Simulating IoT Backend Systems Device Communication via C++

Alperen Kayhan
alperen-1393@hotmail.com
https://github.com/AlperenKayhan

***Abstract*—This paper presents a Node.js-based simulator that mimics a IoT Backend Systems-enabled embedded device. It performs session initialization, WebSocket connection setup, custom logging, heartbeat-based liveness detection, and remote command handling with simulated faults via Poisson-distributed failures. The project demonstrates a robust integration testing tool for IoT Backend Systems server endpoints and message flows.**

I. Introduction

In many IoT applications, embedded devices must maintain a secure, low-latency connection to a backend server, perform real-time sensor measurements, and provide on-device alerts. This work presents an ESP32-based C++ implementation that connects via HTTPS for session initialization, establishes a Socket.IO over TLS WebSocket link for bidirectional messaging, reads an HC-SR04 ultrasonic distance sensor, injects simulated faults via a Poisson process, and plays error tones through an I^{2}S audio amplifier. The design demonstrates a robust pattern for integration testing and remote control of field devices.

II. Operational Scenario

In our setup, we use an ultrasonic sensor to continuously measure the clearance between a fan's blades and the inner wall of its housing. Because real blades have a curved profile, small shifts or vibrations can bring them dangerously close to the housing corners. We quantify this risk by computing

$$X_n = \bigl(70\,d + 3\bigr) \bmod 4, \tag{1}$$

where d is in centimeters. Depending on X_n, the system follows one of four graduated responses:

1) $X_n = 1$**:** Marginal clearance. Log a warning and light the status LED.
2) $X_n = 2$**:** Critical proximity. In addition to the warning LED, advise the operator to perform a manual reboot.
3) $X_n = 3$**:** Unsafe clearance. Automatically reboot, flash the LED, and sound an audible alarm via the I^{2}S speaker.
4) $X_n = 0$ **or** $X_n = 4$**:** Catastrophic fault. Emit a continuous error tone, shut down the fan into safe mode (fan off, electronics remain powered), and log the event.

This tiered approach ensures that minor misalignments trigger early warnings, while severe blade incursions invoke rapid automated protection.

Manuscript received July 23, 2025.

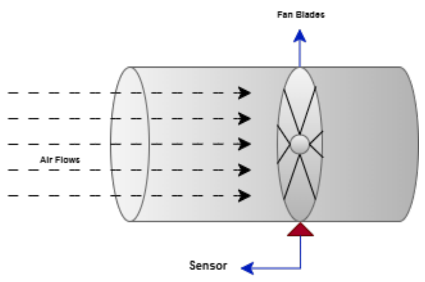

Fig. 1. Schema of Fan

III. System Build

System itself consist of ESP32 and requirement hardware parts. To read and measure the values of its environment.

