PROJECT REPORT

2025/2026

COMCS

G05

Rita Barbosa

1220841

Alfredo Ferreira

1220962

Introduction

This project implements a distributed IoT solution designed for the real-time monitoring of sensitive industrial environments. The system ensures product integrity by tracking temperature and humidity levels, with a specific focus on fault tolerance. It is architected to maintain continuous data logging and anomaly detection.

The solution consists of three interconnected components:

- IoT Monitoring Clients: devices responsible for continuous data acquisition.
- Alert Server: central processing unit for system logic.
- Command Centre: dashboard for operational oversight.

System Architecture & Components

The system architecture, illustrated in Figure 1, relies on a star topology where edge devices communicate directly with central servers. The physical layer consists of two independent sensing nodes, an ESP32 and a Raspberry Pi Pico, designed to provide redundant monitoring of the work cell. Each microcontroller is hardwired to a DHT11 sensor to capture temperature and humidity readings.

To facilitate both visualization and logic processing, the nodes maintain dual communication paths. Sensor readings are transmitted simultaneously to the Command Centre via MQTT for real-time display and to the Alert Server via UDP for differential calculation. Additionally, the Alert Server publishes alerts back to the Command Centre using MQTT when specific error thresholds are breached.

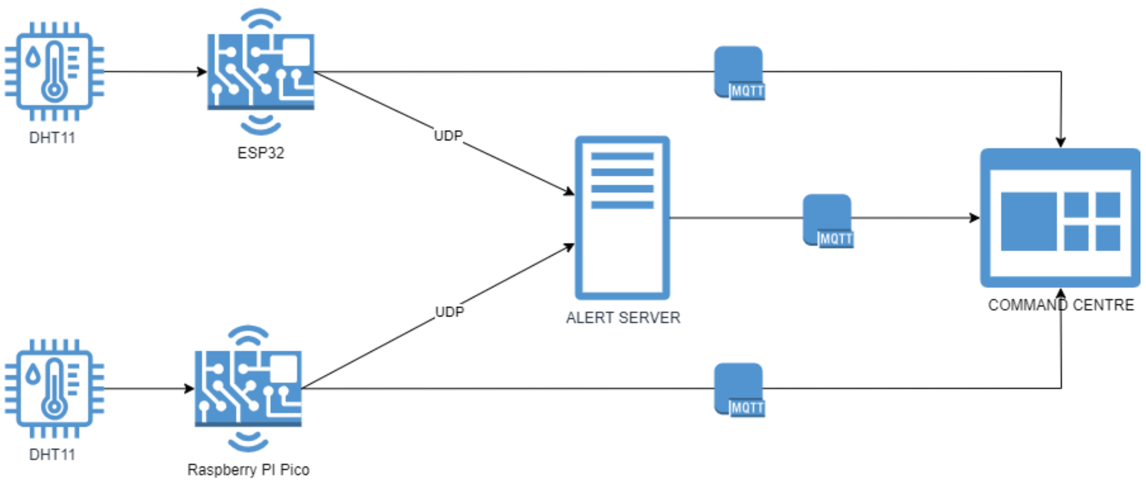

Figure 1: System Architecture.

Implementation Details

The IoT Monitoring Clients handle continuous data acquisition and transmission. For visualization, they utilize secure MQTT (SSL) to publish readings, serialized in SmartData JSON format, to the `/comcs/g04/sensor` topic. The "retained" message flag is enabled, ensuring that the dashboard displays the latest valid reading immediately upon connection, eliminating the wait for the next transmission cycle.

The primary data path relies on the User Datagram Protocol (UDP) combined with a custom Quality of Service (QoS) implementation to ensure message reliability. The client sends telemetry to the Alert Server's configured endpoint with a sequence number (seq). If the server acknowledges receipt (sends an ACK packet within 800ms, validating the sequence number), the delivery is guaranteed. If the ACK is missed, the system employs exponential backoff and up to five retries before classifying the delivery as failed. This protocol mitigates packet loss inherent to UDP, achieving reliability similar to TCP for critical telemetry.

To ensure data persistence during extended network failures, the client implements a rolling window buffer using the device's local flash memory. If the UDP QoS mechanism fails after reaching the maximum number of retries (MAX_RETRIES), the message payload is immediately logged to the /telemetry_log.txt file. This backlog is held until the next successful transmission of live data. Upon a confirmed delivery to the Alert Server, the `transmitStoredData/0` function attempts to re-send the entire backlog, effectively emptying the file and guaranteeing that no data is lost upon device restart or temporary network isolation. This process is further optimized by Adaptive Interval Throttling, where the time between generating new data (current_delay) is dynamically extended if the backlog exceeds a threshold (10 packets), ensuring the client prioritizes clearing old data over saturating local storage.

