Semáforo IoT: Arquitectura por capas con ESP32 y aplicación Python

Maximiliano Aranda León[1][0009−0004−8453−9157], Juan Erick Alvarado Cruz[1][0009−0009−2115−3878], Omar Hernández Luis[1][0009−0008−8793−213X], and Josué Vázquez Jiménez[1][0009−0001−5918−8509]

Benemérita Universidad Autónoma de Puebla (BUAP), Facultad de Ciencias de la Computación, Puebla, México
{maximiliano.aranda,josue.vazquez,juan.alvaradoc,
omar.hernandezlu}@alumno.buap.mx

Abstract. El proyecto "Semáforo" implementa una interfaz gráfica en *Python* (*Tkinter*) que controla una réplica visual de un semáforo y sincroniza su comportamiento con un *ESP32* via *HTTP REST*. La aplicación soporta control manual (botones por cada luz), ajuste dinámico de duraciones mediante `Scale` y una rutina automática cíclica que respeta los tiempos indicados. Las peticiones al *ESP32* se centralizan en un cliente *HTTP* reutilizable.

Keywords: IoT · HCI · REST · ESP32 · Python · Arduino · Arquitectura de Software

1 Introducción

Los procesos informáticos se encuentran profundamente integrados en la vida cotidiana: desde la comunicación personal y la publicación de noticias hasta el monitoreo de pacientes en hospitales, las operaciones bancarias o el seguimiento ambiental y urbano. Esta ubicuidad genera una creciente dependencia del ser humano hacia la tecnología y consolida el campo de la interacción Humano-Computadora (*Human-Computer Interaction*, HCI).

En este contexto, las *Arquitecturas de Software* desempeñan un papel esencial. Una arquitectura es la base conceptual que define cómo se organiza un sistema, qué responsabilidades asume cada componente y qué normas guían su diseño y evolución [8].

Como ejercicio práctico, se desarrolla una aplicación que simula y controla un semáforo. La solución consiste en una aplicación de escritorio en *Python* (*Tkinter*), que se comunica con un dispositivo *ESP32* mediante una *API REST*. El sistema permite control manual (botones para cada luz), ajuste dinámico de duraciones y una rutina automática que emula el ciclo real de un semáforo. La implementación se apoya en el principio de separación de responsabilidades: interfaz de usuario, lógica de control y adaptadores de infraestructura (cliente *HTTP*). Este diseño facilita el análisis arquitectónico bajo un enfoque por capas.

El objetivo principal es mostrar, a través de un caso sencillo pero representativo, cómo aplicar principios de arquitectura por capas en un sistema embebido conectado.

2 Estado del arte

Se han desarrollado proyectos de *IoT* referentes a la manipulacion y monitoreo de semaforos. Un ejemplo relevante es el diseño de un sistema de monitoreo de fallas en semáforos utilizando la aplicación *Blynk* y el microcontrolador *ESP32*. Este sistema permite a los técnicos supervisar en tiempo real el estado de las luces de tráfico en múltiples carriles, detectando posibles fallos como apagones o malfuncionamientos que podrían generar congestión o accidentes [2]. El prototipo presentado en ese trabajo simula el cruce de carreteras con cuatro semáforos y define procesos para hacer las simulaciones de fallos usando variaciones de voltaje para cada semaforo. Este trabajo es una referencia para el uso del microcontrolador *ESP32* pues los autores comparten diagramas de los pines ocupados para simular los semaforos asi como algoritmos de sus simulaciones.

Hay proyectos que proponen luces de trafico automatizadas como el que proponen Ramadhan et al [7]. Los autores proponen un sistema inteligente de semáforo basado en *Arduino Mega*, sensores ultrasónicos y una cámara *ESP32*, capaz de detectar la densidad vehicular en tiempo real y ajustar automáticamente el tiempo de luz verde, además de capturar infracciones y enviarlas a una base de datos mediante *Telegram* [7]. A diferencia de dicho enfoque, el presente proyecto se centra en el desarrollo de un semáforo controlado por una placa *ESP32*, utilizando un servidor *HTTP* embebido en el microcontrolador, lo que permite la manipulación remota del sistema a través de la red. Esta diferencia marca un contraste entre la automatización basada en sensores físicos y visión por computadora, y el control manual remoto mediante interfaz web, ofreciendo una alternativa más accesible y flexible para entornos educativos.

