IoT Alarm System with Bed Presence Detection

Internt of Things a.a. 2023/2024

Accornero Andrea
Alma Mater Studiorum
University of Bologna
Bologna, Italy
andrea.accornero@studio.unibo.it

Magazzù Gabriele
Alma Mater Studiorum
University of Bologna
Bologna, Italy
gabriele.magazzu@studio.unibo.it

I. Introduction

This project implements an **IoT-based alarm system**, which enhances the traditional wake-up alarm by incorporating bed presence detection to improve the user experience. By integrating a pressure sensor, the system ensures that the alarm is triggered only when the user is physically present in bed, preventing unnecessary disturbances and making the wake-up process less intrusive.

The project follows a complete IoT data pipeline, beginning with data acquisition (where sensor readings are collected and processed by the microcontroller), followed by data storage (where information is recorded in a time-series database), and concluding with data visualization (enabling real-time monitoring and analysis of the collected data). The system also supports remote configuration via MQTT-based communication, allowing users to efficiently manage alarm settings. Additionally, users can control and customize the alarm system through a web application or a Telegram bot, enabling them to modify alarm schedules, activate or deactivate the alarm, change their location, and perform other relevant actions.

In the following sections of this report, we provide a detailed analysis of the hardware and software architecture of the system, discuss the implementation choices, and evaluate the results in terms of system accuracy, response time, and overall effectiveness.

II. Architecture

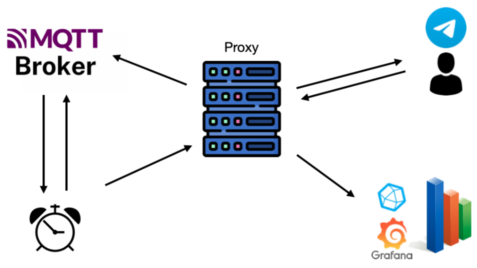

Fig. 1. Architecture

In Figure 2, we present a concise representation of the system architecture, where the proxy server is positioned at the center, acting as the core communication hub between the main components of the system. The proxy facilitates data exchange between the ESP32 microcontroller, the database, the web application, and the Telegram bot.

Below is a list of the hardware components that constitute the IoT-based alarm system:

- *ESP32 and DevKitC-32:* [1] A low-power ESP32 microcontroller, featuring built-in Wi-Fi and Bluetooth, used for data processing and communication, with a development board.
- *Speaker:* [2] An audio output device responsible for playing alarm sounds when triggered by the system.
- *Pressure Sensor:* [3] A sensor designed to detect pressure variations, allowing the system to determine whether the user is present in bed.

The software component of the system is designed to efficiently manage and process data from the hardware elements. It consists of the following key elements:

- *Data Proxy:* A Python-based application responsible for facilitating communication with users for alarm configuration, managing data transmission to the database, and sending settings to the MQTT broker.
- *MQTT Broker:* An intermediary that enables real-time synchronization between user commands and the ESP32 system by facilitating the exchange of alarm settings.
- *InfluxDB:* A time-series database used for efficient storage and organization of sensor data, enabling historical analysis and real-time monitoring.
- *Data Analysis & Evaluation Module:* A Python-based application that retrieves data from InfluxDB, processes it, and provides insights into system performance and user behavior.

The system includes two user interaction components: a **Web application** for configuring and monitoring the alarm and a **Telegram bot** for remote control and notifications.

III. Implementation

Below are the implementation choices for both the hardware and software components of the system.

A. Alarm

The alarm system is implemented using an ESP32 microcontroller, which serves as the central unit responsible for handling sensor data and controlling the alarm activation. The ESP32 is connected to a computer or a power source, ensuring continuous operation. It is also connected to a Wi-Fi network, allowing it to communicate with the data proxy via HTTP requests for sending sensor data, and with the MQTT broker to receive configuration settings and control commands.

1) Communication and Data Flow

- The ESP32 sends sensor data to the data proxy using HTTP requests, ensuring real-time monitoring of user presence in bed.
- The ESP32 subscribes to MQTT topics, allowing it to:
  - Receive updates on the **sampling rate** for sensor readings.
  - Get notified when the alarm should **trigger**.
  - Receive a command to stop the alarm.
  - Receive a command to adjust the alarm sound based on the weather conditions of the selected location.
- When the alarm is activated and the ESP32 receives the trigger command, the alarm plays the default sound through the connected speaker.

This implementation ensures that the alarm system is responsive and adaptable, leveraging both real-time sensor readings and remote configuration options to provide a smart wake-up experience.

The ESP32 is connected to a speaker and a pressure sensor, which is placed under the bed. The pressure sensor detects the user's presence, while the speaker plays the alarm sound when triggered.