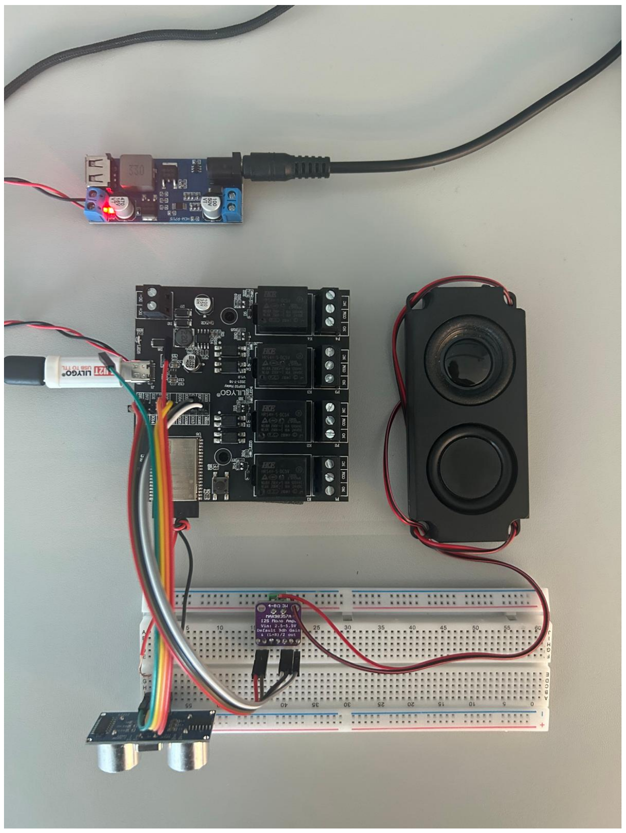

Fig. 2. Figure of System

IV. System Architecture

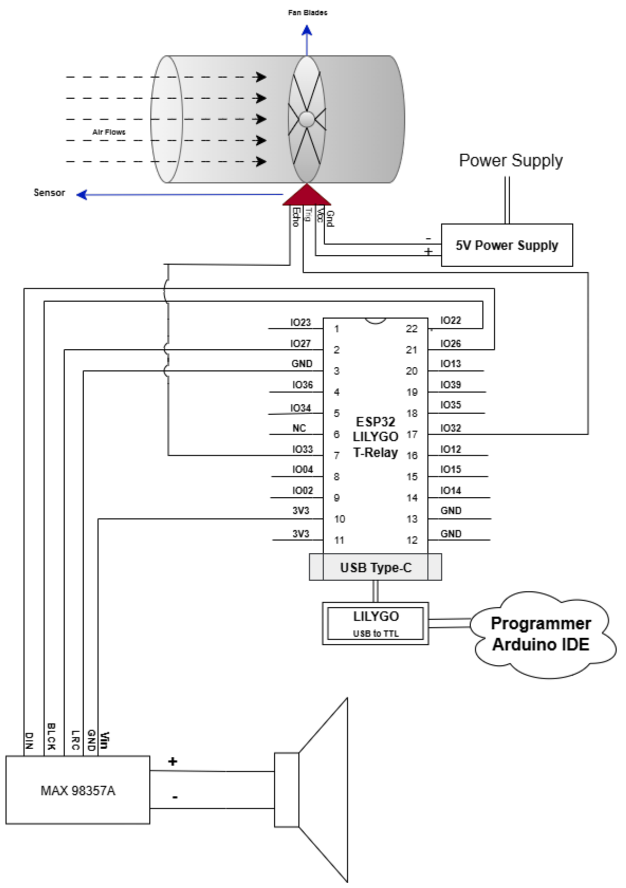

Fig. 3. Sketch of System

- *Session Initialization:* HTTPS GET to '/dv/DvOp' builds or reuses a session token stored in NVS.
- *Real-Time Link:* Socket.IO client over TLS for heartbeat ('ping'/'pong') and server-issued commands.
- *Sensor and Fault Injection:* HC-SR04 distance measurement on GPIO 32/33, Poisson-based anomaly injection activates on each 'pong'.
- *Alerting:* I^{2}S peripheral drives a MAX98357A amplifier and speaker for audible error tones.
- *Status LED:* NeoPixel RGB LED on GPIO 12 indicates connection and fault states.

V. Software Implementation

A. Library Dependencies

- `WiFi.h`, `WiFiClientSecure.h`: connect to SSID and perform HTTPS requests.
- `Preferences.h`: nonvolatile storage for session tokens.
- `SocketIOclient.h`: Socket.IO v4 client over secure WebSocket.
- `driver/i2s.h`, `freertos/task.h`: configure ^{2}S and manage RTOS timing.
- `Adafruit_NeoPixel.h`: single-pixel RGB status LED.
- `ArduinoJson.h`: parse and serialize JSON payloads.
- `math.h`: Poisson random number generation.

B. Configuration Macros

Key parameters are defined at the top of the source:

- I^{2}S audio settings (`SAMPLE_RATE`, `TONE_FREQ`, buffer sizes).
- Ultrasonic sensor pins (`ULTRASONIC_TRIG_PIN`, `ULTRASONIC_ECHO_PIN`) and threshold.
- Wi-Fi credentials and backend host.
- NeoPixel pin and brightness.
- Poisson parameter $\lambda = 5$.

C. Session Management

`buildDvOpPath()` assembles the '/dv/DvOp' query string with:

- Millisecond timestamp ('pts').
- Stored or newly fetched session token.
- Device identifiers: serial numbers, MAC, IP, firmware, model, site IDs.

`sendDeviceOpenRequest()` performs the HTTPS GET, skips headers, reads the JSON body, extracts `data.S` as the session token, stores it via `Preferences`, and then configures the Socket.IO client with the 'cookie: S=¡session¿' header.

D. Real-Time Communication

The global callback `socketIOEvent()` handles:

- `sIOtype_CONNECT` / `sIOtype_DISCONNECT`: updates `socketConnected` and NeoPixel color.
- `sIOtype_EVENT`:
  - On '"pong"', triggers distance measurement, computes $X_n = \mathit{fmod}(7 \cdot d + 3, 4)$, and enacts one of four fault conditions (console warning, manual reboot advice, automatic reboot with tone, or system halt).
  - On application messages ('"m"'), deserializes the inner JSON to dispatch commands: `send_msg_log`, `get_d_parameters`, `reboot`, `changed_parameters`, etc.
- `sIOtype_ACK` / `sIOtype_ERROR`: logged to Serial.

