Control de displays 7-seg con ESP32 y 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. This work demonstrates the design and implementation of two common cathode seven-segment displays integrating a control system using `Arduino` with the `ESP32` microcontroller. It works through a "server" to which any client can make requests to the respective End-Points defined in the `ESP32`. In turn, I designed and implemented a `Python` application (using `Tkinter` and `Requests`) as a client; a simple and minimalist design was maintained, with only buttons that indicate the activation of functionalities allowed by the microcontroller, which are described below. Through this architecture, the system supports both an ascending/descending counter and a basic calculator whose results—when limited to two digits—are directly rendered on the physical displays. This project demonstrates not only the practical aspects of hardware multiplexing and microcontroller-based server deployment, but also the potential of combining embedded systems with higher-level applications for interactive learning environments, remote monitoring, and educational tools in computer engineering curricula.

Keywords: IoT · HCI · REST · `ESP32` · `Python` · `Arduino` · Display

1 Introducción

Los sistemas embebidos se encuentran profundamente arraigados en la vida cotidiana: detección y ajuste de temperatura automático en las "casas inteligentes", monitoreo del movimiento automovilístico en las calles, acciones de la bolsa, etc. Esta ubicuidad demuestra la dependencia del ser humano hacia la tecnología. Así, la interacción entre hardware y software configura escenarios donde el ser humano delega cada vez más tareas a sistemas conectados y consolida el campo de la interacción humano-computadora (*Human-Computer Interaction*, HCI) y su convergencia con el *Internet of Things* (IoT).

En este contexto, los microcontroladores como el `ESP32` demuestran y destacan su versatilidad: conexión inalámbrica, poder de procesamiento local y la capacidad de ejecución de diferentes servicios. Esto, sumado a una integración de aplicaciones de alto nivel como las desarrolladas con `Python`, permite el enlace

entre la interacción del mundo físico mediante, tal vez, sensores que proporcionan información a un microcontrolador como lo puede ser el `ESP32`, y poder tomar decisiones a través de un cliente como una aplicación gráfica.

Sumado a esto, 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 [7].

Como ejercicio práctico, se desarrolla un sistema que simula y controla una calculadora mostrando los resultados en dos displays de 7 segmentos. La solución consiste en una aplicación de diseñada para escritorio y desarrollada en `Python` (`Tkinter`), que se comunica con un microcontrolador `ESP32` mediante una `API REST`. El sistema permite operaciones aritméticas y rutinas de conteo (ascendente y descendente) con paro y reinicio. 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 demostrar cómo la combinación de multiplexación en hardware, un servidor embebido en el microcontrolador y una interfaz de usuario de escritorio permite explorar conceptos de arquitectura cliente-servidor, interacción humano-computadora y control remoto en un entorno controlado y educativo.

2 Estado del arte

Se han desarrollado proyectos de *IoT* referentes a la manipulacion y monitoreo. 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.

También existen propuestas para displays 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 facili-

tar 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,

3 Metodología

3.1 Diseño arquitectónico

El desarrollo siguió una metodología orientada a diseño por capas y por componentes, con énfasis en separación de responsabilidades, pruebas incrementales e integración continua manual. Se trabajó primero el diseño lógico y de interacción (`UI`), posteriormente la lógica de aplicación y finalmente la integración con el sistema embebido (`ESP32`).

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].

La arquitectura se adapto de esta manera: Presentación (`Tkinter`), Aplicación (orquestador, validación y memoria), Dominio (reglas de negocio: formateo y límites 00–99) e Infraestructura (`HTTPClient` y `ESP32`).

La memoria de resultados (historial) se mantiene íntegramente en la aplicación `Python`. El `ESP32` actúa como receptor unidireccional de comandos para actualizar los displays físicos (sin retroalimentación ni estado persistente en el dispositivo). Esta decisión facilita el control y la trazabilidad desde la `PC` y evita inconsistencias por falta de sincronización bidireccional.

