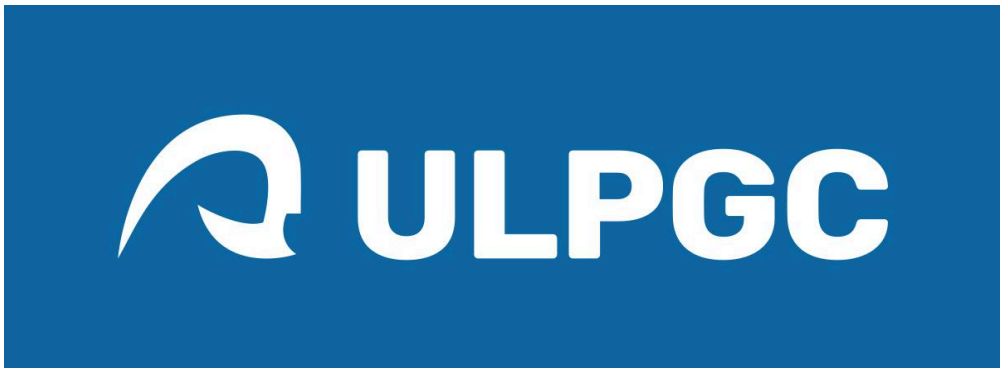
Universidad de

Las Palmas de Gran Canaria

Escuela Ingeniería Informática

MQTT Based Domotics



Autores:

- Wail Ben El Hassane Boudhar
- Félix Miguel Velásquez
- Mohamed O. Haroun Zarkik

Índice de Contenidos

1. INTRODUCCIÓN

El presente proyecto aborda el diseño e implementación de un sistema de Internet de las Cosas (IoT) orientado a la monitorización ambiental y al control automatizado de elementos en entornos domésticos. La domótica representa uno de los campos de aplicación más relevantes del IoT, permitiendo transformar espacios convencionales en ambientes inteligentes capaces de adaptarse a las necesidades de sus usuarios de forma autónoma y eficiente.

La proliferación de dispositivos conectados y el desarrollo de protocolos de comunicación estándares como MQTT han facilitado la creación de ecosistemas IoT escalables y robustos. Este proyecto aprovecha estas tecnologías para construir una solución integral que integra: (a) una red de sensores distribuidos, (b) un sistema de comunicación de largo alcance mediante tecnología LoRa, (c) un puente de integración basado en el protocolo MQTT, y (d) una interfaz web para monitorización y control en tiempo real.

La arquitectura propuesta originalmente contemplaba el uso de cuatro microcontroladores Arduino y una Raspberry Pi actuando como broker MQTT y servidor central. Durante el desarrollo del proyecto, esta arquitectura evolucionó hacia una implementación que mantiene los principios fundamentales de la propuesta inicial, pero incorpora mejoras en la infraestructura de comunicaciones y en la gestión de eventos. El sistema final implementa una red de sensores LoRa, un nodo maestro que actúa como agregador de datos, un puente serial-MQTT desarrollado en Python, y una aplicación web moderna construida con tecnologías contemporáneas.

El proyecto aborda desafíos técnicos relevantes en el ámbito del IoT, incluyendo: la comunicación eficiente entre dispositivos con recursos limitados, la implementación de protocolos de framing para garantizar la integridad de los mensajes, el diseño de una arquitectura desacoplada mediante el patrón publish/subscribe de MQTT, y la creación de una interfaz de usuario reactiva capaz de gestionar flujos de datos en tiempo real.

Esta memoria documenta en profundidad todos los aspectos técnicos del sistema, desde los protocolos de comunicación de bajo nivel hasta la arquitectura de la aplicación web, proporcionando una visión completa de un sistema IoT funcional y extensible.

En el siguiente enlace se puede encontrar el enlace al repositorio con el código fuente: https://github.com/mohamedOharoun/MQTT-Based-Domotics

2. PLANTEAMIENTO DE LA IDEA

2.1. Concepto Original

La propuesta inicial del proyecto contemplaba la construcción de un sistema domótico distribuido compuesto por una red de cuatro microcontroladores Arduino coordinados por una Raspberry Pi. La arquitectura propuesta establecía los siguientes componentes:

- **Dos nodos Arduino esclavos**: cada uno equipado con sensores específicos para la monitorización ambiental. Entre los sensores considerados se incluían: medición de luminosidad, temperatura, detección de movimiento o proximidad (para conteo de personas), sensores de CO_2 o calidad del aire, y medición de humedad ambiental.

- **Un nodo Arduino repetidor/gateway**: actuando como nodo intermedio para agregar y reenviar la información capturada por los sensores hacia el servidor central.

- **Raspberry Pi como nodo maestro**: funcionando simultáneamente como broker MQTT (servidor de mensajería) y servidor web, centralizando la lógica de control, el almacenamiento de datos históricos y la interfaz de usuario.

El protocolo de comunicación elegido fue MQTT, aprovechando su arquitectura publish/subscribe para desacoplar productores y consumidores de información. Este modelo permite que cada sensor publique sus datos en topics específicos, mientras que los consumidores (la interfaz web y los sistemas de automatización) pueden suscribirse selectivamente a los datos que les interesan.

Finalmente a la hora del desarrollo del proyecto se optó porque el nodo que un principio haría de repetidor sea el nodo maestro, interpretando las señales que recibe constantemente de los sensores y atendiendo a la creación de eventos y a la ejecución de los mismos

recibidos por la Raspberry Pi, la cuál haría de servidor web y de interfaz entre usuario y sistema.

2.2. Funcionalidades Planteadas

El sistema propuesto debía proporcionar las siguientes capacidades funcionales:

- **Monitorización en tiempo real**: visualización continua del estado de todos los sensores distribuidos en el entorno.

- **Registro histórico:** almacenamiento persistente de lecturas en archivos log para análisis posterior y detección de patrones.

- **Configuración de umbrales**: definición de valores límite para diferentes parámetros ambientales, permitiendo establecer condiciones de activación para acciones automatizadas.

- **Sistema de eventos y alertas**: capacidad de definir reglas que, al cumplirse, disparan acciones como la activación de actuadores, el envío de notificaciones o la ejecución de secuencias de control (por ejemplo: elevar persianas cuando el nivel de luz sea bajo, activar alarmas ante detección de movimiento inesperado, o ajustar sistemas de climatización según temperatura y humedad).

- **Suscripción selectiva:** mediante la estructura de topics de MQTT, permitir que los usuarios elijan qué información recibir, optimizando el ancho de banda y la carga de procesamiento.

2.3. Evolución Durante el Desarrollo

Durante la fase de implementación, el proyecto experimentó una evolución natural que, manteniendo la filosofía y objetivos originales, introdujo mejoras significativas en varios aspectos técnicos:

- **Tecnología de comunicación inalámbrica**: se adoptó LoRa (Long Range) como tecnología de enlace entre los nodos sensores y el nodo maestro. Esta decisión proporciona

ventajas significativas en términos de alcance (varios kilómetros en campo abierto), bajo consumo energético y tolerancia a interferencias, características especialmente valiosas en despliegues domóticos reales.

-**Arquitectura de bridge**: en lugar de integrar directamente el broker MQTT en una Raspberry Pi, se desarrolló un puente software en Python que actúa como traductor bidireccional entre el protocolo serie (usado por el nodo maestro Arduino) y MQTT. Este diseño modular permite mayor flexibilidad y facilita el mantenimiento.

- **Broker MQTT containerizado**: se utilizó Eclipse Mosquitto desplegado mediante Docker, proporcionando un entorno de ejecución aislado, fácilmente replicable y con soporte tanto para MQTT clásico (puerto 1883) como MQTT sobre WebSockets (puerto 9001) para la integración con la aplicación web.

- **Gestión descentralizada de eventos**: la lógica de evaluación de eventos se implementó tanto en el nodo maestro (para respuestas inmediatas y funcionamiento autónomo) como en la capa de aplicación, creando un sistema híbrido capaz de operar incluso con conectividad intermitente.

