Proyecto Beaglebone

Sistemas Embebidos

Máster Universitario de Informática Industrial y Robótica.

Autor: Álvaro Viña Pérez
Fecha: 25 de noviembre de 2025

Universidad de La Laguna

Índice

1. Descripción general **2**
1.1. Sincronización de código con Visual Studio Code . . . 2
1.2. Configuración de la BeagleBone como Broker MQTT . . . 3
1.2.1. Instalación de Mosquitto . . . 3
1.2.2. Configuración del Broker . . . 4
1.3. Verificación del sistema MQTT . . . 4
1.3.1. Comunicación local: BeagleBone como cliente y broker . . . 4
1.3.2. Integración con el servidor web Flask . . . 4
1.3.3. Comunicación con MQTT Explorer . . . 5
1.4. Desarrollo e integración de la interfaz web . . . 5
1.4.1. Proceso de puesta en marcha y validación . . . 5

2. Código fuente **8**

3. Explicación del código **15**
3.1. Importación de librerías . . . 15
3.2. Inicialización y variables globales . . . 15
3.3. Configuración MQTT . . . 15
3.4. Funciones callback MQTT . . . 16
3.4.1. Callback de conexión . . . 16
3.4.2. Callback de recepción de mensajes . . . 16
3.5. Inicialización del cliente MQTT . . . 16
3.6. Ejecución concurrente del cliente MQTT . . . 17
3.7. Plantilla HTML y estilos CSS . . . 17
3.7.1. Estructura HTML . . . 17
3.7.2. Estilos CSS . . . 17
3.7.3. JavaScript de actualización . . . 17
3.8. Ruta Flask y renderizado . . . 18
3.9. Punto de entrada principal . . . 18
3.10. Flujo de ejecución completo . . . 18

4. Conclusiones **19**

1. Descripción general

Este proyecto constituye una fase preparatoria fundamental para el desarrollo del proyecto final de la asignatura Sistemas Embebidos del Máster Universitario de Informática Industrial y Robótica. La implementación se ha realizado utilizando una placa BeagleBone Black, que desempeña un doble rol en la arquitectura del sistema: actuando simultáneamente como broker MQTT (servidor de mensajería) y como cliente MQTT.

La plataforma BeagleBone Black ha sido configurada con un servidor web basado en Flask, un framework ligero y potente para desarrollo web en Python. Este servidor proporciona una interfaz web intuitiva que permite la visualización en tiempo real de los datos recibidos a través del protocolo MQTT.

El sistema implementa una arquitectura publish-subscribe mediante el protocolo MQTT (Message Queuing Telemetry Transport), ampliamente utilizado en aplicaciones IoT (Internet of Things) por su eficiencia y bajo consumo de recursos. En este esquema, la BeagleBone actúa como:

- **Broker MQTT:** Gestiona la comunicación entre diferentes dispositivos, recibiendo mensajes de los publicadores (publishers) y distribuyéndolos a los suscriptores (subscribers) según los tópicos correspondientes.
- **Cliente MQTT:** Se suscribe a tópicos específicos para recibir datos que posteriormente son procesados y mostrados en la interfaz web.

La interfaz web desarrollada en Flask permite a los usuarios monitorizar de forma remota los datos capturados por el sistema, ofreciendo una solución completa de adquisición, procesamiento y visualización de información en sistemas embebidos. Esta arquitectura proporciona una base sólida para comprender los fundamentos de la comunicación IoT y el desarrollo de aplicaciones web en plataformas embebidas.

1.1. Sincronización de código con Visual Studio Code

Para facilitar el desarrollo y la transferencia de archivos entre el ordenador de desarrollo y la BeagleBone Black, se utilizó la extensión SFTP de Visual Studio Code. Esta herramienta permite sincronizar automáticamente los archivos del proyecto mediante el protocolo SFTP (SSH File Transfer Protocol).

La configuración se realiza mediante un archivo JSON (`sftp.json`) ubicado en la carpeta `.vscode` del proyecto:

```
{
    "name": "BeagleBone",
    "host": "192.168.7.2",
    "protocol": "sftp",
    "port": 22,
    "username": "debian",
    "password": "temppwd",
    "remotePath": "/home/debian/Proyectos",
    "localPath": "./",
    "uploadOnSave": true,
    "syncMode": "update",
    "watcher": {
        "files": "**/*",
```

```
        "autoUpload": true,
        "autoDelete": true
    }
}
```

Los parámetros más relevantes de esta configuración son:

- **host**: Dirección IP de la BeagleBone (192.168.7.2 corresponde a la conexión USB directa).
- **uploadOnSave**: Activa la carga automática de archivos cada vez que se guardan cambios en Visual Studio Code.
- **syncMode**: Modo de sincronización que solo actualiza archivos modificados.
- **watcher**: Observador que detecta automáticamente cambios en los archivos del proyecto y los sincroniza en tiempo real.

