UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Grado en Ingeniería del Software

Sistema de automatización inteligente para invernadero con control remoto y tolerancia a fallos
Smart Greenhouse Automation System with Remote Control and Fault Tolerance

Realizado por
Juan José Gómez Morales

Tutorizado por
Vicente Jesús Benjumea García

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, Febrero 2026

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

SISTEMA DE AUTOMATIZACION INTELIGENTE PARA INVERNADERO CON CONTROL REMOTO Y TOLERANCIA A FALLOS

SMART GREENHOUSE AUTOMATION SYSTEM WITH REMOTE CONTROL AND FAULT TOLERANCE

Realizado por
Juan José Gómez Morales

Tutorizado por
Vicente Jesús Benjumea García

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, FEBRERO DE 2026

Fecha defensa: Marzo de 2026

Resumen

Este **Trabajo de Fin de Grado** aborda el diseño e implementación de un sistema de automatización inteligente para invernaderos, con control remoto, funcionamiento autónomo y tolerancia a fallos. La motivación principal es garantizar la eficiencia y la seguridad en entornos agrícolas mediante tecnologías accesibles y robustas, evitando que errores de lectura o condiciones inseguras comprometan el funcionamiento del sistema. El objetivo consiste en desarrollar una solución integral que permita monitorizar y controlar variables críticas como la humedad del suelo, la temperatura ambiente, el nivel del depósito de agua y el estado de los actuadores, todo ello desde una aplicación móvil conectada mediante un **broker MQTT**.

La metodología combina el uso de microcontroladores **Arduino ESP32**, sensores analógicos y digitales, y una aplicación Android desarrollada con **Jetpack Compose**. El sistema incorpora un **control autónomo basado en PID**, que regula el riego en función de la humedad del suelo y la ventilación según la temperatura ambiente. Además, se implementa una arquitectura modular que asegura la sincronización entre firmware y aplicación, junto con mecanismos de seguridad que bloquean acciones ante fallos de lectura, detienen el riego en caso de error y protegen la bomba cuando el nivel de agua es insuficiente. También se incluye un sistema de eventos, visualización dinámica del estado y registro histórico de alertas y notificaciones, garantizando trazabilidad y fiabilidad.

Los resultados muestran una comunicación robusta entre dispositivos, una experiencia de usuario clara y profesional, y una gestión segura de los eventos del sistema. En conclusión, se valida la viabilidad de aplicar soluciones embebidas y móviles para la automatización agrícola, proponiendo como líneas futuras la integración de **inteligencia artificial para riego predictivo,** así como la implementación de medidas de seguridad adicionales —como autenticación mediante claves, cifrado TLS en el broker MQTT o validación de origen en los mensajes— con el fin de reforzar la integridad y la fiabilidad del sistema.

Palabras clave: Automatización agrícola, invernadero inteligente, Arduino ESP32, MQTT, tolerancia a fallos, Android.

Abstract

This **Final Degree Project** addresses the design and implementation of an intelligent automation system for greenhouses, featuring remote control, autonomous operation, and fault tolerance. The main motivation is to ensure efficiency and safety in agricultural environments through accessible and robust technologies, preventing reading errors or unsafe conditions from compromising the system's operation. The objective is to develop an integrated solution capable of monitoring and controlling critical variables such as soil moisture, ambient temperature, the water tank level, and the state of the actuators, all managed from a mobile application connected through an **MQTT broker**.

The methodology combines the use of Arduino **ESP32** microcontrollers, analog and digital sensors, and an Android application developed with **Jetpack Compose**. The system incorporates an **autonomous PID-based control mechanism** that regulates irrigation according to soil moisture and ventilation according to ambient temperature. In addition, a modular architecture ensures synchronization between firmware and application, together with safety mechanisms that block actions in case of reading failures, stop irrigation when an error occurs, and protect the pump when the water level is insufficient. The system also includes an event manager, dynamic state visualization, and a historical log of alerts and notifications, ensuring traceability and reliability.

The results show robust communication between devices, a clear and professional user experience, and secure management of system events. In conclusion, the feasibility of applying embedded and mobile solutions to agricultural automation is validated, proposing as future work the integration of **artificial intelligence for predictive irrigation**, as well as the implementation of additional security measures—such as key-based authentication, TLS encryption in the MQTT broker, or message origin validation—to strengthen the integrity and reliability of the system.

Keywords: Agricultural automation, smart greenhouse, Arduino ESP32, MQTT, fault tolerance, Android.

Índice

Índice de Figuras

1

Introducción

1.1. Motivación

La agricultura moderna enfrenta el reto de **optimizar recursos y garantizar la sostenibilidad** en un contexto de creciente demanda alimentaria y limitación de agua y energía. Los invernaderos constituyen un entorno controlado que permite mejorar la productividad, pero requieren sistemas de monitorización y control que aseguren la estabilidad de las condiciones internas. Tradicionalmente, estas tareas se realizan de forma manual, lo que implica un elevado consumo de tiempo y una mayor probabilidad de error humano.

La gestión de un invernadero exige controlar múltiples **variables críticas** como la humedad del suelo, la temperatura ambiente y el nivel de agua, además de coordinar actuadores como bombas de riego, ventiladores y sistemas de iluminación. Un error en la lectura de sensores o en la ejecución de órdenes puede provocar **daños materiales**, pérdidas de recursos o incluso la avería de componentes esenciales.

La motivación principal de este Trabajo de Fin de Grado es diseñar un sistema de automatización que no solo permita monitorizar y controlar estas variables de forma remota, sino que también funcione de manera autónoma, incorporando un **control automático basado en PID** para regular el riego y la ventilación según las condiciones del entorno. Además, el sistema debe operar con garantías de seguridad y tolerancia a fallos, deteniendo automáticamente cualquier acción ante una condición insegura, como un error de lectura, un nivel bajo en el depósito de agua o una incoherencia en los datos recibidos. De esta manera, se asegura que la automatización no compromete la integridad del invernadero ni de sus componentes, priorizando siempre la **fiabilidad del sistema**.

1.2. Objetivos

El objetivo general del proyecto es desarrollar un sistema de automatización inteligente para invernaderos que integre hardware embebido y software móvil, permitiendo tanto el **control remoto** como el **funcionamiento autónomo mediante control PID**.
De este objetivo principal se derivaron los siguientes objetivos específicos:

1. **Implementar un firmware en Arduino** capaz de gestionar sensores y actuadores, aplicando control PID para regular riego y ventilación, y

deteniendo automáticamente el sistema ante cualquier error de lectura o condición insegura.
2. **Proteger los componentes críticos**, como la bomba de riego, mediante lógica de seguridad que impida su funcionamiento en caso de nivel insuficiente de agua
3. **Desplegar un servidor MQTT en entorno Linux con Proxmox,** con configuración de acceso externo, garantizando la comunicación segura y estable entre dispositivos.
4. **Diseñar una aplicación Android con Jetpack Compose** que muestre el estado del sistema, registre eventos y permita el control remoto de los actuadores.
5. **Incorporar mecanismos de tolerancia a fallos**, como reconexión automática y bloqueo seguro de acciones, para asegurar la sincronización visual y evitar notificaciones falsas.
6. **Validar el sistema mediante pruebas exhaustivas**, evaluando la fiabilidad de la comunicación, la experiencia de usuario y la capacidad de respuesta ante fallos.

1.3. Estructura del Documento

La presente memoria se organiza en varios capítulos que recogen tanto los fundamentos teóricos como el desarrollo práctico del sistema de automatización para invernaderos. La estructura está diseñada para ofrecer una comprensión progresiva del proyecto, desde su motivación inicial hasta su implementación y validación final.

1. **Resumen y Abstract:** visión general del trabajo en castellano e inglés, incluyendo los objetivos, metodología y resultados principales.
2. **Índice e Índice de Figuras:** herramientas para facilitar la navegación por el documento.
3. **Capítulo 1. Introducción:** motivación, objetivos y estructura general de la memoria.
4. **Capítulo 2. Antecedentes y estado actual del problema:** revisión de trabajos previos y tecnologías relevantes.
5. **Capítulo 3. Tecnologías y herramientas utilizadas:** descripción de Arduino, sensores, MQTT, Jetpack Compose y el control PID.
6. **Capítulo 4. Metodología de trabajo, análisis y requisitos:** metodología aplicada, SCRUM, requisitos funcionales y no funcionales.
7. **Capítulo 5. Diseño funcional del sistema:** casos de uso, diagramas generales, secuencias de eventos y sistema de control PID automático.
8. **Capítulo 6. Diseño físico del sistema:** conexionado, microcontrolador, servidor MQTT y configuración de red.
9. **Capitulo 7. Infraestructura de Red y Comunicación**: Servidor MQTT, Proxmox y Dynamic DNS.
10. **Capítulo 8. Gestión de eventos críticos y respuesta ante fallos:** mecanismos de seguridad, reconexión y registro histórico.
11. **Capítulo 9. Programación del firmware Arduino:** lógica de estados, manejo de topics y control de actuadores.
12. **Capítulo 10. Programación de la App Android:** estructura, funcionalidades principales e integración con MQTT.

13. **Capítulo 11. Resultados y líneas futuras:** resultados obtenidos, gráficas del control PID, conclusiones y mejoras propuestas.
14. **Capítulo 12. Referencias:** listado de fuentes consultadas.
15. **Capítulo 13. Apéndices:** manuales, guías y fragmentos de código relevantes.

2

Antecedentes y estado actual del problema

La automatización en entornos agrícolas ha evolucionado significativamente en los últimos años, impulsada por la necesidad de optimizar recursos, reducir la intervención humana y mejorar la trazabilidad de los procesos. Sin embargo, muchos sistemas actuales siguen siendo cerrados, poco flexibles o dependientes de soluciones propietarias que dificultan su adaptación a escenarios reales.
En el caso concreto de los invernaderos, el control de variables como la humedad, la temperatura o el nivel de agua suele realizarse mediante sistemas manuales o semiautomáticos, con escasa capacidad de respuesta ante fallos o incoherencias. Esto no solo limita la eficiencia operativa, sino que puede comprometer la seguridad de los cultivos y la integridad de los componentes físicos, como bombas o ventiladores.

Por otro lado, la proliferación de tecnologías como microcontroladores con conectividad **WiFi** (por ejemplo, el **ESP32**), protocolos ligeros como **MQTT** y frameworks modernos para desarrollo móvil (**Jetpack Compose en Android**) abre la puerta a soluciones más modulares, escalables y seguras. No obstante, integrar correctamente estas tecnologías requiere una arquitectura bien pensada, especialmente en lo que respecta a la sincronización entre firmware y aplicación, la gestión de eventos en tiempo real y la tolerancia a fallos.
En este contexto, se detecta una carencia clara: la mayoría de los sistemas maker o semiprofesionales no contemplan escenarios críticos como errores de lectura, reconexiones inseguras o condiciones de riesgo (por ejemplo, nivel bajo del depósito de agua durante el riego). Tampoco suelen incorporar mecanismos que bloqueen el sistema ante inconsistencias, lo que puede derivar en daños físicos o pérdida de trazabilidad.

El sistema se apoya en una infraestructura donde el servidor **MQTT** se despliega sobre Linux, alojado y gestionado mediante **Proxmox** para facilitar virtualización y mantenimiento. Para el acceso externo, se utiliza **DuckDNS** como servicio de DNS

dinámico que asigna un dominio a la IP pública cambiante del router, permitiendo alcanzar la web o servicios expuestos desde fuera de la red sin depender de una IP estática. Sobre esta base de comunicación y acceso, la lógica de seguridad del sistema prioriza la protección operativa: ante cualquier condición insegura (errores de lectura, incoherencias o nivel bajo del depósito de agua), se detienen las acciones en curso para proteger tanto el cultivo como los componentes electrónicos.

Además del contexto general, es relevante considerar cómo se abordan actualmente los sistemas de automatización agrícola en el ámbito profesional y en el entorno maker. En los últimos años han surgido múltiples soluciones comerciales basadas en controladores industriales, PLCs y plataformas **IoT** cerradas, que ofrecen estabilidad pero presentan un coste elevado y una capacidad de personalización limitada. Estas soluciones suelen requerir licencias propietarias y no permiten modificar la lógica interna del sistema, lo que dificulta su adaptación a invernaderos pequeños o proyectos educativos.

En paralelo, la comunidad maker ha desarrollado numerosos proyectos basados en microcontroladores como **Arduino** o **ESP32**, orientados a la monitorización básica de variables ambientales. Sin embargo, la mayoría de estas propuestas carecen de mecanismos avanzados de seguridad, tolerancia a fallos o control autónomo mediante algoritmos como **PID**. Tampoco suelen incluir sistemas de registro histórico, sincronización fiable entre dispositivos o arquitecturas escalables que permitan crecer hacia soluciones más complejas.

Por otro lado, el uso de protocolos ligeros como **MQTT** se ha consolidado como estándar en aplicaciones IoT debido a su eficiencia y simplicidad. Existen brokers ampliamente utilizados —como **Mosquitto**, HiveMQ o EMQX— que permiten gestionar comunicaciones en tiempo real entre sensores, actuadores y aplicaciones móviles. No obstante, su integración en proyectos agrícolas requiere una configuración adecuada, especialmente en lo relativo a seguridad, autenticación y acceso remoto.

También es importante destacar que el control automático de variables ambientales mediante algoritmos **PID** es una técnica ampliamente utilizada en la industria, pero poco implementada en proyectos de bajo coste o en invernaderos domésticos. La mayoría de sistemas maker se limitan a umbrales fijos (on/off), lo que reduce la precisión y puede generar oscilaciones o un uso ineficiente de recursos como el agua o la ventilación.

Finalmente, la tendencia actual en automatización agrícola apunta hacia sistemas híbridos que combinen control local, monitorización remota, registro histórico y capacidad de aprendizaje automático. Este proyecto se sitúa en esa línea, integrando tecnologías accesibles con una arquitectura modular que permite evolucionar hacia soluciones más avanzadas.

3

Tecnologías y herramientas utilizadas

3.1. Arduino y sensores embebidos

El núcleo del sistema se basa en un microcontrolador **Arduino ESP32**, una plataforma que combina la simplicidad del ecosistema Arduino con la potencia de un procesador de doble núcleo y conectividad **WiFi** integrada. Esta elección permite desarrollar un firmware flexible, capaz de gestionar múltiples sensores y actuadores en tiempo real sin comprometer la estabilidad del sistema. Además, el ESP32 ofrece un consumo energético reducido y una capacidad de procesamiento suficiente para ejecutar lógica de control autónomo, como el algoritmo PID implementado en este proyecto.

El uso de **sensores embebidos** resulta fundamental para obtener una lectura fiable de las variables ambientales del invernadero. Para la humedad del suelo se emplean sensores analógicos calibrados, mientras que la temperatura ambiente se mide mediante sensores digitales que garantizan precisión y estabilidad frente a interferencias. El nivel del depósito de agua se monitoriza mediante un sensor ultrasónico, cuya lectura permite proteger la bomba de riego ante situaciones de nivel insuficiente. Todos estos sensores se integran en una **arquitectura modular** que facilita su sustitución, ampliación o recalibración sin necesidad de modificar el resto del sistema.

El microcontrolador se encarga de procesar las señales de los sensores, aplicar filtros básicos para reducir ruido y generar eventos que reflejan el estado real del entorno. A partir de estas lecturas, el firmware ejecuta la lógica de control, tanto en modo manual como en modo autónomo mediante **PID**, regulando actuadores como la bomba de riego, los ventiladores o cualquier otro elemento conectado. La comunicación con la aplicación móvil se realiza mediante **MQTT**, lo que permite enviar actualizaciones en tiempo real y recibir comandos de forma ligera y eficiente.

Esta combinación de hardware embebido, sensores calibrados y un firmware estructurado en módulos proporciona una base sólida para un sistema de automatización fiable, escalable y fácil de mantener, adaptado a las necesidades reales de un invernadero moderno.

3.2. Protocolo MQTT y broker Mosquitto

El sistema utiliza **MQTT** como protocolo principal de comunicación entre el microcontrolador y la aplicación móvil. MQTT destaca por su ligereza y eficiencia, lo que lo convierte en una opción ideal para entornos donde los dispositivos deben enviar datos de forma continua sin saturar la red ni consumir recursos innecesarios. Su arquitectura basada en un modelo ***publish/subscribe*** permite desacoplar completamente los emisores y receptores de información, facilitando la escalabilidad del sistema y evitando dependencias directas entre los distintos componentes.

Para gestionar estas comunicaciones se emplea un broker compatible con MQTT, en este caso **Mosquitto**, instalado en un servidor local dentro de la red doméstica (192.168.1.20). Este **broker** actúa como punto central de intercambio de mensajes entre el **ESP32** y la aplicación Android, ofreciendo baja latencia, estabilidad y control total sobre la infraestructura. Además, permite monitorizar los mensajes publicados mediante herramientas como **mosquitto_sub**, lo que facilita la depuración del sistema durante el desarrollo.

El uso de MQTT también simplifica la integración del control autónomo basado en PID, ya que el microcontrolador puede publicar periódicamente el estado de los sensores y recibir ajustes de referencia sin necesidad de establecer conexiones complejas. La estructura jerárquica de los topics permite organizar la información de forma clara, separando lecturas, comandos, alertas y eventos del sistema.

En conjunto, la combinación de **MQTT** y **Mosquitto** proporciona una comunicación fiable, rápida y flexible, adaptada a las necesidades de un invernadero automatizado donde la sincronización entre dispositivos es esencial para garantizar un funcionamiento seguro y coherente.

3.3. Broker y red

El sistema se apoya en una infraestructura de comunicaciones diseñada para ser estable, segura y fácil de mantener. El broker MQTT —pieza central en la comunicación entre el firmware y la aplicación móvil— se despliega sobre una máquina virtual **Debian**, alojada en un entorno **Proxmox VE**. Esta elección permite aislar el servicio, simplificar las tareas de mantenimiento y garantizar que cualquier actualización o fallo del sistema operativo no afecte al resto de servicios del servidor físico.

El uso de **Proxmox** aporta varias ventajas prácticas: snapshots rápidos para recuperar el sistema ante errores, gestión centralizada de recursos y la posibilidad de migrar o ampliar la máquina virtual sin interrumpir el funcionamiento del broker. Sobre esta base se instala y configura el servidor MQTT optimizado para manejar conexiones ligeras y mantener la comunicación en tiempo real con el **ESP32** y la aplicación Android.

Para permitir el acceso externo al **broker** sin depender de una **IP fija**, se utiliza **DuckDNS** como servicio de DNS dinámico. Este mecanismo asigna un dominio estable que apunta automáticamente a la IP pública del router, facilitando la

conexión remota desde la aplicación móvil incluso cuando la dirección cambia. La configuración se completa con reglas de reenvío de puertos y medidas básicas de seguridad para evitar accesos no autorizados.

Toda esta infraestructura —virtualización con Proxmox, servidor Debian, broker **MQTT y DNS dinámico**— proporciona un entorno robusto, reproducible y escalable. Además, permite que el sistema pueda crecer en el futuro, añadiendo nuevos servicios o ampliando la capacidad del broker sin necesidad de modificar la arquitectura principal.

3.4. Android Jetpack Compose

La aplicación móvil del sistema está desarrollada con **Android Jetpack Compose**, el **framework** moderno de Google para construir interfaces declarativas. Su enfoque basado en estados encaja especialmente bien con un proyecto donde la información cambia en tiempo real, ya que permite actualizar la interfaz de forma automática cada vez que llega un nuevo mensaje desde el broker MQTT.

Compose facilita la creación de pantallas limpias, reactivas y fáciles de mantener. En lugar de depender de vistas tradicionales y estructuras complejas, la interfaz se construye mediante funciones composables que representan cada elemento visual del sistema: tarjetas de estado, indicadores de sensores, botones de control o paneles de notificaciones. Esto permite que la aplicación sea más modular y que cada componente pueda evolucionar sin afectar al resto.

Además, **Jetpack Compose** se integra de forma natural con otras herramientas del ecosistema Android, como **ViewModel**, **LiveData** o **Kotlin** Coroutines, lo que simplifica la gestión del estado y la comunicación con el firmware. Gracias a esta arquitectura, la app puede mostrar en tiempo real la humedad del suelo, la temperatura ambiente, el nivel del depósito de agua o el estado de los actuadores, reflejando también el funcionamiento autónomo del control PID.
Otro aspecto relevante es la capacidad de Compose para adaptarse a diferentes tamaños de pantalla, algo especialmente útil en un proyecto que puede utilizarse tanto en móviles como en tablets. La interfaz se mantiene coherente, clara y profesional, incluso cuando se añaden nuevas funcionalidades o se reorganizan los elementos visuales.

En conjunto, Jetpack Compose aporta velocidad de desarrollo, claridad estructural y una experiencia de usuario moderna, convirtiéndose en una herramienta clave para la parte móvil del sistema.

3.5. Streaming de video mediante RTSP y LibVLC

El sistema incorpora una cámara IP que transmite vídeo en tiempo real mediante el protocolo **RTSP (Real Time Streaming Protocol)**. Este flujo se integra directamente en la aplicación Android utilizando la librería **LibVLC**, que permite decodificación por hardware y reproducción fluida incluso en dispositivos móviles.
La app accede al flujo mediante una **URL RTSP** protegida y lo muestra dentro de un componente, sin necesidad de servidores intermedios ni procesamiento adicional en el **ESP32**.

Esta funcionalidad complementa la monitorización del invernadero, permitiendo al usuario supervisar visualmente el estado del cultivo desde cualquier lugar.

3.6. Control PID aplicado a riego y ventilación

El sistema incorpora un **control automático basado en PID** (Proporcional-Integral-Derivativo) para gestionar de forma autónoma dos de las variables más importantes del invernadero: la humedad del suelo y la temperatura ambiente. Frente a los sistemas tradicionales basados en umbrales fijos –que simplemente activan o desactivan un actuador cuando se supera un límite– el PID permite una regulación mucho más fina, estable y eficiente, evitando oscilaciones bruscas y reduciendo el desgaste de los componentes.

En el caso del riego, el controlador PID ajusta la activación de la **bomba de riego** en función de la diferencia entre la humedad real del suelo y el valor de referencia establecido. Esto permite que el sistema responda de manera progresiva: si la humedad está ligeramente por debajo del objetivo, el riego se activa durante intervalos cortos; si la diferencia es mayor, la respuesta es más intensa. De esta forma se evita tanto el exceso de agua como los ciclos de encendido y apagado continuos que suelen aparecer con controles más simples.

Para la ventilación ocurre algo similar. El **PID** analiza la temperatura ambiente y regula la activación de los ventiladores de forma proporcional al desvío respecto al valor deseado. Esto permite mantener el invernadero en un rango térmico estable, evitando picos de calor que podrían afectar al cultivo y reduciendo el consumo energético al no depender de activaciones bruscas.
El firmware del **ESP32** ejecuta el cálculo del **PID** en tiempo real, utilizando lecturas filtradas de los sensores para minimizar el ruido y garantizar una respuesta estable. Además, el sistema publica periódicamente los valores del proceso (PV), la referencia (SP) y la salida del controlador (CV) mediante **MQTT**, lo que permite visualizar en la aplicación móvil la evolución del control autónomo y registrar su comportamiento para análisis posteriores.

La integración del **PID** aporta un nivel de autonomía que va más allá del simple control remoto. El sistema es capaz de mantener por sí mismo las condiciones óptimas del invernadero, reaccionar ante cambios ambientales y adaptarse a diferentes escenarios sin intervención del usuario, lo que mejora la eficiencia y la fiabilidad del conjunto.

4

Metodología de trabajo. Análisis y requisitos

4.1. Metodología de Desarrollo modular

El desarrollo del sistema se ha planteado siguiendo una arquitectura modular, separando cada componente en bloques independientes pero perfectamente integrados entre sí. Esta estrategia permite trabajar de forma aislada en el firmware, la aplicación móvil, la comunicación MQTT o el control PID sin que un cambio en un módulo afecte al resto.

En el firmware del microcontrolador, la modularidad se ha aplicado mediante **funciones y archivos separados**, organizando la lectura de sensores, la gestión de actuadores, la comunicación MQTT y la lógica de control en bloques independientes. Aunque Arduino permite el uso de clases al estar basado en C++, en este proyecto se ha optado por una estructura funcional clara y directa, más adecuada para un sistema embebido de recursos limitados.

En la aplicación Android, la modularidad se implementa mediante **clases y componentes propios de Jetpack Compose**, separando la interfaz, la gestión del estado, la comunicación MQTT y los modelos de datos. Esta organización facilita la mantenibilidad y permite ampliar la aplicación sin afectar al resto de módulos.

Gracias a esta estructura modular, ha sido posible iterar rápidamente en la lógica de sensores, ajustar la interfaz en Android o modificar la estructura de topics MQTT sin comprometer la estabilidad del conjunto.

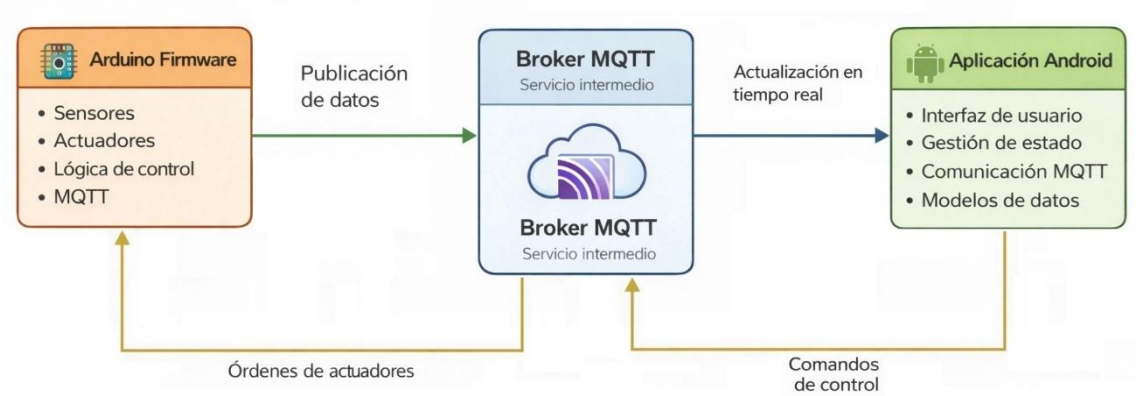

Fig. 1. Arquitectura modular del sistema.

4.2. Metodologías ágiles (SCRUM) y ciclos de desarrollo

Para organizar el trabajo se ha seguido una adaptación ligera de **SCRUM**, centrada en ciclos cortos de desarrollo y validación continua. El proyecto se ha dividido en **sprints funcionales**, cada uno orientado a completar una parte concreta del sistema: lectura de sensores, control PID, interfaz móvil, sincronización MQTT, gestión de eventos, etc.

Al final de cada sprint se realizaba una revisión del estado del sistema, identificando mejoras, errores y prioridades para el siguiente ciclo. Esta metodología ha permitido mantener un ritmo constante, detectar problemas de integración de forma temprana y asegurar que cada avance estuviera completamente probado antes de continuar.