- **Stack tecnológico web moderno:** la interfaz de usuario se construyó utilizando Bun como runtime de JavaScript, React para la construcción de componentes UI, y MQTT.js para la comunicación en tiempo real, resultando en una aplicación web altamente reactiva y eficiente.

Estas adaptaciones no alteran el concepto fundamental del proyecto —un sistema distribuido de monitorización y control basado en MQTT— sino que lo refinan, proporcionando mayor robustez, escalabilidad y características técnicas superiores a las originalmente contempladas.

3. OBJETIVOS DEL PROYECTO

Los objetivos del proyecto se estructuran en dos categorías: objetivos funcionales, relacionados con las capacidades que debe proporcionar el sistema, y objetivos técnicos, vinculados a los requisitos de implementación y calidad del software desarrollado.

3.1. Objetivos Funcionales

- Implementar un sistema de agregación de datos procedentes de múltiples sensores distribuidos, utilizando comunicación inalámbrica de largo alcance mediante LoRa.

- Centralizar la lógica de gestión de eventos en el nodo maestro, permitiendo la definición de reglas basadas en: umbrales configurables (valores numéricos límite), tipos de disparo (condiciones de activación: mayor que, menor que, igual a), estado de activación/desactivación de reglas, y mecanismos de anti-rebote temporal para evitar disparos múltiples.

- Proporcionar un bus de integración estándar basado en MQTT que permita tanto la monitorización de lecturas como la configuración dinámica del sistema.

- Desarrollar una interfaz web de usuario que ofrezca las siguientes funcionalidades: visualización en tiempo real de todas las lecturas de sensores y sus estados clasificados, consulta de mensajes de estado del sistema, visualización de alertas generadas por eventos disparados, y gestión completa del ciclo de vida de eventos (creación, edición, activación/desactivación y eliminación).

3.2. Objetivos Técnicos

- Diseñar e implementar un protocolo binario personalizado sobre LoRa que optimice el uso del ancho de banda disponible, incluyendo cabeceras de red local (direccionamiento, identificadores de mensaje) y estructura de payload para diferentes tipos de sensores.

- Implementar un protocolo de enmarcado robusto para la comunicación serie entre el nodo maestro y el puente Python, que garantice la detección de límites de mensaje y tolere transmisiones fragmentadas o con ruido.

- Establecer una estrategia coherente de Quality of Service (QoS) y mensajes retenidos (retained) en MQTT, diferenciando entre datos de sensores (efímeros), estado del sistema (persistentes) y configuración de eventos (retenidos con QoS 2).

- Crear una arquitectura desacoplada que permita que cada componente (sensores, maestro, puente, broker, web) pueda desarrollarse, probarse y desplegarse de forma independiente.

- Demostrar la viabilidad de una arquitectura IoT completa que abarque desde el nivel físico (radio LoRa) hasta la capa de aplicación (interfaz web), pasando por todos los niveles intermedios de integración.

4. ARQUITECTURA DEL SISTEMA

El sistema implementado sigue una arquitectura en capas que separa claramente las responsabilidades de cada componente. Esta separación facilita el mantenimiento, testing y escalabilidad del sistema completo.

4.1. Vista General de Componentes

La arquitectura se compone de los siguientes componentes principales, organizados en cuatro capas funcionales:

4.1.1. Capa de Sensores (Nodos Esclavos)

Constituida por los microcontroladores Arduino equipados con sensores específicos. En la implementación actual se incluyen dos tipos de nodos:

Nodo de luminosidad (LIGHT_01): utiliza un sensor VEML6030 conectado por I²C (dirección 0x48) para medir la intensidad lumínica ambiental. El sensor se configura con ganancia 0.125 y tiempo de integración de 100 ms, proporcionando lecturas en lux. La dirección LoRa local de este nodo es 0xCC.

Nodo de distancia ultrasónica (DISTANCE_01): emplea un sensor SRF02 por I²C para medición de distancia mediante ultrasonidos. El sensor opera enviando el comando 0x51 al registro 0x00, esperando 70 ms para la medición, y leyendo dos bytes desde el registro 0x02 que representan la distancia en centímetros. La dirección LoRa local es 0xBB.

Ambos nodos realizan lecturas periódicas cada 5000 ms (configurables mediante TX_LAPSE_MS y SENSOR_READ_INTERVAL respectivamente), clasifican el estado según umbrales predefinidos, empaquetan los datos en tramas binarias y los transmiten vía LoRa al nodo maestro (dirección 0xAA).

4.1.2. Capa de Agregación (Nodo Maestro)

El nodo maestro (dirección LoRa 0xAA) actúa como coordinador central de la red de sensores y como traductor entre dos dominios de protocolo: LoRa en el lado de los sensores y Serial/JSON en el lado del puente de integración. Sus responsabilidades incluyen:

- **Recepción y parsing de tramas LoRa**: implementa la lógica de decodificación del protocolo binario propietario, validando cabeceras, extrayendo payloads y discriminando entre diferentes tipos de sensores.

- **Visualización local**: muestra información relevante en una pantalla OLED, permitiendo monitorización directa sin dependencia de la infraestructura de red.

- **Timestamp local:** incorpora a cada mensaje un timestamp obtenido de un RTC (Real-Time Clock), reduciendo la carga de datos transmitidos por LoRa.

- **Generación de mensajes JSON:** traduce las tramas binarias LoRa a objetos JSON estructurados, que incluyen metadatos como node_id, msg_type, timestamp, sensor_type y los datos específicos de cada sensor.

- **Enmarcado de mensajes serie**: envuelve cada mensaje JSON con delimitadores explícitos (comm:start$ y $comm:end) para permitir parsing robusto en el receptor.

- **Gestión de eventos local**: mantiene una tabla configurable de hasta 10 eventos, cada uno con su event_id, sensor_type, trigger_type (above/below/equal), trigger_threshold, is_active, y alert_message. Evalúa continuamente las condiciones contra las lecturas recibidas e implementa anti-rebote temporal (mínimo 5 segundos entre disparos consecutivos).

- **Configuración dinámica:** acepta comandos JSON enmarcados desde el puente Python para crear, actualizar o eliminar eventos, permitiendo reconfiguración en tiempo real sin necesidad de reprogramar el microcontrolador.

4.1.3. Capa de Integración (Puente Serial-MQTT)

Implementado en Python (mqtt-server/python-serial-link/), este componente actúa como traductor bidireccional entre el mundo Arduino y el ecosistema MQTT. Su diseño modular se estructura en varios módulos especializados:

- **arduino_lib.py**: gestiona la comunicación serie con el nodo maestro, implementando un autómata de estados para el parsing de tramas enmarcadas. Detecta START_MARKER (comm:start$), acumula datos en un buffer, identifica END_MARKER ($comm:end), extrae y valida el JSON interior.

- **mqtt_lib.py**: encapsula toda la lógica de interacción con el broker MQTT, incluyendo establecimiento de conexión, gestión de suscripciones, publicación con diferentes niveles de QoS y retain, y manejo de callbacks para mensajes recibidos.

- **mqtt_payload.py:** define la estructura de datos para los diferentes tipos de mensajes (sensor_data, alert, event, clear_event) y proporciona funciones de serialización/deserialización entre objetos Python y payloads JSON.

- **terminal_cli.py**: ofrece una interfaz de línea de comandos para testing y debugging, permitiendo inyectar mensajes falsos tanto en el canal serie como en MQTT.

El flujo de datos en esta capa es bidireccional: en dirección uplink (Arduino → MQTT), parsea mensajes de tipo sensor_data y alert, y los publica en topics específicos; en dirección downlink (MQTT → Arduino), se suscribe a params/event/# para recibir configuración de eventos desde la web, y los reenvía enmarcados al nodo maestro.

4.1.4. Capa de Mensajería (Broker MQTT)

Se utiliza Eclipse Mosquitto como broker MQTT, desplegado mediante Docker Compose con la siguiente configuración:

- **Listener TCP en puerto 1883**: protocolo MQTT estándar para la comunicación con el puente Python.