Esta configuración agiliza significativamente el proceso de desarrollo, permitiendo editar el código en el entorno familiar de Visual Studio Code y visualizar los cambios inmediatamente en la BeagleBone sin necesidad de transferencias manuales mediante SCP o FTP.

1.2. Configuración de la BeagleBone como Broker MQTT

El primer paso para implementar el sistema de comunicación MQTT consiste en configurar la BeagleBone Black como broker MQTT. Para ello, utilizaremos Mosquitto, un broker MQTT de código abierto ligero y eficiente, ideal para sistemas embebidos.

1.2.1. Instalación de Mosquitto

La instalación de Mosquitto en la BeagleBone Black se realiza mediante los siguientes comandos en el sistema Debian:

```
# Actualizar la lista de paquetes
sudo apt-get update

# Instalar Mosquitto broker y cliente
sudo apt-get install mosquitto mosquitto-clients

# Habilitar el servicio para que inicie automaticamente
sudo systemctl enable mosquitto

# Iniciar el servicio Mosquitto
sudo systemctl start mosquitto
```

Para verificar que el servicio está ejecutándose correctamente:

```
# Comprobar el estado del servicio
sudo systemctl status mosquitto
```

1.2.2. Configuración del Broker

Una vez instalado, es necesario configurar Mosquitto para permitir las conexiones. El archivo de configuración se encuentra en `/etc/mosquitto/mosquitto.conf`. Se debe editar con los siguientes comandos:

```
# Editar el archivo de configuracion
sudo nano /etc/mosquitto/mosquitto.conf
```

Se debe añadir o modificar las siguientes líneas para permitir conexiones:

```
listener 1883
allow_anonymous true
```

Tras modificar la configuración, es necesario reiniciar el servicio:

```
# Reiniciar el servicio Mosquitto
sudo systemctl restart mosquitto
```

1.3. Verificación del sistema MQTT

1.3.1. Comunicación local: BeagleBone como cliente y broker

Una vez configurado el broker, se realizaron pruebas de comunicación local donde la BeagleBone actúa simultáneamente como broker y cliente MQTT. Para ello se utilizaron dos terminales:

Terminal 1 (Suscriptor):

```
mosquitto_sub -h localhost -t test/topic
```

Terminal 2 (Publicador):

```
mosquitto_pub -h localhost -t test/topic -m "Hola desde
    BeagleBone"
```

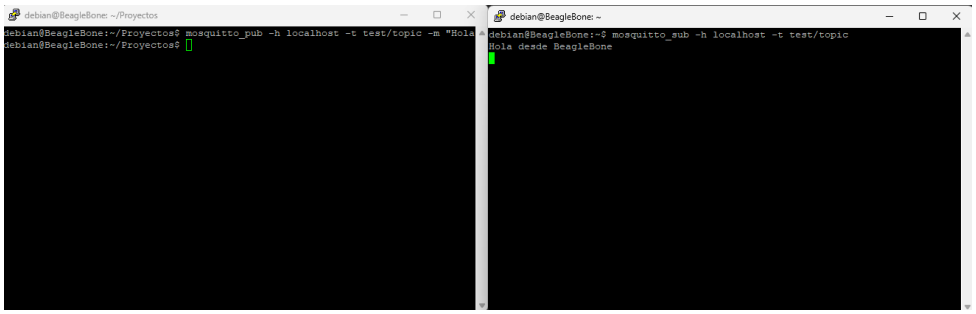

Figura 1: Comunicación MQTT local entre cliente y broker en la BeagleBone

1.3.2. Integración con el servidor web Flask

El siguiente paso consistió en verificar la comunicación entre un cliente MQTT que publica mensajes y el servidor web Flask, que se suscribe a los tópicos correspondientes para mostrar los datos recibidos en tiempo real mediante la interfaz web.

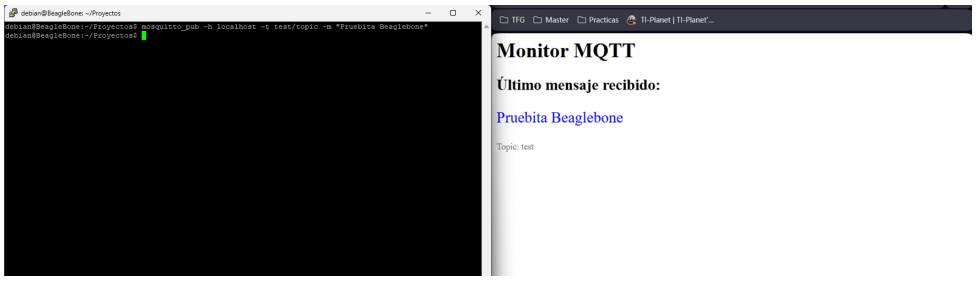

Figura 2: Visualización de mensajes MQTT en la interfaz web Flask desde el navegador

1.3.3. Comunicación con MQTT Explorer

Finalmente, se utilizó MQTT Explorer, una herramienta gráfica para la gestión y monitorización de comunicaciones MQTT, para validar el correcto funcionamiento del sistema completo. Esta herramienta permite visualizar todos los tópicos disponibles, publicar mensajes y suscribirse a tópicos de forma intuitiva.

