EMBEDDED SYSTEMS SPECIALIZATION PROGRAM

Practica 3. CoAP

Sistemas De Comunicación Para Sistemas Embebidos E Internet De Las Cosas

Dorian Ricardo Martinez Preciado

Email: dorian.martinez@iteso.mx

GitHub of the project

https://github.com/DorianMtzP/Practica-3

4/18/2023

Embedded Systems Specialization Program

www.sistemasembebidos.iteso.mx/alumnos

I T E S O A. C., Universidad Jesuita de Guadalajara

Periférico Sur Miguel Gómez Morín #8585, Tlaquepaque, Jalisco, México

Technical Report Number: ESE-O2014-001

Abstract:

This report explains the implementation and testing of CoAP app that links the ESP32 with a Python script to simulate a generator remote annunciator.

Keywords: *Embedded systems, TCP, CoAP, Android, FreeRTOS, IoT, ESP32, IO, Internet of things, ISR*

1. Tabla de contenidos.

2. Introducción

En esta práctica vamos a ejercitar el uso de Wifi y CoAP creando un dispositivo conectado a una red WiFi que interactúa con otros dispositivos en su red local a través de CoAP. También vamos a aprender sobre el protocolo mDNS y lo utilizaremos en nuestro dispositivo para permitir que este pueda ser fácilmente encontrado por otros dispositivos en la misma red.

Para el desarrollo de esta práctica planteáremos un escenario ficticio sobre el desarrollo de un producto. Esto nos ayudara a imaginarnos mejor las características del reto al que nos enfrentamos. Aprender a utilizar el protocolo CoAP.

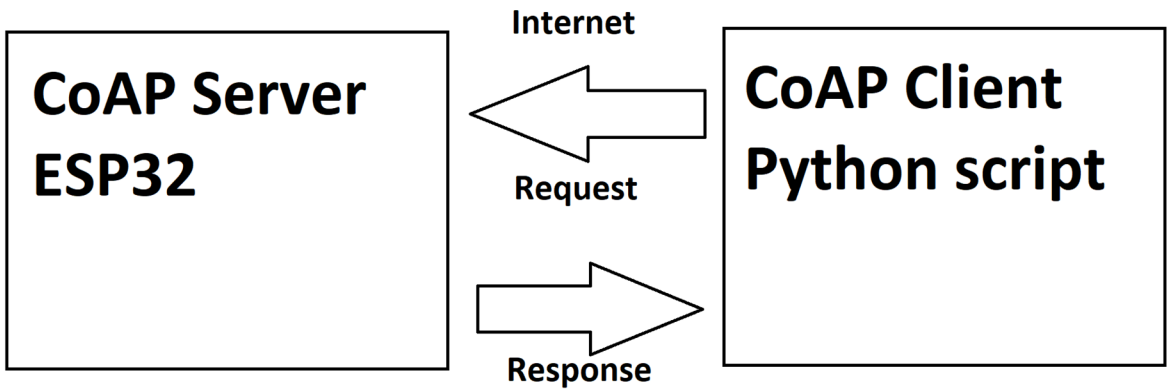

Figura 1. *Diagrama de flujo*

3. Requerimientos

En forma individual:

1. Desarrolla el dispositivo que definiste para tu práctica 2, pero ahora con CoAP en lugar de MQTT.
2. Añade mDNS a tu dispositivo para que pueda ser encontrado fácilmente por otros dispositivos en la red.

Requerimientos del reporte:

- El nombre del reporte debe comenzar con los apellidos.
    - Ej. Perez_Roman_practica1.pdf
- El API de la nueva capa debe estar bien documentada en el reporte.
- Breve descripción del código implementado.
- Diagramas (flujo, estados, secuencia, etc.) según corresponda.
- Una breve descripción de los problemas a los que se enfrentaron.
- Conclusiones.
  El código se debe subir a un repositorio público en GitHub. La dirección del repositorio se debe incluir en el reporte.

4. Descripción Funcional

El proyecto se encuentra almacenado en https://github.com/DorianMtzP/Practica-3

Mi tarea simula un anunciador remoto de un generador eléctrico (planta de luz). Los anunciadores remotos son utilizados para conocer el status de los equipos y poder actuar de manera proactiva sobre posibles alarmas o anormalidades.

El Generador tiene estados los siguientes parámetros.

- Generador Encendido
- Generador Apagado
- Generador en estado de Alarma
- Voltaje de batería se actualiza constantemente.

4.1 Hardware.

Se hace uso de los dos botones y dos LEDs y un Buzzer de la siguiente manera:

Botón 1 (D35) y LED (D13)

- D35 - “Enciende” el generador
- LED D13– Indica que el generador está corriendo.
- Buzzer (D13) – Proporciona indicación auditiva.

Botón 2 (D33) y LED (D15)

- D33 - Activa la alarma
- LED D15– Indica que el generador esta en estado de Alarma.

Arreglo de valores

- Simula cambios en el voltaje de la batería.

4.2 URIs

4.2.1 Configuración de Uri disponibles en el servidor

App Subscriptores

Uri	**GET**	**PUT**	**Payload**
Alarm/status	Disponible	N/A	N/A
Alarm/Reset	N/A	Disponible	Ignorado
Status	Disponible	Disponible	‘e’ -> Enciende Gen ‘a’ -> Apaga Gen
Battery	Disponible	N/A	N/A

4.3 Acciones de URIs

Respuestas del servidor CoAP

URI	Solicitud	Respuesta	Actuador físico
Alarm/status	GET	Regresa estado de las alarmas - Generador en Alarma - Generador OK - No Alarmas	LED y buzzer (D13) - Encendido = Alarma - Apagado = No Alarma
Alarm/Reset	PUT	Resetea el estado de Alarma, **ignora** el Payload	LED y buzzer (D13) se apagan
Status	GET	Regresa el estado del generador - Generador Encendido - Generador Apagado	LED (D15) - Encendido = Gen On - Apagado = Gen Off
Status	PUT Payload = 'e'	Enciende el generador	LED (D15) - Encendido = Gen On
Status	PUT Payload = 'a'	Apaga el generador	LED (D15) - Apagado = Gen Off
Battery	GET	Regresa el voltaje de la Batería	Ninguno.