- **Listener WebSocket en puerto 9001**: MQTT sobre WebSockets para permitir la conexión directa desde navegadores web.

- **Autenticación:** configurado con allow_anonymous true (apropiado para entornos de laboratorio y desarrollo local, no recomendado para producción).

- **Persistencia:** habilitada para mantener mensajes retenidos y colas de suscriptores desconectados.

El broker actúa como punto central de distribución de mensajes, desacoplando completamente productores y consumidores. Gracias al modelo publish/subscribe, múltiples clientes pueden conectarse simultáneamente sin conocer la existencia de los demás.

4.1.5. Capa de Presentación (Aplicación Web)

La interfaz de usuario es una Single Page Application (SPA) desarrollada con tecnologías modernas:

- **Runtime**: Bun, un runtime de JavaScript/TypeScript de alto rendimiento que proporciona ejecución rápida y herramientas de desarrollo integradas.

- **Framework UI:** React con TypeScript, proporcionando un desarrollo con tipado estático y componentes reutilizables.

- **Cliente MQTT:** mqtt.js configurado para conexión WebSocket (ws://localhost:9001), con gestión de estados de conexión (connecting, connected, disconnected) y reconexión automática.

La aplicación se estructura en dos vistas principales: MonitorView para visualización de datos en tiempo real y EventManagerView para gestión del ciclo de vida de eventos. Un contexto React (MessageContext) centraliza la lógica de recepción y normalización de mensajes MQTT, reconstruyendo mensajes completos a partir de topics fragmentados.

5. PROTOCOLOS DE COMUNICACIÓN

El sistema implementa múltiples protocolos de comunicación en diferentes capas de la arquitectura. Esta sección documenta en detalle cada uno de estos protocolos, constituyendo el núcleo técnico del proyecto.

5.1. LoRa como Transporte Físico

LoRa (Long Range) proporciona la capa física de comunicación entre los nodos Arduino. Es crucial entender que LoRa define únicamente la modulación y los parámetros de radio, no la estructura de los mensajes de aplicación. Por ello, el proyecto define un protocolo binario propio encima de LoRa.

5.1.1. Parámetros de Configuración LoRa

Tanto el nodo maestro como los nodos esclavos utilizan una configuración LoRa idéntica para garantizar la compatibilidad en el aire:

- **Frecuencia**: 868 MHz (banda ISM europea, también válida en Canarias)

- **Spreading Factor (SF)**: 12 (máximo alcance, menor tasa de datos, mayor inmunidad a interferencias)

- **Bandwidth (BW)**: 125 kHz (índice 7 en la librería LoRa)

- **Coding Rate (CR)**: 4/8 (alto nivel de redundancia para corrección de errores)

- **Potencia de transmisión**: 20 dBm (100 mW, máxima permitida en banda ISM)

- **Longitud de preámbulo**: 16 símbolos

- **CRC**: habilitado (detección de errores de transmisión)

- **SyncWord**: 0x12 (diferencia esta red de otras redes LoRa cercanas)

Esta configuración prioriza alcance y robustez sobre velocidad de datos, apropiado para un sistema de monitorización con lecturas periódicas de baja frecuencia.

5.2. Protocolo Binario sobre LoRa

Sobre la capa física LoRa, el proyecto implementa un protocolo de aplicación binario que estructura los datos transmitidos. Este protocolo se compone de dos niveles: una cabecera de red local y un payload de aplicación.

5.2.1. Cabecera de Red Local (5 bytes)

La cabecera proporciona información de enrutamiento básico y control de flujo. Los bytes se envían en el siguiente orden exacto:

- **Byte 0 - DESTINATION (1 byte)**: dirección del destinatario. En los esclavos siempre es 0xAA (dirección del maestro).

- **Byte 1 - SENDER (1 byte):** dirección del emisor. Identifica el nodo que envía el mensaje. En este proyecto, el SENDER coincide además con el SENSOR_ID: 0xBB para ultrasonidos, 0xCC para luz.

- **Bytes 2-3 - MSG_ID (2 bytes, big-endian):** identificador único del mensaje, implementado como un contador incremental de 16 bits. Permite al receptor detectar pérdidas de paquetes o reordenamiento.

- **Byte 4 - PAYLOAD_LENGTH (1 byte)**: longitud del payload en bytes. En la implementación actual, siempre es de 6 bytes para ambos tipos de sensores.

5.2.2. Payload de Aplicación (6 bytes)

El payload tiene también una estructura definida, consistente entre ambos tipos de sensores:

- **Byte 0 - MSG_TYPE**: 0x05 (MSG_TYPE_SENSOR), identifica este paquete como datos de sensor.

- **Byte 1 - SENSOR_ID**: 0xBB (ultrasonidos) o 0xCC (luz), redundante con SENDER pero útil para validación.

- **Byte 2 - SENSOR_TYPE:** 0x01 para distancia (ultrasónico), 0x02 para lux (luz).

- **Bytes 3-4 - DATA (2 bytes, big-endian)**:el valor principal de la medición, representado como uint16. Para ultrasonidos, distancia en centímetros. Para luz, valor de lux truncado a entero de 16 bits.

- **Byte 5 - STATE**: clasificación discreta del estado basada en umbrales. Los valores varían según el tipo de sensor.

Estados del sensor ultrasónico:

- 0x00: error de lectura

- 0x01: cerca (distancia < 30 cm)

- 0x02: medio (30 cm ≤ distancia < 100 cm)

- 0x03: lejos (distancia ≥ 100 cm)

Estados del sensor de luz:

- 0x00: oscuro (lux < 200)

- 0x01: tenue (200 ≤ lux < 800)

- 0x02: brillante (lux ≥ 800)

5.2.3. Validación y Parsing en el Maestro

El nodo maestro implementa un parser robusto que ejecuta las siguientes operaciones al recibir un paquete LoRa:

1. Detecta paquetes entrantes mediante LoRa.parsePacket()

2. Lee y descarta el byte DESTINATION (ya cumplió su función en el enrutamiento)

3. Lee SENDER, MSG_ID (2 bytes) y PAYLOAD_LENGTH

4. Aplica un límite de seguridad: si PAYLOAD_LENGTH > 30, rechaza el paquete (protección contra paquetes malformados)

5. Lee el payload completo en un buffer

6. Verifica que buf[0] == MSG_TYPE_SENSOR (0x05)

7. Usa buf[1] (SENSOR_ID) para discriminar entre sensor de luz y ultrasonidos

8. Reconstruye el valor DATA como: value = (buf[3] << 8) | buf[4]

9. Traduce STATE (buf[5]) a una cadena de texto legible

10. Muestra la información en la pantalla OLED

11. Reporta los datos hacia el puente serie (mediante funciones serialbridge_report_*)

12. Invoca checkEvents() para determinar si se dispara alguna alerta

Este flujo de validación garantiza que solo se procesan paquetes válidos y que la información se propague correctamente hacia las capas superiores del sistema.

5.3. Protocolo de Enmarcado sobre Serial

La comunicación serie entre el Arduino maestro y el puente Python requiere un mecanismo para delimitar mensajes, ya que el canal serie es un flujo continuo de bytes sin estructura inherente de mensajes. El proyecto implementa un protocolo de enmarcado basado en delimitadores de texto.

5.3.1. Delimitadores de Trama

Se utilizan dos marcadores de texto fijo para delimitar cada mensaje:

- **Marcador de inicio**: comm:start$

- **Marcador de fin**: $comm:end

Los mensajes completos tienen la estructura:

`comm:start$<PAYLOAD_JSON>$comm:end`

5.3.2. Formato del Payload Serie

El contenido entre delimitadores es siempre un objeto JSON. El sistema define dos tipos principales de mensajes en dirección uplink (Arduino → Python):

Mensaje tipo sensor_data: Representa una lectura de sensor.

Estructura:

- **node_id**: identificador textual del nodo (derivado del SENDER, ej: "LIGHT_01", "DISTANCE_01")

- **msg_type**: siempre "sensor_data"

- **timestamp**: epoch Unix (segundos desde 1970-01-01) obtenido del RTC del maestro

- **msg_id**: identificador del mensaje LoRa original

- **sensor_type**: "light" o "ultrasonic"

- **data**: objeto con los campos específicos del sensor y su estado clasificado

Mensaje tipo alert: Enviado cuando se dispara un evento.