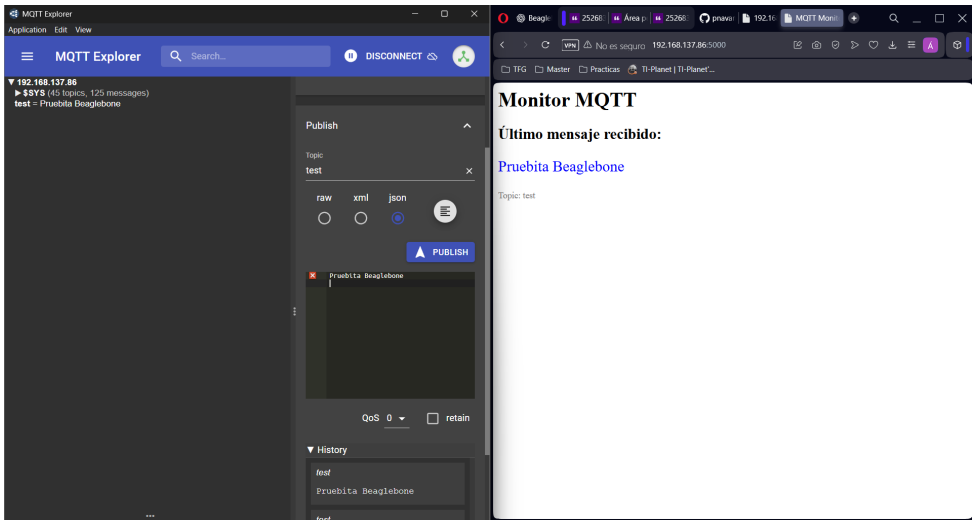

Figura 3: Comunicación entre MQTT Explorer y el broker en la BeagleBone, con visualización simultánea en la interfaz web

Estas pruebas confirmaron el correcto funcionamiento de la arquitectura implementada, validando tanto la capacidad del broker para gestionar múltiples clientes como la integración exitosa con la interfaz web de visualización.

1.4. Desarrollo e integración de la interfaz web

Una vez validada la comunicación MQTT y confirmado el correcto funcionamiento del broker, se procedió al desarrollo de la interfaz web utilizando el framework Flask. Esta interfaz permite visualizar en tiempo real los datos recibidos a través del protocolo MQTT, proporcionando una solución completa de monitorización remota.

1.4.1. Proceso de puesta en marcha y validación

El proceso de validación del sistema completo se realizó siguiendo los siguientes pasos:

Paso 1: Inicialización del servidor Flask El servidor web se ejecuta desde la BeagleBone mediante el siguiente comando en terminal:

```
python -m flask --app Interfaz run --host=0.0.0.0
```

Este comando inicia el servidor Flask en modo accesible desde cualquier dispositivo en la red local. El código completo de la aplicación `Interfaz.py` se detalla en la sección codigo fuente.

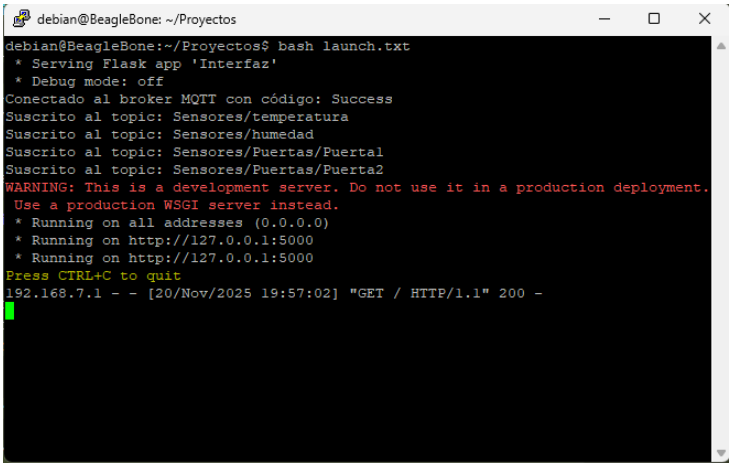

Figura 4: Ejecución del servidor Flask en la BeagleBone

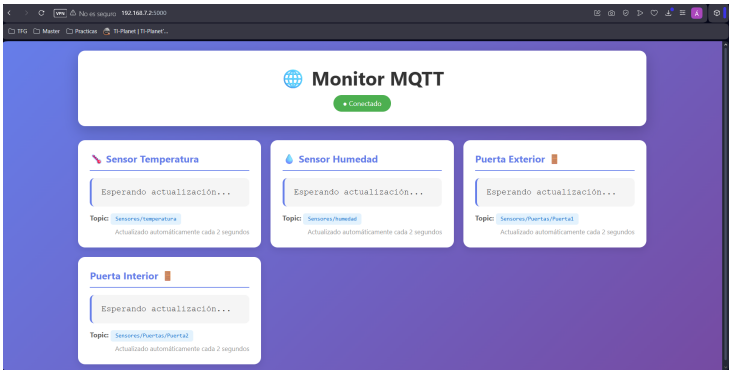

Figura 5: Interfaz web flask

Paso 2: Configuración de MQTT Explorer Se configura MQTT Explorer estableciendo la conexión con el broker mediante la dirección IP de la BeagleBone. Esta herramienta permite gestionar de forma gráfica las publicaciones y suscripciones a los diferentes tópicos MQTT configurados en el sistema.