Aunque se trata de un proyecto individual, la filosofía ágil ha sido clave para mantener una estructura de trabajo ordenada, flexible y orientada a resultados, evitando bloqueos y permitiendo adaptar el desarrollo a los problemas reales que iban surgiendo.

4.3. Análisis y diseño de requisitos

El análisis de requisitos se ha realizado identificando primero las necesidades reales del sistema y del entorno de uso. A partir de estas necesidades se han definido los requisitos funcionales y no funcionales que guían el diseño del firmware, la aplicación móvil y la infraestructura de comunicación.

Este proceso ha permitido establecer qué debe hacer el sistema, cómo debe comportarse ante situaciones anómalas y qué condiciones mínimas debe cumplir para considerarse fiable y seguro.

Los requisitos se han organizado en dos categorías principales: funcionales y no funcionales.

4.3.1. Requisitos Funcionales

Los requisitos funcionales describen **las acciones que el sistema debe ser capaz de realizar**:

1. **Monitorización de sensores en tiempo real:** El sistema deberá leer periódicamente los valores de humedad del suelo, temperatura y nivel de agua, enviándolos mediante MQTT a la aplicación Android.
2. **Control remoto de actuadores:** El usuario podrá activar o desactivar la bomba de riego, el ventilador, las puertas o las luces LEDs desde la aplicación Android, enviando comandos al firmware a través del broker MQTT.
3. **Control automático mediante lógica PID**: El firmware deberá ejecutar algoritmos PID para regular el riego y la ventilación en función de los valores de los sensores y los umbrales configurados.

4. **Modo seguro**: El sistema deberá entrar en modo seguro cuando se detecten condiciones anómalas (fallo de sensor, desconexión MQTT, valores fuera de rango), desactivando los actuadores y notificando al usuario.
5. **Historial de eventos:** La aplicación Android deberá almacenar en una base de datos local (SQLite) los eventos relevantes del sistema, incluyendo cambios de estado, alertas y acciones del usuario.
6. **Visualización del estado del sistema:** La aplicación deberá mostrar en tiempo real el estado de conexión MQTT, los valores de los sensores y el estado de los actuadores.
7. **Configuración del sistema:** El usuario podrá modificar parámetros como:
    1. Humedad Ideal
    2. Temperatura Ideal
    3. Color de luces LED
    4. IP del broker MQTT
8. **Visualización local en pantalla OLED:** El sistema deberá mostrar en la pantalla OLED los valores de humedad del suelo, temperatura ambiente, nivel del depósito y estado del riego en tiempo real, permitiendo la supervisión local sin necesidad de utilizar la aplicación Android.
9. **Visualización del video en tiempo real:** La aplicación Android deberá mostrar en tiempo real el vídeo procedente de la cámara IP instalada en el invernadero.
   La transmisión se realiza mediante el protocolo RTSP, y la reproducción se integra directamente en la interfaz mediante la librería LibVLC.

4.3.2. Requisitos No Funcionales

Los requisitos no funcionales definen las características de calidad que debe cumplir el sistema:

1. **Baja latencia en la comunicación MQTT:** La comunicación entre firmware y aplicación deberá ser prácticamente instantánea, garantizando una actualización fluida del estado del sistema.
2. **Robustez ante fallos:** El firmware deberá continuar funcionando incluso si la aplicación se desconecta, manteniendo el control automático activo.
3. **Persistencia local:** La aplicación deberá almacenar los eventos en SQLite de forma eficiente y sin bloquear la interfaz.
4. **Eficiencia energética:** El firmware deberá minimizar el uso de energía, especialmente en sensores y actuadores, para garantizar un funcionamiento prolongado.
5. **Interfaz intuitiva:** La aplicación Android deberá presentar una interfaz clara, compacta y adaptada a pantallas pequeñas, facilitando la interacción del usuario.
6. **Escalabilidad:** El sistema deberá permitir la incorporación futura de nuevos sensores o actuadores sin modificar la arquitectura principal.
7. **Integridad de datos:** Los mensajes MQTT deberán publicarse con formato consistente y validado para evitar estados incoherentes.
8. **Compatibilidad hardware:** El firmware deberá funcionar correctamente en microcontroladores compatibles con Arduino y sensores estándar de humedad, temperatura y nivel.
9. **Disponibilidad del sistema:** El firmware deberá reiniciarse automáticamente en caso de bloqueo, garantizando la continuidad del servicio.

10. **Seguridad básica en la comunicación:** El sistema deberá evitar comandos no válidos y validar los mensajes recibidos para prevenir comportamientos inesperados.
11. **Reconexión automática MQTT:** El sistema deberá intentar reconectarse automáticamente al broker en caso de pérdida de conexión.
12. **Sincronización inicial de estados:** Al iniciar la aplicación, deberá solicitar los valores actuales de sensores y actuadores para sincronizar la interfaz con el estado real del sistema.
13. **Gestión de mensajes retenidos:** El firmware deberá publicar estados críticos como mensajes retenidos para garantizar que la aplicación reciba la información incluso si se conecta más tarde.
14. **Legibilidad y bajo consumo de la pantalla OLED:** La pantalla OLED deberá ser legible en interiores y mantener un consumo energético reducido para no afectar a la estabilidad del sistema ni interferir con la alimentación del ESP32.

5

Diseño funcional del sistema

5.1. Descripción general del sistema

El sistema desarrollado corresponde a un invernadero automatizado capaz de monitorizar variables ambientales, controlar actuadores y operar tanto de forma manual como autónoma mediante un controlador PID. El sistema está compuesto por los siguientes elementos principales:

- **Recinto del invernadero**: estructura aislada que contiene las plantas, equipada con ventanas motorizadas y ventiladores para la regulación térmica.
- **Depósito de agua**: incluye un sensor de distancia por ultrasonidos para medir el nivel del agua y una bomba encargada del riego.
- **Sensores ambientales**: sensores de humedad del suelo, temperatura y humedad relativa del ambiente.
- **Sistema de iluminación**: tira de LEDs controlable desde el firmware y la aplicación Android.
- **Pantalla OLED integrada**: muestra en tiempo real los valores de humedad, temperatura, nivel del depósito y estado del riego, permitiendo una supervisión local sin necesidad de acceder a la aplicación.
- **Cámara integrada**: permite capturar imágenes periódicas del estado del cultivo y activar o desactivar la emisión en streaming.
- **Firmware en ESP32**: encargado de leer sensores, ejecutar el control PID, gestionar actuadores y comunicarse mediante MQTT.
- **Aplicación Android**: interfaz principal del usuario para visualizar datos, recibir alertas y controlar el sistema.
- **Broker MQTT**: encargado de distribuir los mensajes entre el firmware y la aplicación.

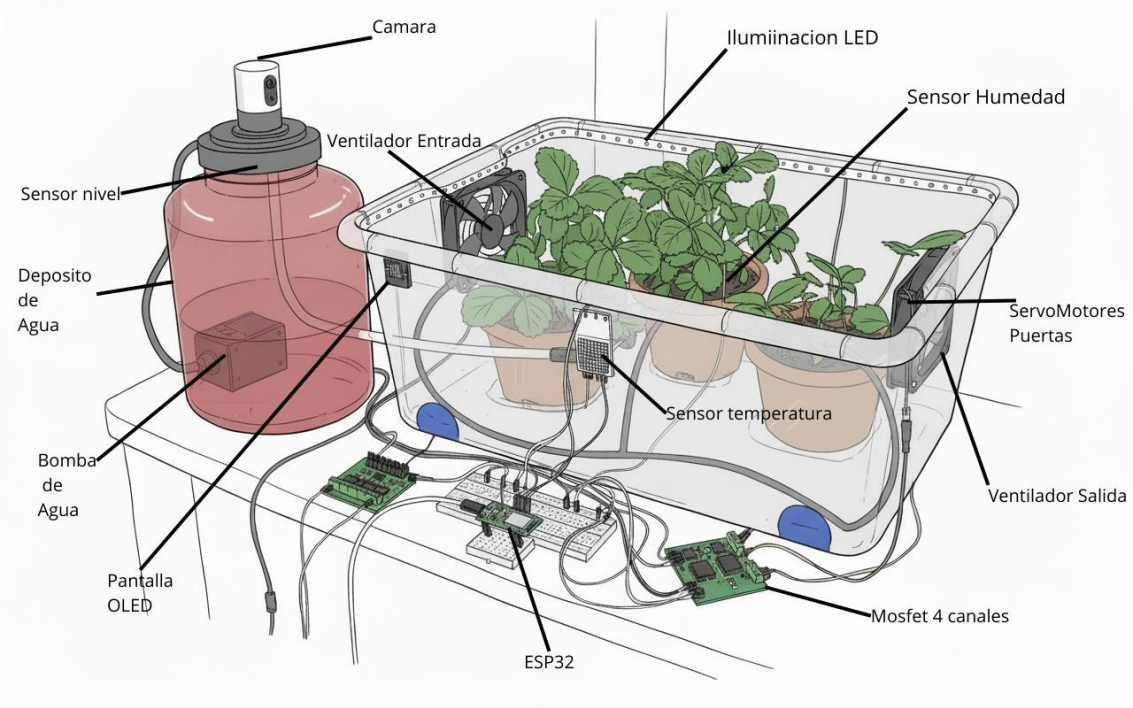

Fig. 2 Vista física del sistema

5.2. Comportamiento general del sistema

El funcionamiento del sistema se basa en la monitorización continua y el control automático de las variables críticas del invernadero:

- El **depósito de agua** almacena el agua utilizada para el riego. Su nivel se supervisa mediante un sensor ultrasónico, y la bomba se activa cuando el sistema lo requiere.
- El **ESP32** monitoriza periódicamente los sensores y publica sus valores en el broker MQTT, desde donde la aplicación Android los recibe en tiempo real.
- Desde la aplicación Android, el usuario puede **activar o desactivar** los actuadores: bomba de agua, ventiladores, ventanas y luces LED.
- El sistema incorpora un **modo seguro**, que desactiva los actuadores y notifica al usuario si se detectan fallos de sensores, niveles críticos o desconexiones.
- Cuando el modo automático está activado, el firmware ejecuta un **control PID** para mantener la humedad del suelo y la temperatura ambiente dentro de los valores configurados, actuando sobre la bomba, ventiladores y ventanas según sea necesario.
- La cámara integrada captura imágenes periódicas para generar un **histórico visual**, y permite activar o desactivar el streaming desde la aplicación.
- El sistema almacena un **registro histórico** de sensores y actuadores, accesible desde la aplicación Android.
- La **pantalla OLED** integrada muestra en tiempo real los valores de humedad, temperatura, nivel del depósito y estado del riego, permitiendo una supervisión local sin necesidad de acceder a la aplicación.

5.3. Casos de uso del sistema

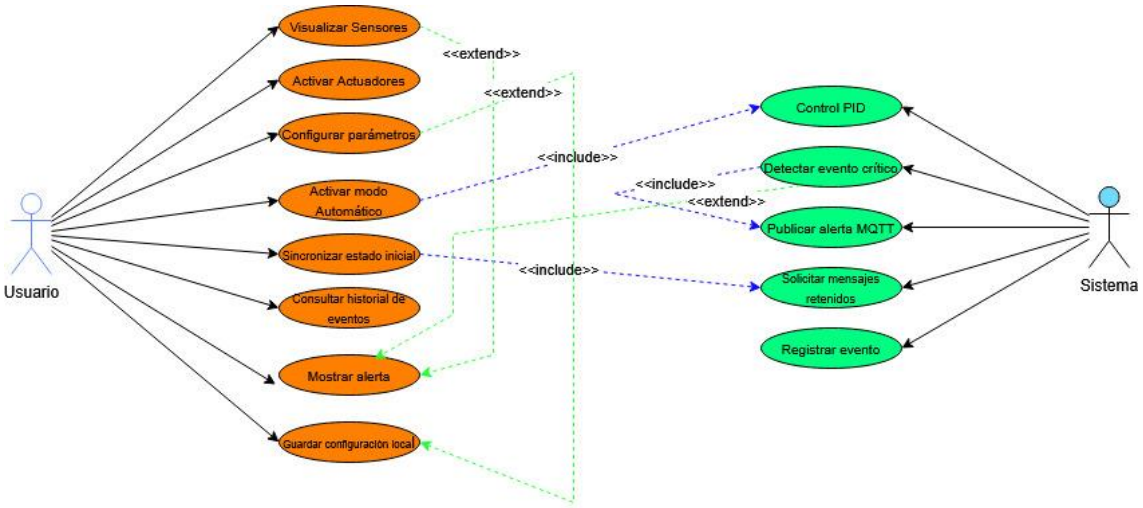

Fig. 3 Diagrama de Casos de Uso del Sistema.

El sistema desarrollado permite la monitorización y el control remoto de un invernadero automatizado mediante una arquitectura modular basada en firmware embebido, comunicación MQTT y una aplicación Android.
Para describir el comportamiento esperado desde la perspectiva del usuario, se definen los siguientes **casos de uso**, que representan las interacciones principales con el sistema:

CU1 – Visualizar sensores
Actor: Usuario
Descripción: El usuario consulta en la aplicación Android los valores actuales de humedad, temperatura y nivel de agua.
Precondición: El sistema debe estar conectado al broker MQTT.
Postcondición: La interfaz muestra los valores actualizados en tiempo real.

CU2 – Activar actuadores
Actor: Usuario
Descripción: El usuario enciende o apaga la bomba de riego, el ventilador u otros actuadores disponibles desde la aplicación.
Precondición: Conexión activa con el broker MQTT.
Postcondición: El firmware ejecuta la orden y publica el nuevo estado.

CU3 – Configurar parámetros
Actor: Usuario
Descripción: El usuario modifica umbrales de humedad, temperatura objetivo, humedad objetivo, color de LEDs, intervalo de muestreo o la IP del broker MQTT.
Precondición: La aplicación debe estar sincronizada con el firmware.
Postcondición: Los nuevos parámetros quedan almacenados y aplicados.

CU4 – Activar modo automático
Actor: Usuario
Descripción: El usuario activa el modo automático para que el sistema gestione riego y ventilación mediante control PID.

Precondición: Parámetros de control configurados y sensores operativos.
Postcondición: El sistema pasa a funcionar en modo automático.

CU5 – Ejecutar control PID
Actor: Sistema
Descripción: El firmware regula los actuadores según los valores de los sensores y los parámetros fijados mediante el algoritmo PID.
Precondición: Modo automático activado (CU4).
Postcondición: Los actuadores se ajustan automáticamente para mantener los valores dentro de los rangos objetivo.

CU6 – Detectar evento crítico
Actor: Sistema
Descripción: El sistema identifica condiciones anómalas, como valores fuera de rango, fallos de sensores o pérdida de conexión.
Precondición: Lectura periódica de sensores y supervisión del estado del sistema.
Postcondición: Se genera un evento crítico que desencadena la gestión de alertas.

CU7 – Gestión de alertas
Actor: Sistema
Descripción: El sistema publica una alerta MQTT y muestra una notificación en la aplicación cuando se detecta un evento crítico.
Precondición: Detección previa de un evento crítico (CU6).
Postcondición: El usuario recibe la alerta en la aplicación.

CU8 – Sincronizar estado inicial
Actor: Usuario
Descripción: Al iniciar la aplicación, el usuario solicita los estados actuales del sistema para sincronizar la interfaz.
Precondición: Inicio de la aplicación y conexión con el broker.
Postcondición: La interfaz refleja el estado real del sistema.

CU9 – Solicitar mensajes retenidos
Actor: Sistema
Descripción: El firmware publica los estados actuales como mensajes retenidos para garantizar la sincronización con la aplicación.
Precondición: El sistema debe tener estados válidos.
Postcondición: La aplicación puede sincronizarse incluso si se conecta más tarde.

CU10 – Consultar historial de eventos
Actor: Usuario
Descripción: El usuario accede a los eventos registrados en la base de datos local (SQLite).
Precondición: Deben existir eventos almacenados.
Postcondición: Se muestran los detalles del evento seleccionado.

CU11 – Registrar evento
Actor: Sistema
Descripción: El sistema guarda en SQLite los cambios relevantes, como

activación de actuadores, alertas o variaciones críticas.
Precondición: Se ha producido un evento significativo.
Postcondición: El evento queda registrado para su consulta posterior.

CU12 – Mostrar alerta
Actor: Usuario
Descripción: La aplicación muestra una alerta visual o una notificación cuando se recibe un mensaje de alerta desde el broker MQTT.
Precondición: Se ha publicado previamente una alerta MQTT (CU7) y la aplicación está conectada al broker.
Postcondición: El usuario es informado del evento crítico a través de la interfaz o de una notificación.

CU13 – Guardar parámetros de configuración
Actor: Usuario
Descripción: El usuario almacena localmente en la aplicación los parámetros de configuración modificados, como humedad objetivo, temperatura objetivo, umbrales de alerta, intervalo de muestreo o la IP del broker.
Precondición: El usuario ha modificado al menos un parámetro de configuración.
Postcondición: Los parámetros quedan guardados y se aplicarán en la siguiente comunicación con el firmware

5.4. Diagrama general del sistema

La arquitectura general del sistema ya ha sido presentada en el apartado 4.1 mediante la **Fig. 1,** donde se muestra la estructura modular compuesta por el firmware embebido, el broker MQTT y la aplicación Android. Dicha organización permite visualizar los componentes principales y sus interacciones, sirviendo como base para los diagramas funcionales y de secuencia que se presentan a continuación.

5.5. Diagrama de secuencia de eventos

Los **diagramas de secuencia** permiten representar de forma detallada el flujo temporal de mensajes entre los distintos componentes del sistema. A diferencia del diagrama de casos de uso, que describe las funcionalidades desde una perspectiva externa, los diagramas de secuencia muestran cómo se coordinan el firmware, el broker **MQTT** y la aplicación **Android** para ejecutar cada proceso.

En este apartado se analizan tres secuencias fundamentales del sistema: la activación automática de riego y ventilación, la gestión de notificaciones y alertas, y la sincronización entre el firmware y la aplicación móvil. Cada una de estas secuencias refleja un comportamiento clave del sistema, destacando la interacción entre sensores, actuadores, lógica de control y comunicación **MQTT**.

Estos diagramas permiten comprender con precisión el orden de los eventos, los mensajes intercambiados y las dependencias entre módulos, facilitando tanto la validación del diseño como su futura ampliación.

5.5.1. Diagrama general del ciclo automático de riego

Este diagrama representa el flujo completo del proceso de riego automático, incluyendo la lectura de sensores, la validación de condiciones, la activación y supervisión de la bomba, la gestión de eventos críticos y la comunicación con la aplicación Android.

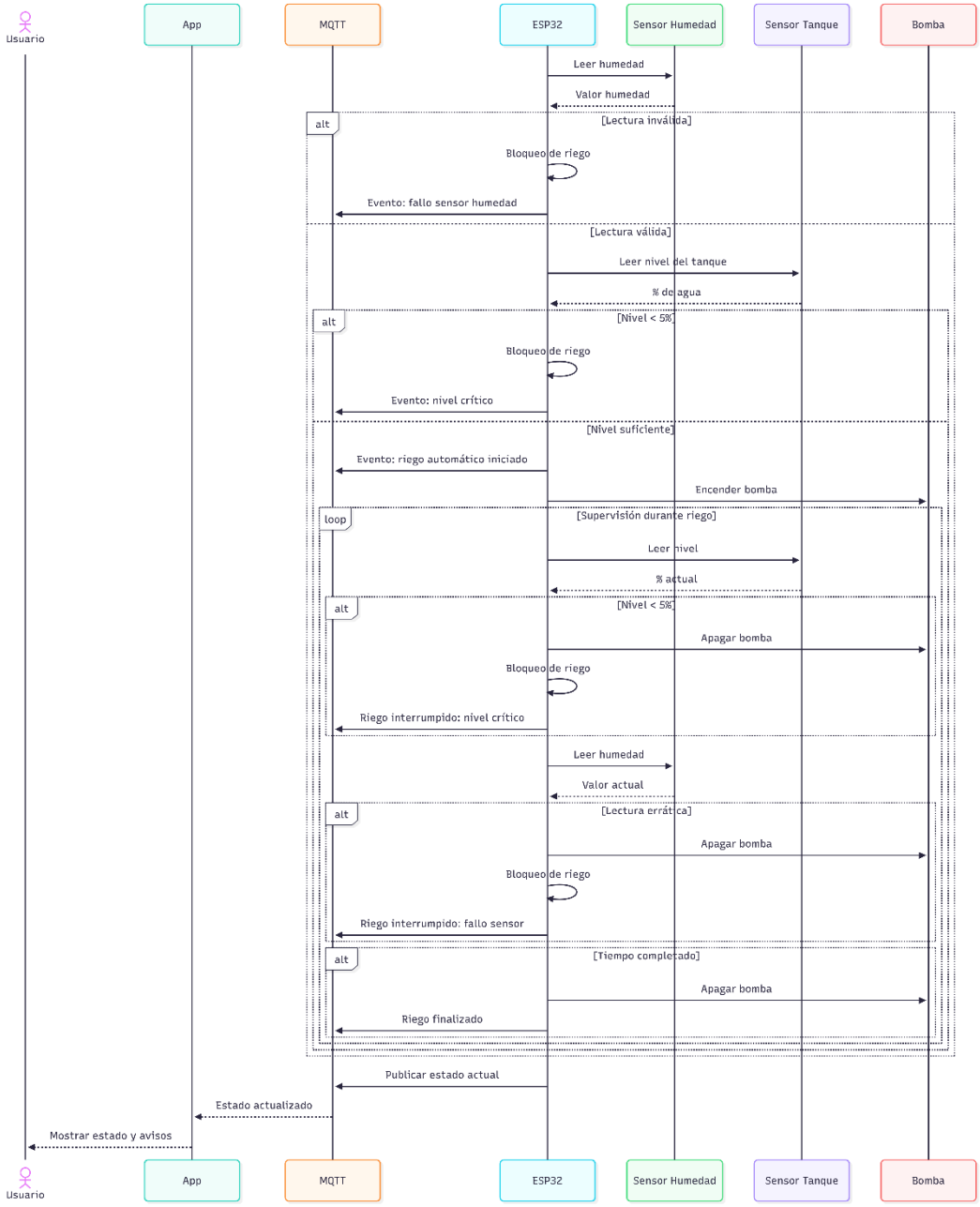

Fig. 4 Diagrama Secuencia Riego automático

5.5.2. Activación de riego y ventilación automáticos

Esta secuencia describe el proceso mediante el cual el usuario activa el modo automático desde la aplicación, lo que desencadena la ejecución del **control PID** en el firmware. En función de los valores de los sensores, el sistema decide activar o desactivar los actuadores correspondientes (bomba de riego y ventilador), manteniendo las condiciones ambientales dentro de los rangos establecidos.

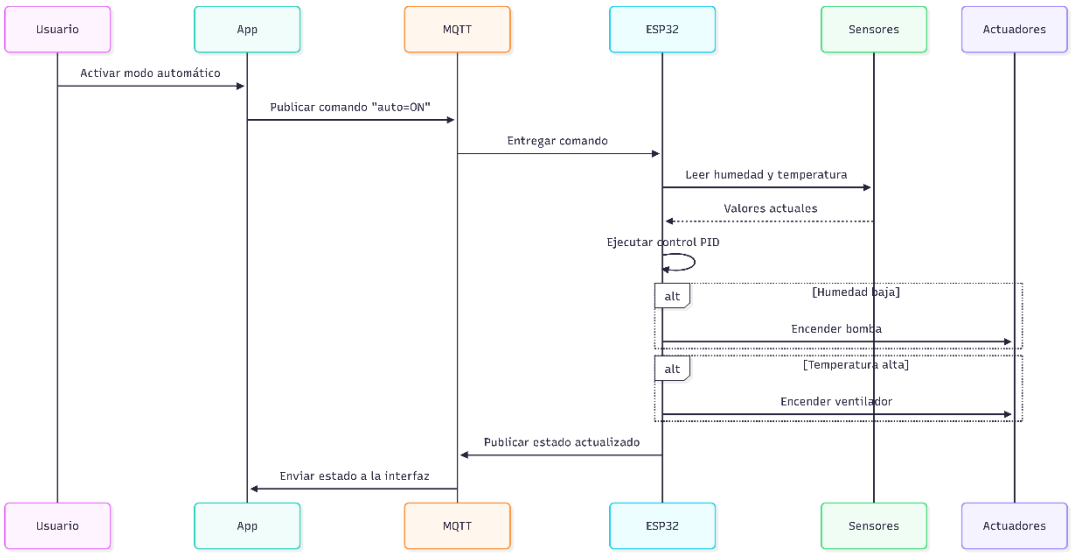

Fig. 5 Activación modo automático

5.5.3. Gestión de notificaciones y alertas

En esta secuencia se detalla cómo el firmware detecta un evento crítico —como un valor fuera de rango— y publica una alerta mediante **MQTT**. El broker distribuye el mensaje a la aplicación Android, que muestra una notificación al usuario. Este flujo garantiza una comunicación inmediata y fiable ante situaciones que requieren atención.

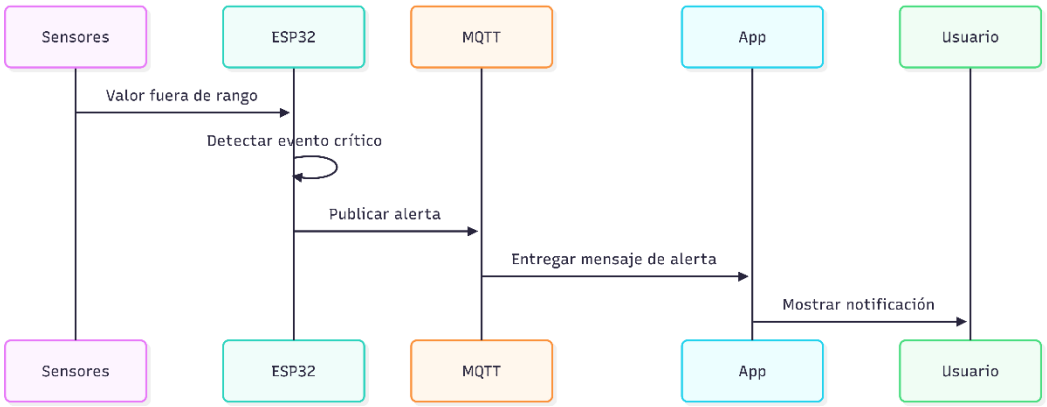

Fig. 6 Gestión notificaciones y alertas

5.5.4. Sincronización entre firmware y app

Esta secuencia representa el proceso de sincronización inicial cuando la aplicación se abre o recupera la conexión. La app solicita el estado actual al broker **MQTT**, que responde mediante mensajes retenidos. El firmware actualiza su estado si es necesario, asegurando que la interfaz muestre siempre información coherente y actualizada.