The Alert Server is architected to handle concurrent data streams from multiple remote clients (including the Pico W device). Using a single UDP socket bound to port 5005, the server efficiently manages traffic from all sources without maintaining persistent connections. The server implements Guaranteed Delivery QoS by: extracting the unique client ID, sequence number (seq), and QoS flag from the incoming JSON payload; checking for duplicate packets against the client's last_seq state (to prevent duplicate data processing); and if the packet is new and requests QoS 1, the server immediately sends a corresponding ACK packet back to the client's source address, confirming successful reception and processing. Furthermore, the server maintains an internal state for every connected device to track its last activity time, providing crucial client monitoring and enabling alerts for client inactivity.

The Alert Server enforces system integrity by validating incoming data against strict operational criteria. It verifies that temperature (0-50º C) and humidity (20-80%) remain within safety limits and simultaneously calculates the differential between the two sensor nodes, flagging anomalies if the variance exceeds 3ºC or 20%. Furthermore, the server monitors client inactivity to detect potential device failures. If any of these conditions are met, the server immediately dispatches an alert via MQTT to the Command Centre, ensuring operators are instantly notified of the specific issue.

The Command Centre serves as the primary operational dashboard, designed to aggregate and visualize real-time data from the distributed sensors. It utilizes line charts to track historical trends for temperature and humidity across both devices, while simultaneously computing and displaying live averages on gauge indicators for quick assessment. Critical warnings dispatched by the Alert Server are prioritized via immediate pop-up notifications in the bottom-right corner of the interface. For persistent tracking, a dedicated tab maintains a log of the ten most recent alerts in reverse chronological order, accompanied by a control feature that allows operators to acknowledge and clear the list as needed.

Compilation & Build Instructions

To successfully compile the solution, specific software environments and libraries are required for each system component. The firmware for the ESP32 and Raspberry Pi Pico is developed using the Arduino IDE. Successful compilation relies on three specific dependencies being installed via the Library Manager: PubSubClient by Nick O'Leary for MQTT communication, the DHT Sensor Library by Adafruit for sensor interfacing, and ArduinoJson by Benoit for data serialization.

The Alert Server is a C-based application designed for a Linux environment capable of running GCC. Prior to compilation, the development environment must be prepared by installing the necessary C libraries for MQTT and JSON handling. This is achieved by executing the installation commands for `libpaho-mqtt3c-dev` and `libcjson-dev` using the system package manager. Finally, the visualization layer utilizes the pre-configured Node-RED Virtual Machine provided via the course platform, while the MQTT communication requires an active HiveMQ Cloud server instance configured with valid user credentials.

The Alert Server includes a dedicated Makefile to streamline the build process. To compile the application, the `make server` command should be executed within the server directory, which automatically handles the GCC compilation and linking of the required libraries. Additionally, a `make clean` target is available to remove previous build artifacts when a fresh compilation is needed.

Installation & Setup

To deploy the firmware, the project source code must first be opened in the Arduino IDE. Before uploading, the configuration variables at the top of the file, specifically the Wi-Fi SSID, password, and HiveMQ credentials, must be updated to match the local environment. Once configured, the appropriate board definition ("ESP32 Dev Module" or "Raspberry Pi Pico W") should be selected from the Tools menu. The process is finalized by connecting the device via USB and clicking the Upload button, which compiles the code and flashes the binary to the microcontroller.

Once the server binary is successfully compiled, the application can be launched immediately from the terminal using the `make run` command. This initiates the UDP listener and establishes the MQTT connection to the broker.

To configure the visualization layer, the provided Virtual Machine is launched using a virtualization tool such as VirtualBox. After logging in, the runtime is initiated by executing the node-red command, making the editor interface accessible via a web browser at `localhost:1880`. The project logic is applied by navigating to the main menu, selecting Import, and uploading the provided `node-red.json` file. To activate the logic, it is essential to click the Deploy button, after which the operational dashboard becomes viewable at `localhost:1880/ui`.

Conclusion

The project successfully delivered a robust, distributed IoT system capable of monitoring sensitive industrial environments in real-time. By integrating dual-protocol edge devices with a custom UDP Alert Server and a Node-RED Command Centre, the solution achieves the core objectives of data visualization, anomaly detection, and fault tolerance.

One of the primary challenges encountered during development was the initial configuration of the MQTT connection which initially resulted in persistent connection failures. However, through rigorous research and critical clarifications provided by the course teacher regarding the MQTT connection, the team was able to establish the connection.

To further enhance the system's reliability, future iterations would benefit from the implementation of redundant communication pathways. Specifically, integrating Bluetooth Low Energy (BLE) as a fallback mechanism would allow the edge devices to transmit critical alerts to a local gateway even if the primary Wi-Fi network becomes unavailable, thereby significantly increasing the overall resilience of the monitoring infrastructure.