Paso 3: Publicación de mensajes Utilizando MQTT Explorer, se selecciona uno de los tópicos configurados en el sistema y se publica un mensaje de prueba. La interfaz permite especificar tanto el tópico de destino como el contenido del mensaje a publicar.

Paso 4: Visualización en la interfaz web Finalmente, se observa en la interfaz web cómo el valor del tópico se actualiza automáticamente en tiempo real, reflejando el

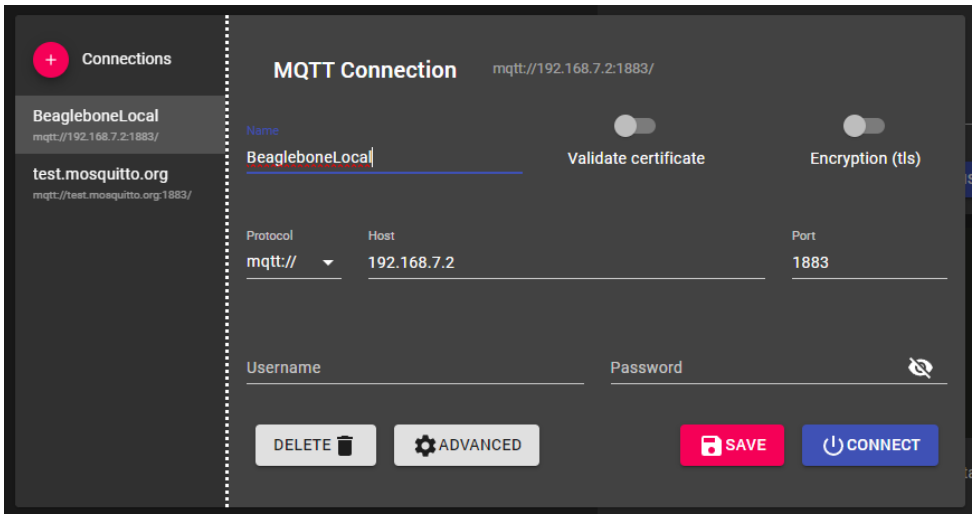

Figura 6: Configuración de la conexión en MQTT Explorer

mensaje publicado. Este comportamiento confirma el correcto funcionamiento de toda la cadena de comunicación: desde la publicación del mensaje, su gestión por el broker, hasta su visualización en la aplicación web.

Este proceso de validación demuestra la correcta integración de todos los componentes del sistema: el broker MQTT, el cliente de publicación, el servidor Flask y la interfaz de usuario web.