Inclusive hay soluciones que van mas allá de una optimización por algoritmos, hay propuestas de usar Inteligencia Artificial, como la propuesta de Hauck et al [5]. que propone un conjunto de herramientas de código abierto que facilitan el diseño de ideas de negocio basadas en *IoT* e *IA*, organizadas según sus capacidades y con propuestas de extensión para crear bibliotecas accesibles para *PyMEs*. Estas herramientas permiten la implementación de sistemas distribuidos de manera sencilla, fomentando la innovación tecnológica en entornos con recursos limitados [5].

Por último, otro enfoque relevante es el presentado por Abu et al. [4], quienes desarrollaron un sistema de control inteligente de semáforos enfocado en facilitar el paso de vehículos de emergencia como ambulancias, patrullas y camiones de bomberos en intersecciones congestionadas. El sistema combina tecnologías como *PLC* (Controlador Lógico Programable), *NodeMCU ESP32*, la aplicación *IFTTT* y la plataforma *Adafruit IO* para integrar sensores y automatizar la gestión del tráfico. Esta solución permite detectar la presencia de vehículos prioritarios y modificar el estado del semáforo en tiempo real para garantizar su

paso seguro y rápido [4]. A diferencia de este enfoque, el presente proyecto se basa en una arquitectura más ligera y accesible,

Trabajo	**Hardware / Plataforma**	**Enfoque / Control**	**Ventajas**	**Limitaciones / Relevancia**
Asry et al. (2024)	ESP32 + Blynk (app)	Monitoreo y simulación de fallas de semáforos mediante app móvil	Monitorización en tiempo real; diagramas de pines y estrategias de simulación	Orientado a detección de fallas; depende de servicio/ciudadano móvil; referencia útil para uso de ESP32 y mapeo de pines
Ramadhan et al. (2021)	Arduino Mega + sensores ultrasónicos + ESP32 (cámara)	Semáforo inteligente adaptativo por detección de densidad vehicular (sensores y visión)	Ajuste dinámico de verde según tráfico; captura de infracciones y envío a DB	Mayor complejidad HW/SW y coste; enfoque autónomo (contrasta con control manual remoto)
Hauck et al. (2019)	Herramientas abiertas IoT/IA (software)	Framework / bibliotecas para integrar IoT con IA en soluciones distribuidas	Facilita integración IA/IoT; extensible para PYMEs y prototipos	Requiere infraestructura adicional y conocimientos IA; relevante como opción para mejora futura (optimización automática)
Este trabajo (proyecto BUAP)	ESP32 (firmware REST) + App Python (Tkinter)	Control remoto vía REST: control manual + rutina configurable (scales) y sincronización HTTP	Arquitectura por capas clara; fácil de reproducir y didáctico; integración HW–SW simple	No incorpora sensores ni AI; dependiente de red; adecuado como caso de estudio pedagógico

Table 1. Comparativa de trabajos relacionados en control y monitorización de semáforos.

3 Metodología

3.1 Diseño arquitectónico

La práctica se fundamentó en el patrón arquitectónico de capas, una estructura ampliamente adoptada en el diseño de sistemas de software por su capacidad de separar responsabilidades y facilitar la mantenibilidad. Esta arquitectura divide el sistema en tres capas principales: presentación, negocio e infraestructura Asi,

cada capa agrupa funcionalidades propias y proporciona servicios a la capa superior dividiendo las responsabilidades [3] Según Jiménez-Torres et al. [6], muchos lenguajes de patrones de arquitectura derivan de este modelo, ya que permite organizar soluciones reutilizables en dominios específicos mediante patrones de negocio, integración, aplicación y ejecución [6].