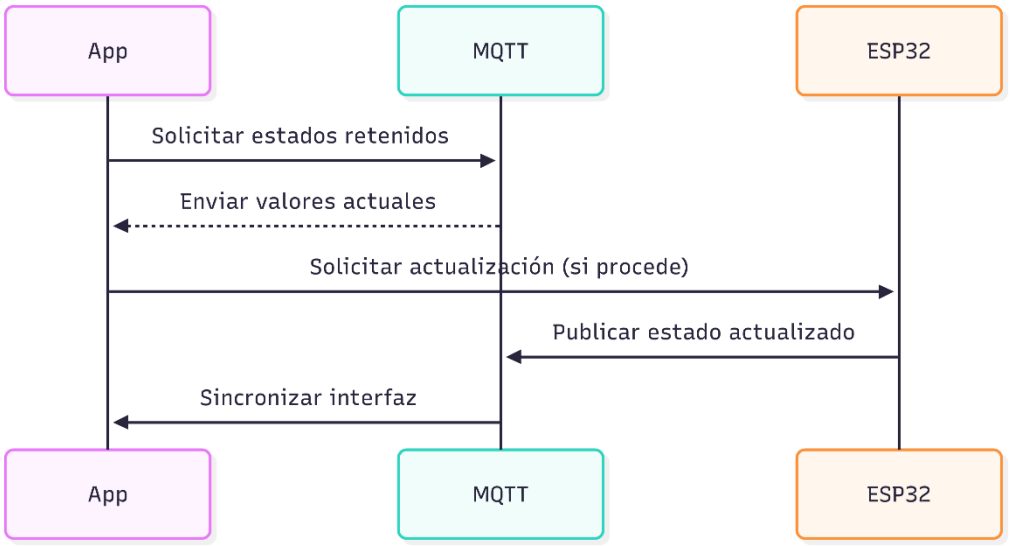

Fig. 7 Sincronización entre firmware y app

5.5.5. Desactivación del modo automático

Esta secuencia describe el proceso mediante el cual el usuario desactiva el modo automático desde la aplicación Android. Al recibir la orden, el firmware detiene la ejecución del controlador PID, desactiva los actuadores que estuvieran funcionando automáticamente y publica el nuevo estado del sistema mediante MQTT. De esta forma, el sistema pasa a operar exclusivamente en modo manual.

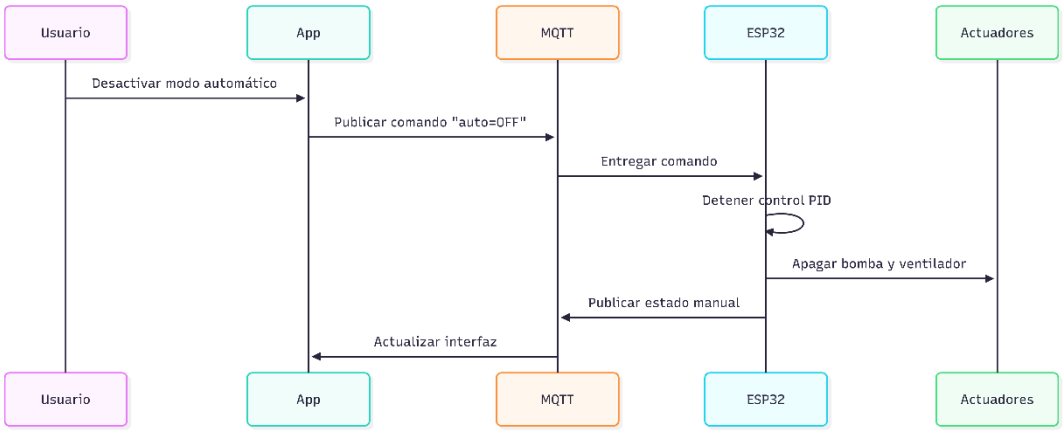

Fig. 8 Desactivación modo automático

5.5.6. Activación manual de actuadores

En esta secuencia se representa cómo el usuario activa manualmente cualquiera de los actuadores del sistema (bomba de riego, ventiladores, ventanas o iluminación). La aplicación envía el comando correspondiente al broker MQTT, que lo distribuye al firmware. El ESP32 ejecuta la acción solicitada y publica el estado actualizado para mantener la interfaz sincronizada.

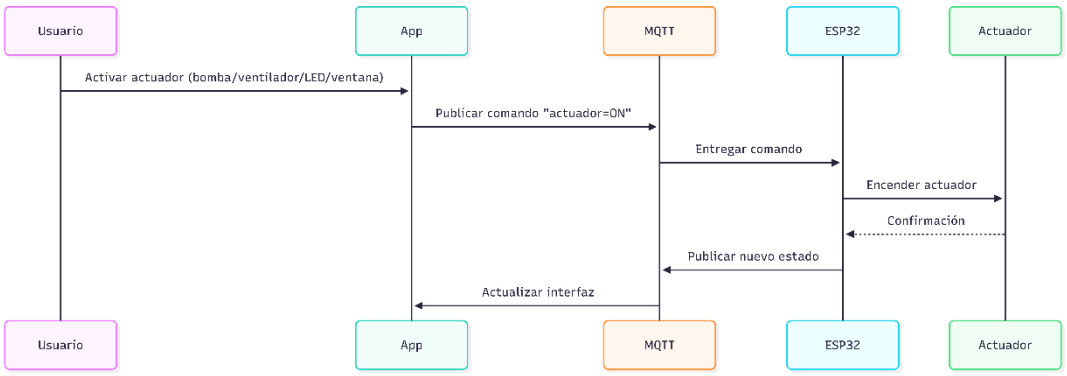

Fig. 9 Activación manual de actuadores

5.5.7. Desactivación manual de actuadores

Esta secuencia muestra el proceso inverso al anterior: el usuario desactiva manualmente uno o varios actuadores desde la aplicación. El comando se envía mediante **MQTT** al firmware, que apaga el actuador correspondiente y publica el nuevo estado. Este flujo garantiza un control manual preciso y coherente con la interfaz de usuario.

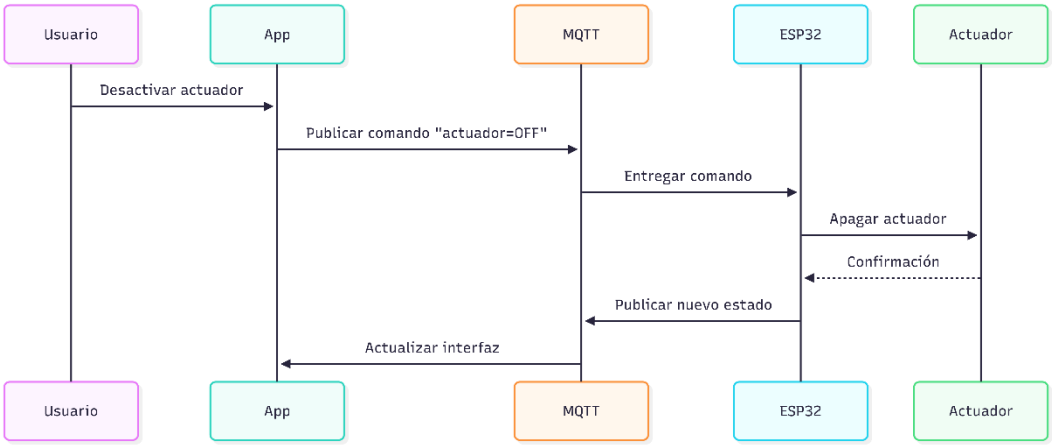

Fig. 10 Desactivación manual de actuadores

5.6. Sistema de Control PID automático

El sistema implementa un **control PID** (Proporcional–Integral–Derivativo) para regular de forma autónoma tanto la humedad del suelo como la temperatura del invernadero. Este enfoque permite que el sistema actúe de manera continua y adaptativa, corrigiendo desviaciones respecto a los valores objetivo sin intervención del usuario. El **PID** es especialmente adecuado para entornos donde los sensores presentan ruido y los actuadores no pueden funcionar de manera lineal, como ocurre con bombas de riego y ventiladores.

5.6.1. Fundamentos del control PID

El controlador PID calcula una señal de control $u(t)$ a partir del error entre el valor objetivo $r(t)$ y el valor medido $y(t)$:

$$\boldsymbol{e(t) = r(t) - y(t)}$$

La salida del PID se obtiene mediante:

$$\boldsymbol{u(t) = K_p e(t) + K_i \int e(t)\, dt + K_d \frac{de(t)}{dt}}$$

Donde:

- $\boldsymbol{K_p}$ controla la reacción proporcional al error instantáneo.
- $\boldsymbol{K_i}$ corrige errores acumulados en el tiempo.
- $\boldsymbol{K_d}$ suaviza la respuesta anticipando cambios bruscos.

En este proyecto se han utilizado dos controladores independientes:

- **PID de humedad**: K_p =2.5, K_i =0.08, K_d =0.05
- **PID de temperatura**: K_p =2.0, K_i =0.1, K_d =0.5

Estos valores se han ajustado empíricamente para obtener una respuesta estable sin oscilaciones excesivas.

5.6.2. Suavización y filtrado del comportamiento

El sistema no aplica directamente la salida del PID al actuador, sino que utiliza una **estrategia de suavización basada en ventanas temporales**:

1. El PID **solo se evalúa cada 30 segundos**, evitando oscilaciones rápidas debidas al ruido del sensor.
2. La salida del PID se convierte en **tiempos discretos de activación** (duración del riego o ventilación).
3. Se aplica un **cooldown** tras cada ciclo para evitar activaciones repetidas.

Ejemplo para humedad:

1. Si el PID devuelve un valor bajo → riego mínimo de 1 segundo.
2. Si devuelve un valor medio → riego proporcional.
3. Si devuelve un valor alto → riego máximo permitido (8 segundos).

Para temperatura:

1. Si el PID es negativo → se activa ventilación proporcional.
2. Si el PID es positivo → se apaga el ventilador y se cierran las compuertas.

Este enfoque híbrido (PID continuo + actuadores discretos) es ideal para hardware real, donde no se puede modular la potencia de una bomba o un ventilador con precisión analógica.

5.6.3. Seguridad y validación de sensores

Antes de ejecutar cualquier acción automática, el sistema realiza **verificaciones críticas**:

1. Nivel de agua adecuado
2. Sensor de humedad del suelo operativo
3. Sensor DHT funcionando correctamente
4. Modo manual desactivado

Si alguna condición falla, el sistema **bloquea el ciclo PID**, muestra un estado de alerta y evita daños en el hardware.

5.6.4. Funcionamiento del PID de Humedad

El sistema de control PID de humedad evalúa el estado del invernadero cada 5 segundos. En cada ciclo, se calcula la salida del PID en función del error entre la humedad actual y la humedad objetivo. Esta salida se traduce en un tiempo de riego mediante la función **calcularTiempoPID().**

Si el tiempo calculado es mayor que cero, se activa la bomba de riego durante ese intervalo, se envía un evento MQTT y se inicia un ciclo de riego controlado. Al finalizar el tiempo asignado, la bomba se apaga y se inicia un periodo de enfriamiento (**cooldown**) antes de la siguiente evaluación.

Este mecanismo permite ajustar la duración del riego de forma proporcional al error, evitando riegos excesivos y manteniendo la humedad dentro de un rango estable sin generar oscilaciones significativas

5.6.5. Funcionamiento del PID de Temperatura

El PID de temperatura actúa sobre dos elementos del sistema: **los ventiladores** y las compuertas del invernadero (controladas mediante **servomotores**). Su función es regular la temperatura interior mediante ventilación natural, ya que el proyecto no incorpora un sistema de calentamiento activo.

Cuando la salida del PID es negativa, se interpreta que la temperatura interior supera el valor objetivo. En ese caso:

- **Se activan los ventiladores.**
- **Se abren las compuertas.**
- **La ventilación se mantiene durante un tiempo proporcional al error térmico.**

Si la salida del PID es positiva, se considera que la temperatura está por debajo del objetivo, por lo que:

- **Se apagan los ventiladores.**
- **Se cierran las compuertas.**

Disposición de los ventiladores
Los ventiladores se han colocado **uno frente al otro**, generando una corriente de aire cruzada a través del invernadero. Este diseño permite que:

- Un ventilador **introduzca aire más frío** desde el exterior.
- El otro ventilador **expulse aire caliente** hacia fuera.

Este flujo cruzado funciona de forma similar al sistema de ventilación de un PC tradicional, donde un ventilador mete aire fresco y otro extrae aire caliente, creando un circuito de renovación constante y eficiente.

Limitaciones del control térmico
Es importante destacar que este sistema no enfría activamente el aire, sino que depende de la temperatura exterior. Por ejemplo, si fuera hay 12 °C, la temperatura interior puede mantenerse en torno a 20 °C gracias al efecto invernadero, pero los ventiladores no pueden reducirla por debajo de la temperatura ambiente.

Las plantas de fresa prosperan con temperaturas medias entre 15 °C y 20 °C, siendo ideales 18–22 °C durante el día y 10–15 °C por la noche. Debido a que las pruebas se realizaron en invierno, no ha sido posible evaluar completamente el comportamiento del PID de ventilación, ya que la temperatura interior rara vez superó los valores objetivo. Aun así, el sistema queda implementado y operativo para futuras pruebas en condiciones climáticas más cálidas.

5.6.6. Gráfica del comportamiento autónomo

La **Figura 3** muestra un fragmento ampliado del comportamiento del sistema de riego automatizado mediante control **PID**. Se trata de un zoom temporal que permite observar con mayor detalle la dinámica de respuesta del controlador ante un cambio de la humedad.
En este caso, se establece una humedad objetivo de 65 %, mientras que la humedad inicial del suelo es de aproximadamente 63 %. El sistema permanece en reposo hasta que la humedad desciende por debajo del umbral configurado. En ese momento, el **PID genera un primer impulso** de riego. Tras este riego inicial, la humedad vuelve a superar el valor objetivo y el sistema espera nuevamente a que la humedad descienda para volver a actuar. Cuando el valor **vuelve a caer por debajo del objetivo**, el PID calcula un nuevo tiempo de riego y activa la bomba. Este proceso se repite hasta que la humedad se estabiliza en torno al valor deseado.

La gráfica incluye tres elementos clave:

- **Línea azul:** representa la humedad actual del suelo, medida por el sensor capacitivo.
- **Línea naranja discontinua:** indica la humedad objetivo configurada por el usuario.
- **Segmentos verdes verticales:** muestran los instantes en los que se activa la bomba de riego, con altura proporcional al tiempo de riego calculado por el PID.

Es importante destacar que la humedad del suelo no responde de forma inmediata, a diferencia de otras variables físicas como la temperatura del aire. La humedad asciende y desciende lentamente, y su medición presenta pequeñas variaciones inherentes al propio sensor, incluso aplicando un suavizado en el código de Arduino. Por este motivo, la gráfica muestra ligeros picos y oscilaciones dentro de un rango completamente normal para este tipo de sensores.

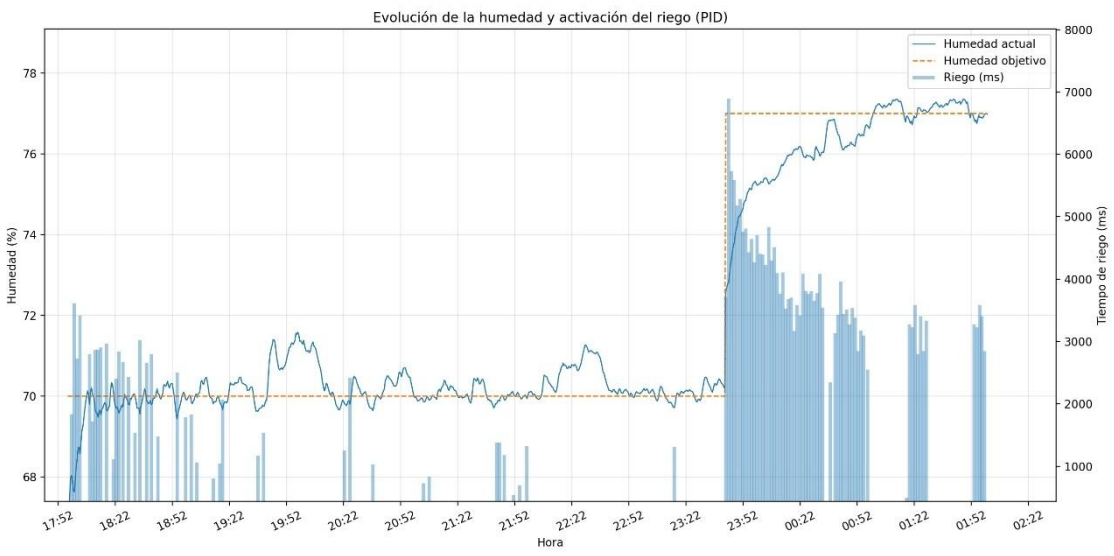

Fig. 11 Grafica Funcionamiento PID

A diferencia del control de temperatura —donde un calentador o un sistema de refrigeración pueden modificar el ambiente de forma rápida—, el riego influye en la humedad de manera gradual. El sistema está diseñado para mantener la humedad

por encima de un valor mínimo, estabilizándola en un rango definido (por ejemplo, entre X y X+2 %) sin generar sobrepasos ni caídas bruscas.

Además de esta figura ampliada, **se incluye también una gráfica adicional con un periodo de tiempo mucho mayor**, donde puede observarse el comportamiento del PID durante un intervalo prolongado. Esta gráfica permite verificar que el sistema mantiene la humedad estable durante todo el día sin oscilaciones significativas.

Finalmente, **se acompañan archivos .txt con formato JSON** que contienen capturas completas de larga duración **de funcionamiento real del PID**, incluyendo todos los valores de humedad, tiempos de riego y eventos registrados. Estos archivos sirven como evidencia del comportamiento autónomo del sistema y permite reproducir o analizar el funcionamiento del controlador en detalle.

5.7. Estructura de la Base de Datos Local (SQLite)

La aplicación Android utiliza una base de datos local **SQLite** para almacenar el historial de eventos generados por el sistema. Esta base de datos permite consultar alertas y notificaciones incluso sin conexión al broker **MQTT**.

La tabla utilizada se denomina **eventos** y almacena un registro por cada mensaje recibido desde MQTT.

Fig. 12 Tabla eventos

Cada evento contiene los siguientes campos:

- **id** *(INTEGER, PRIMARY KEY AUTOINCREMENT)*
  Identificador único del evento.
- **mensaje** *(TEXT)*
  Texto descriptivo del evento mostrado en la interfaz (por ejemplo: “Riego interrumpido por nivel demasiado bajo”).
- **topico** *(TEXT)*
  Topic MQTT desde el que se recibió el mensaje (por ejemplo: invernadero/alertas o invernadero/notificaciones).
- **timestamp** *(LONG)*
  Marca temporal en milisegundos, utilizada para ordenar y agrupar los eventos por día.

- **tipo** *(TEXT)*
  Categoría del evento, utilizada para aplicar formato visual en la interfaz (alerta, info, etc.).

El acceso a la base de datos se realiza mediante un **DAO (Data Access Object)** que permite:

- Obtener los eventos más recientes.
- Consultar todos los eventos en orden descendente.
- Insertar nuevos eventos.
- Eliminar eventos antiguos para evitar crecimiento ilimitado de la base de datos.

Esta estructura es suficiente para mantener un historial claro y eficiente del funcionamiento del sistema.

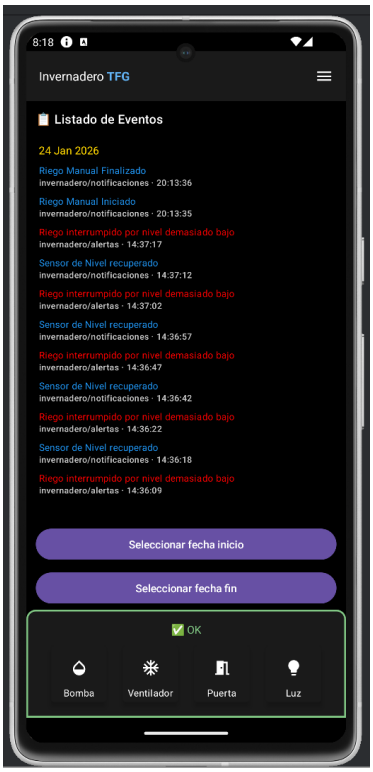

Fig. 13 Captura Pantalla Eventos

5.8. Estructura de comunicación MQTT

El sistema organiza la comunicación entre el firmware (**ESP32**) y la aplicación Android mediante una jerarquía de **topics** MQTT bajo el prefijo común.
Estos **topics** se dividen en tres grupos principales: comandos, lecturas y alertas/notificaciones.

5.8.1. Topic de comandos

Se utilizan para enviar órdenes desde la aplicación Android hacia el firmware:

- **invernadero/bomba/cmd**
  Comando para activar o desactivar la bomba de agua.
- **invernadero/bomba/estado**
  Publicación del estado actual de la bomba (ON/OFF).
- **invernadero/bomba/max**
  Configuración del tiempo máximo de riego permitido.

- **invernadero/led/power**
  Encendido o apagado de la tira LED.
- **invernadero/led/mode**
  Selección del modo de iluminación (color).
- **invernadero/ventiladores/cmd**
  Control de los ventiladores de entrada y salida.
- **invernadero/optimo/temperatura**
  Temperatura ideal configurada por el usuario.
- **invernadero/optimo/humedad**
  Humedad ideal configurada por el usuario.
- **invernadero/servomotor1/cmd**
  Comando para abrir o cerrar las ventanas mediante el servomotor.

5.8.2. Topic de lecturas

Publicados periódicamente por el ESP32 para informar del estado del invernadero:

- **invernadero/suelo/humedad**
  Lectura del sensor de humedad del suelo.
- **invernadero/aire/temperatura**
  Lectura de temperatura ambiente.
- **invernadero/aire/humedad**
  Lectura de humedad relativa del aire.
- **invernadero/tanque/nivel**
  Nivel del depósito de agua medido mediante ultrasonidos.

5.8.3. Topic de alertas y notificaciones

Usados para comunicar eventos importantes o críticos:

- **invernadero/alertas**
  Alertas críticas (fallo de sensor, modo seguro, desconexión, etc.).
- **invernadero/notificaciones**
  Mensajes informativos no críticos (cambios de estado, avisos).

6

Diseño físico del Sistema

6.1. Conexionado de sensores y actuadores

El sistema integra un conjunto de sensores y actuadores conectados al microcontrolador **ESP32 DevKit V1**, organizados sobre protoboard durante la fase de pruebas para permitir ajustes rápidos y verificación funcional. El conexionado se ha realizado siguiendo una distribución lógica que separa señales, potencia y alimentación externa para garantizar estabilidad.
El sistema emplea un único sensor de humedad del suelo, conectado al pin **GPIO34**, que proporciona una lectura analógica continua del sustrato. El sensor DHT22, encargado de medir temperatura y humedad ambiental, se conecta al pin **GPIO4**.
El sensor ultrasónico HC-SR04 no solo mide la distancia, sino que permite determinar el **nivel real del depósito de agua**, garantizando que la bomba no funcione en vacío. Esta lectura es esencial para activar alertas de nivel bajo y para que el sistema pueda decidir si es seguro iniciar un ciclo de riego.
Para la medición de distancia mediante ultrasonidos se utiliza un módulo HC-SR04, con las señales **TRIG** y **ECHO** conectadas a **GPIO12** y **GPIO33**, respectivamente.
Los actuadores se distribuyen de la siguiente forma:

- La **tira LED RGB** se controla mediante un MOSFET de 4 canales, utilizando los pines **GPIO25** (rojo), **GPIO26** (verde) y **GPIO27** (azul).
- El cuarto canal de este mismo MOSFET gestiona **dos ventiladores**, controlados desde **GPIO32**.
- La bomba de agua se activa mediante un MOSFET independiente conectado al pin **GPIO23**.
- Los **dos servomotores** se conectan a **GPIO13** y **GPIO14**, respectivamente. Los servomotores controlan la **apertura y cierre de las ventanas** del invernadero. Su función es permitir que los ventiladores trabajen de forma eficiente, facilitando la renovación del aire y ayudando a regular la temperatura interior. Cuando el sistema activa la ventilación, los servos abren las ventanas; cuando no es necesaria, las cierran para mantener la estabilidad térmica.

- La **pantalla OLED** se conecta al bus I2C del ESP32, utilizando los pines SDA (GPIO21) y SCL (GPIO22), sin requerir pines digitales adicionales.

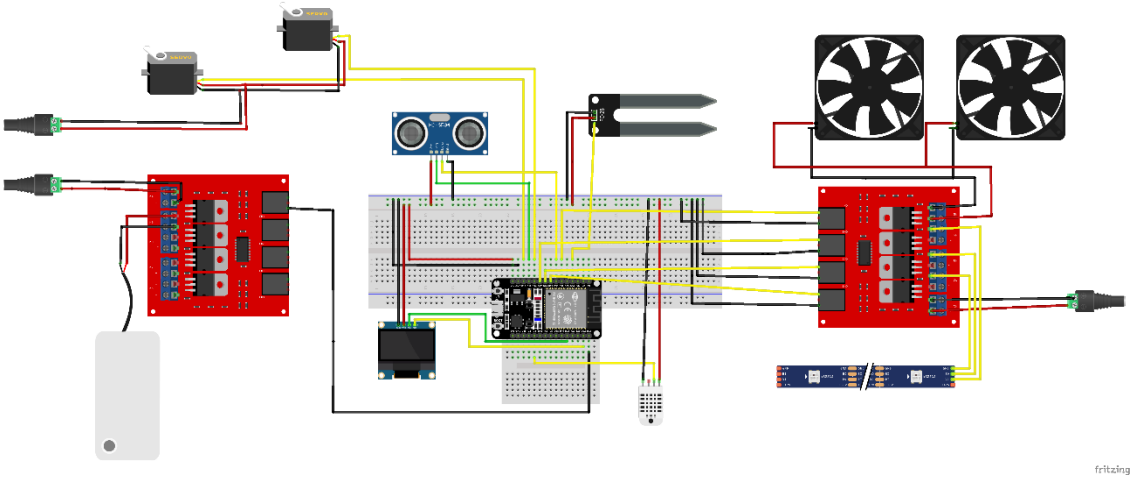

Fig. 14 Montaje físico del sistema de control: sensores, actuadores y microcontrolador integrados sobre protoboard para pruebas funcionales.

6.2. Microcontrolador y alimentación

El sistema está gobernado por un **ESP32 DevKit V1**, un microcontrolador de arquitectura dual-core que integra conectividad **WiFi 2.4 GHz** y **Bluetooth**, permitiendo comunicación inalámbrica directa con el servidor MQTT y facilitando la supervisión remota del sistema. Su capacidad para gestionar múltiples entradas analógicas y digitales, junto con temporizadores internos y PWM por hardware, lo convierte en una plataforma adecuada para controlar sensores ambientales y actuadores de distinta naturaleza.