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

Resumen de conexión

- Pines *GPIO* (`ESP32`) utilizados:
  - Segmento A: `GPIO21`
  - Segmento B: `GPIO19`
  - Segmento C: `GPIO18`
  - Segmento D: `GPIO5`
  - Segmento E: `GPIO4`

- Segmento F: `GPIO2`
- Segmento G: `GPIO15`
- Cátodo común (Display 1): `GPIO22`
- Cátodo común (Display 2): `GPIO23`

– Interfaz de visualización: displays de siete segmentos de cátodo común conectados directamente a las salidas del `ESP32`. Se emplea multiplexación temporal para mostrar de manera alternada los dos dígitos con persistencia visual.
– Alimentación: fuente de 3.3 V suministrada por el propio `ESP32`. Todos los segmentos y cátodos comparten referencia de tierra (`GND`).

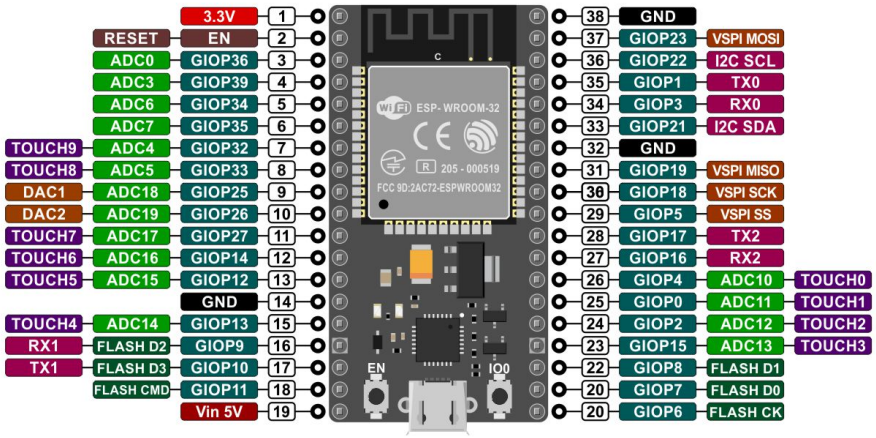


Fig. 1: Esquema de Pinout del controlador *ESP32*

Esquemático y materiales El esquema de conexión detallado y la lista de materiales se muestran en la Figura 2 y la Tabla 1, respectivamente.

Componente	Cantidad	Observaciones
`ESP32` + Shield	1	Con cable de conexión
LEDs (7 segmentos)	28	Del mismo color, para dos displays completos
Estructura para displays	1	Material de elección libre
Cables para protoboard	3 m	Cables macho-macho y macho-hembra
Cautín	1	Para soldadura
Soldadura	—	Cantidad suficiente

Table 1: Lista de materiales para la construcción de los displays y conexión con `ESP32`.

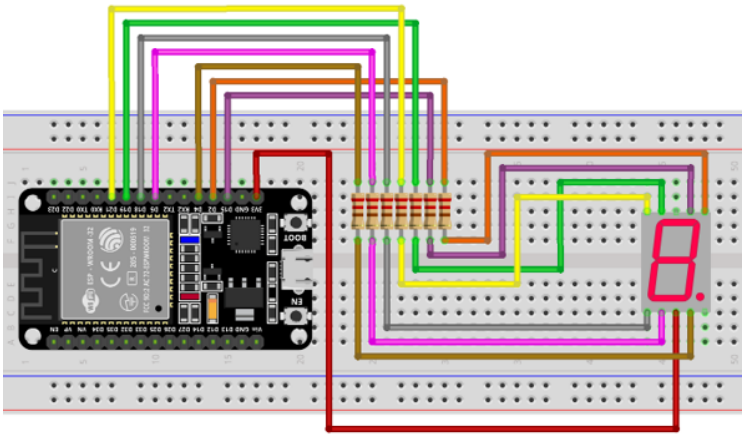


Fig. 2: Esquema de conexion del Display

3.2 Diseño de componentes

El proyecto sigue el paradigma de Programación Orientada a Objetos (POO) para facilitar la separación de responsabilidades, la reutilización y la prueba unitaria. Cada componente encapsula su estado y comportamiento concreto y expone una interfaz clara para el resto del sistema.