E. Sensor Integration and Fault Injection

`readDistanceMeter()` sends a 10µs trigger pulse, measures echo time via `pulseIn()`, and converts to cm. `poissonRandom(double)` implements Knuth's algorithm for Poisson sampling, used to probabilistically activate fault logic in the heartbeat handler.

F. Audio Alerts

`initI2S()` configures the ESP32 I^{2}S peripheral (pins 26/27/22) for 16-bit stereo at 44.1kHz. `playErrorTone()` fills a DMA buffer with a reduced-amplitude square-wave pattern and toggles between beeps and pauses while driving the MAX98357A amplifier.

G. Main Loop

`setup()` performs:

1) Serial, NeoPixel, and NVS initialization.
2) I^{2}S and ultrasonic pin setup.
3) Wi-Fi connection (with retry logic).
4) Session initialization or reuse.
5) Socket.IO client startup and initial registration.

`loop()` continuously calls `socketIO.loop()` to service the WebSocket, sends a ‘ping‘ event every 5s when connected, and checks for a shutdown flag.

VI. Methodology

Here’s an updated snippet that introduces the discrete fault-state variable X_n and explains how it maps each Poisson draw k into a system action:

A. Heartbeat Mechanism

A custom ping is sent every 5 seconds to simulate liveness. This ping cycle is also used to trigger fault values via a Poisson process:

$$P(k;\lambda) = \frac{e^{-\lambda}\,\lambda^k}{k!}, \tag{2}$$

where $\lambda = 5$. On each cycle we draw $k \sim \text{Poisson}(\lambda)$ and set

$$X_n = \begin{cases} k, & k \le 4, \\ 4, & k > 4, \end{cases}$$

so that all higher counts are treated as the most severe state. We then interpret X_n according to:

$$X_n = \begin{cases} 1, & \text{Minor fault: visual warning,} \\ 2, & \text{Moderate fault: advise manual reboot,} \\ 3, & \text{Critical fault: automatic reboot + audible alarm,} \\ 4, & \text{Fatal fault: immediate shutdown into safe mode.} \end{cases}$$

Whenever $X_n \le 2$ a warning is issued; for $X_n = 3$ the system takes control and reboots itself with an alarm; and for $X_n = 4$ it enters safe-mode shutdown.

B. Command Handlers

Table I lists all supported messages received via WebSocket.

TABLE I
WebSocket Message Handlers

Command	Action
send_msg_log	Print and log the message
reboot	Restart the client script
get_d_items	Log network parameters
changed_parameters	Trigger fault simulation
Power_Off	Cleanly exit the process

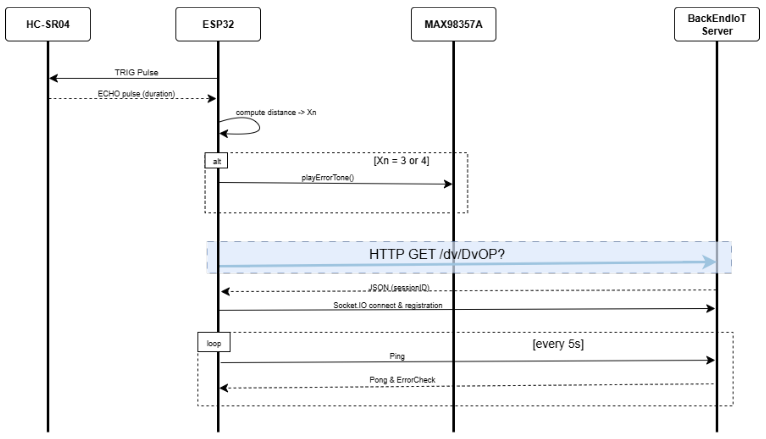

Fig. 4. Sequence Diagram

VII. Sequence Diagram

Figure **??** shows the sequence of interactions between the ultrasonic sensor (HC-SR04), the ESP32-S3 MCU, the MAX98357A audio amplifier, and the KodX cloud server. Each vertical lifeline represents one of these components, and horizontal arrows represent messages or events passed between them. *1. Sensor Trigger and Echo*

– The MCU emits a 10µs `TRIG` pulse to the HC-SR04.
– The HC-SR04 returns an `ECHO` pulse whose width encodes the round-trip time of the ultrasonic wave.
– The MCU measures this pulse width with `pulseIn()`, converting it into a distance in centimeters.