Incluye:

- **node_id**: nodo que generó la alerta

- **msg_type**: "alert"

- **event**: objeto con la configuración completa del evento disparado (event_id, sensor_type, trigger_type, trigger_threshold, alert_message, etc.)

5.3.3. Parsing Robusto en Python (Uplink)

El puente Python implementa un autómata de estados finitos para el parsing de tramas:

1. Estado inicial: busca START_MARKER en cada línea leída del puerto serie

2. Al detectar START_MARKER: transita a estado "in_frame" y comienza a acumular texto en un buffer

3. En estado "in_frame": continúa acumulando caracteres hasta detectar END_MARKER

4. Al detectar END_MARKER: cierra la trama, extrae el JSON interior (eliminando los delimitadores), parsea el JSON con json.loads()

5. Retorna el objeto Python resultante

6. Si hay error de parsing JSON, registra el error y descarta la trama

7. Regresa a estado inicial y repite el proceso

Este diseño tolera situaciones problemáticas comunes en comunicaciones serie: mensajes fragmentados en múltiples llamadas a read(), ruido o datos espurios entre tramas válidas, y velocidades de transmisión variables.

5.3.4. Downlink (Python → Arduino)

El flujo inverso utiliza el mismo protocolo de enmarcado. Cuando el puente Python recibe configuración de eventos desde MQTT:

1. Serializa el objeto EventType a JSON

2. Lo envuelve con los delimitadores: comm:start$<JSON>$comm:end

3. Escribe la trama completa al puerto serie

4. Fuerza flush del buffer serie para garantizar envío inmediato

El maestro Arduino mantiene por su parte un parser incremental que funciona carácter por carácter en su loop() principal: busca FRAME_START, acumula en un buffer hasta encontrar FRAME_END, y llama a handleJsonCommand() para procesar el mensaje.

Este diseño simétrico (mismo protocolo de framing en ambas direcciones) simplifica la implementación y el debugging, además de proporcionar un mecanismo consistente de validación de mensajes.

5.4. MQTT como Bus de Integración

Una vez que los mensajes JSON llegan al puente Python, la integración seria del sistema ocurre vía MQTT. Este protocolo actúa como el bus de eventos central que desacopla completamente los productores de datos (sensores) de los consumidores (aplicación web, sistemas de automatización futuros).

5.4.1. Broker y Despliegue

El broker MQTT utilizado es Eclipse Mosquitto, desplegado mediante Docker Compose con la siguiente configuración específica:

- Puerto 1883: MQTT sobre TCP, protocolo clásico para clientes nativos MQTT (usado por el puente Python)

- Puerto 9001: MQTT sobre WebSockets, fundamental para permitir conexiones directas desde navegadores web

- allow_anonymous true:configuración apropiada para entornos de laboratorio o redes locales de confianza. En producción debería implementarse autenticación

- persistence true: habilita el almacenamiento persistente de mensajes retenidos y suscripciones de clientes, sobreviviendo reinicios del broker

- listener 9001 protocol websockets: declaración explícita del listener WebSocket en la configuración de Mosquitto

5.4.2. Modelo de Topics (Jerarquía de Temas)

El diseño de la jerarquía de topics es crítico en MQTT, ya que determina cómo se organizan y filtran los datos. El sistema implementa una estructura granular que separa claramente diferentes tipos de información:

Topics para sensor de luz:

- sensors/brightness/<NODE_ID>/light → valor en lux (float)

- sensors/brightness/<NODE_ID>/als → valor bruto del sensor ALS (analog light sensor)

- sensors/brightness/<NODE_ID>/estado → clasificación textual ("Oscuro", "Tenue", "Brillante")

- status/<NODE_ID>/last_update → timestamp de la última actualización

Topics para sensor de ultrasonidos:

- sensors/distance/<NODE_ID>/distance_cm → distancia medida en centímetros

- sensors/distance/<NODE_ID>/estado → clasificación textual ("Error", "Cerca", "Medio", "Lejos")

- status/<NODE_ID>/last_update → timestamp de la última actualización

Topics para eventos y alertas:

- params/event/<event_id> → configuración completa de un evento específico. Este topic usa retain=true y qos=2 cuando se publica desde la web, convirtiendo MQTT en una base de datos de configuración distribuida

- params/event/alerts/<node_id> → alertas disparadas por eventos. Publicadas desde el puente Python con qos=1

Esta estructura de topics permite suscripciones muy específicas. Por ejemplo, un cliente interesado solo en el estado de luminosidad puede suscribirse a sensors/brightness/+/estado (donde + es un wildcard para cualquier NODE_ID), mientras que un sistema de logging podría suscribirse a sensors/# para capturar todas las lecturas.

5.4.3. QoS y Retain (Decisiones de Protocolo)

MQTT ofrece tres niveles de Quality of Service (QoS) y la opción de marcar mensajes como retenidos (retained). El sistema hace uso estratégico de estas características:

Topics de estado (status/.../last_update):

- retain = true: el broker almacena el último mensaje publicado. Cuando un nuevo cliente se suscribe, recibe inmediatamente el último estado conocido sin esperar a la siguiente actualización

- qos = 2: garantiza entrega exactamente una vez, evitando duplicados en información crítica de estado

Topics de sensores (sensors/...):

- retain = false: los datos de sensores son flujos continuos. No tiene sentido retener un valor antiguo; los suscriptores deben esperar la siguiente lectura

- qos = 0: entrega at-most-once (sin confirmación). Prioriza bajo overhead y baja latencia sobre garantías de entrega. Si se pierde un paquete, la siguiente lectura (5 segundos después) compensará la pérdida

Configuración de eventos desde la web (params/event/<event_id>):

- retain = true: el último evento publicado queda almacenado en el broker. Si el nodo maestro se reinicia o se conecta un nuevo consumidor, pueden recuperar la configuración actual de eventos sin necesidad de un servidor de configuración separado

- qos = 2: la configuración de eventos es crítica y debe llegar exactamente una vez al destino para evitar comportamientos inconsistentes

Borrado de eventos:

El borrado de eventos sigue una convención MQTT estándar: publicar un payload vacío (empty string) con retain=true en el topic del evento. Esto elimina el mensaje retenido del broker, efectivamente "borrando" la configuración del evento de la base de datos distribuida.

5.4.4. Downlink MQTT → Serial

El puente Python se suscribe al wildcard params/event/# para recibir todas las actualizaciones de configuración de eventos. Cuando recibe un mensaje en esta suscripción:

- Si el payload está vacío: interpreta la operación como borrado y construye un mensaje JSON con msg_type="clear_event" y el event_id extraído del topic, enviándolo enmarcado al Arduino

- Si el payload contiene JSON válido con msg_type="event": lo parsea como un evento completo y lo reenvía enmarcado al puerto serie

Este mecanismo cierra el ciclo completo de control: Web → MQTT → Puente Python → Serial → Arduino Maestro → Evaluación local de eventos → Disparos de alertas → Serial → Python → MQTT → Web.

5.5. MQTT sobre WebSockets (Web ↔ Broker)

Los navegadores web no pueden establecer conexiones TCP directas (restricción de seguridad del navegador). Por ello, el acceso desde la aplicación web requiere MQTT sobre WebSockets.

- URL de conexión: ws://localhost:9001 (protocolo WebSocket inseguro, apropiado para desarrollo local; en producción debería usarse wss:// sobre TLS)

- Configuración del broker: Mosquitto requiere explícitamente la directiva "listener 9001" con "protocol websockets" en su archivo de configuración para habilitar este tipo de conexión

- Cliente web: utiliza la biblioteca mqtt.js, que abstrae las diferencias entre MQTT sobre TCP y MQTT sobre WebSockets, proporcionando la misma API para ambos casos

- Suscripción: la aplicación web se suscribe al wildcard "#" (todos los topics), permitiéndole recibir todas las actualizaciones del sistema. Esto es apropiado para una interfaz de monitorización completa; aplicaciones más específicas podrían suscribirse solo a subconjuntos de topics

Un aspecto importante del frontend es que reconstruye mensajes lógicamente completos a partir de topics fragmentados. Por ejemplo, para el sensor de luz, la web espera recibir tres topics separados (light, als, estado) y los combina en un objeto MessageRecord coherente antes de mostrarlo en la interfaz. Este proceso de normalización se gestiona en el MessageContext de React.