Resumen de clases y responsabilidades

- `InterfazPrincipal` (Controller / Orquestador)
  - Crea la única instancia de `tk.Tk()`, configura ventana y layout general.
  - Instancia y conecta los componentes: `Contador`, `WidgetsContador`, `Calculadora` y `HTTPClient`.
  - Responsable de iniciar el bucle principal (`mainloop()`) y de pasar referencias (display, httpClient) para inyección de dependencias.
- `Calculadora` (UI / Lógica de interacción)
  - Gestiona la interfaz de la calculadora: entrada (`Entry`), botones, evaluación de expresiones y el historial (`Listbox`).
  - Aplica reglas de negocio relacionadas al formato (limitar/normalizar a dos dígitos) antes de mostrar o enviar valores.
  - Inserta resultados en el historial (memoria de la aplicación) y expone mecanismos para seleccionar y reutilizar entradas históricas.
  - Recibe por inyección la referencia al display y al `HTTPClient` para delegar el envío de resultados a los displays físicos.
- `Contador` (Componente de display)
  - Representa el display local en la UI (Label grande que simula 7-segmentos).
  - Provee métodos simples: `actualizar_display(valor)`, `obtener_valor()` e `insertar(x,y)` para posicionamiento.

  - Mantiene el valor mostrado en memoria de la aplicación (lectura/escritura por las demás clases).
- `WidgetsContador` (Controles y lógica de conteo)
  - Contiene los botones de Inicio (asc./desc.), Pausa y Reinicio, y la lógica asociada de conteo.
  - Realiza scheduling no bloqueante usando `tkinter.after` para conteos periódicos y para permitir pausa/reinicio sin hilos adicionales.
  - Mantiene el estado de conteo (`contando` o id de `after`) y evita peticiones HTTP en bucle: las solicitudes al `HTTPClient` se envían únicamente en los momentos requeridos (p. ej. al finalizar o al mostrar un valor puntual).
  - Se inyecta la referencia al display para leer/actualizar el valor mostrado.
- `HTTPClient` (Adaptador / Infraestructura)
  - Centraliza la comunicación REST con el ESP32: rutas, timeouts y manejo básico de errores.
  - Ofrece métodos claros: `request_conteo(direccion, valor)` y `request_mostrar_valor(valor)`.
  - Diseñado para inyección y mockeo en pruebas; las llamadas deben invocarse de forma que no bloqueen la UI (gestión interna de timeouts/errores o ejecución en hilos si es necesario).

Notas de diseño relevantes - La memoria de resultados (historial) reside íntegramente en la capa de aplicación (`Python`); el `ESP32` es receptor unidireccional sin almacenamiento ni retroalimentación. - Se garantiza una única `tk.Tk()` y se usan referencias compartidas (display, httpClient) para desacoplar responsabilidades. - Las operaciones periódicas usan `after` para no bloquear la `UI` y las peticiones `HTTP` se limitan a eventos puntuales para evitar tráfico innecesario.