En este proyecto, el ESP32 gestiona los siguientes elementos:

- **Sensor de humedad del suelo** conectado al pin **GPIO34** (entrada analógica).
- **Sensor DHT22** para temperatura y humedad del aire, conectado al pin **GPIO4**.
- **Sensor ultrasónico HC-SR04**, con **TRIG** en **GPIO12** y **ECHO** en **GPIO33**.
- **Tira LED RGB**, controlada mediante MOSFETs desde **GPIO25**, **GPIO26** y **GPIO27**.
- **Ventiladores**, gestionados desde **GPIO32** a través del cuarto canal del MOSFET de 4 vías.
- **Bomba de agua**, activada mediante un MOSFET independiente desde **GPIO23**.
- **Servomotores**, conectados a **GPIO13** y **GPIO14**, utilizando PWM para su posicionamiento.
- **Pantalla OLED (I2C)** conectada al bus I2C del ESP32 (SDA=21, SCL=22), utilizada para mostrar en tiempo real los valores de humedad, temperatura, nivel del tanque y estado del riego.

Alimentación distribuida

Durante las primeras pruebas se detectaron reinicios y comportamiento inestable al alimentar todos los actuadores desde la salida de 5 V del propio ESP32. Para garantizar un funcionamiento fiable, se ha implementado una arquitectura de alimentación distribuida basada en tres fuentes independientes:

- **Fuente de 12 V** → alimenta el **MOSFET de 4 canales**, encargado de la tira LED RGB y los dos ventiladores.

- **Fuente de 5 V** → dedicada exclusivamente al **MOSFET de la bomba de agua**, evitando caídas de tensión durante el arranque del motor.
- **Fuente de 2.5 V** → utilizada para los **servomotores**, reduciendo el consumo sobre el regulador del ESP32 y evitando interferencias en la lógica de control.

La **pantalla OLED**, al funcionar mediante el bus I2C y requerir un consumo muy reducido, se alimenta directamente desde los **3.3 V del ESP32**, sin afectar a la estabilidad del sistema.

Todas las fuentes comparten una **masa común** con el ESP32, asegurando referencias estables y evitando lecturas erráticas o activaciones no deseadas. Esta separación de cargas permite que el microcontrolador mantenga una operación estable incluso bajo condiciones de demanda elevada por parte de los actuadores.

6.3. Disposición final del Sistema

La **disposición final del sistema** se ha diseñado para garantizar estabilidad eléctrica, accesibilidad durante las pruebas y una organización clara de sensores, actuadores y módulos de potencia. Todos los componentes se han montado sobre una protoboard central, con el ESP32 situado en la zona superior para facilitar el cableado hacia los sensores y actuadores distribuidos alrededor.

Los **sensores** se colocan en posiciones que permiten una lectura fiable durante las pruebas: el **sensor de humedad del suelo** conectado al GPIO34 se sitúa en la parte frontal para facilitar su inserción en el sustrato, mientras que el **DHT22** se mantiene en una zona abierta para asegurar una correcta circulación de aire. El **sensor ultrasónico HC-SR04** se ubica en el borde de la protoboard, orientado hacia el área de medición para evitar interferencias con otros componentes.

Los actuadores se agrupan según su tipo y alimentación. La tira **LED RGB** y los ventiladores se conectan al **MOSFET** de 4 canales alimentado por la fuente de 12 V, situado en el lateral derecho para mantener separados los cables de potencia de las señales de control. La **bomba de agua**, gestionada por un MOSFET independiente alimentado con 5 V, se coloca en el lado opuesto para evitar acoplamientos eléctricos. Los **servomotores**, alimentados por la fuente de 2.5 V, se sitúan en la parte izquierda del montaje, con sus cables organizados para evitar tensiones mecánicas.

La pantalla **OLED** se coloca en la parte frontal del montaje, conectada al bus **I2C** del **ESP32**. Su posición permite visualizar en tiempo real los valores de humedad, temperatura, nivel del depósito y estado del riego sin necesidad de acceder a la aplicación Android, facilitando la supervisión local durante las pruebas.

Todas las fuentes de alimentación externas convergen en un **punto común de masa** compartida con el **ESP32**, garantizando referencias estables y evitando reinicios o lecturas erráticas. La disposición final prioriza la **claridad visual**, la separación entre potencia y señal, y la facilidad de mantenimiento, permitiendo una **futura migración a una PCB** o caja estanca sin modificar la lógica del sistema.

7

Infraestructura de red y comunicación

7.1. Servidor MQTT en Linux (Proxmox)

El sistema utiliza un servidor MQTT desplegado dentro de una máquina virtual Linux alojada en **Proxmox VE**, una plataforma de virtualización basada en KVM/QEMU que permite gestionar máquinas virtuales y contenedores de forma centralizada. Proxmox ofrece ventajas clave para este proyecto: snapshots rápidos, copias de seguridad programadas, aislamiento entre servicios y la posibilidad de asignar recursos de forma dinámica según las necesidades del sistema.

La máquina virtual destinada al broker MQTT ejecuta una distribución Linux ligera, configurada específicamente para alojar el servicio **Mosquitto**, un broker MQTT ampliamente utilizado en entornos IoT por su estabilidad, bajo consumo de recursos y compatibilidad con dispositivos embebidos como el ESP32. El servicio se ejecuta como demonio en segundo plano y mantiene una comunicación continua con el microcontrolador a través de la red local.

La virtualización en Proxmox permite:

- **Aislar el broker MQTT** del resto de servicios del sistema.
- **Asignar recursos dedicados** (CPU, RAM y almacenamiento) para garantizar estabilidad.
- **Realizar snapshots** antes de cambios críticos, facilitando la recuperación ante fallos.
- **Gestionar copias de seguridad automáticas**, evitando pérdida de datos.
- **Monitorizar el rendimiento** mediante la interfaz web de Proxmox.

El ESP32 publica y recibe mensajes a través de la red local, utilizando tópicos organizados por función (sensores, actuadores, órdenes y estados). Esta

arquitectura permite escalar el sistema fácilmente añadiendo nuevos nodos o servicios sin modificar la infraestructura existente.

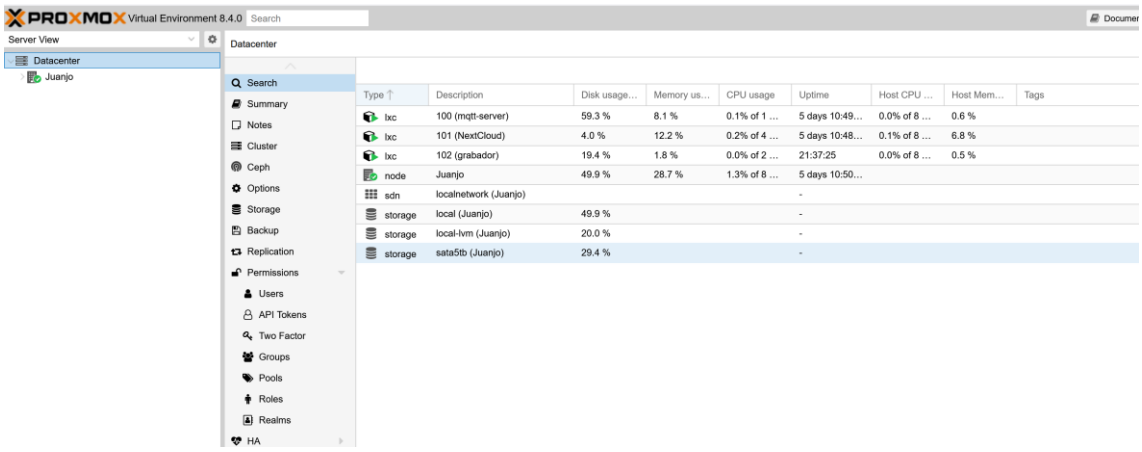

Fig. 15 Captura de máquinas Virtuales usadas en el Proyecto.

7.2. Configuración DNS para acceso desde externo

Para permitir el acceso remoto al sistema sin depender de la IP pública del proveedor, se ha configurado un servicio de DNS dinámico mediante **DuckDNS**, una plataforma gratuita que actualiza automáticamente la dirección IP asociada a un dominio personalizado. Esto permite acceder al servidor MQTT y a otros servicios del sistema desde cualquier ubicación utilizando un nombre de dominio estable.
La configuración se basa en los siguientes elementos:

- **Dominio DuckDNS** asignado al servidor.
- **Script de actualización automática** ejecutado cada hora mediante cron, garantizando que el dominio siempre apunte a la IP pública actual.
- **Apertura de puertos en el router**, redirigiendo las conexiones entrantes hacia la máquina virtual que ejecuta el broker MQTT.
- **Reglas de firewall** para limitar el acceso únicamente a los servicios necesarios.
- **Certificados opcionales de Let's Encrypt**, integrables con DuckDNS si en el futuro se desea habilitar cifrado TLS.

Esta configuración permite:

- Supervisar el sistema desde el exterior.
- Acceder al panel de **Proxmox** o a la cámara IP si se habilitan los puertos correspondientes.
- Integrar aplicaciones móviles o dashboards externos sin necesidad de **VPN**.
- Mantener una infraestructura sencilla y económica sin depender de servicios de pago.

La combinación de **Proxmox**, **Mosquitto** y **DuckDNS** proporciona una arquitectura robusta, flexible y fácilmente ampliable, adecuada para un entorno **IoT** doméstico o de laboratorio.

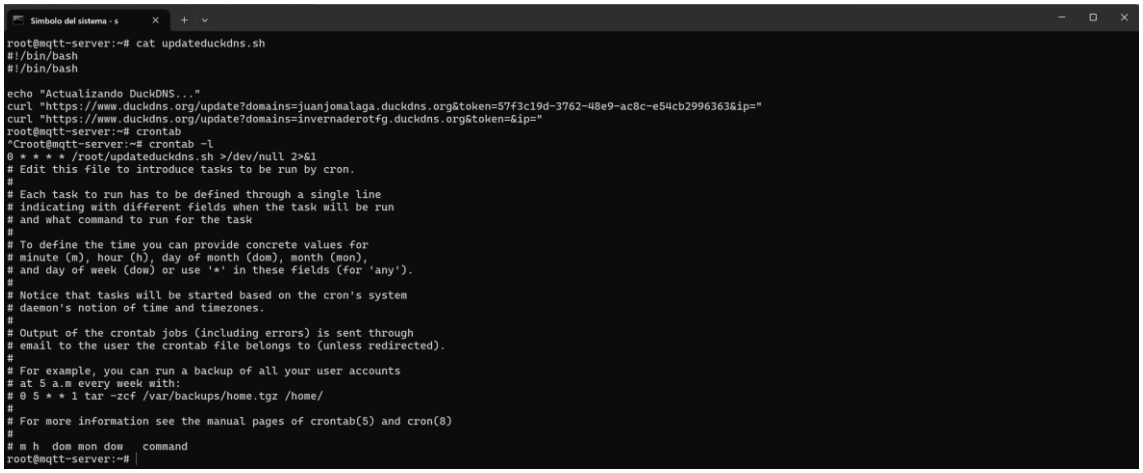

Fig. 16 Captura script updateduckdns.sh y archivo crontab

7.2.1. Servicio Cron

En los sistemas **Linux** existe un servicio llamado **cron** que se encarga de ejecutar tareas programadas automáticamente en segundo plano. Para indicarle qué tareas debe ejecutar y con qué frecuencia, Linux utiliza un archivo llamado crontab. Cada usuario del sistema tiene su propio archivo crontab, y además existe un crontab específico para el usuario **root**. El servicio cron revisa este archivo cada minuto y, cuando coincide la fecha y hora programada con la hora actual, ejecuta la tarea correspondiente. En la práctica, el **crontab** funciona como una agenda donde se definen comandos o scripts que deben ejecutarse de forma periódica sin intervención del usuario.

Para que el servidor actualizara automáticamente su IP pública en DuckDNS, se añadió una entrada al **crontab** del usuario root. Esto se hizo ejecutando el comando:
crontab -e
Dentro del archivo se añadió la siguiente línea:
0* * * * /root/updateduckdns.sh >/dev/null 2>&1
El significado de cada campo es el siguiente:

- 0 indica que la tarea se ejecuta en el minuto 0.
- El asterisco en la posición de la hora significa que se ejecuta cada hora.
- Los siguientes tres asteriscos indican que se ejecuta todos los días del mes, todos los meses y todos los días de la semana.
- A continuación se especifica la ruta del script que debe ejecutarse.
- La parte >/dev/null 2>&1 sirve para ocultar la salida del comando y evitar que cron genere mensajes o archivos de log innecesarios.

El script updateduckdns.sh contiene las llamadas necesarias para actualizar los dominios configurados en DuckDNS con la IP pública actual del servidor. De esta forma, aunque la IP cambie, los dominios siguen apuntando correctamente al servidor MQTT. El script se guardó en /root/updateduckdns.sh y se le dieron permisos de ejecución con chmod +x. Gracias a esta configuración, el servidor actualiza automáticamente su IP pública una vez cada hora sin necesidad de intervención manual.

7.3. Capturas desde cámara IP

Además de los sensores y actuadores conectados al microcontrolador, el sistema incorpora una **cámara IP** destinada a proporcionar supervisión visual del invernadero. Esta cámara está integrada en la red local y transmite vídeo mediante

el protocolo **RTSP (Real Time Streaming Protocol)**, lo que permite tanto la visualización en directo como la captura periódica de imágenes.
Para la parte automatizada, se ha desarrollado un script en **Bash** alojado en el servidor Linux dentro de Proxmox. Este script se conecta al flujo RTSP de la cámara, extrae un fotograma cada hora y lo almacena en el servidor con un nombre basado en la fecha y la hora. Las imágenes quedan organizadas en carpetas cronológicas, facilitando la revisión histórica del estado del invernadero y permitiendo detectar cambios físicos que no generan eventos eléctricos (acumulación de agua, caída de un sensor, obstrucciones, etc.).

Además de la captura automática, la cámara ofrece una **opción de visualización en directo desde la aplicación móvil**, permitiendo comprobar el estado del invernadero en tiempo real desde cualquier ubicación. Esta funcionalidad complementa la monitorización basada en sensores y proporciona una capa adicional de supervisión visual que resulta especialmente útil para validar el funcionamiento de los actuadores o detectar incidencias inesperadas.
La integración de la cámara con el servidor **Proxmox** permite centralizar tanto el almacenamiento como la gestión de los recursos, manteniendo un registro visual continuo sin afectar al rendimiento del resto de servicios del sistema.

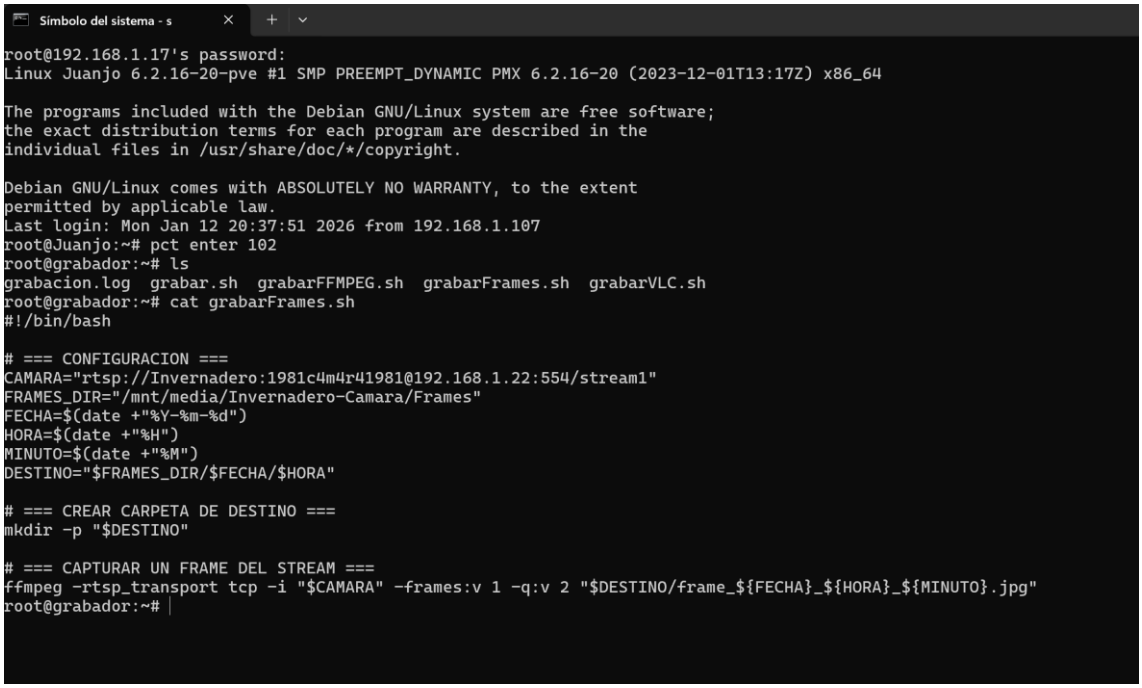

Fig. 17 Captura del Script Bash para hacer capturas

Además de la captura de fotogramas individuales, se desarrolló un segundo script destinado a grabar **vídeos** completos del invernadero. El objetivo inicial de este sistema era poder **monitorizar** las pruebas realizadas durante varios días y disponer de un registro visual continuo que permitiera revisar cualquier incidencia ocurrida en un momento concreto. La grabación de vídeo resulta especialmente útil para detectar problemas que no generan eventos eléctricos, como movimientos inesperados, fallos mecánicos, obstrucciones o comportamientos anómalos de los actuadores.

Dado que los archivos de vídeo ocupan un **espacio** considerable, se decidió conservar únicamente las grabaciones de los últimos tres días. Para ello, el script elimina automáticamente las carpetas con más de tres días de antigüedad, de modo que el almacenamiento se mantiene controlado sin intervención manual. Este

sistema permite disponer siempre de un historial reciente sin comprometer la capacidad del servidor.
La grabación de vídeo y la extracción de fotogramas requieren una cantidad significativa de recursos, especialmente cuando se utiliza **ffmpeg** para procesar el flujo **RTSP** de la cámara. Para evitar que esta carga afectara al rendimiento del servidor principal, se creó un contenedor independiente dentro de **Proxmox** dedicado exclusivamente a la tarea de grabación. De este modo, el procesamiento de vídeo queda aislado y no interfiere con el funcionamiento del broker **MQTT** ni con el resto de servicios del sistema.

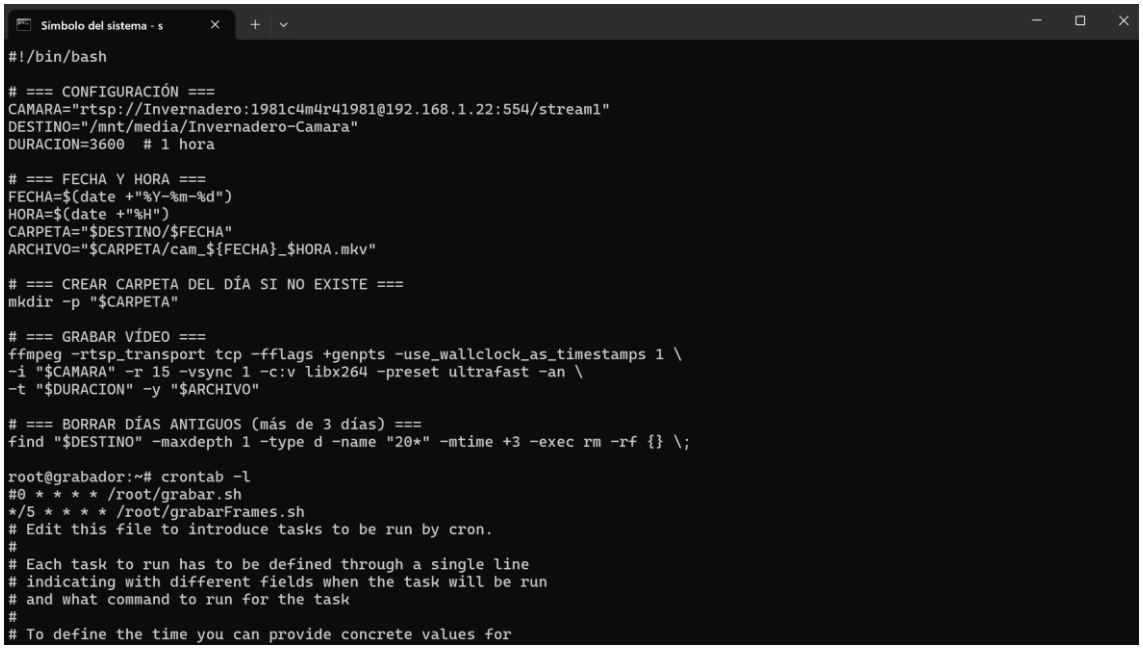

Fig. 18 Captura grabar.sh y crontab servidor grabador

Del mismo modo que ocurre con el script de actualización de **DuckDNS**, la captura automática del fotograma se ejecuta utilizando el servicio cron de Linux. Para ello, se añadió una entrada al archivo **crontab** del usuario root, que permite ejecutar el script de captura una vez cada hora sin intervención manual. La línea añadida al crontab es:
#0 * * * * /root/grabar.sh
***/5 * * * * /root/grabarFrames.sh**

En esta ocasión grabar.sh se ejecuta cada inicio de hora y grabarFrames.sh se ejecuta cada 5 minutos, pero en este caso se utiliza para garantizar que la primera ejecución ocurra en el minuto 5 de cada hora, cuando el script de vídeo ya ha terminado de iniciarse y no hay riesgo de solapamiento. De este modo, ambos **procesos** funcionan de forma **independiente** sin interferirse entre sí y sin provocar caídas de rendimiento en el contenedor encargado del procesamiento de vídeo.

8

Gestión de eventos críticos y respuesta ante fallos

8.1. Detección de eventos y activación de respuestas

El sistema está diseñado para reaccionar de forma inmediata y segura ante cualquier evento crítico, priorizando la protección del hardware y la coherencia del proceso. La lógica del ESP32 no se limita a ejecutar órdenes: incorpora un conjunto de **reglas de seguridad**, **validaciones continuas** y **mecanismos de bloqueo selectivo** que garantizan que ninguna acción pueda ejecutarse si las condiciones no son seguras.
Los eventos se clasifican en tres categorías principales:
1. Eventos sensoriales: lecturas de humedad del suelo, nivel del depósito de agua, temperatura, distancia, etc.
2. Eventos de comunicación: órdenes manuales enviadas desde la app o el servidor MQTT.
3. Eventos internos: fallos de lectura, valores fuera de rango, desconexiones o reinicios del microcontrolador.

A partir de estos eventos, el sistema aplica una lógica de control basada en restricciones **dinámicas**:

- **Protección del depósito de agua**

  Si el nivel del depósito de agua cae por debajo del **5 %**, el sistema **bloquea completamente la función de riego**, tanto manual como automática. Este bloqueo se mantiene **mientras persista la condición de fallo**.

  Si se inicia un riego manual de 60 s y en el segundo 25 el depósito de agua baja del 5 %, el sistema **interrumpe el riego inmediatamente**, desactiva la bomba y envía un aviso.

- **Protección ante fallos del sensor de humedad**
  Si el sensor de humedad devuelve valores erráticos, fuera de rango o deja de responder, el sistema **deshabilita el riego**.
  El bloqueo se mantiene hasta que el sensor vuelva a ofrecer lecturas válidas.
- **Protección ante fallos del sensor de temperatura**
  Si el sensor de temperatura/humedad aire falla o no responde, el sistema **bloquea la ventilación**, evitando que los ventiladores funcionen sin una referencia térmica fiable.
- **Bloqueo selectivo y adaptación dinámica**
  El sistema **no se bloquea por completo** cuando detecta un fallo. En su lugar, aplica un **bloqueo parcial**, manteniendo operativas las funciones que siguen siendo seguras:
  Si falla el sensor de humedad o el depósito de agua está bajo → **solo se permite ventilar**, nunca regar.
  Si falla el sensor de temperatura → **solo se permite regar**, nunca ventilar.
  Si ambos sensores funcionan correctamente → todas las funciones están disponibles.
  Esta lógica convierte al sistema en un entorno **adaptativo**, capaz de seguir operando incluso en condiciones degradadas sin comprometer la seguridad.
- **Reacción inmediata ante condiciones inseguras**
  El ESP32 evalúa continuamente las condiciones mientras un actuador está activo.
  Si surge un evento crítico durante una acción en curso, el sistema:
    - Interrumpe la acción inmediatamente
    - Actualiza el estado interno
    - Bloquea la función afectada
    - Envía un aviso al servidor
- **Funcionamiento autónomo incluso sin red**
  Si se pierde la conexión con el servidor MQTT, el ESP32 mantiene toda la lógica local activa, incluyendo:
    - Reglas de seguridad
    - Bloqueos selectivos
    - Validación de sensores
    - Interrupción de acciones peligrosas
    - Cuando la conexión vuelve, sincroniza estados y eventos pendientes.

8.2. Registro histórico y trazabilidad

El sistema incorpora un mecanismo de registro histórico diseñado para garantizar la trazabilidad completa de todos los eventos relevantes, tanto normales como críticos. Este registro permite reconstruir el comportamiento del sistema en cualquier momento, analizar fallos, validar decisiones automáticas y verificar que las reglas de seguridad se han aplicado correctamente.

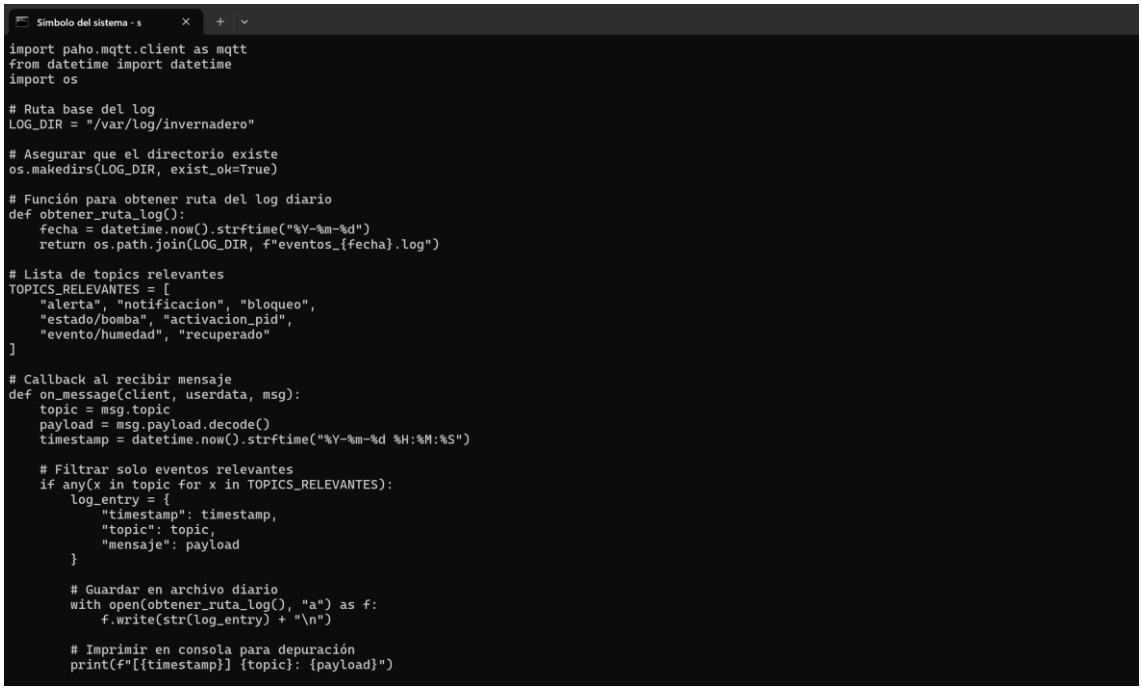

Fig. 19 Captura de Script en Python

El registro se gestiona desde un script en **Python** alojado en el servidor **Linux** dentro de **Proxmox**. Este script recibe los eventos publicados por el **ESP32** a través de **MQTT** y los almacena en un archivo estructurado por fecha. Cada entrada incluye información detallada sobre el evento, el estado del sistema y las condiciones que lo provocaron.

