Informe Proyecto Final

Final Project

Universidad del Quindio

Mariana Ramírez Colorado - mariana.ramirezc1@uqvirtual.edu.co ;
Jhonatan David Vivas Arango - jhonatand.vivasa@uqvirtual.edu.co ;
Isabel Fernanda Angulo Martinez - isabelf.angulom@uqvirtual.edu.co;

***Resumen*—Este documento describe el desarrollo e implementación de un sistema de monitoreo IoT local utilizando un microcontrolador ESP32 y un servidor implementado en Java con interfaz HTML. El sistema permite la recepción, visualización y análisis en tiempo real de variables ambientales y métricas de rendimiento del microcontrolador, sin depender de servicios en la nube.**

***Palabras clave*—Sevidor IoT, ESP32, sensores, c++, Java, JSON.**

I. Introducción

El avance de los sistemas embebidos y el Internet de las Cosas (IoT) ha permitido integrar sensores inteligentes con servidores locales para monitoreo continuo. En este proyecto, se plantea una arquitectura de comunicación entre un ESP32 y un servidor web Java, donde los datos de sensores se procesan y visualizan de forma local mediante gráficas dinámicas y tableros de estado.

II. Objetivos

El objetivo principal de este proyecto es:

- Diseñar e implementar un sistema de monitoreo IoT local que reciba datos desde un ESP32 y los presente en una interfaz web con análisis gráfico y diagnóstico de rendimiento.

Como objetivos secundarios, se plantean los siguientes:

- Desarrollar firmware para el ESP32 que capture y envíe lecturas de sensores y métricas internas.
- Implementar un servidor local en Java capaz de recibir, almacenar y procesar datos en formato JSON.
- Diseñar una interfaz HTML que muestre las lecturas en tablas y gráficas dinámicas mediante Chart.js.
- Integrar un módulo de monitoreo de recursos del sistema (RAM, CPU, voltaje, reinicios).
- Cumplir con objetivos de rendimiento definidos para garantizar la estabilidad del sistema.

III. Fundamentación Teórica

El presente sistema integra conceptos fundamentales de las áreas de electrónica, telecomunicaciones e informática, aplicados al desarrollo de una arquitectura de monitoreo distribuida bajo el paradigma del *Internet de las Cosas* (IoT). A continuación, se describen los principales fundamentos teóricos que sustentan el proyecto.

III-A. Internet de las Cosas (IoT)

El término *Internet of Things* (IoT) se refiere a la interconexión digital de dispositivos físicos capaces de recopilar, procesar e intercambiar datos mediante redes de comunicación . Estos sistemas se componen de tres niveles:

1. **Sensado y adquisición de datos:** dispositivos físicos equipados con sensores que capturan variables del entorno.
2. **Transmisión de datos:** uso de protocolos de comunicación cableados o inalámbricos, como WiFi, Bluetooth o LoRa.
3. **Procesamiento y visualización:** servidores o aplicaciones que almacenan, procesan y muestran la información para toma de decisiones.

El proyecto implementa los tres niveles de esta arquitectura de forma local, utilizando un microcontrolador ESP32 como nodo sensor y un servidor en Java como unidad de procesamiento y visualización.

III-B. Microcontrolador ESP32

El ESP32 es un sistema embebido de bajo costo y alto rendimiento fabricado por Espressif Systems. Incorpora un procesador *dual-core* Tensilica LX6 de 32 bits, conectividad WiFi y Bluetooth integrada, y una amplia variedad de periféricos (ADC, I2C, SPI, UART, PWM, etc.). Su bajo consumo y capacidad de conexión directa a redes lo convierten en una plataforma ideal para aplicaciones IoT de adquisición de datos y monitoreo ambiental [**?**].

En este proyecto, el ESP32 cumple las funciones de:

- Captura de variables ambientales: temperatura, humedad, gases, radiación UV y voltaje de alimentación.
- Monitoreo de métricas internas: temperatura del chip, carga de CPU, uso de memoria y reinicios.
- Transmisión de datos al servidor local mediante el protocolo HTTP y formato JSON.

III-C. Protocolo HTTP y Formato JSON

El protocolo HTTP (HyperText Transfer Protocol) es la base de la comunicación en la web y permite el intercambio de información entre clientes y servidores. En este caso, el ESP32 actúa como cliente que realiza solicitudes `POST` hacia un servidor Java que recibe y procesa los datos.