Interacción y patrones aplicados

- **Composición**: `InterfazPrincipal` crea y contiene las instancias de `Calculadora`, `Contador`, `WidgetsContador` y `HTTPClient`. Actúa como orquestador: construye la UI, pasa referencias compartidas (display, httpClient) e inicia el bucle principal (`mainloop()`).
- **Callback / Observer ligero**: la comunicación UI interna se realiza mediante bindings y callbacks de Tkinter (por ejemplo `«ListboxSelect»` ligado a `obtener_seleccion` en el historial). Los controles (botones) invocan métodos expuestos por otros componentes; esto permite notificaciones puntuales sin un sistema de eventos complejo.
- **Adapter**: `HTTPClient` actúa como adaptador de la capa de infraestructura (REST/ESP32). Se inyecta en `Calculadora` y `WidgetsContador` para desacoplar la lógica de la UI del protocolo de envío y facilitar el mockeo en pruebas.
- **Temporización y concurrencia**: las tareas periódicas (conteo ascendente/descendente) usan `tkinter.after` para scheduling no bloqueante en el hilo de la GUI. Las operaciones de red no deben bloquear la UI: `HTTPClient` debería ejecutar las peticiones con timeouts y/o en hilos secundarios (o exponer una API asíncrona) para mantener la capacidad de respuesta.

– **Inyección de dependencias y testabilidad**: pasar referencias (display, httpClient) en los constructores permite sustituir implementaciones por mocks en pruebas unitarias y aislar la lógica de negocio de la infraestructura física.

La FIgura 5 muestra el diagrama de clases utilizado.

3.3 Implementación

Tecnologías y librerías

– Aplicación (PC): `Python 3.x`, `Tkinter` (interfaz gráfica), `requests` (cliente HTTP) y `Pillow` (recursos/imagenes si se requiere). La aplicación centraliza la comunicación mediante un adaptador `HTTPClient` (en `main.py` se instancia con la IP del ESP32: `"192.168.100.99"`).
– Firmware (dispositivo embebido): `ESP32` (Arduino o ESP-IDF) con servidor HTTP REST (p. ej. `WebServer` en Arduino) que recibe comandos para actualizar los displays físicos (2×7-segmentos).
– Entorno de ejecución: estación de trabajo (PC) conectada a la misma LAN/Wi-Fi que el ESP32; la app envía peticiones HTTP hacia el microcontrolador por su dirección IP.

Mecanismos no bloqueantes La aplicación UI mantiene la capacidad de respuesta mediante programación basada en eventos:

– Las temporizaciones y rutinas periódicas (conteo ascendente/descendente, parpadeos visuales) usan `tkinter.Widget.after()` para scheduling en el hilo de la GUI, evitando hilos adicionales para la lógica de UI.
– Las operaciones de red (envío a ESP32) deben ejecutarse con timeouts y/o en hilos secundarios (o mediante una interfaz asíncrona) para no bloquear el hilo principal; `HTTPClient` actúa como punto único para aplicar estas políticas (reintentos, timeouts, manejo de errores) [1].
– En el firmware del ESP32 la lógica se implementa con una máquina de estados no bloqueante (uso de `millis()` o estructuras de estado) y evitando `delay()`, de modo que la atención del servidor HTTP y la actualización de pines/displays coexistan sin retardos perceptibles.

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: bibliotecas `requests`, `WebServer`, `Pillow`.

4.3 Casos de prueba

ID	Acción	Verificar	Registro / Observaciones
CT-01	Presionar “Iniciar conteo ASCENDENTE”.	El display incrementa un paso por segundo; al llegar a 99 muestra “00” y se detiene.	Captura de pantalla y comprobante del paquete HTTP enviado (si aplica).
CT-02	Presionar “Iniciar conteo DESCENDENTE” desde un valor > 0.	Decremento un paso por segundo; al llegar a 00 se detiene sin negativos.	—
CT-03	Iniciar conteo, presionar “Pausar contador”, luego “Iniciar” de nuevo y “Reiniciar”.	Pausas efectivas (sin cambio de valor), reinicio a 00, no múltiples timers activos (sin aceleración).	—
CT-04	Iniciar conteo hasta su fin (o forzar evento que deba enviar valor).	Sólo se envía la petición HTTP prevista (no llamadas repetidas por ciclo).	Revisar logs del servidor o captura de paquetes.
CT-05	Realizar operaciones cuyo resultado exceda dos dígitos (p.ej. $100 + 5$) y operaciones normales.	Valor mostrado y enviado formateado a dos dígitos (00–99) o indicador de overflow (“–”); comprobar inserción en historial y selección de ítems.	—

Table 2: Casos de prueba del sistema de control y visualización de los displays.