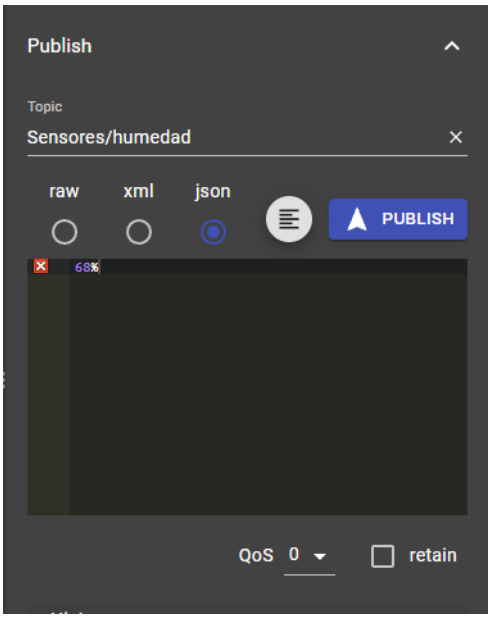

Figura 7: Publicación de mensaje en un tópico mediante MQTT Explorer

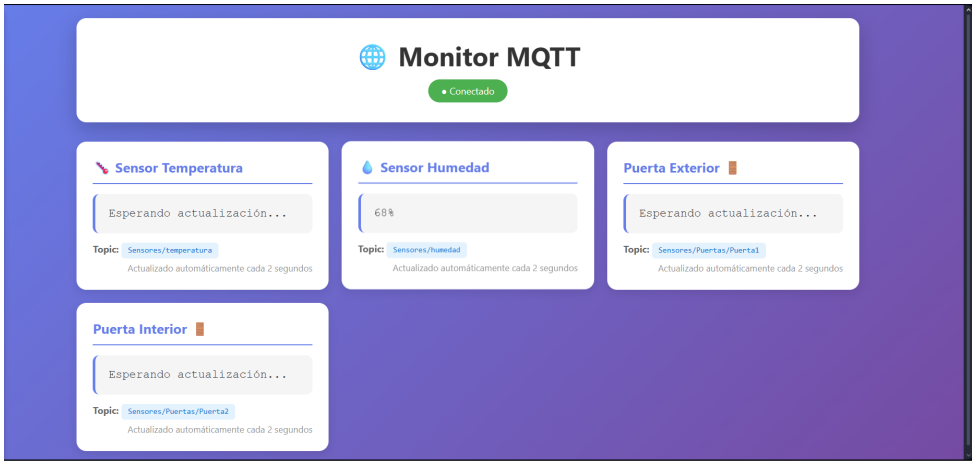

Figura 8: Actualización en tiempo real de los datos en la interfaz web

2. Código fuente

```
1
2  from flask import Flask, render_template_string
3  import paho.mqtt.client as mqtt
4  import threading
5
6  app = Flask(__name__)
7
8  # Variable global para almacenar el último mensaje recibido
9  ultimo_mensaje_temp = "Esperando actualización..."
10 ultimo_mensaje_hum = "Esperando actualización..."
11 ultimo_mensaje_door1 = "Esperando actualización..."
12 ultimo_mensaje_door2 = "Esperando actualización..."
13
```

```
# Configuración MQTT
MQTT_BROKER = "localhost"
MQTT_PORT = 1883
MQTT_TOPIC_TEMP = "Sensores/temperatura"
MQTT_TOPIC_HUM = "Sensores/humedad"
MQTT_TOPIC_DOOR_1 = "Sensores/Puertas/Puerta1"
MQTT_TOPIC_DOOR_2 = "Sensores/Puertas/Puerta2"

# Callback cuando se conecta al broker MQTT
def on_connect(client, userdata, flags, rc, properties=None):
    print(f"Conectado al broker MQTT con código: {rc}")
    client.subscribe(MQTT_TOPIC_TEMP)
    client.subscribe(MQTT_TOPIC_HUM)
    client.subscribe(MQTT_TOPIC_DOOR_1)
    client.subscribe(MQTT_TOPIC_DOOR_2)
    print(f"Suscrito al topic: {MQTT_TOPIC_TEMP}")
    print(f"Suscrito al topic: {MQTT_TOPIC_HUM}")
    print(f"Suscrito al topic: {MQTT_TOPIC_DOOR_1}")
    print(f"Suscrito al topic: {MQTT_TOPIC_DOOR_2}")



# Callback cuando se recibe un mensaje
def on_message(client, userdata, msg):
    global ultimo_mensaje_temp, ultimo_mensaje_hum,
       ultimo_mensaje_door2,ultimo_mensaje_door1

    # Verificar de qué topic viene el mensaje
    if msg.topic == MQTT_TOPIC_TEMP:
        ultimo_mensaje_temp = msg.payload.decode()
        print(f"Mensaje temperatura recibido: {
           ultimo_mensaje_temp}")
    elif msg.topic == MQTT_TOPIC_HUM:
        ultimo_mensaje_hum = msg.payload.decode()
        print(f"Mensaje humedad recibido: {ultimo_mensaje_hum}")
    elif msg.topic == MQTT_TOPIC_DOOR_1:
        ultimo_mensaje_door1 = msg.payload.decode()
        print(f"Mensaje humedad recibido: {ultimo_mensaje_door1}"
           )
    elif msg.topic == MQTT_TOPIC_DOOR_2:
        ultimo_mensaje_door2 = msg.payload.decode()
        print(f"Mensaje humedad recibido: {ultimo_mensaje_door2}"
           )

# Inicializar cliente MQTT
mqtt_client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
mqtt_client.on_connect = on_connect
mqtt_client.on_message = on_message

# Conectar al broker en un hilo separado
def start_mqtt():
```

```
    mqtt_client.connect(MQTT_BROKER, MQTT_PORT, 60)
    mqtt_client.loop_forever()

# Iniciar MQTT en segundo plano
mqtt_thread = threading.Thread(target=start_mqtt, daemon=True)
mqtt_thread.start()

HTML_TEMPLATE = """
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
       scale=1.0">
    <title>Monitor MQTT - Sistema de Sensores</title>
    <style>
        * {
            margin: 0;
            padding: 0;
            box-sizing: border-box;
        }

        body {
            font-family: 'Segoe UI', Tahoma, Geneva, Verdana,
               sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764
               ba2 100%);
            min-height: 100vh;
            padding: 20px;
        }

        .container {
            max-width: 1200px;
            margin: 0 auto;
        }

        header {
            background: white;
            padding: 30px;
            border-radius: 15px;
            box-shadow: 0 10px 30px rgba(0,0,0,0.2);
            margin-bottom: 30px;
            text-align: center;
        }

        h1 {
            color: #333;
            font-size: 2.5em;
            margin-bottom: 10px;
        }