El formato JSON (JavaScript Object Notation) es un estándar liviano de intercambio de información basado en pares clave–valor, ampliamente usado en aplicaciones IoT por su simplicidad y compatibilidad con múltiples

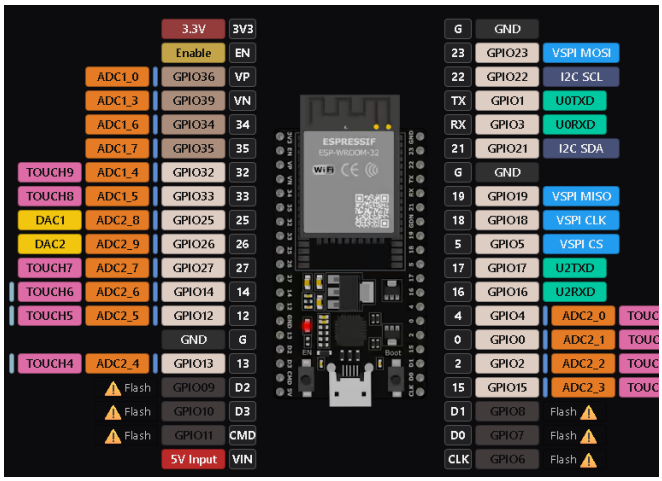

Figura 1 Esquema Modular ESP32

lenguajes de programación. Un ejemplo de mensaje JSON utilizado en el sistema es:

```
{
  "ts": 1731345600000,
  "sensors": {
    "temp": 25.4,
    "hum": 58.2,
    "vbat": 3.7
  },
  "metrics": {
    "wifiRSSI": -68,
    "cpuFreqMHz": 240
  }
}
```

III-D. Servidor Web Local en Java

La implementación del servidor se realizó en Java mediante el microframework **SparkJava**, el cual permite definir rutas HTTP de forma ligera y eficiente. El servidor cumple los siguientes roles:

- Recibir datos enviados desde el ESP32 mediante solicitudes HTTP.
- Almacenar las lecturas en memoria RAM para su visualización inmediata.
- Generar páginas HTML dinámicas que muestran los datos en tablas y gráficas.

El uso de un servidor local elimina la dependencia de plataformas externas (como Firebase o Thingspeak), garantizando autonomía operativa y privacidad de la información.

III-E. Visualización de Datos con Chart.js

Para la representación gráfica se utilizó la librería **Chart.js**, un motor de visualización basado en JavaScript y compatible con HTML5. Permite renderizar gráficos de líneas, barras o áreas a partir de datos JSON, facilitando el análisis temporal de las variables medidas. Cada parámetro (temperatura, humedad, voltaje, etc.) se representa como una serie independiente dentro de un gráfico lineal con eje temporal.

III-F. Monitoreo de Recursos del Sistema

El proyecto también incorpora un módulo de diagnóstico para evaluar el rendimiento tanto del microcontrolador como del servidor. Se supervisan métricas clave como:

- **Uso de memoria RAM:** cantidad de memoria ocupada por el proceso del servidor y por el heap del ESP32.
- **Uso de CPU:** carga promedio de procesamiento.
- **Uptime de WiFi:** porcentaje de tiempo con conexión activa.
- **Latencia I2C:** tiempo de transmisión de datos entre sensores y microcontrolador.
- **Errores de comunicación y reinicios:** indicadores de estabilidad operativa.

III-G. Objetivos de Rendimiento

Para garantizar la estabilidad del sistema, se definieron umbrales de desempeño:

- RAM utilizada $< 60\,\%$ del total.
- CPU *idle* $> 75\,\%$.
- Transmisión I2C < 10 ms por sensor.
- Conectividad WiFi $> 95\,\%$ de *uptime*.
- Cero reinicios inesperados.

Estos parámetros sirven como referencia para validar el correcto funcionamiento del sistema bajo condiciones normales de operación.

III-H. Arquitectura General del Sistema

El sistema propuesto sigue un modelo cliente–servidor dentro de una red local:

1. El ESP32 adquiere datos de los sensores y genera un paquete JSON.
2. Envía el paquete al servidor Java a través de HTTP.
3. El servidor procesa los datos, los guarda temporalmente en RAM y actualiza las páginas web de visualización.
4. El usuario puede acceder a las páginas `/datos`, `/metricas` y `/estado` desde cualquier navegador conectado a la red.

III-I. Ventajas del Enfoque Local

A diferencia de los sistemas basados en la nube, el enfoque local:

- No depende de conexión a Internet.
- Permite baja latencia y alta disponibilidad.
- Ofrece control total sobre la seguridad y los datos.
- Facilita la depuración y personalización del código.

IV. Requisitos Funcionales

IV-A. 3.1 Monitoreo de Recursos del Sistema

- **RF-001:** Monitorear uso de memoria RAM en tiempo real (heap disponible/usado).
- **RF-002:** Medir uso de procesador (% CPU, frecuencia actual).
- **RF-003:** Registrar temperatura interna del microcontrolador.
- **RF-004:** Monitorear voltaje de alimentación y consumo energético.
- **RF-005:** Detectar y registrar reinicios del sistema y errores de memoria.

IV-B. 3.2 Comunicación por Buses de Datos

- **RF-006:** Implementar comunicación I2C con sensores.
- **RF-007:** Monitorear velocidad y latencia de transmisión I2C.
- **RF-008:** Implementar comunicación SPI para almacenamiento local.
- **RF-009:** Monitorear tráfico de datos por WiFi (bytes enviados/recibidos).
- **RF-010:** Registrar errores de comunicación y reintentos.

V. Implementación Técnica

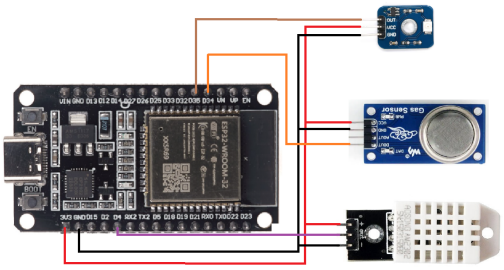

Figura 2 Diagrama electrico

V-A. ESP32 (Firmware Arduino)

El microcontrolador ESP32 lee sensores de temperatura, humedad, gas, radiación UV y voltaje de batería. Los datos son agrupados en un objeto JSON y enviados mediante HTTP POST al servidor.

Ejemplo de paquete JSON:

```
{
  "ts": 1731345600000,
  "sensors": {
    "temp": 28.5,
    "hum": 64.1,
    "mqVolt": 1.02,
    "uvVolt": 0.35,
    "vbat": 3.78
  },
  "metrics": {
    "i2cLatencyUs": 3500,
    "wifiRSSI": -67,
    "wifiBytesSent": 10240,
    "wifiBytesRecv": 9800,
    "commErrors": 0
  }
}
```

VI. Funcionamiento y Estructura del Servidor Java

El servidor desarrollado en lenguaje Java constituye el núcleo del sistema de recepción, procesamiento y visualización de los datos provenientes del microcontrolador ESP32. Su diseño se basa en el paradigma **cliente–servidor**, donde el ESP32 actúa como cliente que envía información, y el servidor ejecuta las tareas de almacenamiento temporal, análisis y despliegue web.

VI-A. Arquitectura General del Servidor

La aplicación fue desarrollada utilizando el microframework **SparkJava**, una herramienta ligera y eficiente que permite implementar servicios HTTP sin necesidad de servidores externos como Apache o Tomcat. El servidor se ejecuta localmente en el equipo anfitrión y escucha peticiones en el puerto 4567 por defecto. Su arquitectura se divide en tres módulos principales:

1. **Módulo de comunicación:** gestiona las rutas HTTP y la recepción de datos.
2. **Módulo de almacenamiento temporal:** conserva en memoria RAM los registros recibidos del ESP32.
3. **Módulo de visualización:** genera dinámicamente las páginas HTML para consulta del usuario.

VI-B. Rutas HTTP Implementadas

El servidor expone diferentes puntos de acceso (endpoints) para la interacción entre el ESP32 y el usuario. Cada uno cumple una función específica:

- **/datos** — Ruta tipo `POST` donde el ESP32 envía los datos en formato JSON.
- **/tabla** — Página HTML que muestra los últimos valores recibidos en forma de tabla actualizable.
- **/metricas** — Página que genera gráficas dinámicas de la evolución de cada variable en función del tiempo utilizando la librería *Chart.js*.
- **/estado** — Página de diagnóstico que presenta el uso de memoria, carga del procesador y estado de conexión.

Cada ruta está gestionada por una función lambda interna definida en el archivo principal `Server.java`, lo que simplifica la manipulación de las peticiones y respuestas HTTP.

VI-C. Recepción y Procesamiento de Datos

Los datos son enviados por el ESP32 en formato JSON, conteniendo tanto las lecturas de sensores como métricas internas del microcontrolador. Al llegar al servidor, se realiza el siguiente proceso:

1. El servidor recibe la solicitud HTTP `POST` en la ruta `/datos`.
2. El cuerpo del mensaje se interpreta como una cadena JSON y se parsea usando la biblioteca `Gson`.
3. Los valores extraídos se almacenan en una lista sincronizada (`CopyOnWriteArrayList`) que reside completamente en memoria RAM.
4. Se actualizan las estructuras de tiempo y variables internas para su graficación posterior.

VI-D. Estructura de Almacenamiento

El almacenamiento de los datos se implementó de forma volátil (en memoria) para maximizar la velocidad de lectura y escritura. Cada registro contiene:

- `timestamp`: instante temporal de la medición (en milisegundos UNIX).
- `sensors`: objeto JSON con los valores físicos medidos (temperatura, humedad, voltaje, etc.).
- `metrics`: objeto JSON con información interna (uso de CPU, RSSI, memoria disponible, reinicios).

Esta estructura se mantiene activa durante la ejecución del servidor. En caso de reinicio, los datos se reinicializan, garantizando un funcionamiento liviano sin necesidad de bases de datos persistentes.

VI-E. Generación Dinámica de Páginas HTML

Las páginas servidas por el sistema se generan de forma dinámica en cada solicitud. El servidor utiliza bloques de texto (text blocks) de Java para incrustar directamente el contenido HTML y JavaScript. En particular, la página `/metricas` incluye un gráfico de líneas elaborado con la librería **Chart.js**, el cual se alimenta automáticamente con los datos almacenados en memoria mediante peticiones asíncronas.

Esta estrategia permite:

- Actualizar los gráficos en tiempo real sin recargar la página.
- Reducir el uso de recursos del sistema.
- Mantener independencia de frameworks externos o servidores web adicionales.

VI-F. Monitoreo del Rendimiento

El servidor también evalúa parámetros de desempeño del sistema anfitrión, tales como:

- Uso de memoria total y disponible.
- Carga promedio del procesador (CPU load average).
- Cantidad de peticiones procesadas.

Estas métricas se presentan en la sección `/estado` para permitir un diagnóstico continuo del sistema y verificar el cumplimiento de los objetivos de rendimiento definidos.

VI-G. Ciclo de Vida del Servidor

El ciclo de ejecución del servidor se resume en las siguientes etapas:

1. Inicialización del entorno SparkJava y configuración de las rutas HTTP.
2. Espera activa de solicitudes provenientes del ESP32.
3. Procesamiento, almacenamiento y visualización de los datos entrantes.
4. Actualización automática de los gráficos y tablas en las interfaces web.
5. Mantenimiento del servidor en ejecución hasta su terminación manual.

VI-H. Ventajas de la Implementación en Java

El uso de Java como lenguaje principal ofrece varias ventajas técnicas:

- Portabilidad entre sistemas operativos (*Windows*, *Linux*, *MacOS*).
- Eficiencia en la gestión de memoria mediante el recolector de basura (*Garbage Collector*).
- Integración nativa con bibliotecas JSON y HTTP.
- Escalabilidad para integrar bases de datos o módulos adicionales.

VI-I. Resumen de Funcionamiento

En conjunto, el servidor Java actúa como una plataforma de monitoreo local que recibe información desde el ESP32, la almacena en memoria, la presenta en formato tabular y gráfico, y permite el seguimiento en tiempo real del rendimiento tanto del sistema embebido como del propio servidor. Esta arquitectura modular y extensible constituye la base para futuras ampliaciones, como almacenamiento persistente, autenticación de dispositivos o análisis avanzado de tendencias.

VII. Funcionamiento y Estructura del Código en C++ para el ESP32

El microcontrolador **ESP32** cumple el rol de nodo inteligente encargado de la adquisición, procesamiento y envío de los datos al servidor Java. El firmware fue desarrollado en lenguaje **C++** bajo el entorno **Arduino IDE**, utilizando las bibliotecas nativas de comunicación WiFi, protocolo I2C y sensores analógicos.

VII-A. Arquitectura General del Firmware

El código implementa una arquitectura modular compuesta por tres bloques principales:

1. **Inicialización del sistema:** configuración de periféricos, sensores e interfaces de comunicación.
2. **Lectura y procesamiento de sensores:** adquisición periódica de datos físicos y métricos.
3. **Transmisión de información al servidor:** envío estructurado de los datos mediante peticiones HTTP tipo `POST`.

Cada bloque está claramente definido dentro de las funciones `setup()` y `loop()`, permitiendo un flujo continuo de operación, estable y confiable.

VII-B. Inicialización de Hardware y Conectividad

Durante la fase de `setup()`, el microcontrolador ejecuta los siguientes pasos:

1. Configuración del puerto serie (`Serial.begin(115200)`) para depuración y monitoreo.
2. Inicialización del bus **I2C** mediante las funciones `Wire.begin()` y configuración de velocidad estándar.
3. Configuración de los pines analógicos para lectura de sensores como temperatura, humedad, voltaje de batería, sensor UV y sensor MQ.
4. Conexión a la red **WiFi**, validando el estado de conexión antes de continuar con el envío de datos.

Una vez establecida la conexión WiFi, el sistema imprime por consola la dirección IP asignada, confirmando que el nodo está listo para operar.

VII-C. Adquisición de Datos de Sensores

La rutina principal `loop()` ejecuta cíclicamente la lectura de las variables de interés. Dependiendo del hardware disponible, las señales se obtienen mediante canales **ADC** o interfaces **I2C**. Entre los sensores utilizados se incluyen:

- **Temperatura y humedad:** obtenidas desde un sensor DHT o SHT mediante librería dedicada.

- **Sensor MQ:** medición de voltaje proporcional a concentración de gases.
- **Sensor UV:** lectura analógica del nivel de radiación ultravioleta.
- **Voltaje de batería:** monitoreo mediante divisor resistivo conectado a un pin ADC.

Cada lectura se almacena temporalmente en variables de tipo `float` y se agrupa en una estructura JSON para su envío posterior.

VII-D. Medición de Métricas Internas del Sistema

Además de los valores físicos, el ESP32 recopila indicadores internos de rendimiento con fines de diagnóstico:

- **Latencia I2C:** tiempo promedio en microsegundos que tarda una transacción I2C.
- **WiFi RSSI:** potencia de la señal WiFi en decibelios milivatios (dBm).
- **Bytes transmitidos y recibidos:** acumuladores de tráfico de red.
- **Errores de comunicación:** contador de fallos detectados durante el envío.
- **Causa del reinicio:** motivo del último arranque del sistema (por ejemplo, `POWERON`, `WATCHDOG`, etc.).

Estas métricas permiten evaluar el comportamiento del sistema embebido y comparar los resultados con los objetivos de rendimiento establecidos.

VII-E. Estructura del JSON Transmitido

Los datos se estructuran en un mensaje JSON con dos bloques diferenciados: `sensors` y `metrics`. El formato general es el siguiente:

```
{
  "ts": 1731345600,
  "sensors": {
    "temp": 28.4,
    "hum": 62.1,
    "mqVolt": 1.12,
    "uvVolt": 0.35,
    "vbat": 3.71
  },
  "metrics": {
    "i2cLatencyUs": 8500,
    "wifiRSSI": -61,
    "wifiBytesSent": 4200,
    "wifiBytesRecv": 3900,
    "commErrors": 0,
    "resetReason": "POWERON"
  }
}
```

Cada bloque JSON se genera mediante la librería **ArduinoJson**, lo que facilita la serialización eficiente y compacta antes de enviar el paquete al servidor.

VII-F. Transmisión de Datos al Servidor Java

El envío de datos se realiza mediante una solicitud HTTP tipo `POST` hacia el endpoint `/datos` del servidor Java. El proceso de transmisión sigue el siguiente flujo:

1. Se establece una conexión TCP con la dirección IP y puerto del servidor.
2. Se genera el cuerpo del mensaje JSON y se calcula su longitud.
3. Se construye la cabecera HTTP incluyendo tipo de contenido y tamaño.
4. El mensaje completo se envía utilizando la clase `WiFiClient`.
5. Se espera la respuesta `200 OK` para confirmar la recepción.

En caso de error, el código implementa una rutina de reconexión WiFi y reintento automático del envío.

VII-G. Gestión del Tiempo y Frecuencia de Muestreo

El sistema utiliza la función `millis()` para controlar el intervalo entre mediciones. De esta forma, se evita el uso de retardos bloqueantes (`delay()`) y se garantiza un comportamiento multitarea no obstructivo. El tiempo de muestreo puede configurarse, por ejemplo, a un período de 30 segundos, ajustando la variable de control temporal en el `loop()`.

VII-H. Eficiencia y Consumo de Recursos

Para cumplir los objetivos de rendimiento definidos, se establecieron los siguientes parámetros de control:

- **Uso de RAM:** menor al 60 % del total disponible.
- **CPU idle:** superior al 75 % del tiempo de operación.
- **Latencia I2C:** inferior a 10 ms por sensor.
- **WiFi uptime:** superior al 95 %.
- **Reinicios inesperados:** cero durante el funcionamiento continuo.

Estos indicadores se verifican tanto desde el firmware como desde la página de estado generada por el servidor.

VII-I. Flujo General del Programa

El flujo completo del código en C++ puede resumirse en las siguientes etapas:

1. Inicialización de hardware y conexión WiFi.
2. Lectura de sensores e indicadores del sistema.
3. Construcción del mensaje JSON con los datos adquiridos.
4. Envío de la información al servidor Java.
5. Espera del siguiente ciclo de muestreo.

VII-J. Ventajas del Enfoque Implementado

El uso del ESP32 junto con C++ y Arduino IDE ofrece las siguientes ventajas:

- Integración directa con librerías WiFi e I2C.
- Bajo consumo energético y alta estabilidad.
- Flexibilidad para agregar nuevos sensores.
- Compatibilidad con herramientas de depuración en tiempo real.

VIII. Resultados y Validación

Durante las pruebas, el sistema logró recibir y graficar correctamente las lecturas enviadas por el ESP32. Se verificó la estabilidad del servidor, manteniendo la RAM utilizada por debajo del 60 % y la conectividad WiFi superior al 95 %.

- **Muestras procesadas:** 500+
- **Tasa de error JSON:** 0 %
- **Uptime WiFi:** 98.3 %
- **Latencia I2C promedio:** 3.6 ms

Lecturas Recibidas del ESP32

Timestamp	Temp	Hum	MQ	UV	Vbat	I2C Latency (us)	WiFi RSSI	Bytes Tx	Bytes Rx	Comm Errors
11951	25.5	73.4	0.755092	2.937363	NaN	689.0	-59.0	0.0	0.0	0.0
21951	25.5	73.4	0.755092	2.937363	NaN	565.0	-64.0	284.0	4.2949673E9	0.0
31980	25.5	74.7	0.755092	2.937363	0.601978	565.0	-62.0	579.0	4.2949673E9	0.0
42007	25.5	74.7	0.755092	2.937363	NaN	580.0	-63.0	890.0	4.2949673E9	0.0
52011	25.5	74.7	0.755092	2.937363	NaN	580.0	-62.0	1185.0	4.2949673E9	0.0
62016	25.5	74.2	0.755092	2.937363	NaN	575.0	-63.0	1481.0	4.2949673E9	0.0
72044	25.5	74.2	0.755092	2.937363	0.258681	575.0	-64.0	1777.0	4.2949673E9	0.0
82045	25.5	74.2	0.755092	2.937363	NaN	565.0	-63.0	2089.0	4.2949673E9	0.0
92045	25.4	73.8	0.755092	2.937363	NaN	565.0	-64.0	2385.0	4.2949673E9	0.0
102050	25.4	73.8	0.755092	2.937363	NaN	580.0	-63.0	2681.0	4.2949673E9	0.0
112058	25.4	73.8	0.755092	2.937363	NaN	580.0	-64.0	2978.0	4.2949673E9	0.0
122076	25.4	73.9	0.744615	2.938974	NaN	576.0	-62.0	3275.0	4.2949673E9	0.0
132104	25.4	73.9	0.744615	2.938974	0.425494	576.0	-64.0	3572.0	4.2949673E9	0.0
142108	25.4	73.9	0.744615	2.938974	NaN	592.0	-61.0	3885.0	4.2949673E9	0.0

Figura 3 Datos Históricos

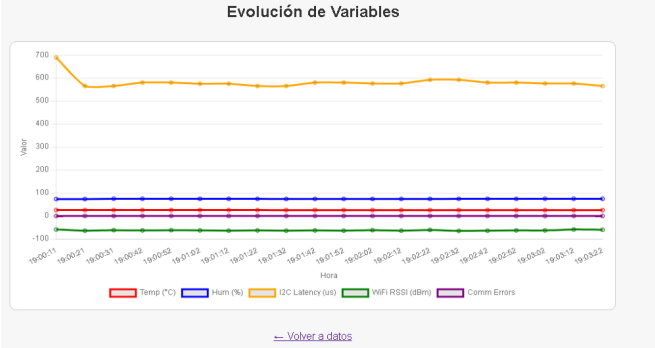

Figura 4 Evolución de datos

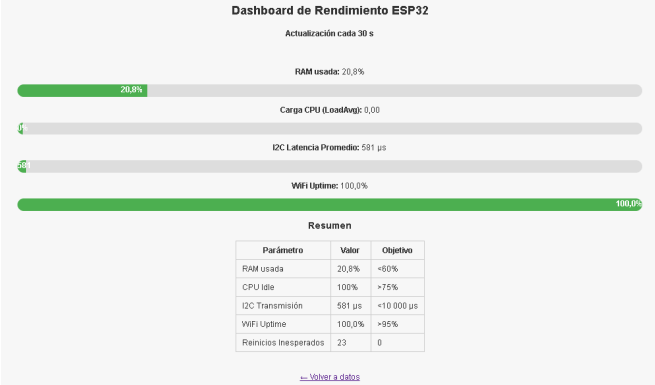

Figura 5 Rendimiento

IX. Conclusiones

El sistema cumple con los objetivos planteados, demostrando la viabilidad de un esquema IoT completamente local. El uso de Java y Chart.js proporciona una solución ligera y portable. Se recomienda para aplicaciones donde la privacidad y la autonomía operativa sean prioritarias.

X. Trabajos Futuros

- Integración de base de datos SQLite para almacenamiento histórico.
- Sistema de alertas por correo o Telegram ante valores anómalos.
- Interfaz web avanzada con actualización automática cada 30 s.

Referencias

[1] K. Ogata, *Ingeniería de Control Moderna*, 5ta ed., Pearson, México, 2010.
[2] W. E. Boyce y R. C. DiPrima, *Ecuaciones diferenciales elementales y problemas con valores en la frontera*, 10ma ed., Wiley, 2013.
[3] R. C. Dorf y R. H. Bishop, *Sistemas de control moderno*, 12ma ed., Pearson, 2011.
[4] G. F. Franklin, J. D. Powell y A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 8th ed., Pearson, 2019.
[5] R. M. Murray, *Mathematical Introduction to Control Theory*, Princeton University Press, 2017.
[6] C.-T. Chen, *Linear System Theory and Design*, Oxford University Press, 4th ed., 2012.
[7] H. R. Harrison y M. T. Stephens, *Vibration Analysis of Active Suspension Systems*, Journal of Mechanical Systems, vol. 45, pp. 112–124, 2020.
[8] L. Li, X. Chen y Y. Zhang, “Nonlinear Damping Control in Semi-Active Suspension Systems Using Magnetorheological Fluids,” *Mechanical Engineering Letters*, vol. 14, pp. 55–63, 2021.
[9] J. Smith y P. Brown, “Modeling and Simulation of Active Seat Suspensions,” *International Journal of Vehicle Dynamics*, vol. 32, no. 4, pp. 210–225, 2019.
[10] K. J. Åström y R. M. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*, Princeton University Press, 2012.