4.4 Resultados esperados

- Conteos ascendente/descendente con paso 1 s y paro en 00/99 según dirección.
- Pausa/reinicio deterministas (ningún `after` colgado).
- HTTPClient envía sólo las peticiones puntuales definidas (no envío en cada iteración del `after`).
- Calculadora guarda resultados en historial y limita/sanitiza salidas a 2 dígitos para compatibilidad con displays físicos.

4.5 Problemas conocidos y mitigaciones

- Si el ESP32 no responde, las peticiones HTTP pueden tardar hasta el timeout: registrar y, si procede, ejecutar pruebas con un mock server local para aislar la verificación de la lógica de la app.
- Riesgo de múltiples timers: comprobar que `contando` guarda el id de `after` y que `pausar_contador` cancela correctamente con `after_cancel`.
- Overflow de cálculo: casos extremos deben mapearse explícitamente a indicador válido (`"–"`) antes de enviar a hardware.

5 Resultados

5.1 Observaciones cualitativas

Dado que no se realizaron medidas instrumentadas de rendimiento, los resultados provienen de pruebas funcionales e integración manual de la aplicación Python y del firmware en el ESP32:

Aspecto	**Observaciones / Implementación**
Responsividad de la GUI	La interfaz gráfica permanece responsiva durante la ejecución de las rutinas de conteo gracias al uso de `tkinter.Widget.after()` para temporización, evitando bloqueos del hilo principal.
Conteos ascendente/descendente	Funciona según lo esperado (paso de 1 cada segundo), respetando límites 00–99; al llegar a 99/00 se muestra "00" y se detiene el ciclo.
Control de timers	Se evita la creación de múltiples timers activos. La variable `contando` almacena el estado/ID de `after` y `pausar_contador` cancela el temporizador con `after_cancel`, previniendo aceleraciones o timers colgantes.
Historial de resultados	Un `Listbox` almacena en memoria los resultados de la calculadora; seleccionar un ítem del historial recupera y expone el valor (por ejemplo, en consola) para su reutilización.
Formato y validación de resultados	Los valores de la calculadora se formatean a dos dígitos (00–99) para compatibilidad con los displays; en caso de overflow se muestra un indicador controlado ("–").
Comunicación con ESP32	Se realiza mediante `HTTPClient`; las peticiones se envían de forma puntual para evitar tráfico innecesario. En condiciones de red estable, las peticiones se completan correctamente.
Manejo de errores de red	La aplicación captura excepciones del cliente HTTP y registra mensajes en la consola; actualmente no hay UI dedicada para errores de conectividad.
Limitaciones observadas	La memoria de resultados reside en la aplicación Python; el `ESP32` actúa como receptor unidireccional sin retroalimentación de estado, lo que impide verificar el estado físico de los displays desde la app.

Table 3: Observaciones y verificación de la implementación de la aplicación y el control de displays.

6 Conclusiones

La implementación presentó retos hardware-prácticos: construir y cablear los displays de 7 segmentos exigió atención al tipo (cátodo), multiplexado y alimentación/level-shifting para el ESP32, además de pruebas físicas repetidas para verificar segmentos y continuidad. La ausencia de retroalimentación desde el microcontrolador dificultó la verificación automatizada del estado físico, obligando a depuraciones manuales durante la integración inicial.

En la capa de software los principales problemas fueron de sincronización y diseño: garantizar una única instancia de tk.Tk(), evitar bloqueos de la GUI usando tkinter.after(), asegurar que los valores se limiten y formateen a dos dígitos antes de enviarlos y evitar llamadas HTTP en bucle. La inyección de dependencias (pasar display y HTTPClient) y el uso de mocks facilitaron las pruebas, pero requirieron refactorizaciones para mantener el acoplamiento bajo y la trazabilidad de resultados en la aplicación.