2. Initial Cloud Handshake

– On first boot, the MCU performs an HTTPS GET request to the KodX server’s `/dv/DvOp` endpoint.
– The server responds with JSON containing a session ID, which the MCU stores in nonvolatile preferences.

3. Socket.IO Registration

– The MCU opens a secure WebSocket (Socket.IO) connection to the server using the saved session cookie.
– It immediately emits a registration event {`n` : sessionID, `r` : ”dev”}.

4. Heartbeat Loop

– Every 5s, the MCU emits a `ping` event over the Socket.IO connection.
– The server replies with a `pong` (and may include an `ErrorCheck` instruction).
– On each `pong`, if error simulation is enabled, the MCU re-runs the Poisson X_n logic, potentially triggering another reboot or alarm.

By following this sequence, the device continuously monitors its environment, locally evaluates fault conditions, and stays synchronized with the cloud backend—ensuring robust, timed error handling and remote management.

VIII. Hardware Components

- **ESP32 (LILYGO T-Relay)**
  Dual-core Tensilica LX6 MCU @ 240 MHz with integrated Wi-Fi. Hosts the HTTPS/Socket.IO client, I^2S audio, ultrasonic sensing, fault logic, and status LED.

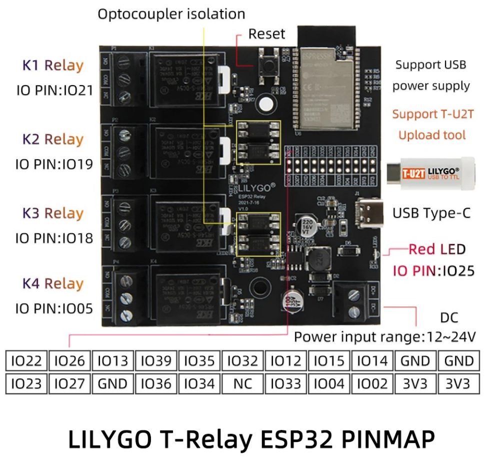

Fig. 5. LILYGO T-Relay

  * **LILYGO T-U2T**
    USB-to-serial adapter and Li-Po charger onboard. Powers and programs the ESP32 and provides a convenient USB interface for logs.

Fig. 6. LILYGO T-U2T

    · **MAX98357A I^2S Class-D Audio Amplifier**

      Accepts the ESP32's I^2S digital audio (BCLK→GPIO 26, LRCLK→GPIO 27, DIN→GPIO 22). Drives the stereo speaker with up to 3 W.

    · **8 ohm Stereo Speaker**

      Connected to the MAX98357A outputs. Emits the square-wave error beeps generated in `playErrorTone()`.

MAX98357A

  * **5 V External Power Supply**

    Feeds the MAX98357A and speaker (separate from the ESP32's 3.3 V rails). Ensures clean, stable audio output under load.

  * **HC-SR04 Ultrasonic Sensor**

    TRIG pin → GPIO 32, ECHO pin → GPIO 33. Measures distance (2 cm–400 cm) each heartbeat to feed the Poisson-based fault logic.

HC-SR04 Ultrasonic Sensor

- **Adafruit NeoPixel (WS2812) RGB LED**

  Data line → GPIO 12. Shows yellow during Wi-Fi/session init, green on successful connection, and red on errors.

- **Breadboard & Jumper Wires**

  Prototyping platform and wiring to interconnect all modules (ESP32 pins → sensors, amp, LED, power).

IX. Conclusion

The HC-SR04, when powered at 5V and read via GPIO 32/33 on the T-Relay ESP32, delivers reliable distance measurements in the 10–30cm range. This validated sensor performance can now be integrated into the main IoT sketch to trigger warnings and automatic reboots whenever the measured distance falls below the 30cm threshold. Thus, The presented C++ implementation on ESP32 provides a complete pattern for:

- Secure session management via HTTPS and NVS.
- Low-latency real-time messaging with Socket.IO over TLS.
- Sensor polling, Poisson-based fault injection, and automated recovery.
- On-device audio and visual alerts via I^2S and NeoPixel.

This architecture is readily extensible for production-grade IoT deployments or integration testing of backend systems.

Acknowledgment

I would like to thank the IoT Backend Systems engineering team and thier support during the development of this project.