Smart Water Bowl For Pets
IoT Exam Project

Marco Acerbis
Master's Degree in Artificial Intelligence
University of Bologna
marco.acerbis@studio.unibo.it

***Abstract*—In this work I am going to present my implementation of a smart water bowl for pets. In particular the system has been designed to monitor the water level and send an alarm when it results under a given, user configurable, threshold. As for the hardware, the project has been developed with the ESP32 VROOM board connected to two sensors: an ultrasonic ranging sensor (HC-SR04) and an temperature/humidity sensor (DHT22). The implemented data pipeline is based on the HTTP protocol to send sensor data to a *Data Proxy* that save the collected information in an InfluxDB and proceed to forcast the expected water lever using two different prediction models.**

The complete code developed for the project is available on GitHub.

I. Introduction

Pet owners faces many challenges to assure that their animal friends live a happy life. One of these challenges is providing fresh water to the animal(s), especially during the hot summer season, through the day. For these reasons, the Smart Water Bowl would offer an helping hand by providing constant information about the remaining water in the bowl and the statistics of relevant data, like temperature, humidity and average water level during the day.
The full system architecture, discussed in details in Sec. II, is based on the ESP32 VROOM which controls two sensors: an ultrasonic raging sensor to measure the water level and a temperature/humidity sensor to also account for the water evaporation during the day and the extra water consumption during hot days.
The data pipeline described in Sec. II-B consists in two modules, using different protocols to collect sensor data in a database and to allow the user to change some setting on the ESP32 board. Additionally, the collected data are also used by two forecast models that predicts the water level based on different input data.
The whole project has been developed using Python and the ArduinoIDE(C++), the complete code and an explanation on how to run the system can be found on this project GitHub repository.

II. System Architecture

Fig. 2 summarizes the system architecture that can be seen as divided into three parts: the sensors that collect relevant data, the data pipeline that manages, stores and makes those data available to a series of different services, and the forecasting models that make predictions on the available water in the bowl based on the information received from the sensors. These ones are discussed in details in sec. III.

A. Sensors

The system makes use of two different kind of sensors:

- the **HC-SR04**, an ultrasonic distance sensor located above the bowl in order to collect the water level;
- the **DHT22** temperature and humidity sensor that collects the information used by the forecasting model.

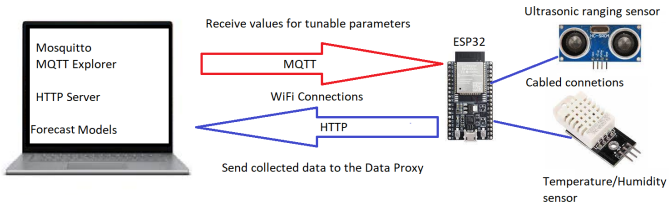

Fig. 1. Summary scheme of the implemented system architecture

B. Data Pipeline

In order to develop the required system, two communication protocols have been implemented. In particular, MQTT has been used to allow the user to tune some parameters, while the collected data are being managed by HTTP in order to be sent to the *Data Proxy* running on a laptop.

1) MQTT - Parameters Tuning: The MQTT protocol has been implemented to allow the user to change the following parameters on the the ESP32 board while the system is running:

- **Base water level (calibration)**: the user can specify the starting water level, i.e. the distance between the ranging sensor and the water in the bowl. This way the user is not forced to fill the bowl till a precise point and can adapt the amount of water, for example, based on water consumption.
- **Threshold**: this value specify the threshold level after which an alarm level is triggered. In particular, the threshold represent the maximum accepted value for the for the difference between the observed water level and the starting base water level, as summarize in eq. 1.

$$threshold = max\{observedLevel - baseLevel\} \quad (1)$$

- **Sampling rate**: this value allow the user to control the sampling rate of the data collection. The default value is 10s, and it can be changed to any desired time interval even if values lower than 2s can create some overheating problems in the DHT22 sensor.
- **Alarm Counter**: this values let the user decide how many alarm events are actually required before sending an alert for the low water level.

2) HTTP - Data Acquisition: The data collected by the sensors needs to be saved in a database, for data visualization and monitoring, and made available to the forecasting models to generate the expected water level. For these reason, an HTTP server has been deployed to establish a client/server communication between the ESP32 and the laptop running the Data Proxy services.

Fig. 2. The implemented system running

III. Implementation

In fig. 1, we can see the implemented system running: the data are collected by the ESP32 and sent to the Data Proxy using HTTP. At this point, data are stored in an database, and made available to *Grafana* for data visualization.
In the following each implementation of the system described in sec. II is discussed in details.

A. MQTT

On the ESP32, the MQTT communication has been implemented using the *PubSubClient.h* library for *Arduino*. Once the board is connected to the WiFi, it tries to establish a connection to an MQTT Broker service; if the connection succeeds the MQTT client proceeds to subscribe to four topics, one for each tunable parameter, otherwise a new try is attempted every 5 seconds.
As broker the for the publish/subscribe protocol, I decided to use *Eclipse Mosquitto* because it was easy to run, supported by Windows and free to use.
On the laptop, I have also installed *MQTT Explorer* in order to simulate a remote controller that allows to change the paramenters by publish the desired valued on the relative topic.