6. COMPONENTES DEL SISTEMA Y FUNCIONAMIENTO DETALLADO

Esta sección describe en profundidad el funcionamiento interno de cada componente del sistema, explicando cómo se implementan las funcionalidades descritas en la arquitectura.

6.1. Sensor de Luz (arduino/LoRa/LightSlave/)

Rol: nodo esclavo que mide luminosidad ambiental y la transmite periódicamente al maestro.

6.1.1. Inicialización

En la función setup():

- Inicializa el bus I²C mediante Wire.begin()

- Configura el sensor VEML6030 con dirección I²C 0x48

- Establece ganancia del sensor en 0.125 (rango dinámico alto, apropiado para entornos con variaciones grandes de luminosidad)

- Configura tiempo de integración de 100 ms (balance entre velocidad de lectura y precisión)

- Inicializa el módulo LoRa con todos los parámetros especificados en la sección 5.1.1 (frecuencia 868 MHz, SF=12, etc.)

- Registra callback onTxDone para gestionar el final de transmisiones

6.1.2. Ciclo de Operación

En loop(), cada TX_LAPSE_MS (5000 ms) se ejecuta el siguiente proceso:

1. Lee luminosidad del VEML6030, obteniendo un valor float en lux

2. Lee valor bruto ALS (analog light sensor) como uint16

3. Trunca el valor de lux a uint16 para transmisión (limitación del protocolo binario)

4. Clasifica el estado según umbrales: lux < 200 → Oscuro (0x00), lux < 800 → Tenue (0x01), lux ≥ 800 → Brillante (0x02)

5. Construye payload binario de 6 bytes: [MSG_TYPE_SENSOR=0x05] [SENSOR_ID_LIGHT=0xCC] [SENSOR_TYPE_LUX=0x02] [DATA_HIGH] [DATA_LOW] [STATE]

6. Construye cabecera LoRa: [DESTINATION=0xAA] [SENDER=0xCC] [MSG_ID_HIGH] [MSG_ID_LOW] [PAYLOAD_LENGTH=6]

7. Verifica que no hay transmisión en curso (flag txInProgress)

8. Inicia transmisión LoRa: LoRa.beginPacket(), escribe cabecera y payload, LoRa.endPacket(true) (modo async)

9. Marca txInProgress = true

10. Incrementa msgCount para el próximo mensaje

Cuando la transmisión finaliza, el callback onTxDone() marca txInProgress = false, permitiendo la siguiente transmisión. Este mecanismo evita que se inicien nuevas transmisiones antes de completar la anterior, lo cual causaría corrupción de datos en el módulo LoRa.

6.2. Sensor de Ultrasonidos (arduino/LoRa/Slave/)

Rol: nodo esclavo que mide distancia mediante ultrasonidos y la transmite al maestro.

6.2.1. Inicialización

Similar al sensor de luz, en setup() se inicializan I²C y LoRa con los mismos parámetros para garantizar compatibilidad.

6.2.2. Proceso de Medición con SRF02

El sensor SRF02 requiere un protocolo específico de lectura por I²C:

1. Escribe el comando 0x51 ("ranging mode in centimeters") en el registro 0x00 del SRF02

2. Espera 70 ms para que el sensor complete la medición (tiempo necesario para el pulso ultrasónico de ida y vuelta)

3. Lee 2 bytes desde el registro 0x02: el primer byte es el MSB (byte alto) y el segundo es el LSB (byte bajo) de la distancia

4. Reconstruye la distancia: distance_cm = (msb << 8) | lsb

5. Si la lectura I²C falla (requestFrom retorna 0), marca la distancia como 0 (error)

6.2.3. Clasificación y Transmisión

Cada SENSOR_READ_INTERVAL (5000 ms):

1. Ejecuta el proceso de medición descrito arriba

2. Clasifica estado: distance == 0 → Error (0x00), distance < 30 → Cerca (0x01), distance < 100 → Medio (0x02), distance ≥ 100 → Lejos (0x03)

3. Si hay error, normaliza distance_cm a 0 para el payload (evita valores indefinidos en el protocolo)

4. Construye payload: [MSG_TYPE_SENSOR=0x05] [SENSOR_ID_ULTRA=0xBB] [SENSOR_TYPE_DISTANCE=0x01] [DATA_HIGH] [DATA_LOW] [STATE]

5. Construye cabecera LoRa con SENDER=0xBB

6. Transmite vía LoRa con el mismo mecanismo de control de transmisión que el sensor de luz

6.3. Nodo Maestro (arduino/LoRa/Master/)

El nodo maestro es el componente más complejo del sistema Arduino, actuando como cerebro del edge computing. Coordina múltiples responsabilidades simultáneamente.

6.3.1. Recepción y Parsing de Paquetes LoRa

En cada iteración de loop(), el maestro ejecuta:

1. Llamada a LoRa.parsePacket() que retorna el tamaño del paquete si hay uno disponible

2. Si packetSize > 0, invoca processPacket() que implementa el parser completo del protocolo binario

3. Lee recipient, sender, msgId (2 bytes) y len

4. Valida len ≤ 30 (protección contra paquetes malformados)

5. Lee el payload completo en un buffer

6. Verifica buf[0] == MSG_TYPE_SENSOR

7. Discrimina sensor según buf[1] (SENSOR_ID)

8. Extrae valor: value = (buf[3] << 8) | buf[4]

9. Traduce buf[5] (STATE) a string legible

10. Actualiza pantalla OLED con la información recibida

11. Llama a serialbridge_report_* para generar el mensaje JSON enmarcado hacia el puente Python

12. Invoca checkEvents() para evaluar si la nueva lectura dispara algún evento configurado

6.3.2. Generación de Reportes Serie

El módulo SerialReport.ino implementa funciones especializadas para cada tipo de sensor que construyen los mensajes JSON:

- serialbridge_report_light(): crea JSON con node_id="LIGHT_01", msg_type="sensor_data", timestamp=rtc.getEpoch(), sensor_type="light", y data={lux, als, estado}

- serialbridge_report_ultrasonic(): similar pero con sensor_type="ultrasonic" y data={distance_cm, estado}

- Ambas funciones llaman a **serialbridge_report_packet()** que añade los delimitadores: Serial.print("comm:start$" + payload + "$comm:end")

Un detalle crítico: el maestro incluye el timestamp obtenido de su RTC (Real-Time Clock). Esto reduce la cantidad de datos a transmitir por LoRa (cada byte ahorrado mejora la eficiencia energética de los esclavos) y centraliza la sincronización temporal en el maestro.

6.3.3. Gestión de Eventos

El maestro mantiene un array estático de hasta MAX_EVENTS (10) eventos. Cada evento es una estructura que contiene:

- event_id: string identificador único

- sensor_type: "light" o "ultrasonic"

- trigger_type: "above", "below" o "equal"

- trigger_threshold: float con el valor umbral

- is_active: bool que determina si el evento está habilitado

- alert_message: string descriptivo para la alerta

- last_trigger_time: timestamp del último disparo (para anti-rebote)

Función handleJsonCommand():

Procesa comandos JSON recibidos por serie. Soporta dos operaciones:

- msg_type="event": busca el evento por event_id en la tabla. Si existe, actualiza sus campos. Si no existe y hay espacio, crea uno nuevo. Si la tabla está llena, ignora la solicitud

- msg_type="clear_event": busca el evento por event_id y lo marca como inválido (normalmente mediante un flag o limpiando el event_id)

Función checkEvents():

Invocada tras cada lectura de sensor. Para cada evento activo en la tabla:

1. Verifica que el sensor_type coincide con el sensor que generó la lectura

2. Verifica que is_active == true

3. Evalúa la condición: si trigger_type="above" y value > threshold, dispara; similar para "below" y "equal"

4. Implementa anti-rebote temporal: compara (current_time - last_trigger_time) contra un mínimo (típicamente 5000 ms). Si no ha pasado suficiente tiempo, ignora el disparo

5. Si todas las condiciones se cumplen: actualiza last_trigger_time, muestra alert_message en OLED, y envía un mensaje tipo "alert" enmarcado por serie hacia el puente Python

Este diseño descentralizado permite que el sistema continúe evaluando eventos y disparando alertas incluso si la conectividad con el servidor o la web se interrumpe. El nodo maestro opera de forma autónoma.