```

```
        .status {
            display: inline-block;
            padding: 8px 20px;
            background: #4CAF50;
            color: white;
            border-radius: 20px;
            font-size: 0.9em;
        }

        .dashboard {
            display: grid;
            grid-template-columns: repeat(auto-fit, minmax(300px,
                1fr));
            gap: 20px;
            margin-top: 20px;
        }

        .card {
            background: white;
            padding: 25px;
            border-radius: 15px;
            box-shadow: 0 5px 15px rgba(0,0,0,0.1);
            transition: transform 0.3s ease;
        }

        .card:hover {
            transform: translateY(-5px);
            box-shadow: 0 10px 25px rgba(0,0,0,0.2);
        }

        .card h2 {
            color: #667eea;
            font-size: 1.3em;
            margin-bottom: 15px;
            border-bottom: 2px solid #667eea;
            padding-bottom: 10px;
        }

        .message-box {
            background: #f5f5f5;
            padding: 20px;
            border-radius: 10px;
            border-left: 4px solid #667eea;
            font-family: 'Courier New', monospace;
            font-size: 1.1em;
            color: #333;
            word-wrap: break-word;
        }

        .info {
            margin-top: 15px;
```

```
            font-size: 0.9em;
            color: #666;
        }

        .topic-badge {
            background: #e3f2fd;
            color: #1976d2;
            padding: 5px 10px;
            border-radius: 5px;
            font-family: monospace;
        }

        .timestamp {
            text-align: right;
            color: #999;
            font-size: 0.85em;
            margin-top: 10px;
        }

        @media (max-width: 768px) {
            h1 {
                font-size: 1.8em;
            }

            .dashboard {
                grid-template-columns: 1fr;
            }
        }
    </style>
    <script>
        // Auto-refresh cada 2 segundos
        setTimeout(function(){
            location.reload();
        }, 2000);
    </script>
</head>
<body>
    <div class="container">
        <header>
            <h1>Monitor MQTT</h1>
            <span class="status">Conectado</span>
        </header>

        <div class="dashboard">
            <div class="card">
                <h2>Sensor Temperatura</h2>
                <div class="message-box">
                    {{ mensaje_temp }}
                </div>
                <div class="info">
```

```
                <strong>Topic:</strong> <span class="topic-
                    badge">{{ topic_temp }}</span>
            </div>
            <div class="timestamp">
                Actualizado automáticamente cada 2 segundos
            </div>
        </div>

        <div class="card">
            <h2>Sensor Humedad</h2>
            <div class="message-box">
                {{ mensaje_hum }}
            </div>
            <div class="info">
                <strong>Topic:</strong> <span class="topic-
                    badge">{{ topic_hum }}</span>
            </div>
            <div class="timestamp">
                Actualizado automáticamente cada 2 segundos
            </div>
        </div>

         <div class="card">
            <h2>Puerta Exterior</h2>
            <div class="message-box">
                {{ mensaje_door1 }}
            </div>
            <div class="info">
                <strong>Topic:</strong> <span class="topic-
                    badge">{{ topic_door1 }}</span>
            </div>
            <div class="timestamp">
                Actualizado automáticamente cada 2 segundos
            </div>
        </div>

         <div class="card">
            <h2>Puerta Interior</h2>
            <div class="message-box">
                {{ mensaje_door2 }}
            </div>
            <div class="info">
                <strong>Topic:</strong> <span class="topic-
                    badge">{{ topic_door2 }}</span>
            </div>
            <div class="timestamp">
                Actualizado automáticamente cada 2 segundos
            </div>
        </div>

```

```
            <!-- Aquí irán más cards cuando agregues los topics -->
        </div>
    </div>
</body>
</html>
"""

@app.route("/")
def index():
    return render_template_string(
        HTML_TEMPLATE,
        mensaje_hum=ultimo_mensaje_hum,
        mensaje_temp=ultimo_mensaje_temp,
        mensaje_door1=ultimo_mensaje_door1,
        mensaje_door2=ultimo_mensaje_door2,
        topic_temp=MQTT_TOPIC_TEMP,
        topic_hum=MQTT_TOPIC_HUM,
        topic_door1=MQTT_TOPIC_DOOR_1,
        topic_door2=MQTT_TOPIC_DOOR_2
    )

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=5000)
```

3. Explicación del código

El código desarrollado integra un servidor web Flask con un cliente MQTT, permitiendo la visualización en tiempo real de datos provenientes de diferentes sensores. A continuación se detalla la estructura y funcionamiento de cada componente:

3.1. Importación de librerías

```
from flask import Flask, render_template_string
import paho.mqtt.client as mqtt
import threading
```

El programa utiliza tres librerías principales:

- **Flask**: Framework web ligero que proporciona las funcionalidades de servidor HTTP y renderizado de plantillas HTML.
- **paho.mqtt.client**: Biblioteca cliente MQTT que permite la suscripción a tópicos y la gestión de mensajes mediante callbacks.
- **threading**: Módulo de Python para la ejecución concurrente, necesario para ejecutar simultáneamente el servidor Flask y el cliente MQTT.