Los eventos registrados incluyen:

- Lecturas sensoriales válidas (humedad, temperatura, nivel del depósito de agua, distancia).
- Activaciones de actuadores (riego, ventilación, servos, iluminación).
- Cambios de estado del sistema (modo seguro, bloqueo parcial, desbloqueo).
- Fallos detectados en sensores o actuadores.
- Interrupciones automáticas por condiciones inseguras.
- Reconexiones del **ESP32** al servidor **MQTT**.
- Avisos generados por eventos críticos.

El sistema registra no solo lo que ocurre, sino **por qué ocurre**. Por ejemplo, si el riego se interrumpe en el segundo 25 debido a que el depósito de agua baja del 5 %, el registro incluye:

- Hora exacta
- Nivel del depósito de agua antes y después
- Acción que estaba en curso
- Motivo de la interrupción
- Estado final del sistema (bloqueo de riego activado)

Esta trazabilidad permite validar que las reglas de seguridad funcionan correctamente y facilita la detección de patrones anómalos, como lecturas erráticas repetidas o fallos intermitentes de sensores.

Además, el sistema mantiene un registro de los **bloqueos selectivos** aplicados. Si un sensor falla, el log refleja:

- Tipo de fallo
- Función bloqueada (riego o ventilación)
- Duración del bloqueo
- Momento exacto en que el sistema vuelve a habilitar la función

El registro histórico se convierte así en una herramienta esencial para la supervisión, el mantenimiento y la mejora continua del sistema, permitiendo analizar su comportamiento real bajo condiciones normales y de fallo.

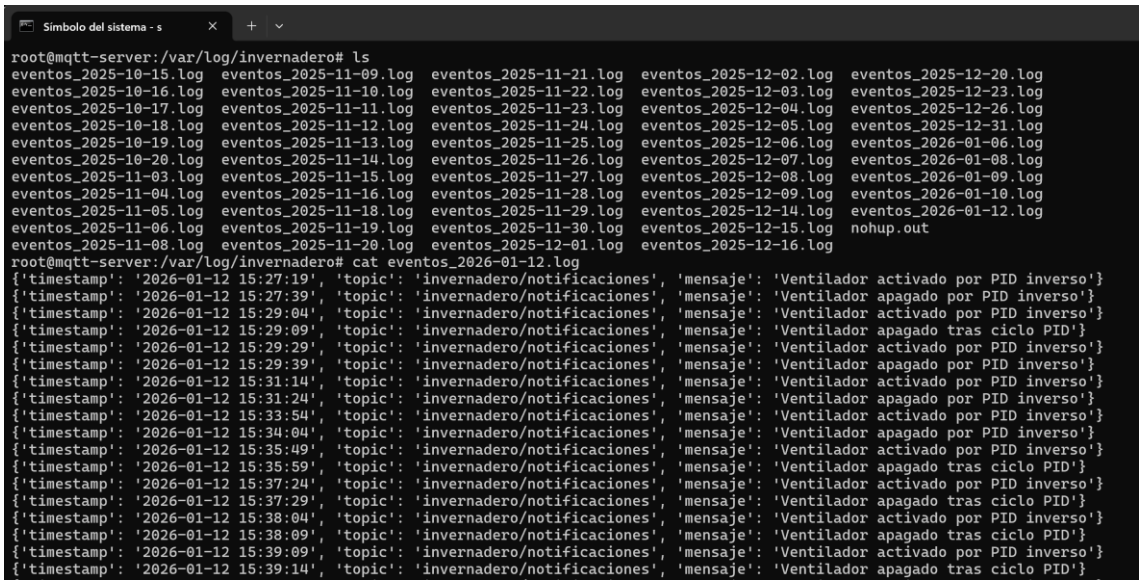

Fig. 20 Captura de carpeta con logs y muestreo de un log.

8.3. Reconexión segura y sincronización visual

El sistema está preparado para mantener su funcionamiento incluso ante pérdidas temporales de conectividad o reinicios inesperados del microcontrolador. Para ello, el firmware del ESP32 implementa un mecanismo de **reconexión segura**, diseñado para restaurar el estado del sistema sin provocar activaciones indeseadas ni inconsistencias entre la lógica interna y la interfaz visual.

Cuando se pierde la conexión con el servidor MQTT, el ESP32 entra en un modo de operación autónoma en el que mantiene activas todas las reglas de seguridad y los bloqueos selectivos definidos previamente. Durante este periodo, el sistema sigue leyendo sensores, aplicando restricciones y reaccionando ante eventos críticos, garantizando que la seguridad no depende de la conectividad externa.

Una vez restablecida la conexión, el microcontrolador ejecuta un proceso de **sincronización de estados**. Este proceso incluye:

- Publicación inmediata del estado actual de todos los sensores.
- Envío del estado de cada actuador (bomba, ventilación, servos, iluminación).
- Comunicación del estado de bloqueo activo (riego bloqueado, ventilación bloqueada, etc.).
- Actualización de la interfaz visual en la aplicación y en la tira LED RGB.

La sincronización visual es un componente clave del sistema. La tira LED RGB actúa como indicador del estado global, mostrando colores específicos para condiciones normales, alertas o fallos. Tras una reconexión, el **ESP32** actualiza automáticamente el color de la tira LED para reflejar el estado real del sistema, evitando discrepancias entre lo que ocurre físicamente y lo que percibe el usuario.

Además, el sistema evita activaciones automáticas durante la fase de reconexión. Si el servidor **MQTT** envía órdenes pendientes o mensajes retenidos, el **ESP32** valida primero las condiciones de seguridad antes de ejecutar cualquier acción. Esto garantiza que, tras un reinicio o una caída de red, el sistema no active la bomba o los ventiladores sin comprobar previamente el estado de los sensores y los bloqueos activos.

Este mecanismo de reconexión segura y sincronización visual asegura que el sistema mantenga coherencia, estabilidad y seguridad incluso en escenarios de fallo, proporcionando una experiencia de control fiable tanto en local como en remoto.

9

Programación del firmware Arduino

9.1. Arquitectura general del Firmware

El firmware del sistema se ha desarrollado siguiendo una estructura modular que permite separar claramente las funciones de sensorización, control, comunicación y visualización. Esta organización facilita el mantenimiento del código, mejora la legibilidad y permite ampliar o sustituir componentes sin afectar al resto del sistema.
El proyecto se divide en varios ficheros .ino y un fichero de cabecera .h, cada uno encargado de un bloque funcional específico:

- **00_main.ino**: contiene el flujo principal del programa, incluyendo la inicialización de módulos y el ciclo loop() donde se ejecutan las tareas periódicas.
- **01_Config.ino**: define pines, constantes físicas, variables globales y los topics MQTT utilizados por el sistema.
- **02_MQTT.ino**: implementa la conexión WiFi, la comunicación MQTT, la suscripción a comandos y la publicación de datos.
- **03_Eventos.ino**: centraliza la gestión de alertas y notificaciones, tanto visuales como enviadas por MQTT.
- **04_Sensores.ino**: gestiona la lectura de los sensores (humedad del suelo, temperatura y humedad ambiental, nivel del depósito) y la validación de su estado.
- **05_Actuadores.ino**: controla la bomba de riego, el ventilador y los servomotores de apertura de la tapa.
- **06_Oled.ino**: gestiona la pantalla OLED y la visualización de información del sistema.
- **07_ControladorPID.ino**: contiene la lógica de control automático mediante PID para riego y ventilación.
- **08_OTA.ino**: implementa la actualización remota del firmware mediante OTA y la comprobación automática de nuevas versiones.
- **09_ControladorLEDs.ino**: gestiona el LED RGB como indicador visual del estado del sistema.

- **PIDControl.h**: define la estructura del controlador PID utilizada para el control proporcional del riego y la ventilación.

Aunque Arduino unifica todos los ficheros .ino en un único programa durante la compilación, **el orden en el que los procesa depende del nombre del archivo**. Por este motivo, se ha adoptado una **numeración explícita** en los nombres, garantizando que las declaraciones y variables globales estén disponibles antes de que otros módulos las utilicen.

Esta numeración asegura que:

- Las **constantes, pines y variables globales** definidas en 01_Config.ino estén disponibles para todos los módulos posteriores.
- Las funciones de **MQTT**, **eventos**, **sensores** y **actuadores** se procesen en un orden lógico y sin dependencias rotas.
- Los controladores **PID**, la lógica de **OTA** y el sistema de **indicadores LED** se compilen después de que todos los elementos que utilizan estén definidos.
- El archivo PIDControl.h pueda incluirse desde cualquier módulo sin generar errores de compilación.

El flujo general del firmware sigue el patrón clásico de Arduino:

1. **setup()** Se inicializan todos los módulos: sensores, actuadores, pantalla OLED, red WiFi, MQTT, controladores PID y sistema OTA.
2. **loop()** Se ejecutan de forma periódica las siguientes tareas:
    - Mantenimiento de la conexión WiFi y MQTT.
    - Gestión de actualizaciones OTA.
    - Lectura de sensores cada 5 segundos.
    - Publicación de datos por MQTT.
    - Actualización del PID de humedad y temperatura.
    - Activación proporcional de bomba y ventilador.
    - Supervisión de seguridad durante el riego.
    - Actualización del estado visual mediante el LED RGB.
    - Refresco de la pantalla OLED.

Esta **arquitectura modular** permite que cada componente del sistema funcione de manera independiente, pero coordinada, garantizando un comportamiento estable y seguro incluso ante fallos de sensores o pérdidas de conexión.

9.2. Inicialización del Sistema

La fase de inicialización es fundamental para garantizar que todos los componentes del sistema —sensores, actuadores, comunicaciones y controladores— se encuentren en un estado estable antes de comenzar el funcionamiento normal. Esta inicialización se ejecuta íntegramente dentro de la función setup() del archivo 00_main.ino, y se apoya en funciones distribuidas en los distintos módulos del firmware.
El proceso de arranque sigue un orden lógico que asegura que cada subsistema esté operativo antes de que otro dependa de él:

9.2.1. Inicialización de actuadores

Se configuran los servomotores, la bomba de riego, el ventilador y el LED RGB. Esto incluye:

- Configuración de pines como salida.
- Posicionamiento inicial de los servos.

- Desactivación inicial de bomba y ventilador.
- Configuración de los canales PWM para el LED RGB.

Esta fase garantiza que ningún actuador quede activado accidentalmente durante el arranque.

9.2.2. Inicialización de sensores

Se configuran los pines de entrada y se inicializan los módulos de lectura:

- Sensores de humedad del suelo (dos sondas analógicas).
- Sensor DHT22 para temperatura y humedad ambiental.
- Sensor ultrasónico para nivel del depósito.

Además, se inicializan variables internas de suavizado y estado de sensores.

9.2.3. Inicialización de la pantalla OLED

El módulo OLED se inicializa mediante la librería **Adafruit_SSD1306**, estableciendo:

- Dirección I2C del dispositivo.
- Resolución de pantalla.
- Buffer de dibujo interno.

La pantalla queda lista para mostrar información del sistema desde el primer ciclo.

9.2.4. Inicialización de la red WiFi

El sistema se conecta a la red WiFi configurada en 01_Config.ino. La conexión se realiza en modo estación (WIFI_STA) y se mantiene automáticamente durante la ejecución mediante la función **mantenerConexiones**().

9.2.5. Inicialización de MQTT

Una vez establecida la conexión WiFi, se configura el cliente MQTT:

- Dirección del broker.
- Puerto.
- Callback para recepción de mensajes.
- Suscripción a los topics de control.

También se publican mensajes iniciales para limpiar estados previos en el broker.

9.2.6. Inicialización de controladores PID

Los controladores **PID** de temperatura y humedad del suelo se configuran con sus constantes:

- **Kp**, **Ki** y **Kd** para temperatura.
- **Kp**, **Ki** y **Kd** para humedad del suelo.

Esto permite que el sistema comience a calcular tiempos de riego y ventilación desde el primer ciclo de lectura.

9.2.7. Inicialización del sistema OTA

Finalmente, se configura el módulo OTA para permitir actualizaciones remotas del firmware:

- Nombre del dispositivo.
- Callbacks de progreso y errores.
- Inicio del servicio OTA.

Esto permite actualizar el firmware sin necesidad de conectar físicamente el ESP32.

En conjunto, esta fase de inicialización garantiza que el sistema arranque en un estado seguro, estable y completamente operativo, dejando preparado el entorno para el ciclo principal de ejecución.

9.3. Adquisición de datos de sensores

El sistema integra varios sensores que permiten monitorizar en tiempo real las variables críticas del invernadero: humedad del suelo, temperatura y humedad ambiental, y nivel del depósito de agua. La lectura de estos sensores se realiza de forma periódica y controlada desde el archivo 04_Sensores.ino, y los valores obtenidos se publican por MQTT para su visualización y registro externo.
La adquisición de datos sigue un ciclo de muestreo de **5 segundos**, definido por la constante INTERVALO_SENSORES. Este intervalo permite obtener datos suficientemente actualizados sin sobrecargar el microcontrolador ni la red MQTT.

9.3.1. Lectura de humedad del suelo

El sistema utiliza **sensor de humedad capacitivo** para medir la humedad del sustrato. La lectura se realiza mediante analogRead() y se aplican varias etapas de procesamiento:

- **Normalización**: los valores crudos se convierten a un porcentaje utilizando los valores calibrados de suelo seco y suelo mojado.
- **Limitación**: los valores se ajustan al rango 0–100%.
- **Suavizado exponencial**: se aplica un filtro de tipo EMA (Exponential Moving Average) para reducir ruido eléctrico y variaciones bruscas.
- **Detección de fallos**: si una lectura cae fuera del rango físico esperado, el sensor se marca como no operativo.

El valor final utilizado por el sistema es el de la sonda principal suavizada, que alimenta tanto la lógica de riego como el controlador PID de humedad.

9.3.2. Lectura de temperatura y humedad ambiental (DHT22)

El sensor **DHT22** proporciona la temperatura y la humedad relativa del aire. La lectura se realiza mediante la librería oficial DHT.h.
El firmware implementa:

- **Comprobación de validez**: si el sensor devuelve NaN, se marca como fallo.
- **Supervisión durante el riego**: un fallo del DHT puede bloquear la ventilación por seguridad.
- **Publicación periódica por MQTT**: temperatura en °C y humedad relativa en %.

Estos valores alimentan el controlador PID de temperatura y permiten activar el ventilador de forma proporcional.

9.3.3. Lectura del nivel del depósito (sensor ultrasónico)

El nivel del depósito se mide mediante un sensor ultrasónico (trigger/echo). El firmware:

- Envía un pulso de 10 µs al pin TRIG.
- Mide el tiempo de retorno en el pin ECHO.
- Calcula la distancia en centímetros.
- Convierte la distancia a un porcentaje de nivel entre depósito lleno y vacío.
- Detecta fallos si:
    - no se recibe eco,
    - la distancia es incoherente,
    - o el sensor queda fuera de rango.

Un fallo en este sensor **bloquea el riego** para evitar que la bomba funcione sin agua.

9.3.4. Validación del estado de los sensores

Cada sensor tiene un mecanismo de validación independiente:

- **Humedad del suelo**: rango físico + lecturas consecutivas válidas.
- **DHT22**: valores no nulos y dentro de un rango razonable.
- **Ultrasonido**: distancia válida y eco recibido.

Si un sensor crítico falla durante el riego, el sistema:

- detiene la bomba,
- genera una alerta,
- publica un mensaje de bloqueo por MQTT,
- cambia el estado visual del LED RGB.

9.3.5. Publicación de datos por MQTT

Cada ciclo de lectura publica los valores actualizados en los siguientes topics:

- **invernadero/suelo/humedad**
- **invernadero/aire/temperatura**
- **invernadero/aire/humedad**
- **invernadero/tanque/nivel**

Los mensajes se envían con retención (**retain = true**) para que cualquier cliente MQTT pueda obtener el último valor sin esperar a la siguiente lectura.

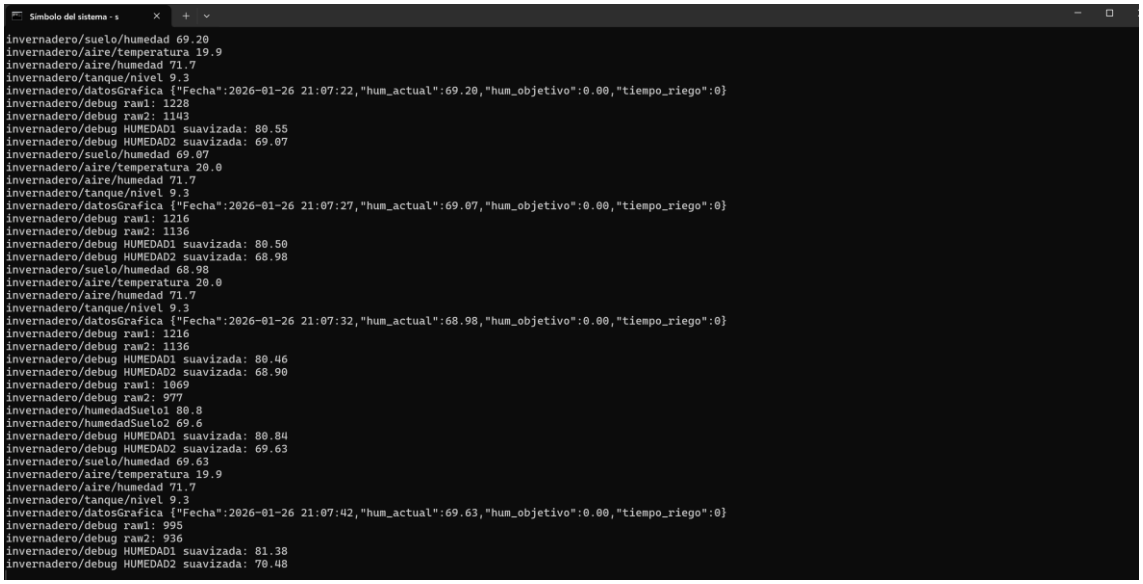

Fig. 21 Captura consola servidor mosquitto con mensajes en topic /invernadero

9.3.6. Integración con el ciclo principal

La lectura de sensores se ejecuta desde el **loop()** únicamente cuando ha transcurrido el intervalo definido. Esto evita bloqueos y permite que el resto del sistema (MQTT, OTA, PID, OLED) siga funcionando sin interrupciones.

9.4. Control de actuadores

El sistema dispone de varios actuadores encargados de ejecutar las acciones físicas del invernadero: la bomba de riego, los ventiladores, los servomotores de apertura de las tapas y la tira LED RGB que actúa como indicador visual del estado del sistema. Toda la lógica asociada a estos elementos se encuentra en los archivos 05_Actuadores.ino y 09_ControladorLEDS.ino.

El control de actuadores está diseñado bajo tres principios fundamentales:

- **Seguridad**: ningún actuador puede activarse si un sensor crítico está fallando.

- **Modularidad**: cada actuador tiene funciones independientes para encendido, apagado y supervisión.
- **Coherencia visual**: el LED RGB refleja en todo momento el estado operativo del sistema.

9.4.1. Control de la bomba de riego

La bomba se activa mediante un relé conectado al pin CH1_IN. El firmware implementa dos modos de funcionamiento:

Modo manual

El usuario puede activar o desactivar la bomba mediante comandos MQTT. Antes de permitir el riego manual, el sistema verifica:

- Estado del sensor de nivel del depósito.
- Estado del sensor de humedad del suelo.

Si alguno falla, el riego se bloquea y se genera una alerta.

Modo automático (PID)

En el modo automático, el controlador PID de humedad compara la humedad actual del suelo con la humedad objetivo configurada por el usuario. A partir de ese error, el firmware calcula un tiempo de riego proporcional.

El funcionamiento real es el siguiente:

- Evalúa el PID cada 5 segundos, obteniendo una salida proporcional al déficit de humedad.
- Convierte la salida del PID en milisegundos de riego mediante la función calcularTiempoPID(), aplicando límites mínimos y máximos.
- Activa la bomba únicamente si el tiempo calculado es mayor que cero, lo que indica que la humedad actual está por debajo del objetivo.
- Mantiene la bomba encendida durante exactamente ese tiempo, controlado por tInicioRiego y duracionRiego.
- Al finalizar el tiempo, la bomba se apaga y se inicia un pequeño periodo de enfriamiento para evitar activaciones demasiado seguidas.

9.4.2. Control del ventilador

El ventilador se controla mediante un relé conectado al pin FAN_CTRL_PIN. Su funcionamiento también puede ser:

Manual

El usuario puede encenderlo o apagarlo desde MQTT. Antes de activarlo, el sistema comprueba el estado del sensor DHT22.

Automático

El controlador PID de temperatura genera una salida negativa cuando la temperatura supera el objetivo. El firmware interpreta esta salida como:

- **Tiempo de ventilación proporcional** al exceso de temperatura.
- Activación del ventilador y apertura de la tapa mediante servos.
- Apagado automático cuando el tiempo calculado finaliza.

Si el DHT falla, la ventilación automática se bloquea por seguridad.

9.4.3. Control de servomotores (apertura de tapa)

El sistema utiliza dos servomotores para abrir o cerrar la tapa del invernadero. Su comportamiento está vinculado al ventilador:

- Cuando el ventilador se activa (manual o PID), los servos se posicionan en apertura.
- Cuando el ventilador se apaga, los servos vuelven a la posición de cierre.

Esto permite mejorar la circulación del aire durante la ventilación.

9.4.4. Control del LED RGB (indicador de estado)

El LED RGB se controla mediante PWM en tres canales independientes. El archivo 09_ControladorLEDS.ino define colores específicos para cada estado del sistema:

- **Verde** → funcionamiento normal
- **Azul** → riego activo
- **Amarillo** → ventilación activa
- **Cian** → riego + ventilación simultáneos
- **Rojo** → bloqueo por fallo en sensor de riego
- **Naranja** → bloqueo por fallo en sensor de ventilación
- **Magenta** → bloqueo total

El LED se actualiza automáticamente desde el loop() mediante la función actualizarEstadoVisual(), garantizando que el color mostrado siempre coincida con el estado real del sistema.

9.4.5. Priorización de seguridad

El firmware incorpora varias capas de protección:

- Si el nivel del depósito es bajo → **se bloquea el riego**.
- Si el sensor de humedad del suelo falla → **se bloquea el riego**.
- Si el DHT falla → **se bloquea la ventilación**.
- Si ambos fallan → **bloqueo total del sistema**.
- Si un actuador está en modo manual → el PID correspondiente se desactiva temporalmente.

Estas medidas garantizan que el sistema nunca ejecute acciones que puedan dañar los componentes o generar un comportamiento inesperado.

9.5. Publicación y manejo de topics MQTT

El sistema incorpora dos controladores PID independientes: uno para la humedad del suelo y otro para la temperatura del invernadero. Ambos se actualizan de forma periódica desde el ciclo principal y su salida se utiliza para activar la bomba de riego o el ventilador de manera proporcional al error respecto al objetivo configurado por el usuario.

Toda la lógica de control automático se encuentra en el archivo **07_Controlador-PID.ino**, mientras que el cálculo interno del PID está implementado en **PIDCon-trol.h.**

9.5.1. Actualización periódica del PID en el ciclo principal

Cada 5 segundos, tras la lectura de sensores, el firmware actualiza ambos controladores PID:

- **pidHum.actualizar(sueloPct, humedadObjetivo)**
- **pidTemp.actualizar(temperaturaActual, temperaturaObjetivo)**

La salida de cada PID se almacena en pidHum.output y pidTemp.output, respectivamente. Estas salidas no se saturan internamente, sino que se procesan después según la lógica de cada actuador.

9.5.2. PID de humedad del suelo: cálculo del tiempo de riego

La salida del PID de humedad representa la intensidad del déficit de humedad. El firmware convierte esta salida en un **tiempo de riego en milisegundos** mediante la función: **calcularTiempoPID(salidaPID, 5000).**

La conversión aplica reglas específicas:

- Si la salida es ≤ 0 → no se riega.
- Si la salida es pequeña → riego mínimo de 1 segundo.
- Si la salida es moderada → riego proporcional.
- Si la salida es alta → riego máximo permitido (5 segundos).

Esto permite un riego suave, progresivo y proporcional al error real.

9.5.3. Activación de la bomba según la salida del PID

Si el tiempo calculado es mayor que cero:

- Se activa la bomba.
- Se registra el instante de inicio (tInicioRiego).
- Se guarda la duración del ciclo (duracionRiego).
- Se publica el evento por MQTT.
- Se actualiza el estado visual (LED azul).

La bomba permanece encendida **exactamente el tiempo calculado**, controlado por: **if (millis() - tInicioRiego >= duracionRiego)**
Al finalizar:

- La bomba se apaga.
- Se publica el evento.
- Se inicia un pequeño cooldown para evitar activaciones demasiado seguidas.

9.5.4. Supervisión del riego durante el ciclo PID

Mientras la bomba está activa, el sistema supervisa continuamente:

- El nivel del depósito.
- El estado del sensor de humedad del suelo.

Si cualquiera de estos sensores falla:

- El riego se detiene inmediatamente.
- Se genera un bloqueo de seguridad.
- Se publica una alerta por MQTT.
- El LED cambia a rojo.

Esto evita daños en la bomba y garantiza un funcionamiento seguro.

9.5.5. PID de temperatura: control inverso del ventilador

El PID de temperatura funciona de forma inversa:

- Si la temperatura está por debajo del objetivo → salida positiva → no se ventila.
- Si la temperatura supera el objetivo → salida negativa → se ventila.

El firmware convierte la salida negativa en un **tiempo de ventilación**:

- Salidas pequeñas → ventilación breve.
- Salidas grandes → ventilación máxima (5 segundos).

Durante la ventilación:

- El ventilador se activa.
- Los servomotores abren la tapa.
- El LED cambia a amarillo.
- Se publica el evento por MQTT.

Al finalizar el tiempo calculado, el ventilador se apaga y la tapa se cierra.

9.5.5.1. Limitación física del sistema de ventilación

Es importante destacar que este PID **no controla un sistema de refrigeración activa**, sino un ventilador. Por tanto:

- El ventilador **no puede bajar la temperatura hasta el objetivo exacto**.
- Si el invernadero está a 50 °C y el objetivo es 20 °C, el sistema **no puede reducir 30 grados**, porque no dispone de un mecanismo de enfriamiento real.