En este contexto, se diseñó un sistema "Semáforo" controlado remotamente, donde la capa de presentación fue implementada en *Python* con *Tkinter*, la lógica de negocio se gestionó mediante solicitudes *HTTP*, y la infraestructura estuvo compuesta por el microcontrolador *ESP32*, relevadores y LEDs.

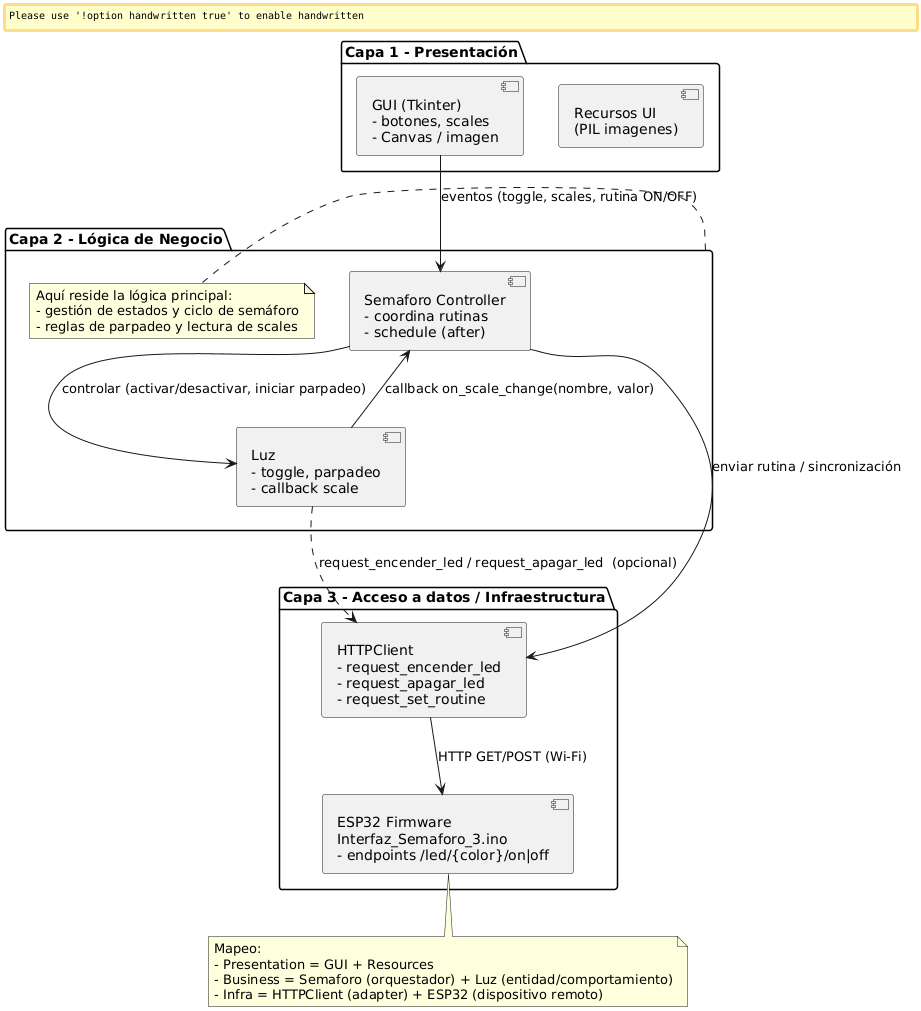

Fig. 1. Diagrama de capas: mapeo de *GUI*, controlador (`Semaforo`), entidades (`Luz`) y adaptadores (*HTTPClient / ESP32*).

El controlador *ESP32* acciona módulos de potencia para las luces del semáforo. A continuación se resume la integración física y las decisiones de diseño.