B. HTTP

I have opted for HTTP over Coap because this protocol is easier to implement, has more available libraries that offer a variety of functions and still performed in the desired times even being an "heavier" protocol.
To expose the **endpoint** needed to the board to establish a connection, I created an HTTP server, that runs on the laptop, by means of *Fast API* and *Uvicorn*.
On the ESP32, the Arduino library *HTTPClient.h* allows to create an HTTP client that performs a **post** request sending a Json file with the following data: sensor name, measured temperature and humidity, observed water level, starting water level, alert status, WiFi receiver signal strength (RSSI) and time. These information are then stored into an *InfluxDB* and made available to other services, like the forecasting models.

C. Data Processing and Visualization

We saw how the data generated by the sensors is then collected and transferred to the data proxy running on the laptop. Here the data are stored in an InfluxDB running in *Docker* container, and also sent to the forecasting models.

1) Forecasting Models: The collected data are used to make predictions on the water remaining in the bowl using two forecasting models:

- The *Simple Model* works with the assumption of a constant water consumption that leads to a loss of 1 cm in water level every hour. Its calculations are reported in eq.2, where SimplePred is the expected water level, BaseLevel is the starting distance between water and sensor, and dt is the up-time of the ESP32.

$$SimplePred = BaseLevel + 0.00278 * dt \quad (2)$$

- The *UpgradedModel* starts from a similar assumption as the previous model, but it also account for temperature and humidity: higher temperature means higher evaporation rate, but the evaporation is limited as the humidity goes up. The model follows eq. 3.

$$UpgradedPred = BaseLevel + 0.0015 * \frac{Temperature}{Humidity} * dt \quad (3)$$

2) InfluxDB and Grafana - Data Visualization: The data received by the data proxy has been saved in an *InfluxDB*, an open source time-series database, running in a *Docker*

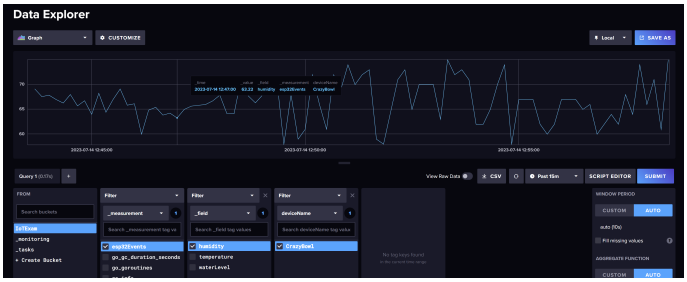

Fig. 3. Example of data stored on InfluxDB

container. In fig. 3, an example of the data collected in the database.

The data were also made available to *Grafana*, an analytics platorm for data visualization. The obtained dashboard, showing the trends for water level, temperature and humidity, is reported in fig. 4.

Fig. 4. Dashboards on Grafana showing the analytics for (top to bottom) water level, temperature and humidity

IV. Results

In order to measure the performances of the system and of our models, the following metrics have been adopted:

- **Latency** to measure the time required to the system to receive the data send from the ESP32;
- **Mean Square Error** between the predicted water level and the observed one, in order to evaluate model accuracy.

Some examples of the outputs are shown in fig 5. To measure the latency of the HTTP post performed by the ESP32, a request to an NTP server is performed in order to get the **epochTime** which is later subtracted to the epochTime measured by the the data proxy at arrival. Normally, the latency ranged between 300 ms and 750 ms, some lower measurements are generated by rounding errors when the ESP32 epochTIme is converted into an integer number.

On the other hand, the performances of the sensors where generally good, but a protract usage of the DHT22 generates some inaccuracies in the registered temperature values. Also, the DHT22 sometimes, as we can see from the results in fig. 4, requires some time to ”start up” and adjust the temperature values and tends to perform badly with a sampling rate lower than 2 seconds.

While the system performed quite well, there can be, in my opinion, some possible improvements, starting with the replacement of the distance sensor, for example, with a weight sensor to estimate the remaining water in the bowl. I wanted to make this consideration because the ranging sensor tends to move very easily when subject to even small vibrations, and the animal is usually limited in its actions or attracted by the sensor that might be confused for a toy o something to eat.

```
No need to refil the bowl.
Forecasted water level (simple model):  [8.84972176]  MSE:  37.2798358170946
Forecasted water level (upgraded model):  [5.7682750182637275]  MSE:  9.146238200578301
Latency:  20.721435546875  ms
RSSI:  -70.0
←[32mINFO←[0m:     127.0.0.1:63279 - "←[1mPOST /data/ HTTP/1.1←[0m" ←[32m200 OK←[0m
No need to refil the bowl.
Forecasted water level (simple model):  [8.86449468]  MSE:  37.460452728674426
Forecasted water level (upgraded model):  [5.941684753105565]  MSE:  10.225186526751413
Latency:  353.2218933105469  ms
RSSI:  -69.0
←[32mINFO←[0m:     127.0.0.1:63280 - "←[1mPOST /data/ HTTP/1.1←[0m" ←[32m200 OK←[0m
No need to refil the bowl.
Forecasted water level (simple model):  [8.87931764]  MSE:  37.6421201386507
Forecasted water level (upgraded model):  [5.892669907394605]  MSE:  9.914120951453787
Latency:  329.28013801574707  ms
RSSI:  -70.0
←[32mINFO←[0m:     127.0.0.1:63283 - "←[1mPOST /data/ HTTP/1.1←[0m" ←[32m200 OK←[0m
Low water level! Refill the bowl!
Forecasted water level (simple model):  [5.893904299999999]  MSE:  9.921895864396037
Forecasted water level (upgraded model):  [2.813301005394721]  MSE:  0.004802602182763473
Latency:  27.241945266723633  ms
RSSI:  -67.0
```

Fig. 5. Examples of system’s outputs