3.2. Inicialización y variables globales

```
app = Flask(__name__)

ultimo_mensaje_temp = "Esperando actualización..."
ultimo_mensaje_hum = "Esperando actualización..."
ultimo_mensaje_door1 = "Esperando actualización..."
ultimo_mensaje_door2 = "Esperando actualización..."
```

Se crea la instancia de la aplicación Flask y se definen cuatro variables globales que almacenarán los últimos mensajes recibidos de cada tópico MQTT. Estas variables actúan como buffer de datos entre el cliente MQTT y la interfaz web.

3.3. Configuración MQTT

```
MQTT_BROKER = "localhost"
MQTT_PORT = 1883
MQTT_TOPIC_TEMP = "Sensores/temperatura"
MQTT_TOPIC_HUM = "Sensores/humedad"
MQTT_TOPIC_DOOR_1 = "Sensores/Puertas/Puerta1"
MQTT_TOPIC_DOOR_2 = "Sensores/Puertas/Puerta2"
```

Se definen los parámetros de conexión al broker MQTT:

- **MQTT_BROKER**: Dirección del broker (localhost indica que el broker se ejecuta en la misma BeagleBone).
- **MQTT_PORT**: Puerto estándar para comunicación MQTT (1883).
- **MQTT_TOPIC_***: Jerarquía de tópicos a los que el cliente se suscribirá.

3.4. Funciones callback MQTT

3.4.1. Callback de conexión

```
def on_connect(client, userdata, flags, rc, properties=None):
    print(f"Conectado al broker MQTT con código: {rc}")
    client.subscribe(MQTT_TOPIC_TEMP)
    client.subscribe(MQTT_TOPIC_HUM)
    client.subscribe(MQTT_TOPIC_DOOR_1)
    client.subscribe(MQTT_TOPIC_DOOR_2)
```

Esta función se ejecuta automáticamente cuando el cliente establece conexión con el broker. El parámetro `rc` (return code) indica el estado de la conexión (0 = éxito). Tras conectarse, el cliente se suscribe a los cuatro tópicos configurados.

3.4.2. Callback de recepción de mensajes

```
def on_message(client, userdata, msg):
    global ultimo_mensaje_temp, ultimo_mensaje_hum,
       ultimo_mensaje_door2, ultimo_mensaje_door1

    if msg.topic == MQTT_TOPIC_TEMP:
        ultimo_mensaje_temp = msg.payload.decode()
        print(f"Mensaje temperatura recibido: {
           ultimo_mensaje_temp}")
    elif msg.topic == MQTT_TOPIC_HUM:
        ultimo_mensaje_hum = msg.payload.decode()
        print(f"Mensaje humedad recibido: {ultimo_mensaje_hum}")
    # ...
```

Esta función se invoca cada vez que llega un mensaje a uno de los tópicos suscritos. El código:

1. Identifica el tópico de origen mediante `msg.topic`.
2. Decodifica el payload (contenido del mensaje) de bytes a string.
3. Actualiza la variable global correspondiente.
4. Registra el evento en consola para depuración.

3.5. Inicialización del cliente MQTT

```
mqtt_client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
mqtt_client.on_connect = on_connect
mqtt_client.on_message = on_message
```

Se crea una instancia del cliente MQTT especificando la versión de la API de callbacks (VERSION2). Posteriormente se asignan las funciones callback definidas anteriormente.

3.6. Ejecución concurrente del cliente MQTT

```
def start_mqtt():
    mqtt_client.connect(MQTT_BROKER, MQTT_PORT, 60)
    mqtt_client.loop_forever()

mqtt_thread = threading.Thread(target=start_mqtt, daemon=True)
mqtt_thread.start()
```

Para que el cliente MQTT y el servidor Flask puedan ejecutarse simultáneamente:

- Se define `start_mqtt()` que establece la conexión y mantiene un bucle infinito escuchando mensajes.
- Se crea un hilo daemon (se cerrará automáticamente cuando termine el programa principal).
- Se inicia el hilo inmediatamente al arrancar la aplicación.

3.7. Plantilla HTML y estilos CSS

La variable `HTML_TEMPLATE` contiene una plantilla HTML completa con:

3.7.1. Estructura HTML

- Diseño responsivo mediante `viewport`.
- Rejilla adaptativa (`grid`) que organiza las tarjetas de sensores.
- Sistema de plantillas Jinja2 (sintaxis {{ `variable` }}) para inyección dinámica de datos.

3.7.2. Estilos CSS

- Degradado de fondo (`linear-gradient`).
- Tarjetas con efecto hover (elevación y sombra).
- Diseño responsivo con `media queries` para dispositivos móviles.
- Sistema de colores consistente para badges y mensajes.

3.7.3. JavaScript de actualización

```
<script>
    setTimeout(function(){
        location.reload();
    }, 2000);
</script>
```

Este script recarga automáticamente la página cada 2 segundos, permitiendo visualizar los datos actualizados sin intervención manual del usuario. Esta aproximación simple es adecuada para este proyecto, aunque en sistemas de producción se recomendaría utilizar WebSockets para actualizaciones en tiempo real sin recargas.