Resumen de conexión

- Pines *GPIO* (*ESP32*) utilizados:
  - PIN_LED_VERDE: *GPIO21*
  - PIN_LED_AMARILLO: *GPIO22*
  - PIN_LED_ROJO: *GPIO23*
- Interfaz de potencia: módulos relé (*5V*) controlados desde las salidas *GPIO* del *ESP32*. Las señales de control son de 3.3 V (*ESP32*); se procuró que los módulos sean compatibles.
- Alimentación: fuente 5 V para relés/LEDs y 3.3 V interna del *ESP32*; compartir tierra (*GND*) entre ambas fuentes.

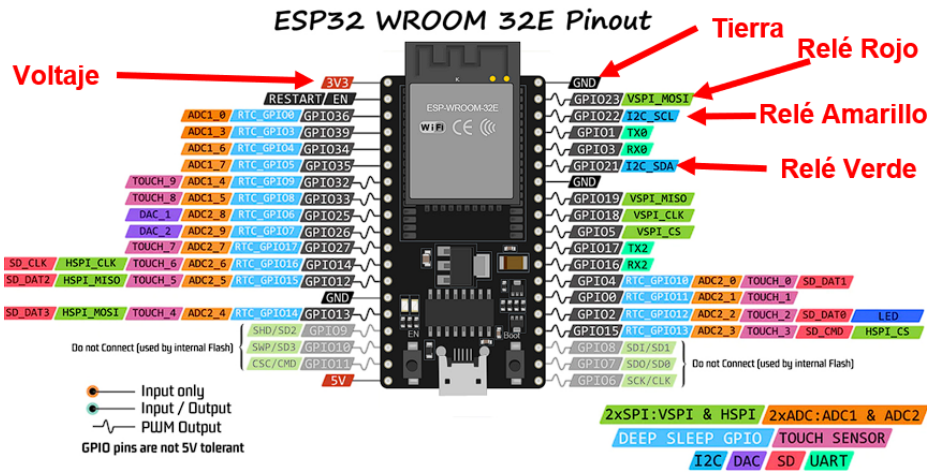

Fig. 2. Esquema de conexion del controlador *ESP32*

Esquemático y materiales El esquema de conexion detallado y la lista de materiales se muestran en las Figura 2 y 3; y la Tabla 2 respectivamente.

3.2 Diseño de componentes

Se sigue el paradigma de programación orientada a objetos (POO) para facilitar la separación de responsabilidades, la reutilización y la prueba de unidades. Cada componente encapsula su estado y comportamiento concreto, exponiendo una *API* clara para que el resto del sistema lo use. La POO permite además aplicar inyección de dependencias (p. ej. pasar una instancia de `HTTPClient` a las luces) para desacoplar la lógica de control de la capa de infraestructura y facilitar las pruebas.

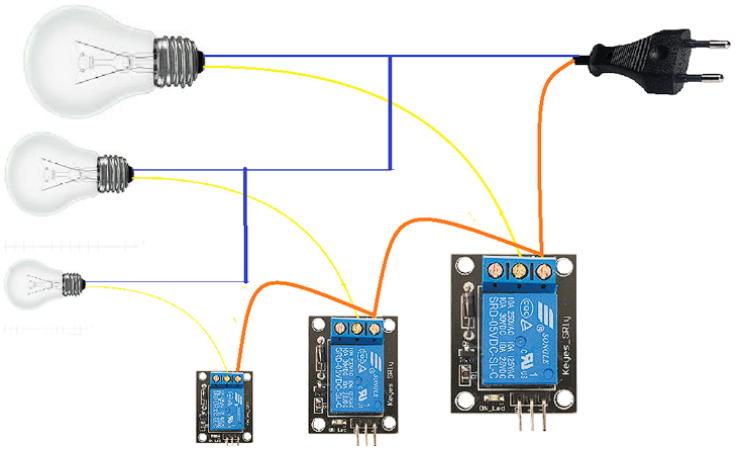

Fig. 3. Esquema de conexion de los relevadores

Componente	Cantidad	Observaciones
ESP32 + Shield	1	Pines: 21,22,23
Módulo relé 1-canal (5V)	3	
LEDs 5V / Lámparas	3	
Sockets para LEDs	3	
Estructura de semáforo	1	
Cables y protoboard	—	

Table 2. Lista de materiales.