- La ventilación solo permite **reducir algunos grados** mediante renovación del aire, pero no puede realizar un control térmico preciso como un aire acondicionado.

El PID se utiliza únicamente para **modular la intensidad de la ventilación**, no para garantizar que la temperatura alcance exactamente el setpoint. Su función es suavizar picos térmicos y mejorar la estabilidad del ambiente dentro de las limitaciones físicas del sistema.

9.5.6. Apertura automática de la tapa mediante servomotores

Los servos están vinculados al ventilador:
Si el ventilador se activa (manual o PID) → servos a 150° (tapa abierta).
Si el ventilador se apaga → servos a 0° (tapa cerrada).
Esto mejora la circulación del aire durante la ventilación automática.

9.6. Gestión de modos de operación

El sistema puede funcionar en dos modos principales: **modo manual** y **modo automático**. Ambos modos se gestionan desde el archivo 02_MQTT.ino, donde se reciben los comandos del usuario, y desde 07_ControladorPID.ino, donde se ejecuta la lógica automática. La coexistencia de ambos modos está diseñada para evitar conflictos entre órdenes manuales y decisiones automáticas del PID.

9.6.1. Modo manual

En el modo manual, el usuario controla directamente los actuadores mediante comandos MQTT. Las acciones disponibles son:

- Encender o apagar la bomba de riego.
- Encender o apagar el ventilador.
- Abrir o cerrar la tapa (a través del ventilador).

Antes de ejecutar cualquier acción manual, el firmware realiza comprobaciones de seguridad:

- Si el sensor de nivel del depósito falla → **no se permite activar la bomba**.
- Si el sensor de humedad del suelo falla → **no se permite activar la bomba**.
- Si el DHT22 falla → **no se permite activar el ventilador**.

Si se detecta un fallo, el sistema:

- Bloquea la acción.
- Publica una alerta por MQTT.
- Cambia el LED a un color de error.

Cuando un actuador está en modo manual, el PID correspondiente se **desactiva temporalmente** para evitar que el sistema intente tomar el control mientras el usuario lo está manipulando.

9.6.2. Modo automático (PID)

En el modo automático, el sistema gestiona el riego y la ventilación sin intervención del usuario. La lógica es la siguiente:

- El PID de humedad calcula un tiempo de riego proporcional al déficit de humedad, basándose en el dato configurado como objetivo por el usuario.
- El PID de temperatura calcula un tiempo de ventilación proporcional al exceso de temperatura.
- Los actuadores se activan únicamente si los sensores están en estado válido.
- El LED RGB refleja el estado actual (riego, ventilación, bloqueo, etc.).
- Se publican datos por MQTT para permitir la visualización en tiempo real.

Si el usuario envía un comando manual mientras el sistema está en automático:

- El sistema **cambia automáticamente a modo manual**.
- Se detiene cualquier riego o ventilación en curso.
- Se notifica el cambio de modo por MQTT.

Esto evita conflictos entre decisiones automáticas y órdenes del usuario.

9.6.3. Priorización de seguridad

Independientemente del modo seleccionado, el sistema aplica una serie de reglas de seguridad que tienen prioridad absoluta:

- **Fallo del sensor de nivel del depósito** → bloqueo del riego.
- **Fallo del sensor de humedad del suelo** → bloqueo del riego.
- **Fallo del DHT22** → bloqueo de la ventilación.
- **Fallo simultáneo de sensores críticos** → bloqueo total del sistema.

Estas reglas garantizan que el sistema nunca ejecute acciones que puedan dañar los componentes o generar comportamientos peligrosos.

9.6.4. Recuperación automática

Cuando un sensor vuelve a funcionar correctamente:

- El sistema elimina el bloqueo.
- El LED vuelve al color normal.
- Se publica un mensaje de recuperación por MQTT.
- El modo automático puede reanudarse sin intervención del usuario.

Esto permite que el sistema se mantenga operativo incluso ante fallos temporales.

9.7. Comunicación MQTT

La comunicación **MQTT** es el eje central que permite la interacción entre el invernadero y el exterior. A través de este protocolo, el sistema envía datos de sensores, notificaciones de estado, eventos de riego y ventilación, y recibe comandos del usuario para controlar los actuadores o modificar parámetros del sistema. Toda la lógica de comunicación se encuentra en el archivo **02_MQTT.ino**.

El firmware utiliza un cliente MQTT ligero y estable, configurado para reconectar automáticamente en caso de pérdida de conexión. Además, todos los mensajes importantes se publican con la opción **retain**, lo que permite que cualquier cliente obtenga el último estado sin esperar a la siguiente actualización.

9.7.1. Conexión al broker MQTT

Durante la inicialización, el sistema:

- Configura la dirección del broker.
- Establece el puerto de comunicación.
- Define un identificador único para el dispositivo.
- Registra la función callback que procesará los mensajes entrantes.

Si la conexión se pierde, el sistema intenta reconectar de forma automática sin bloquear el loop().

9.7.2. Topics suscritos (comandos entrantes)

El firmware se suscribe a varios topics que permiten controlar el sistema desde el exterior:

- invernadero/riego/manual → activar o desactivar la bomba.
- invernadero/ventilacion/manual → activar o desactivar el ventilador.
- invernadero/modo → cambiar entre modo manual y automático.
- invernadero/objetivos/humedad → actualizar la humedad objetivo.
- invernadero/objetivos/temperatura → actualizar la temperatura objetivo.
- invernadero/ota/actualizar → iniciar una actualización OTA.

Cada comando recibido se procesa en la función callbackMQTT(), que valida el mensaje, ejecuta la acción correspondiente y actualiza el estado del sistema.

9.7.3. Topics publicados (datos y eventos)

El sistema publica periódicamente los valores de los sensores y los eventos relevantes del funcionamiento:

Datos de sensores

- invernadero/suelo/humedad
- invernadero/aire/temperatura
- invernadero/aire/humedad
- invernadero/tanque/nivel

Eventos del sistema

- invernadero/riego/evento
- invernadero/ventilacion/evento
- invernadero/alertas
- invernadero/estado

Datos para gráficas

- invernadero/datosGrafica (mensaje JSON con humedad actual, objetivo y tiempo de riego calculado)

Todos los mensajes críticos se envían con **retain = true** para mantener el último estado disponible.

9.7.4. Formato de mensajes

Los mensajes simples (encendido/apagado, valores numéricos) se envían como texto plano. Los datos más complejos, como los utilizados para generar gráficas, se envían en formato **JSON**, por ejemplo:

{"Fecha":"2026-02-12 00:43:04","hum_actual":65.58,"hum_objetivo":66.00,"tiempo_riego":2716}

Este formato facilita la integración con plataformas como Node-RED, Home Assistant o dashboards personalizados.

9.7.5. Integración con el resto del sistema

La comunicación **MQTT** está completamente integrada en el ciclo principal:

- El sistema mantiene la conexión sin bloquear el **loop**().
- Los comandos entrantes pueden cambiar el modo de operación.
- Los datos publicados permiten monitorizar el invernadero en tiempo real.
- Las alertas notifican fallos de sensores o bloqueos de seguridad.
- Los eventos de riego y ventilación permiten generar gráficas históricas.

Gracias a esta integración, el invernadero puede controlarse y supervisarse desde cualquier dispositivo conectado a la red.

9.8. Sistema de alertas y notificaciones

El sistema incorpora un mecanismo de alertas diseñado para detectar fallos en sensores, bloqueos de seguridad y eventos relevantes del funcionamiento del invernadero. Estas alertas se gestionan desde el archivo 03_Eventos.ino y se integran con el LED RGB, la comunicación MQTT y la pantalla OLED.

El objetivo principal es garantizar que el usuario conozca en todo momento el estado del sistema y pueda actuar rápidamente ante cualquier incidencia.

9.8.1. Tipos de Eventos

El firmware distingue entre dos categorías principales:

9.8.1.1. Alertas críticas

Requieren intervención inmediata y provocan bloqueos de seguridad:

- Fallo del sensor de nivel del depósito.
- Fallo del sensor de humedad del suelo.
- Fallo del sensor DHT22.

- Fallo simultáneo de sensores críticos.
- Riego o ventilación interrumpidos por seguridad.

Estas alertas detienen automáticamente los actuadores afectados y cambian el estado visual del LED a un color de error.

9.8.1.2. Notificaciones

No bloquean el sistema, pero notifican eventos relevantes:

- Inicio y fin de un ciclo de riego.
- Inicio y fin de un ciclo de ventilación.
- Cambio de modo (manual/automático).
- Recuperación de un sensor previamente fallado.

9.8.2. Gestión de alertas mediante MQTT

La gestión de alertas y notificaciones se centraliza en la función **gestionarEvento(tipo, mensaje),** ubicada en 03_Eventos.ino. Esta función recibe dos parámetros:

- tipo → "alerta" o "notificacion"
- mensaje → texto descriptivo del evento

En función del tipo recibido, el firmware publica el mensaje en un topic MQTT diferente:

- **Alertas críticas** → se publican en T_ALERTAS
- **Notificaciones informativas** → se publican en T_NOTIFICACIONES

El contenido publicado es **únicamente el texto del mensaje**. El mensaje se envía con la opción **retain = true**, de modo que cualquier cliente MQTT puede recuperar el último evento sin esperar a uno nuevo.
Ejemplo real según el firmware:

- gestionarEvento("alerta", "Fallo sensor nivel") → publica "Fallo sensor nivel" en T_ALERTAS
- gestionarEvento("notificacion", "Riego finalizado") → publica "Riego finalizado" en T_NOTIFICACIONES

Además de la publicación MQTT, la función actualiza la pantalla OLED mostrando:

- Un encabezado ("⚠ ALERTA ⚠" o "**i** Notificación")
- El mensaje recibido

9.8.3. Indicadores visuales mediante LED RGB

El LED RGB actúa como un indicador de estado en tiempo real. Cada alerta o evento cambia el color del LED según la situación:

- **Rojo** → fallo en sensor de riego.
- **Naranja** → fallo en sensor de ventilación.
- **Magenta** → bloqueo total.
- **Azul** → riego activo.
- **Amarillo** → ventilación activa.
- **Verde** → funcionamiento normal.

El LED se actualiza desde la función actualizarEstadoVisual(), que se ejecuta en cada ciclo del loop().

9.8.4. Notificaciones en pantalla OLED

La pantalla OLED muestra información relevante del sistema, incluyendo:

- Estado de conexión WiFi y MQTT.

- Humedad del suelo y objetivo.
- Temperatura y objetivo.
- Nivel del depósito.
- Estado actual (normal, riego, ventilación, bloqueo).

En caso de alerta crítica, la pantalla muestra un mensaje destacado para facilitar la identificación del problema.

9.8.5. Recuperación automática

Cuando un sensor vuelve a funcionar correctamente:

- El sistema elimina el bloqueo.
- Se publica una alerta de recuperación.
- El LED vuelve al color normal.
- El modo automático puede reanudarse sin intervención del usuario.

Esto permite que el sistema se mantenga operativo incluso ante fallos temporales.

9.9. Interfaz visual en pantalla OLED

La pantalla OLED integrada en el sistema proporciona una visualización clara y en tiempo real del estado del invernadero. Su función es complementar la información enviada por MQTT, permitiendo al usuario comprobar rápidamente el estado del sistema sin necesidad de acceder a una aplicación externa. Toda la lógica de visualización se encuentra en el archivo 06_Oled.ino.
La pantalla se actualiza de forma periódica desde el loop(), mostrando datos de sensores, objetivos configurados y el estado actual del sistema (normal, riego, ventilación o alerta).

9.9.1. Inicialización de la pantalla

Durante el arranque, el firmware:

- Inicializa la librería Adafruit_SSD1306.
- Configura la resolución y la dirección I2C.
- Limpia el buffer interno.
- Muestra un mensaje inicial de bienvenida.

Una vez completada la inicialización, la pantalla queda lista para recibir actualizaciones.

9.9.2. Información mostrada en funcionamiento normal

En condiciones normales, la pantalla OLED muestra:

- **Humedad del suelo** (valor actual)
- **Humedad objetivo**
- **Temperatura ambiente**
- **Temperatura objetivo**
- **Nivel del depósito**
- **Estado del sistema** (Normal, Riego, Ventilación, Manual, Automático)

El diseño está optimizado para que toda la información relevante quepa en una sola vista, evitando menús o pantallas adicionales.

9.9.3. Visualización de alertas

Cuando se genera una alerta mediante gestionarEvento("alerta", mensaje):

- La pantalla se limpia completamente.
- Se muestra un encabezado destacado: **“ALERTA”**
- Debajo aparece el mensaje descriptivo del fallo.

Este mensaje permanece visible hasta que el sistema vuelve al estado normal o se genera una nueva notificación.

9.9.4. Visualización de notificaciones

Cuando se llama a gestionarEvento("notificacion", mensaje):

- La pantalla se limpia.
- Se muestra el encabezado **"Notificación"**.
- Debajo aparece el mensaje informativo (por ejemplo, "Riego finalizado").

Las notificaciones no bloquean el funcionamiento normal y se muestran de forma temporal.

9.9.5. Actualización periódica desde el ciclo principal

La pantalla se actualiza desde el loop() mediante una función dedicada (por ejemplo, actualizarPantalla()), que:

- Comprueba si hay una alerta activa (en cuyo caso no sobrescribe la pantalla).
- Muestra los valores actuales de sensores.
- Muestra los objetivos configurados.
- Indica el modo de operación (manual/automático).
- Indica si hay un riego o ventilación en curso.

Esto garantiza que la información visual esté siempre sincronizada con el estado real del sistema.

9.9.6. Integración con el sistema de eventos

La pantalla OLED está completamente integrada con el sistema de alertas:

- Las alertas tienen prioridad absoluta y sustituyen temporalmente la vista normal.
- Las notificaciones informativas también se muestran, pero no bloquean el sistema.
- Cuando la alerta se resuelve, la pantalla vuelve automáticamente al modo normal.

Esta integración permite que el usuario identifique rápidamente cualquier incidencia sin necesidad de revisar MQTT.

9.10. Actualización remota del firmware(OTA)

El sistema incorpora un mecanismo de actualización OTA (Over-The-Air) que permite instalar nuevas versiones del firmware sin necesidad de conectar físicamente el dispositivo. La actualización se realiza de forma **automática**, comprobando periódicamente si existe una versión más reciente disponible en un servidor remoto.
Toda la lógica se encuentra en el archivo 10_OTA.ino.

9.10.1. Inicialización del servicio OTA

Durante el arranque, el firmware configura el servicio OTA integrado de Arduino mediante la función **configurarOTA**(). En esta fase se definen:

- El nombre del dispositivo en la red.
- Los mensajes de depuración para inicio, progreso y fin de la actualización.
- El manejo de errores (autenticación, conexión, recepción, etc.).

Una vez configurado, el servicio queda activo y puede gestionar actualizaciones recibidas a través de la red local mediante ArduinoOTA.handle().

9.10.2. Servidor de versiones y ubicación del firmware

La comprobación de nuevas versiones no depende de servicios externos. El firmware consulta un **servidor privado**, ubicado en: **https://192.168.1.30/**
Este servidor (gestionado mediante Nextcloud) almacena los dos archivos necesarios para la actualización:

- **Version.txt:** Contiene la versión remota disponible. Ruta completa: https://192.168.1.30/index.php/s/ic2k8aePcapxdAn/download/Version.txt
- **firmware.bin:** Archivo binario con la nueva versión del firmware. Ruta completa: https://192.168.1.30/index.php/s/3G6FWaYwZCptrH9/download/firmware.bin

Al tratarse de una red local controlada, se utiliza **WiFiClientSecure** en modo **setInsecure(),** permitiendo conexiones HTTPS sin validación de certificados, lo cual es seguro en este contexto.

9.10.3. Comprobación automática de nuevas versiones

La función **compruebaVersion**() se ejecuta cada 24 horas (o la primera vez tras el arranque). El proceso es el siguiente:

1. Comprobación de la conexión WiFi Si no hay conexión, se publica "WiFi no conectado" por MQTT y se aborta la comprobación.
2. Descarga de Version.txt Se realiza una petición HTTPS al servidor privado. El código HTTP recibido se publica en invernadero/debug/http.
3. Comparación de versiones Si la respuesta es correcta (200 OK), se compara la versión remota con la versión actual (VERSION_FIRMWARE) mediante la función **esVersionSuperior**().
4. Decisión de actualización
    - Si la versión remota es superior → se inicia la descarga del firmware.
    - Si no lo es → se publica un mensaje indicando que no es necesario actualizar.

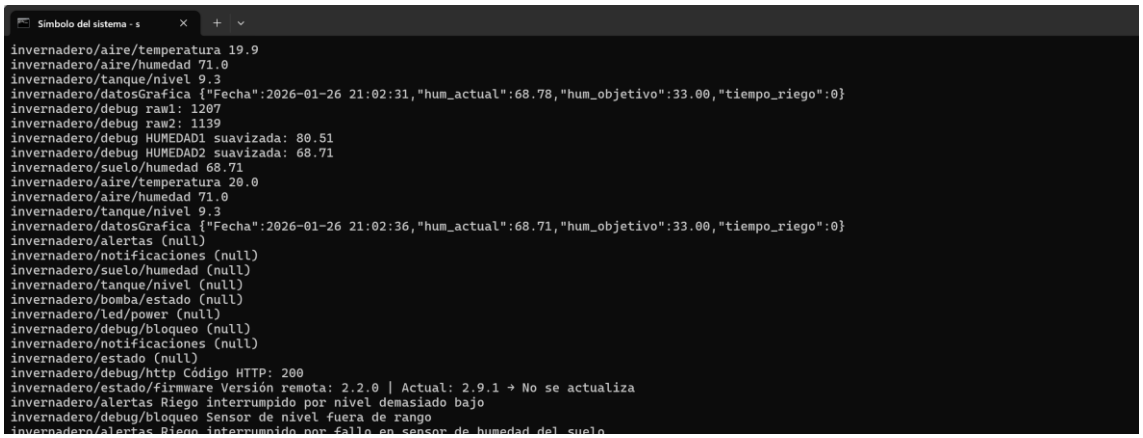

Fig. 22 Captura servidor mosquitto, donde se puede ver como la aplicación comprueba si hay nueva versión

9.10.4. Descarga y aplicación de la actualización

Si existe una versión más reciente:

- Se publica un mensaje indicando la disponibilidad de la nueva versión.
- Se descarga el archivo firmware.bin mediante httpUpdate.update().
- El firmware se escribe directamente en la memoria flash del microcontrolador.

Si la actualización finaliza correctamente:

- Se publica "Actualización OTA completada con éxito. Reiniciando...".
- El dispositivo se reinicia automáticamente.

Si ocurre un error:

- Se publica un mensaje con el código devuelto por httpUpdate.

9.10.5. Manejo de errores y depuración

Durante la comprobación y actualización, el sistema publica mensajes de diagnóstico en varios topics MQTT:

- invernadero/debug/http → códigos HTTP recibidos
- invernadero/estado/firmware → estado de la versión, errores y resultados de la OTA

Esto permite monitorizar el proceso desde cualquier cliente MQTT.

9.10.6. Integración con el ciclo principal

El sistema OTA se integra en el loop() de dos formas:

- gestionarOTA() → mantiene activo el servicio OTA estándar de Arduino.
- compruebaVersion() → ejecuta la comprobación automática cada 24 horas sin bloquear el programa.

Gracias a esta integración, el dispositivo puede actualizarse de forma autónoma y segura, manteniéndose siempre en la versión más reciente disponible en el servidor. Todo el proceso se ejecuta utilizando temporización basada en **millis(),** por lo que no interrumpe la ejecución del resto de funciones.

9.10.7. Ciclo principal del firmware

El ciclo principal **(loop())** coordina todos los módulos del sistema de forma no bloqueante. En cada iteración se mantienen las conexiones **MQTT**, se gestiona el servicio **OTA**, se supervisan los riegos en curso y se actualiza el estado visual. De forma periódica, según el intervalo configurado, se leen los sensores, se publican los datos, se actualiza la pantalla OLED, se ejecutan los **controladores PID** y se activan los actuadores correspondientes. Este diseño modular permite que el sistema funcione de manera estable y reactiva sin interrupciones.

10

Programación de la app Android

10.1. Arquitectura general de la aplicación

La aplicación Android se ha desarrollado en **Kotlin**, utilizando **Jetpack Compose** como framework de interfaz y el patrón **MVVM (Model–View–ViewModel)** para garantizar una separación clara entre la lógica de negocio, la gestión de datos y la presentación visual. Esta arquitectura permite una interfaz reactiva, modular y fácilmente extensible.
El punto de entrada principal es la clase ActividadPrincipal.kt, derivada de ComponentActivity. Sus responsabilidades incluyen:

- Solicitud del permiso de notificaciones en Android 13 o superior.
- Configuración visual global: modo oscuro forzado, paleta de colores personalizada y ajuste de las barras del sistema mediante SystemUiController.
- Creación del NavController para gestionar la navegación interna.
- Instanciación del VistaModeloMQTT, encargado de la comunicación con el firmware del invernadero.
- Inicialización del cliente MQTT dentro de un bloque LaunchedEffect, evitando bloqueos durante el arranque de la interfaz.

10.1.1. Organización modular del proyecto

La aplicación se estructura en paquetes funcionales, cada uno con responsabilidades bien definidas:

- **ui/** Contiene todas las pantallas y componentes visuales implementados con Compose:
  - **principal/** → Pantalla principal, control manual de actuadores, selector de color.
  - **historial/** → Visualización del historial de eventos y alertas.
  - **configuracion/** → Ajustes del servidor MQTT, parámetros del usuario y pantalla “About”.
  - **navegacion/** → Gestión del flujo entre pantallas mediante NavHost.

- **data/** Gestiona la persistencia local de eventos: Evento, EventoBD, EventoDAO, RepositorioEventos. Almacenamiento del historial y sincronización con los mensajes MQTT entrantes.
- **notificaciones/** Incluye el sistema de notificaciones locales, utilizado para mostrar alertas críticas enviadas por el firmware.

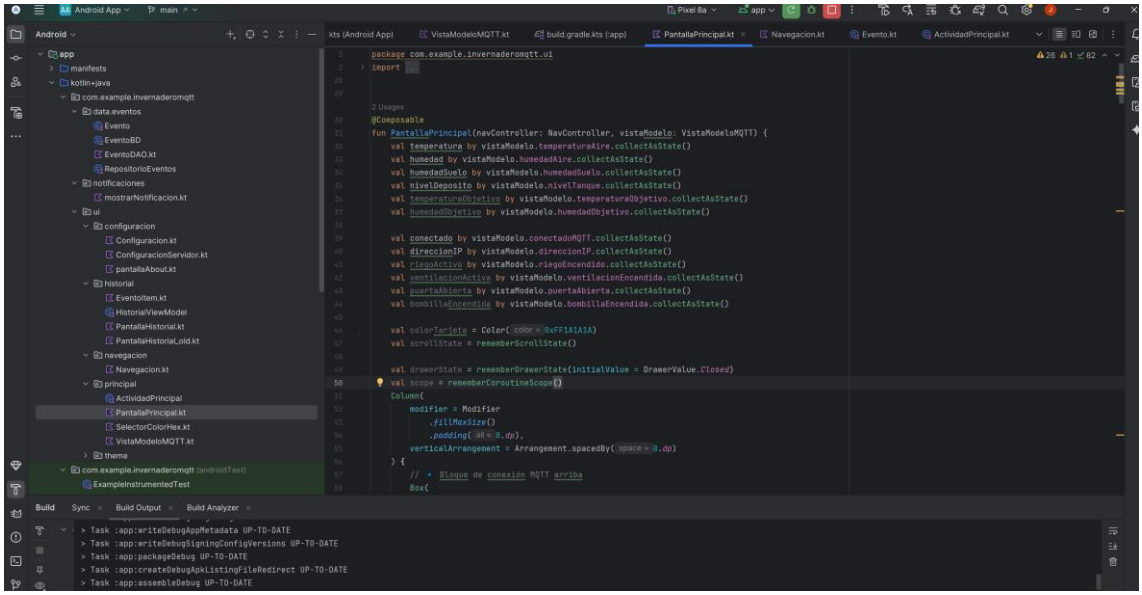

Fig. 23 Captura Android Studio con el proyecto

10.1.2. Navegación y estructura visual

El archivo Navegacion.kt implementa NavHost de Compose Navigation, definiendo todas las rutas internas de la aplicación:

- Pantalla principal
- Configuración del servidor
- Configuración PID
- Historial de eventos
- Pantalla "About"

Cada pantalla se implementa como un bloque @Composable, lo que permite aislar completamente la lógica de presentación y mantener una interfaz reactiva.
La aplicación incorpora elementos de navegación adicionales:

- **MenuGlobalTopBar** → menú superior desplegable para cambiar de pantalla.
- **MenuActuadores** → barra inferior para controlar bomba, ventilador, puerta o luz.

Ambos menús reflejan el estado del sistema mediante colores simbólicos sincronizados con los tópicos MQTT (verde, azul, rojo, amarillo…).

10.1.3. Estilo visual y coherencia estética

La interfaz utiliza un tema oscuro basado en MaterialTheme, con:

- Fondo negro uniforme
- Colores de acento personalizados
- Iconografía clara y minimalista
- Barras del sistema oscurecidas mediante SystemUiController

10.1.4. Modelo reactivo y actualización en tiempo real

Todos los componentes de la interfaz observan el estado expuesto por **VistaModeloMQTT**. Gracias a **Compose** y **MVVM**:

- La UI se actualiza automáticamente cuando llegan nuevos mensajes MQTT.

- Los actuadores reflejan su estado real sin intervención del usuario.
- Los eventos se almacenan y aparecen en el historial en tiempo real.
- Los colores e indicadores visuales cambian dinámicamente según el estado del sistema.

10.1.5. Ventajas de la arquitectura adoptada

La combinación de **Compose + Navigation + ViewModel** proporciona:

- Una interfaz flexible y fácilmente ampliable.
- Desacoplamiento entre lógica y presentación.
- Mantenimiento simplificado y código más limpio.
- Actualización dinámica basada en estados observables.
- Integración fluida con el firmware ESP32 mediante MQTT.

10.2. Gestión de datos y persistencia

La aplicación implementa un sistema de almacenamiento local para registrar los eventos, alertas y notificaciones generados durante el funcionamiento del invernadero. El objetivo es mantener un historial persistente —accesible incluso sin conexión— que refleje la actividad registrada por el firmware a través del protocolo MQTT.

10.2.1. Arquitectura de persistencia

El almacenamiento se desarrolla siguiendo el patrón **Repository**, que abstrae la capa de acceso a datos mediante la librería **Room** (Jetpack). Este enfoque garantiza una manipulación segura y eficiente de los datos, integrando operaciones de lectura y escritura con **corutinas** y **flujos reactivos (Flow)** para actualizar la interfaz en tiempo real.