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. Reynoso, C.B.: Introducción a la arquitectura de software. Universidad de Buenos Aires **33**, 11 (2004)

A Diagramas de Capas

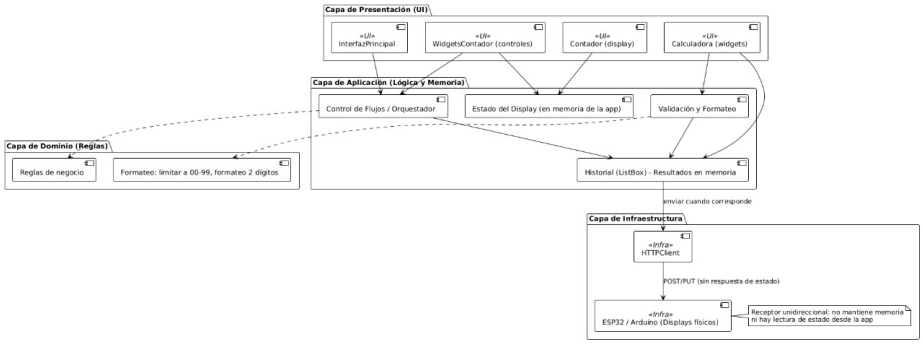


Fig. 3: Diagrama de capas del sistema desarrollado: mapeo de la interfaz gráfica de usuario (`GUI`), el controlador de displays (`ESP32`), las entidades representadas (`Display / Cálculo`) y los adaptadores de comunicación (*HTTPClient / Server*).