Resumen de clases y responsabilidades

- `Semaforo` (Controller / Orquestador)
  - Responsable de coordinar la rutina completa del semáforo (ciclo verde → parpadeo → amarillo → rojo).
  - Mantiene instancias de `Luz` (composición) y gestiona el scheduling no bloqueante mediante `tkinter.Widget.after()`.
  - Escucha cambios de duración (callbacks desde las `Luz`) y recalcula/reprograma el ciclo cuando es necesario.
  - Usa `HTTPClient` para sincronizar la rutina completa con el *ESP32* (opcional).
- `Luz` (Entidad / Componente de UI)
  - Encapsula los elementos de la interfaz relacionados a una luz: botón de toggle, `Scale` para duración y el círculo en el `Canvas`, se hablara de él más adelante.
  - Gestiona su propio estado visual (encendida/apagada) y comportamiento de parpadeo mediante llamadas a `after()`.
  - Expone un callback para notificar cambios de duración (`on_duration_change`) que `Semaforo` registra.

  - Recibe (por inyección) una instancia de `HTTPClient` y delega en ella las peticiones *REST* (encender/apagar) — de forma asíncrona para no bloquear la *UI*.
- `HTTPClient` (Adaptador / Infraestructura)
  - Centraliza la comunicación *HTTP* con el *ESP32*: rutas, timeouts, manejo de errores y retries básicos.
  - Proporciona métodos claros: `request_encender_led(color, velocidad?)`, `request_apagar_led(color)`, `request_set_routine(durs)`.
  - Diseñada para ser inyectada en `Semaforo` y/o en cada `Luz`, permitiendo reemplazo por un mock en pruebas unitarias.
  - Ejecuta las llamadas en hilos secundarios (o permite que el llamador lo haga) para no bloquear el hilo de la *GUI*.

Interacción y patrones aplicados

- Composición: `Semaforo` contiene varias `Luz` y las coordina.
- *Callback* / Observer ligero: `Luz` notifica a `Semaforo` cambios de duración mediante una función registrada.
- Adapter: `HTTPClient` actúa como adaptador de la capa de infraestructura (*REST*) frente a la lógica de negocio.
- Concurrency: las operaciones de red se aíslan para conservar la capacidad de respuesta de la *REST*; las temporizaciones usan mecanismos no bloqueantes.

Clase	**Atributos principales**	**Métodos principales**
`Semaforo`	referencias a `Luz` (verde/amarilla/roja), `HTTPClient`, flags de rutina, lista de ids `after`	`activar_rutina()`, `desactivar_rutina()`, `_schedule()`, `_cancel_scheduled()`, `_on_scale_duration_changed()`
`Luz`	`nombre`, `boton`, `scale`, `circle_id`, estado `encendida`, referencia a `HTTPClient`	`toggle()`, `place_circle()`, `iniciar_parpadeo()`, `detener_parpadeo()`, `on_scale_change()`
`HTTPClient`	`base_url`, timeouts, opciones de retry	`request_encender_led()`, `request_apagar_led()`, `request_set_routine()`

Table 3. Resumen compacto de atributos y métodos clave.

La FIgura 3 muestra el diagrama de clases utilizado.