La capa de persistencia se compone de cuatro elementos principales:

a) Entidad Evento

Archivo: Evento.kt

Define la estructura de los registros almacenados en la base de datos local. Cada evento contiene:

- **id** → identificador autogenerado
- **tipo** → categoría del evento (alerta, info, debug)
- **mensaje** → descripción del suceso
- **tópico** → tópico MQTT desde el que se originó
- **timestamp** → marca temporal en milisegundos (generada automáticamente)

Esta entidad mapea directamente la tabla eventos dentro de la base de datos.

b) Acceso a datos (EventoDao)

Archivo: EventoDao.kt

Define las operaciones permitidas sobre la tabla eventos, empleando anotaciones SQL de Room:

- insertar(evento) → inserta o reemplaza un registro
- obtenerUltimos(limite) → devuelve los eventos más recientes
- limpiarAntiguos(limiteTiempo) → elimina registros anteriores a una fecha
- obtenerTodos() → devuelve un Flow<List<Evento>> ordenado por fecha descendente

El uso de **Flow** permite que la UI se actualice automáticamente cuando se insertan o eliminan registros, sin necesidad de recargar manualmente la vista.

c) Base de datos Room (EventoBD)

Archivo: EventoBD.kt

Declara la base de datos local basada en Room y expone el DAO mediante eventoDao(). Implementa un patrón **singleton seguro** usando synchronized, garantizando que toda la aplicación utilice una única instancia.

La base de datos se crea con:

- nombre: eventos.db
- versión: 1

Esto facilita futuras migraciones si la aplicación crece.

d) Repositorio de eventos (RepositorioEventos)

Archivo: RepositorioEventos.kt

Actúa como puente entre el DAO y las capas superiores (ViewModel y UI). Proporciona una API clara y de alto nivel para registrar y consultar datos:

- registrarEvento() → inserta un nuevo evento con timestamp automático
- obtenerHistorial(limit) → recupera los últimos *n* eventos
- limpiarEventosAntiguos(dias) → elimina eventos con más de *n* días
- obtenerHistorial() → devuelve un Flow reactivo con el historial completo

Este diseño encapsula la lógica de almacenamiento y desacopla completamente la persistencia del resto de la aplicación.

10.2.2. Integración con la capa de presentación

El **VistaModeloMQTT** se comunica directamente con el **RepositorioEventos** para registrar cualquier evento recibido por MQTT (alertas, notificaciones o fallos). Cada inserción en la base de datos se propaga automáticamente a la interfaz mediante los Flow observados desde PantallaHistorial.kt, que muestra el historial en tiempo real sin necesidad de recargar la actividad.

Cuando el usuario abre la pantalla de historial:

- La UI observa los flujos expuestos por el repositorio
- Recibe una lista ordenada y actualizada de eventos
- Se actualiza automáticamente ante cualquier cambio

Además, el método **limpiarEventosAntiguos**() mantiene la base de datos ligera, eliminando entradas anteriores a 30 días.

10.2.3. Ventajas de esta implementación

- Persistencia local confiable basada en Room
- Actualización automática de la interfaz mediante Flow
- Baja carga de recursos, ideal para dispositivos móviles
- Escalabilidad gracias a la separación DAO–Entidad–Repositorio
- Sincronización directa con MQTT, garantizando que todos los eventos del firmware queden registrados en el histórico local

10.3. Interfaz de usuario (UI)

La interfaz de usuario de la aplicación se ha desarrollado íntegramente con **Jetpack Compose**, lo que permite construir pantallas reactivas, modernas y fáciles de mantener. La UI está diseñada para ofrecer una visualización clara del estado del invernadero y permitir al usuario interactuar con los actuadores de forma intuitiva. Todo el sistema se actualiza en tiempo real gracias a la integración directa con el

VistaModeloMQTT, que expone los valores recibidos desde el firmware mediante flujos observables.

10.3.1. Pantalla principal

La pantalla principal (PantallaPrincipal.kt) es el núcleo visual de la aplicación. Su objetivo es mostrar de forma clara y ordenada los valores más importantes del invernadero, así como el estado de los actuadores y la conexión MQTT.

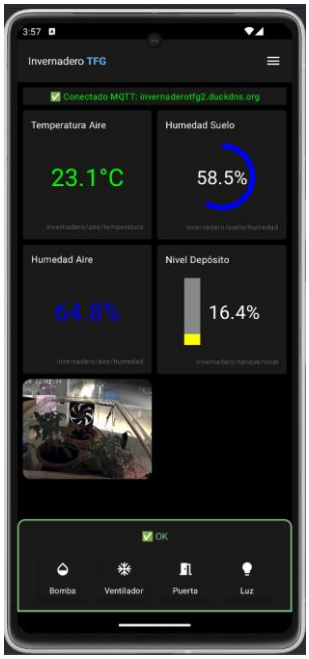

Fig. 24 Captura Pantalla Principal de la Aplicación Android

10.3.1.1. Indicador de conexión MQTT

En la parte superior se muestra un bloque que refleja el estado de la conexión con el broker MQTT:

- Si la conexión está activa, aparece un mensaje en verde indicando la IP del servidor.
- Si no hay conexión, el mensaje aparece en rojo y la tarjeta es clicable, permitiendo acceder directamente a la pantalla de configuración del servidor.

Este diseño facilita la detección rápida de problemas de red y evita que el usuario tenga que navegar por menús para corregir la configuración.

10.3.1.2. Visualización de sensores

La pantalla principal muestra los valores de los sensores en dos columnas:

- **Columna izquierda**
  - Temperatura del aire
  - Humedad del aire
  - Acceso directo a la cámara del invernadero (imagen clicable)
- **Columna derecha**
  - Humedad del suelo (indicador circular)
  - Nivel del depósito (barra vertical animada)

Cada tarjeta utiliza colores dinámicos para facilitar la interpretación:

- Temperaturas bajas → azul
- Valores óptimos → verde
- Valores altos → naranja o rojo
- Humedad baja → rojo
- Humedad media → amarillo

- Humedad alta → verde o azul

Estos colores se calculan mediante funciones auxiliares (obtenerColorTemperatura, obtenerColorAgua), lo que aporta coherencia visual en toda la app.

10.3.1.3. Estado de actuadores

La pantalla también refleja el estado actual de los actuadores:

- Riego activo
- Ventilación activa
- Puerta abierta
- Luz encendida

Estos valores se obtienen directamente del ViewModel mediante collectAsState(), lo que garantiza que la interfaz se actualice automáticamente cuando el firmware envía un cambio de estado.

10.3.1.4. Acceso a la cámara

La tarjeta con la imagen del invernadero actúa como acceso directo a la pantalla de cámara o stream RTSP. Esto permite al usuario comprobar visualmente el estado del cultivo sin salir de la aplicación.

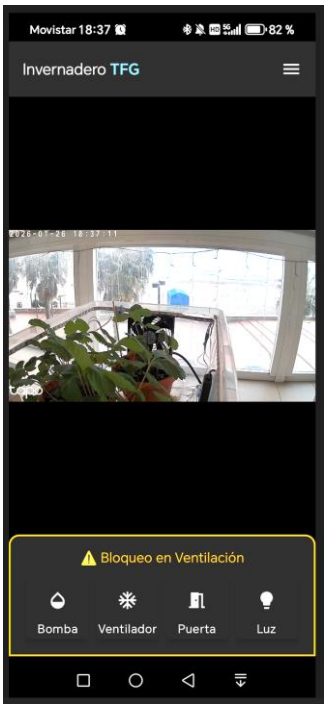

Fig. 25 Captura de Pantalla de Video

10.3.2. Componentes reutilizables

Para mantener un código limpio y modular, la pantalla principal utiliza varios componentes reutilizables:

- **TarjetaSensor**
  Muestra un valor numérico (temperatura, humedad del aire) con:
  - título
  - valor
  - color dinámico
  - tópico MQTT asociado
- **TarjetaCircular:** Indicador circular para la humedad del suelo, con animación y porcentaje centrado.
- **TarjetaDeposito:** Representación vertical del nivel del depósito, con barra que crece desde la parte inferior.

- **BotonControl:** Botón animado para actuadores, con icono, etiqueta y cambio de color cuando está activo.

Estos componentes permiten mantener un estilo visual uniforme y facilitan la ampliación futura de la app.

10.3.3. Estilo visual

La interfaz utiliza un diseño oscuro coherente con el resto del proyecto:

- Fondo negro
- Tarjetas gris oscuro
- Texto blanco
- Iconos Material Design
- Animaciones suaves en botones y tarjetas

El objetivo es ofrecer una interfaz moderna, clara y cómoda para su uso en entornos con poca luz, como un invernadero.

10.3.4. Reactividad y actualización en tiempo real

Gracias a **Jetpack Compose** y al uso de **StateFlow** en el **ViewModel**:

- La UI se actualiza automáticamente cuando llegan nuevos valores **MQTT**
- No es necesario refrescar manualmente la pantalla
- Los actuadores cambian de color en tiempo real
- Los sensores se actualizan sin retrasos perceptibles

Este comportamiento reactivo mejora la experiencia del usuario y garantiza que la información mostrada sea siempre la más reciente.

10.4. Comunicación MQTT

La comunicación entre la aplicación Android y el sistema embebido del invernadero se basa en el protocolo **MQTT**, elegido por su bajo consumo de recursos y su adecuación a sistemas **IoT**. Este protocolo utiliza un modelo de publicación y suscripción en el que los dispositivos intercambian información a través de un broker intermedio, sin necesidad de comunicación directa entre ellos.

En este proyecto, el ESP32 publica las lecturas de sensores y el estado del sistema, además de recibir órdenes de control. La aplicación Android actúa como cliente de supervisión y control remoto, mostrando la información en tiempo real y permitiendo al usuario interactuar con el invernadero.

10.4.1. Cliente MQTT en la aplicación Android

La aplicación implementa la comunicación MQTT mediante un cliente asíncrono basado en la librería **HiveMQ** MQTT Client (com.hivemq:hivemq-mqtt-client), configurado para trabajar con el protocolo MQTT v3.

Toda la lógica relacionada con MQTT se centraliza en la clase **VistaModeloMQTT**, lo que permite separar claramente la comunicación de la interfaz de usuario.
La conexión con el **broker Mosquitto** se inicializa al arrancar la aplicación, utilizando una dirección configurable por el usuario y el puerto estándar 1883. El cliente se configura con un identificador propio, un keep-alive de 30 segundos y una sesión persistente, lo que facilita la recuperación tras pérdidas temporales de conexión.

10.4.2. Gestión de la conexión y suscripciones

El proceso de conexión se realiza de forma asíncrona, evitando bloqueos en la interfaz. Cuando la conexión se establece correctamente, la aplicación se suscribe a los distintos tópicos relacionados con el funcionamiento del invernadero.

Entre estos tópicos se incluyen las lecturas de sensores, el estado de los actuadores, los parámetros de control automático y los canales destinados a alertas y notificaciones. La recepción de mensajes se gestiona mediante un listener global, lo que simplifica el tratamiento de los datos sin necesidad de implementar múltiples callbacks.

El estado de la conexión se expone a la interfaz mediante **StateFlow**, permitiendo mostrar en todo momento si la aplicación está conectada o no al sistema.

10.4.3. Procesamiento de mensajes recibidos

Cada mensaje **MQTT** recibido se procesa en función del tópico al que pertenece. El **ViewModel** interpreta el contenido del mensaje y actualiza las variables internas correspondientes.

Las lecturas de sensores actualizan directamente los valores de temperatura, humedad ambiental, humedad del suelo y nivel del depósito. De la misma forma, los mensajes relacionados con actuadores reflejan el estado real del riego, la ventilación, la puerta o la iluminación.

Los parámetros de control automático, como la temperatura y humedad objetivo o el tiempo máximo de riego, se actualizan dinámicamente sin necesidad de reiniciar el sistema.

Gracias al uso de **StateFlow** y **Jetpack Compose**, cualquier cambio recibido se refleja automáticamente en la interfaz de usuario.

10.4.4. Gestión de alertas y notificaciones

El sistema distingue entre **alertas** y **notificaciones** mediante tópicos específicos. Las alertas se utilizan para indicar situaciones críticas, como fallos de sensores o interrupciones de seguridad, mientras que las notificaciones informan de eventos relevantes o recuperaciones del sistema.

Cuando la aplicación recibe uno de estos mensajes, genera una notificación local para el usuario y registra el evento en una base de datos local mediante **Room**. Para evitar duplicados tras reconexiones, los mensajes **retained** en estos tópicos se ignoran deliberadamente.

10.4.5. Envío de comandos desde la aplicación

Las acciones del usuario en la aplicación se traducen en publicaciones **MQTT** hacia el **broker**. Estas publicaciones permiten controlar los distintos actuadores del invernadero, como el riego, la ventilación, la puerta o la iluminación LED.

Además, la aplicación permite modificar parámetros internos del sistema, como la humedad y temperatura objetivo para el controla automático o el tiempo máximo de riego manual, que actúa como mecanismo de seguridad para evitar activaciones prolongadas de la bomba.

Toda la lógica de publicación se encuentra centralizada en el **ViewModel**, manteniendo la interfaz desacoplada de los detalles de comunicación.

10.4.6. Supervisión de la comunicación

Para aumentar la robustez del sistema, el **ViewModel** implementa un mecanismo de supervisión basado en el tiempo transcurrido desde el último mensaje recibido. Si durante un intervalo prolongado no se reciben datos, la aplicación considera que la conexión se ha perdido y actualiza su estado interno.

Este comportamiento permite detectar fallos de red o caídas del **broker** y mostrarlos de forma clara al usuario.

10.4.7. Valoración de la solución adoptada

La **implementación** de **MQTT** en el proyecto proporciona una comunicación eficiente, flexible y fácilmente ampliable. La **separación** entre firmware y aplicación, junto con el uso de un cliente **asíncrono** y una interfaz reactiva, permite un control en tiempo real del invernadero y una experiencia de usuario fluida.

El diseño adoptado facilita la incorporación de nuevos sensores o actuadores, así como la modificación del comportamiento del sistema sin necesidad de cambios estructurales en la aplicación.

10.5. Sistema de notificaciones

El **sistema de notificaciones** tiene como objetivo informar al usuario de forma inmediata sobre eventos relevantes o situaciones críticas que se producen durante el funcionamiento del invernadero. Este mecanismo permite que el usuario esté al tanto del estado del sistema incluso cuando la aplicación no se encuentra en primer plano.
Las notificaciones se integran directamente con la comunicación MQTT, de modo que los mensajes enviados por el firmware del sistema embebido se traducen automáticamente en avisos visibles en el dispositivo móvil.

10.5.1. Tipos de notificaciones

El sistema distingue entre dos tipos principales de mensajes:

- **Alertas críticas**, asociadas a situaciones anómalas o de riesgo, como fallos de sensores, interrupciones de seguridad o condiciones que requieren atención inmediata.
  Estas alertas se muestran al usuario mediante un **mensaje emergente (popup)** que requiere una acción explícita de confirmación, garantizando que la información crítica no pase desapercibida.
- **Notificaciones informativas**, utilizadas para comunicar eventos relevantes del sistema, como recuperaciones tras un fallo o cambios de estado importantes.
  Este tipo de notificaciones se muestran de forma no intrusiva, generando un aviso sonoro y visual en la **barra de notificaciones de Android**, sin interrumpir el uso normal del dispositivo.

Fig. 26 Captura Notificación en system tray

Esta diferenciación permite priorizar la información mostrada al usuario y evitar la saturación de avisos innecesarios.

10.5.2. Origen de las notificaciones

Las notificaciones se generan a partir de mensajes **MQTT** publicados por el sistema embebido en los siguientes tópicos:

- **invernadero/alertas**
- **invernadero/notificaciones**

Cuando la aplicación Android recibe un mensaje en alguno de estos tópicos, el contenido se procesa en el **ViewModel** y se invoca al sistema de notificaciones local. De este modo, cualquier evento relevante detectado por el firmware se refleja inmediatamente en el dispositivo del usuario.

Para evitar la generación de avisos duplicados tras reconexiones, los mensajes **retained** en estos tópicos se ignoran deliberadamente.

10.5.3. Implementación en la aplicación Android

El sistema de notificaciones se implementa mediante una función específica que encapsula toda la lógica necesaria para mostrar avisos al usuario. Esta función se encarga de:

- Determinar si el mensaje corresponde a una alerta crítica o a una notificación informativa.
- Seleccionar el canal de notificación adecuado.
- Configurar la prioridad y el icono de la notificación.
- Mostrar el aviso de forma inmediata en el sistema operativo Android.

A partir de **Android 13**, la aplicación solicita explícitamente el permiso de notificaciones, garantizando el cumplimiento de los requisitos de seguridad y privacidad del sistema operativo.

10.5.4. Canales de notificación y prioridades

Para mejorar la organización y el control de los avisos, se utilizan canales de notificación diferenciados:

- Un canal específico para **alertas críticas**, configurado con alta prioridad.
- Un canal independiente para **notificaciones informativas**, con prioridad estándar.

Esta separación permite que el usuario gestione desde el sistema operativo el comportamiento de cada tipo de aviso (**sonido, vibración o silenciamiento**), manteniendo un equilibrio entre visibilidad y comodidad de uso.

10.5.5. Integración con el historial de eventos

Además de mostrarse como notificación local, cada alerta o notificación recibida se registra en una **base de datos** local mediante **Room**. Este registro permite mantener un historial persistente de eventos, accesible desde la aplicación incluso aunque el usuario no haya visto el aviso en el momento en que se produjo.

De esta forma, el sistema de notificaciones no solo cumple una función informativa inmediata, sino que también contribuye a la **trazabilidad** y al diagnóstico posterior del funcionamiento del invernadero.

10.5.6. Valoración del sistema de notificaciones

El sistema de notificaciones implementado proporciona una solución eficaz y poco intrusiva para informar al usuario sobre el estado del invernadero. Su integración directa con **MQTT** garantiza que los avisos reflejen fielmente los eventos detectados por el sistema embebido, mientras que el uso de canales y prioridades mejora la experiencia de usuario.

Esta solución permite ampliar fácilmente el número de eventos notificados y adaptar el comportamiento del sistema a futuras mejoras sin modificar la arquitectura general de la aplicación.

10.6. Historial de eventos

El historial de eventos permite registrar y consultar de forma persistente los sucesos relevantes que se producen durante el funcionamiento del invernadero. Su objetivo principal es ofrecer **trazabilidad**, facilitar el **diagnóstico de incidencias** y permitir al usuario revisar alertas o notificaciones pasadas, incluso si no fueron atendidas en el momento en que se generaron.
Este sistema complementa al mecanismo de notificaciones, proporcionando una visión histórica del comportamiento del sistema más allá de los avisos inmediatos.

10.6.1. Tipos de eventos registrados

El historial almacena distintos tipos de eventos generados a partir de la comunicación MQTT:

- **Alertas**, asociadas a situaciones críticas como fallos de sensores, interrupciones de seguridad o condiciones anómalas detectadas por el sistema embebido.
- **Eventos informativos**, relacionados con cambios de estado, recuperaciones tras un fallo o notificaciones relevantes del sistema.

Cada evento incluye información suficiente para su interpretación posterior, permitiendo distinguir claramente su origen y naturaleza.

10.6.2. Origen y captura de eventos

Los eventos registrados en el historial se generan a partir de los mensajes recibidos en los tópicos **MQTT** destinados a alertas y notificaciones. Cuando la aplicación Android recibe un mensaje válido en estos tópicos, el contenido se procesa en el **ViewModel** y se registra automáticamente en la base de datos local.
Este mecanismo garantiza que **todos los eventos relevantes enviados por el firmware queden almacenados**, independientemente de que el usuario tenga la aplicación abierta o haya visualizado la notificación en ese momento.

10.7. Persistencia local de los datos

El historial de eventos se almacena en el dispositivo móvil mediante una base de datos local implementada con **Room**, siguiendo el patrón de arquitectura

Repository. Cada evento se guarda junto con su tipo, mensaje descriptivo, tópico MQTT de origen y una marca temporal generada automáticamente.
El uso de almacenamiento local permite:

- Acceder al historial sin necesidad de conexión al broker MQTT.
- Mantener los datos tras cierres o reinicios de la aplicación.
- Reducir la carga de red y la dependencia de servicios externos.

Para evitar un crecimiento excesivo de la base de datos, el sistema incluye mecanismos de limpieza de eventos antiguos, manteniendo un equilibrio entre persistencia y uso eficiente de recursos.

10.7.1. Visualización del historial en la aplicación

La aplicación dispone de una pantalla específica para la visualización del historial de eventos. En ella se muestra una lista ordenada cronológicamente, donde el usuario puede consultar los eventos más recientes de forma clara e intuitiva.

La interfaz se actualiza automáticamente gracias al uso de flujos reactivos, de modo que cualquier nuevo evento registrado aparece de inmediato en la pantalla sin necesidad de recargar la vista. El diseño prioriza la legibilidad, diferenciando visualmente alertas y notificaciones para facilitar su identificación.

10.7.2. Relación con el sistema de notificaciones

El historial de eventos actúa como complemento del sistema de notificaciones. Mientras que las notificaciones informan al usuario de forma inmediata, el historial permite revisar posteriormente lo ocurrido y analizar la evolución del sistema a lo largo del tiempo.

Esta combinación mejora la fiabilidad del sistema desde el punto de vista del usuario, ya que garantiza que ninguna alerta importante se pierda, incluso si el aviso no fue atendido en el momento de su emisión.

10.7.3. Valoración del sistema de historial

El sistema de historial implementado aporta un valor añadido significativo al proyecto, proporcionando trazabilidad, persistencia y capacidad de análisis del funcionamiento del invernadero. Su integración con la comunicación MQTT y con la interfaz de usuario permite una gestión transparente de los eventos, manteniendo la aplicación ligera y fácil de usar.

La solución adoptada es fácilmente ampliable, permitiendo incorporar nuevos tipos de eventos o filtros de visualización sin modificar la arquitectura general del sistema.

10.8. Configuración del sistema

El sistema incorpora un apartado de configuración en la aplicación Android para ajustar parámetros clave sin necesidad de recompilar ni el firmware ni la app. Esta configuración se divide en dos bloques diferenciados: **conectividad (servidor MQTT)** y **parámetros de funcionamiento (parámetros objetivo y límites de control)**. Con este enfoque se separa lo “infraestructura/red” de lo “comportamiento del invernadero”, mejorando la claridad para el usuario.

10.8.1. Configuración del servidor MQTT

La pantalla **Configuración del Servidor MQTT** permite introducir y modificar la dirección del broker (IP o dominio). El valor se toma del estado expuesto por el ViewModel y puede cambiarse en tiempo de ejecución.
Tras pulsar "Añadir", la aplicación:

1. Actualiza la dirección del broker en el ViewModel.
2. Lanza un intento de conexión MQTT de forma asíncrona.
3. Muestra un resultado al usuario:
    - Si conecta, se informa con un diálogo de confirmación y se vuelve a la pantalla principal.
    - Si falla, se muestra un diálogo de error que permite **guardar igualmente la dirección** (útil cuando el broker está temporalmente inaccesible o se está configurando la red).

Este diseño evita bloquear el uso de la app y facilita cambios rápidos de entorno (por ejemplo, pruebas en red local frente a acceso remoto).
Es importante destacar que si el servidor esta desconectado podemos acceder a esta misma pantalla pulsando directamente sobre la dirección del servidor en la pantalla principal.

10.8.2. Configuración de parámetros del sistema

Además de la conectividad, la aplicación incluye una pantalla de **Configuración PID** (parámetros de control) desde la cual el usuario puede modificar la humedad y temperatura objetivo y límites operativo de riego manual, esto se hace así para evitar que se encienda el riego manual y por un olvido pueda ocasionar un problema en la instalación.
En esta pantalla se ajustan, mediante controles tipo *slider*:

- **Temperatura objetivo (°C)**: para el control automático de ventilación.
- **Humedad objetivo (%)**:asociada al control automático del riego.
- **Tiempo máximo de riego manual (segundos)**: límite de seguridad que evita que la bomba pueda permanecer activada de forma indefinida por error o por pérdida de supervisión.

Cada cambio se envía al sistema en el momento en que el usuario finaliza el ajuste (onValueChangeFinished). Tras el envío se muestra una confirmación visual (" Valor enviado"), reforzando la sensación de control y evitando dudas sobre si el comando se ha transmitido.

10.8.3. Control de iluminación LED y modo de funcionamiento

En la misma pantalla de configuración se incluye el control del sistema de iluminación LED RGB, con dos modos:

- **Modo usuario**: permite seleccionar un color (mediante un selector RGB y colores predefinidos) y enviarlo al sistema embebido.
- **Modo automático**: devuelve la iluminación a su comportamiento automático, mostrando un mensaje de confirmación al usuario.

Esta funcionalidad permite tanto una personalización directa (útil para pruebas o estados visuales manuales) como la recuperación rápida del modo automático sin necesidad de reiniciar el sistema.

10.8.4. Integración con la arquitectura reactiva

Todos los parámetros de configuración se gestionan desde el **VistaModeloMQTT**, que expone su estado mediante **StateFlow**. Esto permite:

- Mostrar en pantalla el valor actual de cada paráemtro.
- Reflejar cambios de forma inmediata en la UI.
- Mantener coherencia entre el estado visual y el estado real recibido por MQTT.

De este modo, la configuración se integra en el patrón **MVVM** de la aplicación sin acoplar la interfaz a la lógica de comunicación.

10.8.5. Ventajas de la configuración implementada

El sistema de configuración aporta:

- **Flexibilidad**: ajuste de red y parámetros sin recompilación.
- **Usabilidad**: feedback inmediato al usuario (diálogos y confirmaciones).
- **Seguridad**: límite de tiempo en riego manual como mecanismo de protección.
- **Escalabilidad**: fácil ampliación con nuevos parámetros o modos sin alterar la arquitectura general.

10.9. Flujo de la navegación

La aplicación Android presenta un flujo de navegación sencillo y directo, diseñado para facilitar el acceso rápido a la información del invernadero y a las principales funciones de control. Se ha priorizado minimizar el número de acciones necesarias por parte del usuario, evitando estructuras profundas o menús complejos.
La navegación se implementa mediante **Compose Navigation**, definiendo las rutas de la aplicación y el comportamiento de cambio de pantallas desde un componente central.

10.9.1. Pantalla principal como punto de entrada

La **pantalla principal** es el punto de entrada y el núcleo funcional de la aplicación. En ella se visualizan:

- Estado de conexión MQTT (con acceso rápido a configuración si hay fallo).
- Lecturas de sensores principales (aire, suelo, depósito).
- Estado de actuadores y controles manuales.
-

Este diseño permite que el usuario, al abrir la app, disponga inmediatamente de una visión global del sistema y pueda actuar sin navegar por varias pantallas.

10.9.2. Pantallas secundarias y accesos

Desde la pantalla principal se accede a pantallas secundarias mediante el menú superior y/o los accesos disponibles en la interfaz. Las pantallas principales del flujo son:

- **Configuración (PID/Parámetros)**: ajuste de temperatura objetivo y humedad objetivo, tiempo máximo de riego manual y control del modo/color de LED.
- **Configuración del servidor MQTT**: pantalla específica para modificar la dirección del broker y comprobar el estado de conexión.
- **Historial de eventos**: consulta de alertas y notificaciones registradas.
- **Pantalla "About"**: información básica del proyecto y versión.