B Diagramas de Clases

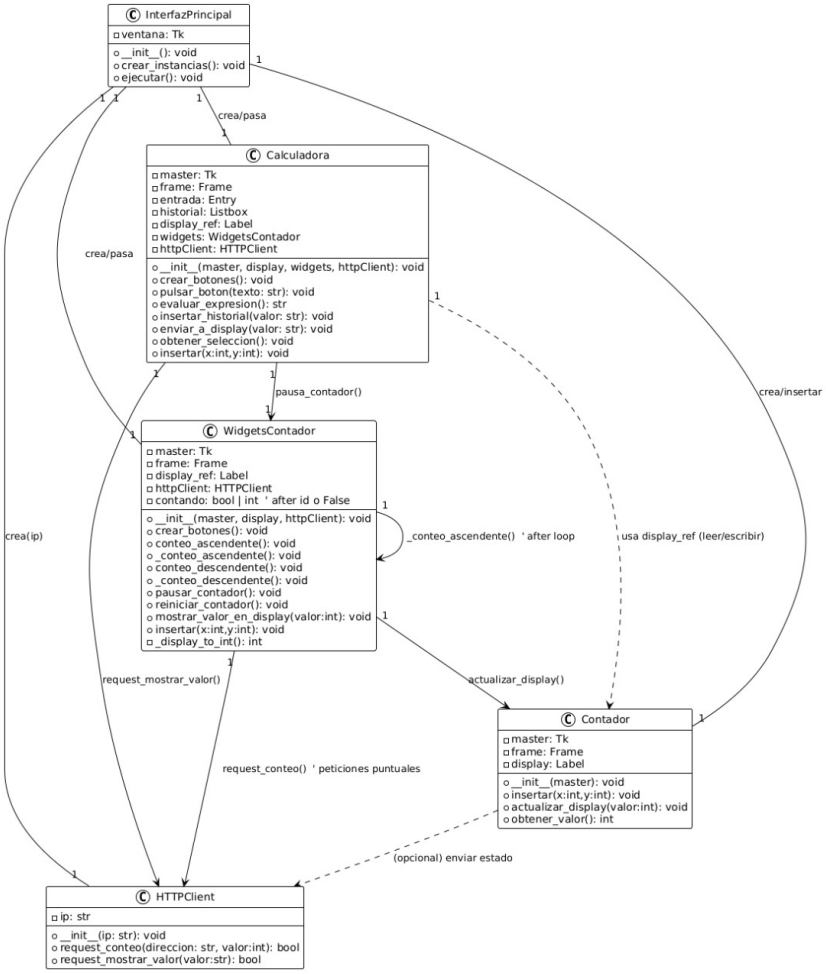


Fig. 4: Diagrama de clases del sistema

C Diagramas de Despliegue

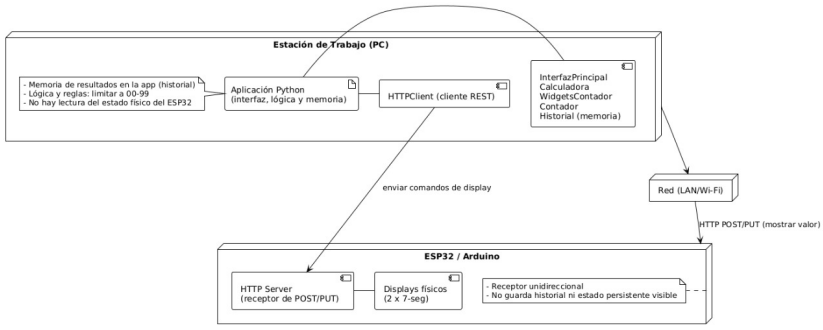
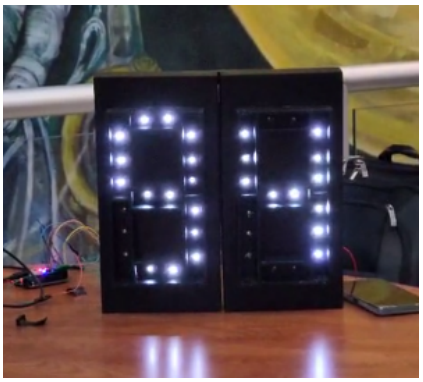


Fig. 5: Diagrama de Despliegue del sistema

D Displays Físicos

(a) Displays apagados



(b) Displays encendidos

Fig. 6: Visualización de los displays de siete segmentos apagados y encendidos, conectados al `ESP32`.

E Aplicación cliente

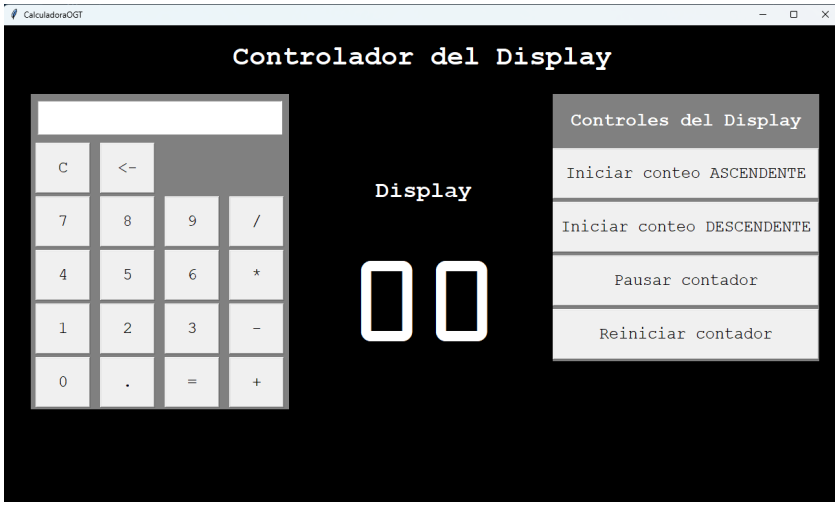


Fig. 7: Aplicacion de Python