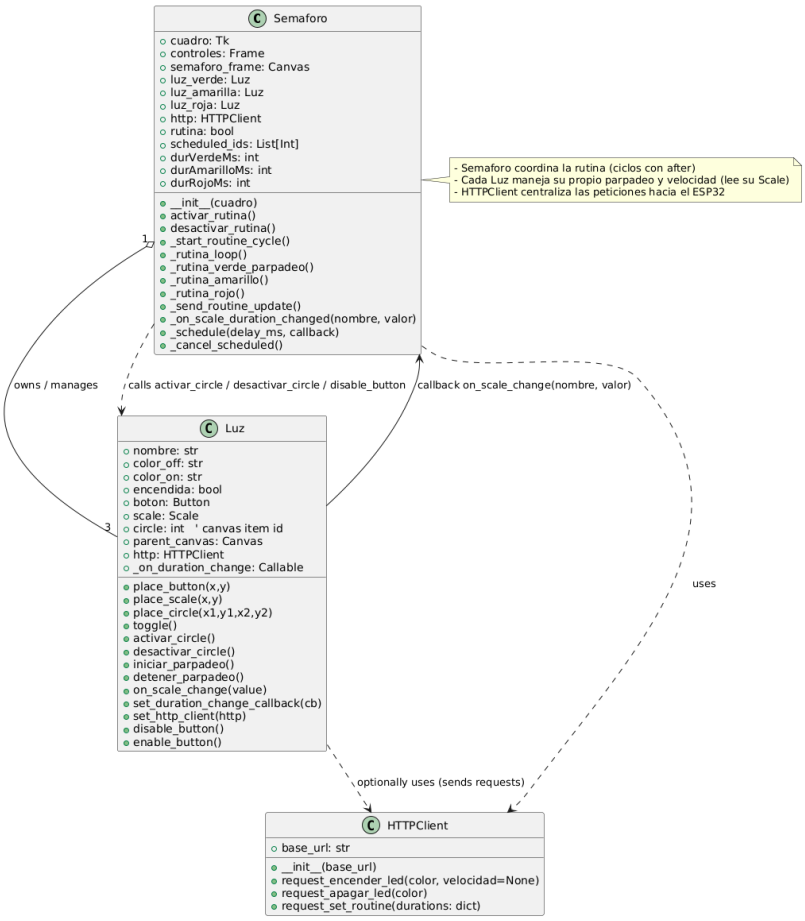

Fig. 4. Diagrama de clases del sistema Semáforo

3.3 Implementación

Tecnologías y librerías

- Aplicación: *Python 3*, *Tkinter (GUI)*, *Pillow* (recursos gráficos), *requests (HTTP)*.
- *Firmware*: *ESP32 (ESP-IDF/Arduino)*, librería *WebServer* para endpoints *REST*.
- Entorno: ejecución en PC con red local (*Wi-Fi*) comunicando con *ESP32* por *IP*.

Mecanismos no bloqueantes La aplicación *UI* evita bloqueos usando `tkinter.Widget.after()` para programar fases y parpadeos, lo que permite mantener la capacidad de respuesta mientras se ejecuta la rutina. En el firmware del *ESP32* la lógica de

la rutina se implementa con una máquina de estados no bloqueante basada en `millis()`, de modo que la gestión de pines y el manejo de cliente *HTTP* coexisten sin retardos por `delay()` [1].

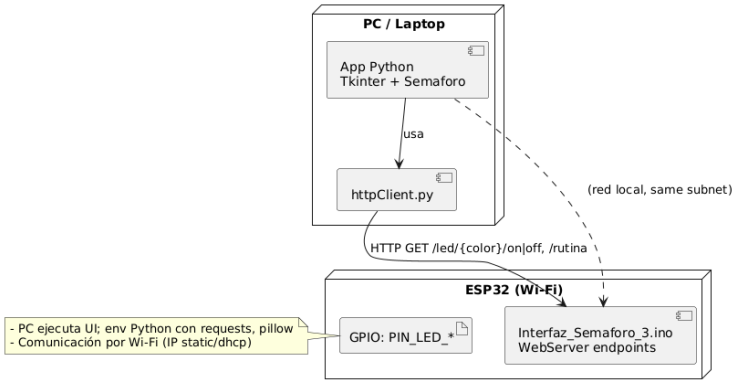

Fig. 5. Diagrama de despliegue: *App Python (Tkinter + HTTPClient)* y *ESP32* con *endpoints REST*.

4 Pruebas experimentales

4.1 Objetivo

Verificar el funcionamiento funcional del sistema integrado (App Python + ESP32) mediante pruebas manuales reproducibles y registro visual del comportamiento.

4.2 Entorno de pruebas

- PC con Python 3.x ejecutando la aplicación Tkinter.
- ESP32 con firmware `Interfaz_Semaforo_3.ino` en la misma red Wi-Fi.
- Versiones relevantes: Python 3.x, librerías `requests`, `Pillow`.

4.3 Casos de prueba funcionales

1. Control manual: enviar `/led/{color}/on` y `/led/{color}/off` desde la GUI; verificar efecto visual (canvas) y físico (LED/relé).
2. Rutina ON/OFF: activar rutina con distintas combinaciones de duraciones y comprobar la secuencia (verde → parpadeo → amarillo → rojo).
3. Cambio dinámico de duraciones: ajustar `Scale` mientras la rutina está activa y verificar reprogramación inmediata del ciclo.
4. Robustez funcional: apagar/encender el ESP32 o desconectar la red y comprobar que la aplicación no bloquea y registra el fallo en consola.

4.4 Registro y observaciones

Las pruebas realizadas fueron manuales y basadas en inspección visual y registros de consola. Observaciones principales:

- La GUI permanece responsiva durante la ejecución de la rutina gracias a `after()`.
- La secuencia del semáforo se sigue correctamente según las duraciones configuradas; el parpadeo verde se realiza conforme al diseño.
- Ajustar una `Scale` durante la rutina provoca la reprogramación esperada y la interfaz refleja el cambio de inmediato.
- Ante pérdida de conectividad del ESP32 las llamadas HTTP generan excepciones capturadas en consola; la aplicación continúa operando localmente (sin sincronización con hardware).

4.5 Tabla de pruebas realizadas

Caso de prueba	**Procedimiento**	**Resultado observado**
Encendido manual	Pulsar botón de cada luz	LED virtual y físico se encienden; petición HTTP registrada en consola
Rutina ON	Iniciar rutina con duraciones diversas	Ciclo correcto; parpadeo verde ok
Cambio dinámico de scale	Ajustar scale durante rutina	Rutina se reprograma y actualiza visualmente
Scale = 0	Poner una duración a 0 para saltar fase	Fase correspondiente se omite correctamente
ESP32 desconectado	Cortar Wi-Fi del ESP32	Excepciones en consola; UI no bloquea; hardware no responde

Table 4. Resumen de pruebas funcionales manuales.

5 Resultados

5.1 Observaciones cualitativas

Dado que en esta práctica no se realizaron medidas instrumentadas de rendimiento, los resultados se basan en observaciones funcionales durante la integración y las pruebas manuales:

- La interfaz gráfica permanece responsiva durante la ejecución de la rutina: el uso de `tkinter.Widget.after()` evita bloqueos visibles en la *GUI*.

– La secuencia del semáforo (verde → parpadeo → amarillo → rojo) se ejecuta según las duraciones configuradas por los `Scale`, y el parpadeo verde se realiza correctamente (tres ciclos de encendido/apagado con la temporización definida).
– Cambiar un `Scale` mientras la rutina está activa provoca que el controlador reprograme el ciclo y la aplicación mantenga la sincronía visual; en la práctica esto se percibe como una actualización casi instantánea en la interfaz.
– Las peticiones *HTTP* al *ESP32* se completan correctamente en condiciones de red local estable; ante pérdida de conectividad la aplicación captura excepciones y muestra mensajes en consola (no existe aún una *UI* de error dedicada).

5.2 Evidencia visual

A continuación se incluyen fotografías y capturas que muestran el sistema en funcionamiento. Sustituya las rutas por las imágenes exportadas del proyecto.