B. Data Proxy

The *data proxy* is a Python-based middleware component that acts as the central communication hub between the **ESP32**, the **InfluxDB database**, and the **user interfaces** (Web Application and Telegram Bot). Implemented using *Flask* for HTTP requests and *paho-mqtt* for real-time communication via MQTT, it ensures seamless interaction and data management.

1) Alarm Configuration via Web Application or Telegram

The web application, hosted using Flask, serves as the primary interface for users to configure alarm settings. Through the web app, users can:

- Set new alarms by specifying the ID, time, and frequency.
- Modify pre-existing alarms by specifying the alarm ID and updating the time, frequency, or activation state.
- Delete existing alarms, either a specific one or all alarms.
- Dynamically update alarm preferences, including sampling_rate and location.

Once an alarm is configured, the proxy processes the request, stores the alarm details, and publishes the trigger command on the corresponding MQTT topic. Additionally, the data proxy runs a separate thread to manage time monitoring for active alarms, ensuring that triggers are executed at the correct time. The detailed implementation of the web application is discussed in a separate section.

2) Alarm Processing and Scheduling - Alarm Clock Thread

The proxy is responsible for managing all active alarms, ensuring that they trigger at the correct time based on the predefined user settings. The system continuously monitors:

- The scheduled time of each alarm.
- The frequency (once, daily, weekdays, weekends, or custom schedules, such as every Monday, etc.).
- The activation state (enabled or disabled).
- The location set by the user to retrieve the weather conditions.

The proxy continuously monitors the alarm list in real time and, when an alarm is due, it processes the request and publishes a notification to the corresponding MQTT topic to trigger the alarm on the ESP32.

3) Data Management & Storage (InfluxDB Integration)

The proxy plays a role in managing sensor data by storing it in InfluxDB, a time-series database specifically optimized for IoT applications. When the ESP32 collects pressure values, it transmits them to the proxy using HTTP requests. The proxy then processes this data and stores it in InfluxDB, ensuring that it is efficiently organized and readily accessible for further analysis. Additionally, this stored data is integrated with Grafana, allowing users to visualize both real-time and historical trends, making it easier to monitor and analyze system performance over time.

C. MQTT Broker

The MQTT broker manages the information published by the proxy across various channels, ensuring that the data is accessible and communicated to the ESP32 in real time.

- `iot/bed_alarm/sampling_rate`: Defines the interval for pressure sensor readings.
- `iot/bed_alarm/stop_alarm`: Sends a stop command to deactivate an alarm that is currently playing.
- `iot/bed_alarm/alarm_sound`: Selects the sound/music played when the alarm triggers.
- `iot/bed_alarm/trigger_alarm`: Sends a notification indicating that the scheduled time has arrived and the alarm should start sounding.

D. Web application

The system includes a Web application that provides an intuitive interface for users to configure and manage their alarms. The web app is built using `Flask` and serves as the primary means of interaction between the user and the system.

`Key Features`

- **Update Sampling Rate**: Allows users to adjust the interval at which pressure sensor data is collected.
- **Stop Alarm**: Provides a button to immediately stop a playing alarm.
- **Set New Alarm**: Users can define a new alarm by specifying a ID, time, and frequency.
- **Modify Alarm**: Enables editing existing alarms, including time, frequency, and activation status.
- **Remove Alarm**: Allows users to delete a specific alarm by entering its ID.

- **Remove All Alarms**: Provides an option to clear all stored alarms.
- **Set Location**: Users can select a location, which can be used for weather-based alarm customization sound.

E. Data Module

The Data Module is responsible for retrieving data from the InfluxDB bucket (or from a synthetic data source). It analyzes the collected pressure sensor data to determine when a specific user is sleeping during the night. Additionally, it calculates the average sleep duration over a selected time period, providing insights into the user's sleep patterns.

F. Telegram Bot

The system includes a Telegram Bot that provides users with an alternative method, compared to the web application, for configuring and managing alarms. The bot, developed in Python using the python-telegram-bot library, communicates with the locally running *Flask* server via HTTP requests (POST, PUT, DELETE).

For each configuration or management operation, the bot implements an error-handling mechanism. For instance, if a user attempts to create an alarm that already exists, modify a non-existent one, or remove an alarm that was never created, the bot receives a corresponding error (400 or 404) from the server and notifies the user. Similarly, if the user inputs a time or frequency in an invalid format, the bot prompts them to re-enter the data, specifying the required format or valid options.

Unlike the frontend, the bot also allows users to view the entire list of configured alarms, although it does not support setting a new sampling_rate value or a new location.

IV. Result

A. Accuracy to detect if a person is in bed or not

1) First approach

The pressure sensor records values ranging from `0` to `4095`. In our initial approach, we collected these values based on the defined sampling rate and conducted practical tests by placing the pressure sensor under the mattress. When a user is lying down, the sensor consistently registers values of `4095` indicating the presence of the user in bed.

However, an issue arises when the user is not lying down while the sensor remains under the mattress. Occasionally, the sensor produces sporadic spikes of `4095`, generating anomalous values that could lead to misclassification.

To address this, we implemented and tested two supervised machine learning models, using labeled data to classify whether the user is in bed or not. Below are the performance metrics for one of the tested models:

TABLE I
Classification Performance Metrics

Metric	Value
Accuracy	0.9435
Precision	0.8996
Recall	1.0000
F1-score	0.9471

Additionally, the confusion matrix breakdown provides a detailed evaluation of the model's classification performance:

TABLE II
Confusion Matrix

Metric	Value
True Positives (TP)	215
False Positives (FP)	24
False Negatives (FN)	0
True Negatives (TN)	186

These results indicate that while the model achieves high accuracy and recall, occasional false positives (FP = 24) occur due to anomalous pressure spikes. Specifically, when the user is not in bed, the sensor sometimes registers unexpected values of 4095, leading to incorrect classifications where the system falsely detects the presence of the user.

2) Refinement with Averaged Sampling

To mitigate the issue of anomalous pressure spikes, we decided to sample the pressure values as an average over multiple readings within the sampling rate interval. This approach helps smooth out occasional anomalous values, as an isolated spike of 4095 is distributed across the average, preventing incorrect classifications.

When the user is lying down, the sensor readings remain consistently at 4095. By averaging the values over time, the system ensures a more stable and reliable classification, effectively filtering out short-term fluctuations.

After implementing the averaging strategy, the model achieves perfect accuracy, as shown in the updated classification metrics.

TABLE III
Improved Classification Performance Metrics

Metric	Value
Accuracy	1.0000
Precision	1.0000
Recall	1.0000
F1-score	1.0000

Additionally, the confusion matrix breakdown confirms that all classifications are correct, with no false positives or negatives:

TABLE IV
CONFUSION MATRIX AFTER AVERAGING METHODS

Metric	**Value**
True Positives (TP)	246
False Positives (FP)	0
False Negatives (FN)	0
True Negatives (TN)	213

B. Mean Latency of the data acquisition process

We have measured the time it takes for the ESP32 (Arduino) to send pressure sensor data via HTTP to the proxy and receive an HTTP response in return. Assuming that the round-trip time is evenly split between the request and the response, we divide the total duration by two to estimate the one-way transmission latency.

Each data point represents the average latency calculated over three samples.

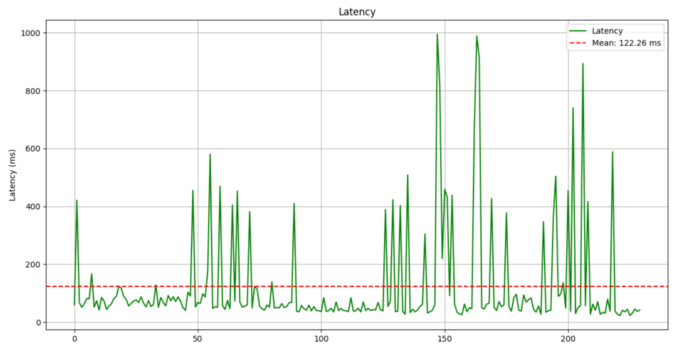

Fig. 2. Mean Latency

DEMO

To set up the system, ensure that the ESP32 is connected to a power outlet. Connect the sensors to the device according to the following pin definitions:

- Speaker - PIN 32
- Pressure Sesnor - PIN 33

After wiring the hardware, start the system by running the `proxy.py` script.

Additionally create the following files:

For the `credentials.h` file:

Create a file named `credentials.h` and include the following variables:

- `const char* ssid` (Wi-Fi network name)
- `const char* password` (Wi-Fi network password)
- `const char* serverName` (EndPoint to receive data)
- `const char* mqtt_server` (Address of the MQTT broker)

For the `.env` file:

Create a file named `.env` and include the following variables:

- `token` (Authorization token for InfluxDB)
- `TELEGRAM_BOT_TOKEN` (Token for the Telegram bot)
- `FLASK_SERVER_URL` (URL of the Flask server)
- `weather_api_key` (API key for the weather service)

REFERENCES

[1] ESP32
[2] Speaker
[3] Pressure Sensor