3.8. Ruta Flask y renderizado

```
@app.route("/")
def index():
    return render_template_string(
        HTML_TEMPLATE,
        mensaje_hum=ultimo_mensaje_hum,
        mensaje_temp=ultimo_mensaje_temp,
        mensaje_door1=ultimo_mensaje_door1,
        mensaje_door2=ultimo_mensaje_door2,
        topic_temp=MQTT_TOPIC_TEMP,
        topic_hum=MQTT_TOPIC_HUM,
        topic_door1=MQTT_TOPIC_DOOR_1,
        topic_door2=MQTT_TOPIC_DOOR_2
    )
```

El decorador `@app.route(/")` define la ruta raíz del servidor web. Cuando un cliente accede a esta URL:

1. Se ejecuta la función `index()`.
2. Se renderiza la plantilla HTML inyectando los valores actuales de las variables globales.
3. Se envía el HTML generado al navegador del cliente.

Esta función se ejecuta cada vez que la página se recarga (cada 2 segundos), mostrando siempre los valores más recientes recibidos por MQTT.

3.9. Punto de entrada principal

```
if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=5000)
```

Esta sección se ejecuta únicamente cuando el script se ejecuta directamente (no cuando se importa como módulo):

- **debug=True**: Habilita el modo de depuración (recarga automática ante cambios en el código y mensajes de error detallados).
- **host='0.0.0.0'**: Permite conexiones desde cualquier dirección IP de la red, no solo localhost.
- **port=5000**: Puerto en el que escucha el servidor Flask.

3.10. Flujo de ejecución completo

El funcionamiento integrado del sistema sigue este flujo:

1. Al iniciar el script, se crea el hilo MQTT que se conecta al broker y se suscribe a los tópicos.

2. El servidor Flask inicia en el hilo principal, quedando a la espera de peticiones HTTP.

3. Cuando llega un mensaje MQTT, `on_message` actualiza las variables globales.

4. Un usuario accede a la interfaz web mediante su navegador.

5. Flask renderiza la plantilla HTML con los valores actuales.

6. JavaScript recarga la página cada 2 segundos, repitiendo el proceso.

Esta arquitectura permite una separación clara de responsabilidades: el cliente MQTT gestiona la comunicación con el broker de forma asíncrona, mientras Flask se encarga exclusivamente de servir la interfaz web con los datos más recientes disponibles.

4. Conclusiones

Este proyecto ha permitido implementar con éxito un sistema completo de monitorización basado en el protocolo MQTT, integrando una BeagleBone Black que actúa simultáneamente como broker y cliente MQTT, junto con una interfaz web desarrollada en Flask para la visualización de datos en tiempo real.

Los principales logros alcanzados incluyen:

- **Configuración exitosa del broker Mosquitto:** Se ha establecido un broker MQTT funcional en la BeagleBone Black, capaz de gestionar múltiples clientes y tópicos de forma eficiente.
- **Desarrollo de una interfaz web intuitiva:** La implementación en Flask proporciona una visualización clara y accesible de los datos provenientes de cuatro tópicos diferentes (temperatura, humedad y dos sensores de puerta), con actualización automática cada 2 segundos.
- **Validación exhaustiva del sistema:** Las pruebas realizadas con comunicación local, MQTT Explorer y la interfaz web han confirmado el correcto funcionamiento de todos los componentes de la arquitectura.
- **Arquitectura modular y escalable:** El diseño implementado permite añadir fácilmente nuevos sensores o tópicos MQTT simplemente extendiendo las variables globales y la plantilla HTML.

Durante el desarrollo del proyecto se han adquirido competencias fundamentales en:

- Configuración y administración de sistemas embebidos basados en Linux (Debian).
- Implementación de comunicaciones IoT mediante el protocolo MQTT.
- Desarrollo de aplicaciones web en Python utilizando el framework Flask.
- Programación concurrente mediante threading para gestión de múltiples procesos.
- Integración de diferentes tecnologías (MQTT, Flask, HTML/CSS/JavaScript) en una solución completa.

Este trabajo constituye una base sólida para el desarrollo del proyecto final de la asignatura, habiendo establecido una infraestructura robusta de comunicación y visualización que podrá ser extendida con funcionalidades adicionales como:

- Almacenamiento persistente de datos históricos en bases de datos.
- Implementación de actuadores controlables remotamente vía MQTT.
- Mejora de la interfaz web mediante WebSockets para eliminación de recargas.
- Incorporación de sistemas de autenticación y seguridad en las comunicaciones.
- Análisis y visualización avanzada de datos mediante gráficas temporales.

En conclusión, el proyecto ha cumplido satisfactoriamente con los objetivos planteados, demostrando la viabilidad de implementar sistemas IoT completos en plataformas embebidas como la BeagleBone Black, y proporcionando una experiencia práctica valiosa en el diseño e implementación de arquitecturas publish-subscribe para sistemas embebidos.