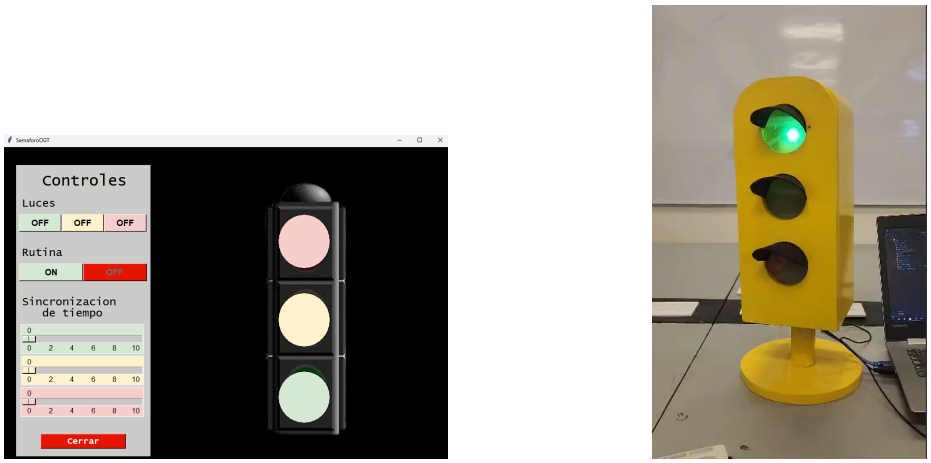

Fig. 6. Izquierda: captura de la aplicación Python (*GUI*). Derecha: montaje físico del semáforo controlado por *ESP32* y módulos de potencia.

6 Conclusiones

Se diseñó e implementó un sistema educativo que integra una aplicación de escritorio en *Python (Tkinter)* con un microcontrolador *ESP32* mediante una *API REST*. El prototipo soporta control manual de cada luz, ajuste dinámico de duraciones y una rutina automática que emula el ciclo de un semáforo. A nivel arquitectónico se adoptó un modelo por capas (Presentación, Lógica de negocio,

Infraestructura) y un enfoque orientado a objetos para encapsular responsabilidades en `Semaforo`, `Luz` y `HTTPClient`.

En la práctica, los retos más relevantes fueron de naturaleza física: el cableado y la interconexión con módulos de potencia (relés), el dimensionamiento de la alimentación, y la verificación de la lógica de activación de los relés. Estas dificultades reforzaron la necesidad de documentar el mapeo de pines, garantizar una masa común segura. A nivel software, el uso de `after()` en la *GUI* y de una máquina de estados no bloqueante en el firmware (basada en `millis()`) permitió mantener la responsividad y la concurrencia cooperativa entre interfaz, temporización y comunicaciones.

References

1. tkinter — python interface to tcl/tk, https://docs.python.org/3/library/tkinter.html
2. Asry, A.I., Lutfi, L., Umar, A., Ahmad, R.: Monitoring system for traffic light lamp damage using blynk application based on iot esp32. Jurnal Teknologi Transportasi Dan Logistik **5**(1), 59–66 (2024)
3. Castro, L.: Arquitectura del Software. Cengage Learning Editores (2015)
4. Gupta, R., Singh, A.K., Dabral, S., Tewari, P.P.: Iot based traffic light control based on traffic density. In: 2023 Second International Conference on Informatics (ICI). pp. 1–5. IEEE (2023)
5. Hauck, M., Machhamer, R., Czenkusch, L., Gollmer, K.U., Dartmann, G.: Node and block-based development tools for distributed systems with ai applications. IEEE Access **7**, 143109–143119 (2019)
6. Jimenez-Torres, V.H., Tello-Borja, W., Rios-Patiño, J.I.: Lenguajes de patrones de arquitectura de software: una aproximación al estado del arte. Scientia et technica **19**(4), 371–376 (2014)
7. Ramadhan, Z.A., Mohammed, B.K., Alwaily, A.H., et al.: Design and implement a smart traffic light controlled by internet of things. Periodicals of Engineering and Natural Sciences **9**(4), 542–548 (2021)
8. Reynoso, C.B.: Introducción a la arquitectura de software. Universidad de Buenos Aires **33**, 11 (2004)