6.3.4. Parsing Incremental de Comandos Serie (Downlink)

En loop(), el maestro también lee continuamente del puerto serie:

1. Lee caracteres uno a uno con Serial.read()

2. Busca la secuencia FRAME_START ("comm:start$")

3. Acumula caracteres en serialBuf hasta detectar FRAME_END ("$comm:end")

4. Extrae el JSON interior (removiendo los delimitadores)

5. Invoca handleJsonCommand(serialBuf) para procesar el comando

6. Limpia el buffer y regresa al estado inicial de búsqueda

Este parser carácter por carácter es necesario en Arduino debido a las limitaciones de memoria y la naturaleza asíncrona del puerto serie. No se puede asumir que un mensaje completo llegará en una sola llamada a Serial.read().

6.4. Puente Serial-MQTT (mqtt-server/python-serial-link/)

El puente Python es el componente de integración crítico que conecta el mundo Arduino con el ecosistema MQTT. Su arquitectura modular facilita el mantenimiento y la extensibilidad.

6.4.1. Módulo arduino_lib.py

Gestiona toda la comunicación con el puerto serie. Implementa una clase ArduinoSerial que mantiene:

- Objeto serial.Serial conectado al puerto del Arduino (típicamente /dev/ttyACM0 en Linux)

- Estado del autómata de parsing: "idle" (buscando inicio), "in_frame" (acumulando), "complete" (trama lista)

- Buffer de acumulación: string donde se concatenan los caracteres recibidos

Método read_framed_message():

1. Lee líneas del puerto serie (readline() con timeout)

2. Para cada línea, verifica si contiene START_MARKER

3. Si está en estado "in_frame", acumula el texto en el buffer

4. Al detectar END_MARKER, extrae el contenido entre marcadores usando expresiones regulares o string slicing

5. Parsea el JSON con json.loads()

6. Retorna el objeto Python resultante

7. Si hay error de parsing JSON, registra el error y descarta la trama

Método write_framed_message():

1. Recibe un diccionario Python

2. Serializa a JSON con json.dumps()

3. Construye el mensaje enmarcado: f"comm:start${json_str}$comm:end"

4. Escribe al puerto serie con serial.write()

5. Fuerza flush del buffer serie para garantizar envío inmediato

6.4.2. Módulo mqtt_lib.py

Encapsula toda la lógica MQTT. Utiliza la biblioteca paho-mqtt para Python. Define una clase MQTTClient que mantiene:

- Cliente paho-mqtt configurado con broker_host="localhost", broker_port=1883

- Callbacks registrados: on_connect, on_message, on_disconnect

- Cola de mensajes recibidos (para procesamiento asíncrono si es necesario)

Método connect():

- Establece conexión con el broker: client.connect(host, port)

- Inicia loop de red en thread separado: client.loop_start() (modo threaded para no bloquear el main loop)

Método subscribe():

- Suscribe a topics específicos: client.subscribe(topic, qos)

- En este proyecto se suscribe a "params/event/#" para recibir configuración de eventos

Método publish():

- Publica mensajes: client.publish(topic, payload, qos, retain)

- Permite especificar QoS (0/1/2) y retain flag según el tipo de mensaje

Callback on_message():

Invocado automáticamente por paho-mqtt cuando llega un mensaje en un topic suscrito. Ejecuta:

1. Extrae topic y payload del mensaje

2. Si topic comienza con "params/event/": parsea el payload como JSON

- Si está vacío, extrae event_id del topic y construye mensaje {"msg_type":"clear_event","event_id":...}

- Si contiene JSON, valida que msg_type="event" y prepara el mensaje completo

3. Llama a arduino_serial.write_framed_message() para enviar el comando al maestro

6.4.3. Módulo mqtt_payload.py

Define dataclasses o diccionarios tipados para los diferentes tipos de mensajes del sistema. Proporciona funciones de serialización y validación:

- SensorData: node_id, msg_type, timestamp, sensor_type, msg_id, data (dict con campos específicos del sensor)

- AlertData: node_id, msg_type, event (objeto Event completo)

- EventData: event_id, msg_type, sensor_type, trigger_type, trigger_threshold, is_active, alert_message

- Funciones parse_sensor_data(), parse_alert(), parse_event() que validan la estructura y tipos de datos

6.4.4. Módulo main.py (Loop Principal)

Orquesta el funcionamiento general del puente:

1. Inicializa ArduinoSerial y MQTTClient

2. Conecta al broker MQTT

3. Suscribe a "params/event/#"

4. Entra en loop infinito:

- Lee mensajes del Arduino con arduino.read_framed_message() (con timeout para no bloquear indefinidamente)

- Si llega un mensaje: parsea el msg_type

- Para "sensor_data": valida la estructura, publica cada campo del sensor en su topic correspondiente

- Luz → sensors/brightness/<node_id>/{light,als,estado}

- Ultrasonidos → sensors/distance/<node_id>/{distance_cm,estado}

- Publica timestamp en status/<node_id>/last_update con retain=true y qos=2

- Para "alert": publica el mensaje completo en params/event/alerts/<node_id> con qos=1

- Gestiona excepciones: reconexión automática al broker si se pierde la conexión, log de errores

6.4.5. Módulo terminal_cli.py

Proporciona una interfaz de línea de comandos para testing y debugging. Permite:

- Comando "fake": inyecta un mensaje JSON directamente al loop de procesamiento, simulando recepción desde Arduino sin necesidad de hardware conectado

- Comando "send": envía mensajes JSON enmarcados al puerto serie, permitiendo probar la recepción de comandos en el Arduino sin pasar por MQTT

- Comando "mqtt": publica mensajes directamente en topics MQTT, simulando publicaciones desde la web

- Comando "status": muestra estado actual de conexiones (serie y MQTT), estadísticas de mensajes procesados, última actividad de cada sensor

6.5. Aplicación Web (web/)

La interfaz web es una Single Page Application moderna que proporciona una experiencia de usuario fluida para monitorización y control del sistema.

6.5.1. Estructura del Proyecto y Stack Tecnológico

- Runtime: Bun (alternativa a Node.js, significativamente más rápido para desarrollo y build)

- Framework UI: React 18 con TypeScript, proporcionando desarrollo con tipos estáticos y detección temprana de errores

- Cliente MQTT: mqtt.js, biblioteca estándar de facto para MQTT en JavaScript

- Gestión de estado: React Context API para estado global (mensajes MQTT), useState/useEffect para estado local de componentes

- Estilado: CSS modules o styled-components

6.5.2. Componente App.tsx (Raíz de la Aplicación)

El componente principal gestiona la conexión MQTT y el routing básico:

- Inicialización: en useEffect con dependencias vacías, establece conexión MQTT: const client = mqtt.connect('ws://localhost:9001', {clientId: 'web-client-' + Math.random(), clean: true, reconnectPeriod: 1000})

- Gestión de estados de conexión: mantiene estado [connectionStatus, setConnectionStatus] que puede ser 'connecting', 'connected', 'disconnected'

- Suscripción global: tras conectar exitosamente, client.subscribe('#', {qos: 0}) para recibir todos los topics

- Provisión de contexto: envuelve toda la aplicación en <MessageProvider client={client}> para que los componentes hijos accedan al cliente MQTT

- UI de estado: muestra un indicador de conexión en la parte superior (badge verde "Conectado" / rojo "Desconectado")

- Navegación: implementa tabs o routing entre MonitorView y EventManagerView

6.5.3. Hook MessageContext.tsx (Gestión Centralizada de Mensajes)

Este es uno de los componentes más sofisticados del frontend. Implementa la lógica de reconstrucción de mensajes lógicos a partir de topics fragmentados.