La separación entre "Configuración de parámetros" y "Configuración del servidor" permite mantener una organización clara: por un lado el comportamiento del sistema y por otro la conectividad.

10.9.3. Navegación orientada a la recuperación ante fallos

Un aspecto importante del flujo es el acceso rápido a la **Configuración del servidor MQTT** cuando se detecta un problema de conexión. Si el sistema no está conectado, la propia interfaz facilita la navegación a la pantalla de servidor para corregir la dirección o reintentar la conexión sin recorrer varios menús.

Este enfoque reduce el tiempo de diagnóstico y mejora la usabilidad en escenarios reales (cambios de red, broker inaccesible, etc.).

10.9.4. Retorno y mantenimiento del estado

El flujo de navegación está diseñado para ser **no destructivo**: al volver a la pantalla principal se mantiene el estado de la aplicación y no se pierde la información mostrada. Los cambios realizados en las pantallas de configuración se aplican de forma inmediata y el usuario puede regresar al control principal sin interrupciones. La navegación respeta además el comportamiento estándar de Android, permitiendo el uso del botón "atrás" y una experiencia coherente con el sistema operativo.

10.9.5. Ventajas del flujo de navegación

El flujo de navegación implementado aporta:

- Acceso rápido a la información crítica.
- Separación clara entre control, configuración, conectividad e histórico.
- Facilidad de recuperación ante problemas de conexión MQTT.
- Experiencia consistente y predecible para el usuario.

10.10. Comunicación con el sistema embebido

La comunicación entre la aplicación Android y el sistema embebido del invernadero constituye un elemento clave del proyecto, ya que permite la supervisión en tiempo real y el control remoto de sensores y actuadores. Esta comunicación se basa en un modelo desacoplado, en el que ambas partes intercambian información sin depender directamente una de la otra.

El uso del protocolo MQTT permite establecer un canal de comunicación bidireccional, eficiente y robusto, adecuado para sistemas IoT distribuidos como el desarrollado en este trabajo.

10.10.1. Roles de la aplicación y del sistema embebido

En la arquitectura del sistema, cada componente tiene responsabilidades bien definidas:

- El **sistema embebido (ESP32)** se encarga de:
  - Leer y procesar los valores de los sensores.
  - Ejecutar la lógica de control automático (riego, ventilación, seguridad).
  - Publicar estados, lecturas y eventos relevantes.
  - Recibir y ejecutar comandos enviados desde la aplicación.

- La **aplicación Android** actúa como:
  - Interfaz de supervisión en tiempo real.
  - Herramienta de control manual de actuadores.
  - Sistema de configuración de parámetros.
  - Receptor y visualizador de alertas y notificaciones.

Esta separación permite que ambos sistemas evolucionen de forma independiente.

10.10.2. Comunicación bidireccional basada en eventos

La interacción entre la aplicación y el sistema embebido se realiza mediante el intercambio de mensajes MQTT asociados a eventos. El sistema embebido publica información cuando se produce un cambio relevante, mientras que la aplicación responde a las acciones del usuario enviando comandos específicos.
Este enfoque evita la necesidad de sondeos constantes (*polling*) y reduce el tráfico de red, ya que solo se transmiten datos cuando es necesario. Además, facilita la escalabilidad del sistema y la incorporación de nuevos sensores o actuadores sin modificar la estructura general de la comunicación.

10.10.3. Sincronización del estado del sistema

La sincronización entre ambos extremos se basa en la publicación continua del estado del sistema y de las lecturas más recientes. Gracias al uso de mensajes retenidos (*retained*), la aplicación puede recuperar el estado actual del invernadero tras una reconexión, sin necesidad de esperar a nuevas publicaciones.
Desde el punto de vista del usuario, esto se traduce en una interfaz que refleja siempre el estado real del sistema embebido, incluso después de interrupciones de red o reinicios de la aplicación.

10.10.4. Gestión de errores y situaciones anómalas

El sistema embebido es responsable de detectar situaciones anómalas, como fallos de sensores o condiciones inseguras durante el riego o la ventilación. Cuando se produce una de estas situaciones, se genera un evento que se comunica inmediatamente a la aplicación mediante los canales de alertas y notificaciones.
La aplicación, por su parte, muestra estos eventos al usuario y los registra en el historial, proporcionando una visión completa de lo ocurrido y facilitando el diagnóstico posterior.

10.10.5. Robustez y desacoplamiento del sistema

Uno de los principales beneficios de la comunicación adoptada es el **desacoplamiento** entre la aplicación y el sistema embebido. Ninguno de los dos componentes necesita conocer el estado interno del otro más allá de los mensajes intercambiados.
Este diseño aporta varias ventajas:

- Mayor tolerancia a fallos de red.
- Facilidad de mantenimiento y ampliación.
- Posibilidad de sustituir o actualizar componentes de forma independiente.
- Comportamiento predecible ante reconexiones.

10.10.6. Valoración de la comunicación con el sistema embebido

La comunicación entre la aplicación Android y el sistema embebido cumple los requisitos de un sistema **IoT** real: es bidireccional, eficiente y robusta frente a fallos. La combinación de **MQTT**, una arquitectura basada en eventos y una interfaz reactiva permite un control preciso del invernadero y una supervisión clara del estado del sistema.

La solución adoptada sienta una base sólida para futuras ampliaciones, como la incorporación de nuevos módulos de control, sensores adicionales o interfaces alternativas.

10.11. Interfaz visual y estilo

La interfaz visual de la aplicación Android se ha diseñado con el objetivo de proporcionar una visualización clara, coherente y fácilmente interpretable del

estado del invernadero. Al tratarse de una aplicación de supervisión y control, se ha priorizado la **legibilidad de la información**, la **rapidez de interpretación** y la **consistencia visual** frente a elementos puramente decorativos.
La interfaz se ha desarrollado íntegramente con **Jetpack Compose**, lo que permite una construcción declarativa de la UI y una actualización automática de los elementos visuales en función del estado del sistema.

10.11.1. Diseño general y tema visual

La aplicación utiliza un **tema oscuro fijo** en todas sus pantallas. Esta decisión se ha tomado por varios motivos:

- Mejora de la visibilidad en entornos con poca luz.
- Reducción de la fatiga visual durante usos prolongados.
- Coherencia estética con aplicaciones de monitorización y control.
- Menor consumo energético en dispositivos con pantallas OLED.

El diseño se basa en fondos oscuros uniformes, texto claro y tarjetas bien delimitadas, lo que facilita la separación visual de la información y evita la saturación de la pantalla.

10.11.2. Uso del color como elemento funcional

El color se emplea como un elemento funcional para representar el estado del sistema, no como un recurso decorativo. Los distintos valores y estados se asocian a colores fácilmente reconocibles, permitiendo al usuario interpretar la situación del invernadero de un vistazo.

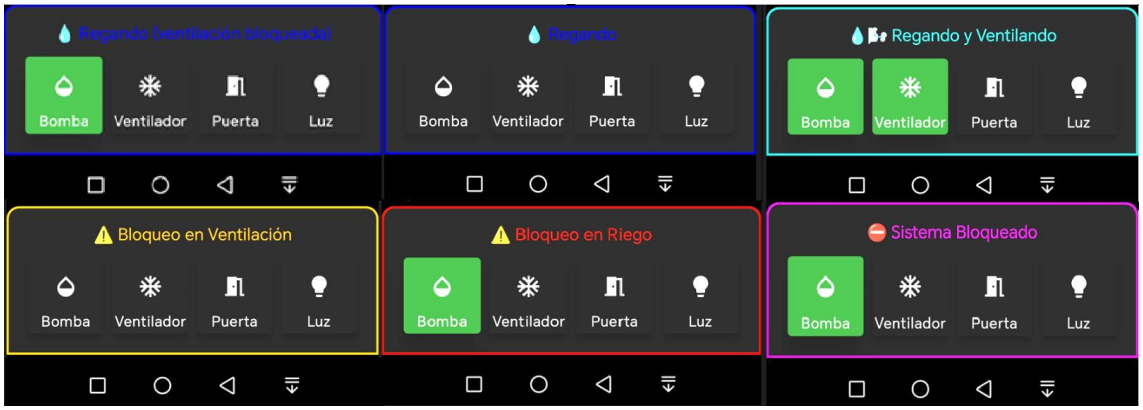

Fig. 27 Diferentes estados por los que pasa el Sistema

De forma general, se siguen los siguientes criterios:

- **Verde**: estado normal o valores dentro del rango óptimo.
- **Azul**: estados relacionados con el riego o el uso de agua.
- **Amarillo / Naranja**: advertencias o estados intermedios.
- **Rojo**: alertas o situaciones críticas.

Este criterio se aplica de forma consistente en sensores, actuadores y elementos visuales, reduciendo la ambigüedad y mejorando la experiencia de uso.

10.11.3. Representación visual de sensores

Los sensores se muestran mediante tarjetas que combinan:

- Valor numérico.
- Unidad de medida.
- Color dinámico asociado al estado del valor.

En el caso de magnitudes como la humedad del suelo o el nivel del depósito, se utilizan representaciones gráficas (indicadores circulares o barras verticales) que facilitan la interpretación visual del dato, especialmente en situaciones en las que el valor exacto es menos relevante que su tendencia o nivel relativo.

10.11.4. Representación y control de actuadores

Los actuadores se representan mediante botones claramente identificables, que permiten tanto visualizar su estado actual como interactuar con ellos. El cambio de color de estos controles actúa como confirmación visual inmediata de la acción realizada.

Este enfoque reduce la incertidumbre del usuario y refuerza la sensación de control sobre el sistema embebido, especialmente en acciones críticas como el riego o la ventilación.

10.11.5. Coherencia visual y experiencia de usuario

Todas las pantallas de la aplicación mantienen una estructura y estilo coherentes: tipografía uniforme, disposición similar de los elementos y uso consistente de colores e iconos. La navegación es directa y evita sobrecargar la interfaz con opciones innecesarias.

Gracias a la integración con el modelo reactivo de la aplicación, cualquier cambio recibido desde el sistema embebido se refleja de forma inmediata en la interfaz, sin necesidad de refrescos manuales ni acciones adicionales por parte del usuario.

10.11.6. Valoración de la interfaz visual

La interfaz visual desarrollada cumple su función como herramienta de supervisión y control del invernadero, ofreciendo una presentación clara de la información y una interacción sencilla con el sistema. El uso de **Jetpack Compose**, junto con un diseño sobrio y funcional, permite mantener una aplicación moderna, mantenible y fácilmente ampliable.

El equilibrio entre simplicidad visual y funcionalidad técnica contribuye a una experiencia de usuario adecuada tanto para pruebas como para un uso continuado del sistema.

10.12. Pruebas y validación

El proceso de pruebas y validación del sistema se ha llevado a cabo con el objetivo de comprobar el correcto funcionamiento de la aplicación Android, la comunicación con el sistema embebido y la respuesta global del invernadero ante distintas situaciones de uso. Las pruebas se han realizado de forma progresiva, validando primero cada componente de manera individual y posteriormente el sistema completo de forma integrada.

10.12.1. Pruebas de comunicación MQTT

Se realizaron pruebas específicas para verificar la estabilidad y fiabilidad de la comunicación **MQTT** entre la aplicación Android y el sistema embebido. Estas pruebas incluyeron:

- Conexión y reconexión al broker **MQTT**.
- Recepción correcta de lecturas de sensores en tiempo real.
- Envío de comandos desde la aplicación y ejecución correcta en el sistema embebido.

- Comprobación del comportamiento tras pérdidas temporales de conexión.

Los resultados mostraron una comunicación estable, con actualización inmediata del estado en la aplicación tras cada mensaje recibido.

10.12.2. Pruebas de control de actuadores

Se validó el control manual de los distintos actuadores (riego, ventilación, puerta e iluminación) desde la aplicación. Para cada actuador se comprobó:

- Envío correcto del comando MQTT.
- Ejecución inmediata de la acción en el sistema embebido.
- Actualización del estado visual en la interfaz.
- Coherencia entre el estado mostrado y el estado real del sistema.

Estas pruebas confirmaron que el sistema responde de forma fiable a las acciones del usuario.

10.12.3. Pruebas de sensores y visualización

Las lecturas de los sensores fueron comprobadas tanto en condiciones normales como en situaciones límite. Se verificó:

- La correcta recepción y visualización de temperatura, humedad y nivel del depósito.
- La actualización dinámica de colores e indicadores gráficos según los valores recibidos.
- La estabilidad de la interfaz ante variaciones rápidas de las lecturas.

La visualización resultó clara y consistente, permitiendo interpretar el estado del invernadero de forma rápida.

10.12.4. Pruebas del sistema de notificaciones y alertas

Se realizaron pruebas forzando distintos eventos desde el sistema embebido para comprobar el comportamiento del sistema de notificaciones:

- Generación de alertas críticas y aparición de mensajes emergentes.
- Generación de notificaciones informativas y visualización en la barra de notificaciones.
- Registro automático de eventos en el historial.
- Comprobación de que los mensajes retained no generan duplicados tras reconexiones.

El sistema respondió correctamente, garantizando que los eventos relevantes se comunican y registran adecuadamente.

10.12.5. Pruebas de configuración

Se validaron las distintas opciones de configuración disponibles en la aplicación:

- Cambio de servidor MQTT y reconexión correcta.
- Modificación de parámetros objetivo (temperatura, humedad).
- Ajuste del tiempo máximo de riego manual y verificación de su aplicación en el sistema embebido.
- Cambio entre modo automático y modo usuario en el control de la iluminación LED.

Los cambios se aplicaron de forma inmediata y coherente, sin afectar negativamente al funcionamiento general de la aplicación.

10.12.6. Validación del sistema completo

Finalmente, se realizaron pruebas de funcionamiento continuo del sistema completo, simulando un uso real del invernadero durante periodos prolongados. Estas pruebas permitieron validar:

- La estabilidad general de la aplicación.
- La coherencia entre el estado del sistema embebido y la información mostrada.
- El correcto registro de eventos y notificaciones.
- La robustez ante situaciones de red variables.

Los resultados obtenidos confirman que el sistema cumple los objetivos definidos en el proyecto y es apto para su uso en un entorno real de monitorización y control.

10.12.7. Valoración final de las pruebas

Las pruebas realizadas demuestran que la solución desarrollada es funcional, estable y coherente con los requisitos planteados. La integración entre aplicación Android, comunicación MQTT y sistema embebido permite un control fiable del invernadero y una supervisión clara de su estado.
El sistema queda preparado para futuras ampliaciones, tanto a nivel de software como de hardware, manteniendo una base sólida y validada.

11

Resultados y líneas futuras

11.1. Resultados obtenidos

Como resultado de este Trabajo de Fin de Grado se ha desarrollado un sistema completo de automatización inteligente para un invernadero de pequeña escala, compuesto por un sistema embebido basado en ESP32 y una aplicación Android para supervisión y control remoto.

El sistema permite la monitorización en tiempo real de variables críticas como temperatura, humedad ambiental, humedad del suelo y nivel del depósito, así como el control de los principales actuadores del invernadero (riego, ventilación, iluminación y apertura de la puerta). La comunicación entre el sistema embebido y la aplicación se realiza mediante el protocolo MQTT, garantizando una transmisión de datos eficiente, desacoplada y fiable.

Se han implementado mecanismos de **tolerancia a fallos**, incluyendo la detección de errores en sensores, condiciones inseguras durante el riego y situaciones de bloqueo, generando alertas y notificaciones que se comunican al usuario y quedan registradas en un historial persistente. Asimismo, la aplicación permite configurar remotamente parámetros de funcionamiento como valores objetivo de temperatura y humedad, tiempo máximo de riego manual y modos de control de la iluminación.
Las pruebas realizadas en un entorno real confirman que el sistema es funcional, estable y cumple los objetivos definidos en el anteproyecto, proporcionando una solución práctica para la automatización y supervisión de invernaderos a pequeña escala.

11.2. Conclusiones

El desarrollo de este proyecto ha permitido integrar de forma coherente conceptos de sistemas empotrados, comunicaciones IoT y desarrollo de aplicaciones móviles, dando lugar a una solución completa y modular.

El uso del protocolo MQTT ha demostrado ser especialmente adecuado para este tipo de sistemas distribuidos, permitiendo una arquitectura desacoplada, escalable

y robusta frente a fallos de red. La separación clara entre el sistema embebido y la aplicación Android facilita el mantenimiento del sistema y su posible evolución futura.
Desde el punto de vista de la aplicación, la arquitectura basada en MVVM y el uso de Jetpack Compose han permitido desarrollar una interfaz reactiva, clara y fácilmente ampliable. La integración de notificaciones y persistencia local mejora la experiencia de usuario y aporta trazabilidad al funcionamiento del sistema.

Aunque el sistema PID implementado funciona correctamente y mantiene la humedad dentro de un rango estable, el estudio detallado del comportamiento dinámico del proceso podría ampliarse en trabajos futuros para obtener una caracterización más profunda del sistema físico.

En conjunto, el proyecto cumple los objetivos planteados inicialmente y demuestra la viabilidad de implementar sistemas de automatización inteligentes de bajo coste con control remoto y tolerancia a fallos, sentando una base sólida para desarrollos más avanzados en el ámbito del Internet de las Cosas.

11.3. Propuestas de mejora y líneas futuras

A partir del sistema desarrollado, se identifican diversas líneas de mejora que podrían abordarse en trabajos futuros.

Realizar un estudio más exhaustivo del control PID, incluyendo pruebas prolongadas con diferentes sustratos y sensores, con el fin de analizar en mayor detalle la respuesta dinámica del sistema y su comportamiento frente a variaciones ambientales.

Otra posible ampliación consiste en reforzar la **seguridad de las comunicaciones**, incorporando mecanismos de autenticación y cifrado (TLS) en el broker MQTT, especialmente relevante en escenarios con acceso desde redes externas.
Asimismo, el sistema podría evolucionar hacia un mayor grado de **automatización inteligente** mediante la incorporación de técnicas de visión artificial. Aprovechando la cámara RTSP ya integrada, sería posible analizar imágenes del cultivo para estimar el estado visual de las plantas, detectando situaciones como estrés hídrico, estrés térmico o posibles anomalías. Esta información podría utilizarse para ajustar de forma automática las consignas de control (temperatura y humedad objetivo) o para generar alertas adicionales al usuario.

Desde el punto de vista arquitectónico, este módulo de visión podría implementarse como un servicio independiente conectado al broker MQTT, publicando sus resultados en tópicos específicos. De este modo, se mantendría el desacoplamiento del sistema y se garantizaría que las decisiones automáticas estén siempre acotadas por los mecanismos de seguridad ya existentes en el sistema embebido.
Otras líneas de mejora incluyen la gestión de múltiples invernaderos desde una única aplicación, el almacenamiento de datos en la nube, el acceso multiusuario con distintos niveles de permisos y la incorporación de visualizaciones avanzadas de datos históricos.

Estas ampliaciones permitirían aumentar la versatilidad del sistema y adaptarlo a escenarios más complejos, manteniendo la arquitectura desarrollada como base sólida.

Referencias

Sistemas embebidos y ESP32

[1] Espressif Systems. ESP32 Series Datasheet.
https://www.espressif.com/

[2] Espressif Systems. ESP-IDF Programming Guide.
https://docs.espressif.com/projects/esp-idf/

[3] Arduino. ESP32 Arduino Core Documentation.
https://docs.arduino.cc/

[4] Arduino. Arduino IDE – Software oficial.
https://www.arduino.cc/en/software

[5] Arduino Forum. Foro oficial de Arduino.
https://forum.arduino.cc/

[6] Fritzing. Fritzing – Diseño de esquemas y prototipos.
https://fritzing.org/

Comunicación MQTT e IoT

[7] Eclipse Foundation. MQTT Version 3.1.1 Specification.
https://mqtt.org/

[8] HiveMQ. MQTT Essentials.
https://www.hivemq.com/mqtt-essentials/

[9] HiveMQ. Getting Started with MQTT.
https://www.hivemq.com/blog/how-to-get-started-with-mqtt/

[10] HiveMQ. HiveMQ MQTT Client for Java/Kotlin.
https://github.com/hivemq/hivemq-mqtt-client

[11] Nick O'Leary. PubSubClient Library for Arduino.
https://github.com/knolleary/pubsubclient

[12] Stack Overflow. MQTT ESP32 & Android Discussions.
https://stackoverflow.com/

Android, Kotlin y desarrollo móvil

[13] https://developer.android.com/develop/ui/views/notifications

[14] Android Developers. Kotlin Coroutines and StateFlow.
https://developer.android.com/kotlin/coroutines

RTSP, vídeo y cámaras IP

[15] VideoLAN. LibVLC Android SDK.
https://code.videolan.org/videolan/libvlc-android

[16] Real Time Streaming Protocol (RTSP). RFC 2326.
https://www.rfc-editor.org/

[17] TP-Link. Tapo C200 – RTSP Configuration Guide.
https://www.tp-link.com/

Virtualización, servidores y sistemas

[18] Proxmox Server Solutions. Proxmox VE Documentation.
https://www.proxmox.com/en/proxmox-ve

[19] Debian Project. Debian GNU/Linux Documentation.
https://www.debian.org/doc/

[20] Docker. Docker Documentation.
https://docs.docker.com/

[21] Mosquitto. Eclipse Mosquitto MQTT Broker.
https://mosquitto.org/

[22] Linux Documentation Project. Linux Networking.
https://tldp.org/

[23] Python, automatización y scripting

[24] Python Software Foundation. Python Documentation.
https://docs.python.org/3/

[25] GNU Project. Bash Reference Manual.
https://www.gnu.org/software/bash/manual/

Diseño, diagramas y documentación
[26] Mermaid.js. Mermaid – Diagramas como código.
https://mermaid.js.org/
[27] draw.io (diagrams.net). Herramienta de diagramas online.
https://www.diagrams.net/
[28] Markdown Guide. Markdown Documentation.
https://www.markdownguide.org/

Visión artificial e inteligencia artificial (líneas futuras)
[29] OpenCV. Open Source Computer Vision Library.
https://opencv.org/
[30] Szeliski, R. Computer Vision: Algorithms and Applications. Springer, 2010.
[31] TensorFlow Lite. ML for Edge Devices.
https://www.tensorflow.org/lite

Canales técnicos (YouTube – apoyo práctico)
[32] Andrés J. Moya – Electrónica y ESP32 (ES)
https://www.youtube.com/
[33] Luis Llamas – Electrónica, IoT y Arduino (ES)
https://www.luisllamas.es/
[34] The Engineering Mindset – Systems & Control (EN)
https://www.youtube.com/
[35] GreatScott! – Electrónica aplicada (EN)
https://www.youtube.com/

Apéndice A. Código fuente y recursos

En este apéndice se recogen los distintos entregables generados durante el desarrollo del Trabajo de Fin de Grado. Estos materiales complementan la memoria escrita y permiten validar, reproducir y analizar el funcionamiento del sistema desarrollado.
Los entregables se proporcionan en formato digital junto con la memoria del proyecto.

A.1. Repositorio del Proyecto

El código fuente completo y el material asociado al proyecto se encuentran disponibles en un repositorio Git, que incluye el historial de desarrollo y la organización del trabajo realizado.
Repositorio GitHub del proyecto: https://github.com/Juanjo81/Invernadero-tfg/

El repositorio contiene tanto el firmware del sistema embebido como la aplicación Android.

A.2. Vídeos de funcionamiento del sistema

Se incluyen distintos vídeos demostrativos del proyecto, entre ellos:

- Vídeo general del funcionamiento completo del sistema.
- Vídeo del control PID en funcionamiento.
- Vídeo de la aplicación Android durante el control del sistema.
- Vídeo timelapse del crecimiento de las plantas.

Estos vídeos permiten observar de forma clara el comportamiento del sistema y su evolución en el tiempo. Los vídeos se entregan como archivos adjuntos y/o mediante enlace a una plataforma de alojamiento en línea.

A.3. Firmware del sistema embebido (ESP32)

Se entrega el código fuente completo del firmware desarrollado para el microcontrolador ESP32, implementado en C++ sobre el entorno Arduino.
El firmware incluye:

- Lectura de sensores ambientales y de nivel.
- Control de actuadores (riego, ventilación, iluminación y servomotores).
- Comunicación MQTT con el broker Mosquitto.
- Lógica de protección y tolerancia a fallos.
- Supervisión del estado del sistema.

El código se entrega estructurado y documentado mediante repositorio Git.

A.4. Aplicación Android

Se entrega el código fuente completo de la aplicación Android desarrollada en Kotlin utilizando Jetpack Compose.
La aplicación incluye:

- Visualización en tiempo real de sensores y estados.
- Control remoto de actuadores.
- Configuración de parámetros de funcionamiento.
- Sistema de notificaciones y registro de eventos.
- Acceso a cámara mediante RTSP.

A.5. Material gráfico del proyecto

Se incluyen fotografías del sistema físico desarrollado, que muestran:

- Montaje del hardware.
- Sensores y actuadores instalados.
- Disposición general del invernadero y los elementos de control.

Este material permite contextualizar visualmente el desarrollo y validar la implementación real del sistema.

A.6. Scripts de soporte y automatización

Se entregan distintos scripts desarrollados como apoyo al sistema principal, implementados en **Bash** y **Python**, entre los que se incluyen:
Script de registro y almacenamiento de logs del sistema.
Script de actualización dinámica de DNS.
Script Bash para captura automática de imágenes y vídeos desde la cámara RTSP.
Script Python para procesamiento de datos en formato JSON.

Estos scripts complementan el sistema y facilitan tareas de mantenimiento, monitorización y análisis.

A.7. Captura y análisis de datos del control PID

Se incluye un conjunto de datos registrados durante el funcionamiento del sistema de control PID, almacenados en formato **JSON**, correspondientes a un periodo prolongado de operación.
A partir de estos datos se generan:

- Gráficas de funcionamiento del control PID.
- Análisis de la estabilidad del sistema.
- Evaluación del comportamiento ante cambios de consignas.

A.8. Gráfica de funcionamiento continuo

Se adjuntan gráficas representativas del funcionamiento del sistema durante un periodo continuo, mostrando la evolución de las variables controladas y la respuesta del sistema ante distintas condiciones.
Este material permite validar la estabilidad y el comportamiento del sistema en condiciones reales de uso prolongado.

A.9. Material adicional

Como material complementario, se entregan:

- Diagramas de secuencia del sistema realizados mediante **Mermaid**, que describen el flujo de comunicación entre la aplicación Android, el broker MQTT y el sistema embebido.
- Diagramas electrónicos del sistema elaborados con **Fritzing**, que representan la conexión de sensores, actuadores y módulos del sistema embebido.
- Ejemplos de mensajes MQTT intercambiados.
- Capturas de pantalla de la aplicación Android.

- Diagramas de arquitectura y flujo general del sistema.

Este material facilita la comprensión del diseño y la arquitectura del sistema desarrollado.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga