data stored in database, then protocols (http and MQTT) were changed or the device placed outside building in a fixed location (for bad RSSI) and process repeated. The packet loss ratio was calculated simply by estimating how many packets were supposed to be sent according to rules given to the end-device and observing how many packets were received by the database.

Protocol	MQTT	Http
Time (avg)	26 ms	31 ms
PLR (RSSI -50 – -60 dBm)	0 %	0 %
PLR (RSSI -80 – -90 dBm)	90 %	90 %

The results are rough estimates of the real performance, as the tests were performed in a field and not in a laboratory. At this time varying the time interval did not change results greatly, therefore they can be trusted.

The results are surprising in a way that MQTT requires a little less time than the http protocol, as node.js cas to process incoming MQTT data and build a JSON from all the topics in order to send it to the database, while http node doesn't do any processing on the server side and immediately sends the received data to the database.

	MSE	STD
Humidity	5.87	4.36
Temperature	0.76	0.32
Gas	3912.64	3207.90

Table 1: MSE and STD of predicted and actual values

4.2. Data Analysis: Forecasting

In order to compute MSE of the predicted humidity, temperature and gas values we queried the influx database to retrieve last hour's data (60 datapoints). Results for temperature are much better as variations indoors are rare due to the use of the air conditioning. What comes to the gas values, they itself variate throughout a day significantly from around 1600 to 2900, that may affected prediction accuracy.

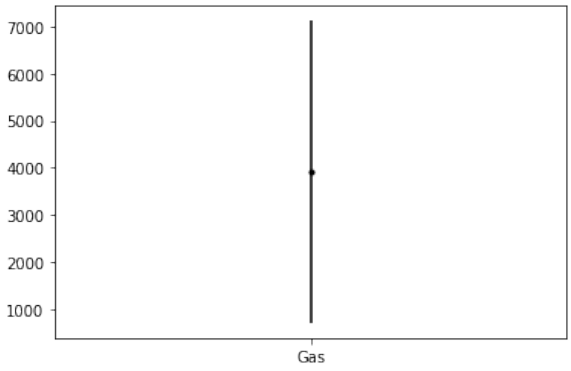

Figure 9: Gas MSE and STD

Figure 10: Humidity and Temperature MSE and STD

Average values of the predictions and actual values were also assessed as you can see on Fig 11, 12, 13.

Figure 8: Grafana dashboard

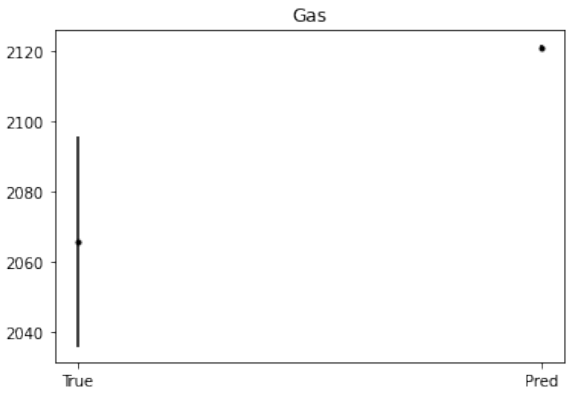

Figure 11: Average and STD of predicted and actual gas values

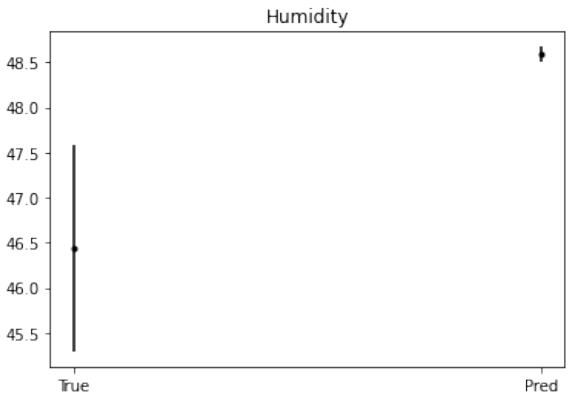

Figure 12: Average and STD of predicted and actual humidity values

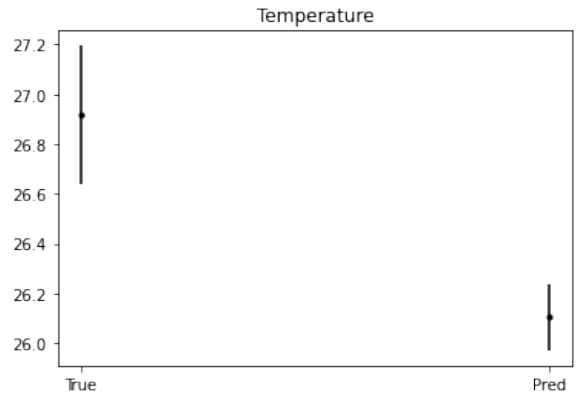

Figure 13: Average and STD of predicted and actual temperature values

4.3. Conclusion

In this paper an architecture and deployment of a simple "bare-bones" internet of things service was deployed and reviewed. It was shown how through having only little skill and close to none budget, custom internet of things services can be deployed. We have analysed some performance and tested service capabilities.

Suggested further improvements for the project could be e.g. making of a custom pcb and adding powering circuitry to the device, possibly choosing a less expensive microcontroller, as ESP-32 can be seen having excessive performance, and adding deep sleep cycling to the embedded side for reducing energy consumption as well as improving a bit the software running on the embedded part, since it was written using not so widely used Arduino extension for C++. Additionally a custom node.js service could be implemented for performance and versatility.

The server side was tested on a simple laptop, while it could be later implemented on a Raspberry Pi -like device, or on a server deployment service like Microsoft Azure or similar.

Predictions can be improved by increasing the amount of data for training, using periodicity of the data and fine tuning the parameters.

During writing project and deployment only few bottlenecks in production flow were noticed; firstly deploying whole system and merging all the processes required a significant workload for debugging, secondly; working with embedded was surprisingly time consuming, as there are simply way too many things that can go wrong.