Problema a resolver:

El puente Python publica cada campo de un sensor en un topic separado. Estos mensajes pueden llegar en cualquier orden. La UI necesita mostrar lecturas completas, no campos individuales.

Solución implementada:

1. Mantiene un mapa de "mensajes parciales" por node_id: partials[node_id] = {light?: number, als?: number, estado?: string}

2. Cuando llega un mensaje MQTT: extrae node_id del topic, determina qué campo está actualizando, actualiza el partial correspondiente

3. Verifica si el partial está "completo": para luz, necesita light + als + estado; para ultrasonidos, necesita distance_cm + estado

4. Cuando el partial está completo: construye un MessageRecord completo con todos los campos

5. Añade el MessageRecord a la lista de mensajes: setMessages(prev => [newMessage, ...prev])

6. Limpia el partial para ese node_id

Además, el contexto gestiona directamente mensajes que llegan completos (status updates, alerts, event configs) sin necesidad de reconstrucción.

6.5.4. Vista MonitorView

Componente de visualización en tiempo real:

- Consume mensajes del MessageContext: const {messages} = useMessageContext()

- Implementa filtrado: permite al usuario seleccionar qué tipos de mensajes ver (sensor_data, status, event, alert)

- Filtra mensajes: const filteredMessages = messages.filter(m => selectedTypes.includes(m.msg_type))

- Renderiza lista: utiliza virtualized list si hay muchos mensajes para mantener rendimiento

- Para cada mensaje, utiliza un componente especializado:**

  - <SensorDataCard />: muestra sensor_type, node_id, timestamp, todos los campos de data con unidades. Código de color según estado

  - <StatusCard />: muestra node_id y last_update con formato relativo ("hace 5 segundos")

  - <EventCard />: muestra configuración del evento completa

  - <AlertCard />: destaca visualmente las alertas con color rojo/naranja

- Auto-scroll: implementa scroll automático al último mensaje (opcional, con toggle)

6.5.5. Vista EventManagerView

Interfaz CRUD (Create, Read, Update, Delete) para eventos:

- Lista de eventos actuales: extrae eventos del MessageContext y muestra cada uno

- Formulario de creación/edición: campos para event_id, sensor_type, trigger_type, trigger_threshold, is_active, alert_message

- Función handleCreateEvent():valida campos, construye objeto EventType, publica en params/event/<event_id> con client.publish(topic, JSON.stringify(event), {qos: 2, retain: true})

- Función handleEditEvent(): similar a crear, pero permite modificar un evento existente re-publicando en su mismo topic

- Función handleToggleActive(): lee el evento actual, invierte is_active, re-publica

- Función handleDeleteEvent(): muestra diálogo de confirmación, si confirma: publica payload vacío con retain=true: client.publish(topic, '', {qos: 2, retain: true})

- Validaciones: previene IDs duplicados, valida que trigger_threshold sea número válido, asegura campos obligatorios no vacíos

La combinación de QoS=2 y retain=true en las publicaciones de eventos es crucial: garantiza que el mensaje llegue exactamente una vez y que permanezca en el broker para nuevos suscriptores. Esto convierte MQTT en una base de datos de configuración distribuida y confiable.

7. FLUJO DE DATOS EXTREMO A EXTREMO

Para consolidar la comprensión del sistema completo, esta sección describe el flujo completo de datos desde la captura por un sensor hasta la visualización en la interfaz web, y el flujo inverso de configuración de eventos.

7.1. Flujo Uplink: Sensor → Web

Secuencia completa paso a paso:

1. Captura de datos: un nodo esclavo realiza una lectura física del sensor mediante I²C. Obtiene un valor numérico (lux o distancia en cm)

2. Clasificación: el esclavo clasifica el valor según umbrales predefinidos, generando un código de estado discreto (0x00-0x03)

3. Empaquetado binario: el esclavo construye una trama binaria de 11 bytes: 5 bytes de cabecera + 6 bytes de payload

4. Transmisión LoRa: el esclavo transmite la trama por radio LoRa a 868 MHz con SF12, dirigida a 0xAA (maestro)

5. Recepción LoRa: el maestro detecta un paquete entrante mediante LoRa.parsePacket()

6. Parsing binario: el maestro lee secuencialmente todos los bytes, valida la estructura, reconstruye el valor numérico y el estado

7. Actualización OLED: el maestro muestra la información en su pantalla local

8. Timestamp: el maestro obtiene el epoch actual de su RTC y lo añade al mensaje

9. Traducción a JSON: el maestro construye un objeto JSON con estructura completa

10. Enmarcado serie: el maestro envuelve el JSON con delimitadores y lo escribe al puerto serie

11. Recepción en Python: el puente Python lee del puerto serie hasta completar una trama enmarcada

12. Parsing JSON: el puente extrae el JSON interior y lo parsea con json.loads()

13. Validación: el puente valida la estructura del mensaje

14. Fragmentación MQTT: el puente divide el mensaje en múltiples publicaciones MQTT, cada una en su topic específico

15. Propagación en el broker:Mosquitto distribuye las publicaciones a todos los clientes suscritos

16. Recepción en web: la aplicación web recibe cada topic como evento separado vía WebSocket

17. Reconstrucción de mensaje: el MessageContext acumula los topics en partials[node_id]

18. Creación de MessageRecord: cuando el partial está completo, construye un MessageRecord unificado

19. Actualización de estado React: el nuevo mensaje se añade al estado global

20. Re-render: React re-renderiza MonitorView, mostrando el nuevo mensaje en la UI

Todo este flujo ocurre en menos de 1 segundo bajo condiciones normales, proporcionando una experiencia de monitorización en tiempo real.

7.2. Flujo de Evaluación de Eventos (en Maestro)

Paralelamente al flujo uplink:

1. Inmediatamente después del parsing binario, el maestro invoca checkEvents()

2. checkEvents() itera sobre todos los eventos en su tabla local

3. Para cada evento: verifica coincidencia de sensor_type, verifica que is_active sea true, evalúa la condición (above/below/equal), verifica anti-rebote temporal

4. Si todas las condiciones se cumplen: actualiza last_trigger_time, muestra alert_message en OLED, construye mensaje JSON tipo "alert", lo enmarca y envía por serie

5. El puente Python recibe el mensaje "alert", lo publica en params/event/alerts/<node_id> con qos=1

6. La web recibe la alerta vía MQTT, la añade a la lista de mensajes, MonitorView la muestra con formato destacado

Este flujo demuestra que la lógica de eventos funciona en el edge (maestro Arduino), permitiendo respuestas inmediatas incluso con latencias o desconexiones.

7.3. Flujo Downlink: Web → Maestro (Configuración de Eventos)

1. Usuario crea/edita evento en EventManagerView: llena formulario con todos los campos del evento

2. Validación frontend: la aplicación valida que event_id sea único, trigger_threshold sea número válido, campos obligatorios no vacíos

3. Construcción del objeto EventType: construye objeto JavaScript con todos los campos

4. Publicación MQTT: client.publish('params/event/<event_id>', JSON.stringify(event), {qos: 2, retain: true})

5. Propagación en broker: Mosquitto recibe la publicación, almacena el mensaje como retenido

6. Recepción en puente Python: el callback on_message del cliente MQTT se dispara

7. Extracción del topic: el puente identifica que es un mensaje en params/event/#

8. Parsing del payload: parsea el JSON del evento

9. Validación: verifica que msg_type="event" y que todos los campos obligatorios están presentes

10. Serialización a JSON: prepara el JSON para enviar al Arduino

11. Enmarcado serie: envuelve con comm:start$ ... $comm:end

12. Transmisión serie: escribe al puerto serie con flush

13. Recepción en maestro: el parser incremental del maestro detecta el frame completo

14. Extracción del JSON: elimina delimitadores y obtiene el JSON

15. Invocación de handleJsonCommand():procesa el comando según msg_type

16. Actualización de tabla de eventos: busca el event_id en la tabla, si existe actualiza, si no existe y hay espacio crea nuevo

17. Confirmación visual: actualiza OLED mostrando "Evento configurado: <event_id>"

18. Evaluación inmediata: en la siguiente lectura de sensor, checkEvents() evaluará también el nuevo evento

Este flujo cierra el ciclo completo: la web puede configurar dinámicamente el comportamiento del sistema sin necesidad de reprogramar el firmware del maestro.

8. ESTRUCTURA DEL REPOSITORIO

El repositorio se organiza en tres grandes bloques funcionales:

arduino/

- **LoRa/Master/** - Firmware del nodo maestro Arduino

  - Master.ino - Inicialización LoRa y loop principal

  - SerialReport.ino - Generación de mensajes JSON enmarcados

  - globals.h - Definiciones de constantes y estructuras

- **LoRa/LightSlave/** - Firmware del sensor de luz

  - LightSlave.ino - Lectura VEML6030 y transmisión LoRa

- **LoRa/Slave/** - Firmware del sensor ultrasónico

  - Slave.ino - Lectura SRF02 y transmisión LoRa

mqtt-server/

- **config/** - Configuración del broker Mosquitto

  - mosquitto.conf - Configuración de listeners y persistencia

- **python-serial-link/** - Puente Serial ↔ MQTT

  - main.py - Loop principal del puente

  - arduino_lib.py - Gestión de comunicación serie

  - mqtt_lib.py - Cliente MQTT y callbacks

  - mqtt_payload.py - Definiciones de tipos de mensajes

  - terminal_cli.py - CLI para testing

- docker-compose.yml - Despliegue del broker Mosquitto

web/

- **src/**

  - App.tsx - Componente raíz, conexión MQTT

  - hooks/MessageContext.tsx - Contexto global de mensajes

  - views/MonitorView.tsx - Vista de monitorización

  - views/EventManagerView.tsx - Vista de gestión de eventos

  - models/ - Definiciones TypeScript de tipos de datos

- package.json - Dependencias del proyecto

- tsconfig.json - Configuración TypeScript

Archivos raíz:

- README.md - Documentación del proyecto

- start.sh - Script de inicio automatizado

9. INSTRUCCIONES DE EJECUCIÓN

9.1. Requisitos Previos

Hardware:

- 1 Arduino con módulo LoRa (nodo maestro)

- 2 Arduinos con módulos LoRa (nodos esclavos)

- Sensor VEML6030 (luz)

- Sensor SRF02 (ultrasonidos)

- Pantalla OLED para el maestro

- RTC (Real-Time Clock) para el maestro

Software:

- Docker y Docker Compose

- Python 3.8 o superior

- Bun runtime

- Arduino IDE (para compilar y subir firmware)

9.2. Configuración del Broker MQTT

```bash

cd mqtt-server

docker-compose up -d

```

Esto levanta Mosquitto en puertos 1883 (MQTT) y 9001 (WebSockets).

9.3. Configuración del Puente Python

```bash
cd mqtt-server/python-serial-link
python -m venv venv
source venv/bin/activate  # En Windows: venv\Scripts\activate
pip install -r requirements.txt
python main.py
```

9.4. Configuración de la Aplicación Web

```bash
cd web
bun install
bun dev
```

La aplicación estará disponible en http://localhost:3000

9.5. Configuración de los Arduinos

1. Abrir cada sketch en Arduino IDE

2. Verificar configuración de pines según el hardware

3. Compilar y subir a cada Arduino

4. El maestro debe conectarse al puerto serie que usa el puente Python

9.6. Script de Inicio Automatizado

El repositorio incluye un script `start.sh` que automatiza el inicio de la web y el puente:

```bash
chmod +x start.sh
./start.sh
```

10. USO DE INTELIGENCIA ARTIFICIAL

Durante el desarrollo de este proyecto se ha utilizado inteligencia artificial para:

- Asistencia en la redacción de la memoria: la IA ha ayudado a estructurar y redactar secciones de la memoria, asegurando claridad, coherencia y nivel técnico apropiado para documentación académica.

- Resolución de dudas técnicas: consultas sobre implementación de protocolos de comunicación, mejores prácticas en arquitectura IoT, y debugging de problemas específicos en el código.

- Revisión y corrección de código: detección de errores lógicos, sugerencias de optimización, y mejoras en la estructura del código de los diferentes componentes del sistema.

- Validación de decisiones de diseño: discusión de alternativas técnicas para diferentes aspectos del sistema (protocolos, QoS en MQTT, estrategias de parsing, etc.).

La IA ha actuado como herramienta de asistencia, pero todas las decisiones técnicas fundamentales, el diseño de la arquitectura, y la implementación del sistema han sido realizadas por el equipo de desarrollo.

11. CONCLUSIONES

El proyecto ha logrado implementar exitosamente un sistema IoT completo que integra múltiples tecnologías y protocolos de comunicación, demostrando la viabilidad de arquitecturas distribuidas en el ámbito de la domótica.

11.1. Logros Principales

- Arquitectura completa extremo a extremo: el sistema abarca desde sensores físicos hasta interfaz web, pasando por todos los niveles intermedios de comunicación y procesamiento.

- Protocolos personalizados eficientes: el diseño de un protocolo binario sobre LoRa optimizado para dispositivos con recursos limitados ha demostrado ser efectivo para la comunicación de sensores.

- Gestión descentralizada de eventos: la capacidad del nodo maestro de evaluar eventos de forma autónoma proporciona resiliencia al sistema ante fallos de red.

- Arquitectura modular y desacoplada: cada componente puede desarrollarse, probarse y desplegarse independientemente, facilitando el mantenimiento y la evolución del sistema.

- Integración MQTT robusta: el uso estratégico de QoS y retained messages convierte MQTT en una herramienta poderosa no solo para transmisión de datos sino también para gestión de configuración distribuida.

11.2. Desafíos Superados

- Sincronización de protocolos múltiples: coordinar LoRa (binario), Serial (enmarcado), MQTT (topics) y WebSockets requirió un diseño cuidadoso de cada interfaz.

- Parsing robusto de tramas: implementar parsers que toleren fragmentación, ruido y errores de transmisión fue crítico para la estabilidad del sistema.

- Reconstrucción de mensajes en frontend: el desafío de reconstruir lecturas completas desde topics MQTT fragmentados requirió una lógica sofisticada de acumulación temporal.

11.3. Posibles Extensiones

El sistema actual proporciona una base sólida para múltiples extensiones futuras:

- Más tipos de sensores: temperatura, humedad, CO_2, pueden integrarse siguiendo el mismo patrón de protocolo binario.

- Actuadores: incorporar relés, servomotores o sistemas de control que respondan a eventos disparados.

- Base de datos histórica: almacenar lecturas en una base de datos para análisis temporal y detección de patrones.

- Machine Learning: entrenar modelos predictivos para anticipar eventos basándose en patrones históricos.

- Seguridad: implementar autenticación en MQTT, cifrado de comunicaciones y control de acceso basado en roles.

- Escalabilidad: desplegar múltiples nodos maestros coordinados, permitiendo redes más grandes y distribuidas geográficamente.

11.4. Aprendizajes

Este proyecto ha proporcionado experiencia práctica en:

- Diseño de protocolos de comunicación de bajo nivel

- Arquitecturas publish/subscribe y desacoplamiento de componentes

- Integración de sistemas heterogéneos (microcontroladores, Python, JavaScript)

- Gestión de diferentes paradigmas de comunicación (síncrona/asíncrona)

- Testing y debugging de sistemas distribuidos

- Documentación técnica exhaustiva

El resultado es un sistema funcional que demuestra los principios fundamentales del Internet de las Cosas aplicados a un caso de uso real de domótica, proporcionando una base sólida para futuros desarrollos en este campo.