4.4 Funciones CoAP

Función `static void hnd_alarm_get(coap_resource_t *resource...)`

Descripción Este handler se encarga de responder a las solicitudes GET del URI "Alarm/Status"

Función `static void hnd_reset_put(coap_resource_t *resource...)`

Descripción Este handler se encarga de responder a las solicitudes PUT del URI "Alarm/Reset". También limpia las banderas y apaga los LEDs. **El Payload es completamente ignorado en este handler.**

Función `static void hnd_status_get(coap_resource_t *resource...)`

Descripción Este handler se encarga de responder a las solicitudes GET del URI "Status"

Función `static void hnd_status_put(coap_resource_t *resource...)`

Descripción Este handler se encarga de responder a las solicitudes PUT del URI “Status”. Limpia y establece banderas según sea requerido al igual que enciende o apaga LED dependiendo del payload

Función `static void hnd_battery_get(coap_resource_t *resource...)`

Descripción Este handler se encarga de responder a las solicitudes GET del URI “Battery”, regresa un valor de un arreglo predefinido de valores, esto con el fin de simular una lectura verdadera del voltaje de una batería.

4.6 Funciones del GPIO

Función `static void IRAM_ATTR gpio_isr_handler(void* arg)`

Descripción Agrega las interrupciones a un Queue el cual será atendido posteriormente por la tarea del GPIO

Función `static void configure_IO(void)`

Descripción Configura las dos salidas y dos entradas digitales. Las entradas son configuradas como interrupciones.

Función `static void gpio_task_inputs(void* arg)`

Descripción Tarea encargada de procesar el queue generado por las interrupciones de los I/O. Aquí se establecen y limpian banderas para simular el estado “Alarma” y “Run” del generador.

4.6 Función principal

Función `static void coap_example_server(void *p)`

Descripción Proporciona un nombre al dispositivo por medio de MDNA, obtiene la IP utilizando DHCP, posteriormente se queda en espera de solicitudes de CoAP a las cuales responde.

4.5. MDNS.

MDNS permite asignar un nombre y una lista de servicios/puertos a un dispositivo, el cual hará un broadcast a la red para informar que está disponible.

La funcionalidad MDNS esta aplicada al programa de la practica 3. Se puede observar su broadcast mediante el programa Bonjour.

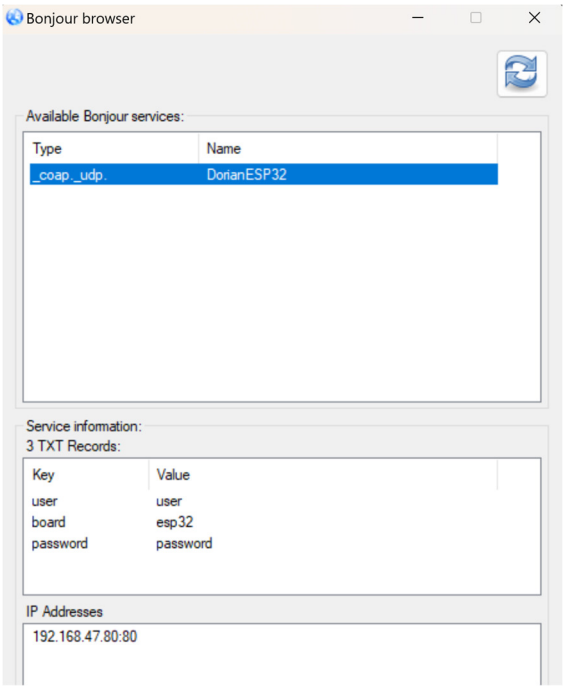

Figura 2. MDSN broadcast in Bonjour browser

También es posible utilizarla mediante el script de Python, simplemente es necesario sustituir la IP con el nombre del dispositivo.

**Para que esto funcionara fue necesario utilizar el puerto 80 en vez del 5683.

```
test_coap.py
1  import logging
2  import asyncio
3
4  from aiocoap import *
5
6  # put your board's IP address here
7  #ESP32_IP = "192.168.47.80"
8  ESP32_IP = "DorianESP32"
```

Figura 3. Dispositivo MSDN Python script.

Anadir MDNS a el ESP32 es muy sencillo, bastan añadir las 8 líneas de código que se muestran a continuación.

```
1. …
2.      mdns_init();
3.      mdns_hostname_set("DorianESP32");
4.
5.      mdns_txt_item_t serviceTxtData[3] = {
6.          {"board", "esp32"},
7.          {"user", "user"},
8.          {"password","password"}
9.      };
10.     mdns_service_add("DorianESP32", "_coap","_udp",80, serviceTxtData, 3);
11. …
```

La línea 10 contiene los parámetros que se incluirán en el broadcast (*Figura 2. MDSN broadcast in Bonjour browser*).

5. Resultados

Se logro implementar exitosamente la funcionalidad del protocolo MDNS y CoAP (en modo servidor) en la tarjeta ESP32.

5.1 Alarm/status.

Se realizo un **Alarm/status GET**, se presionó el botón para activar Alarma y se realizó otro **Alarm/status GET**

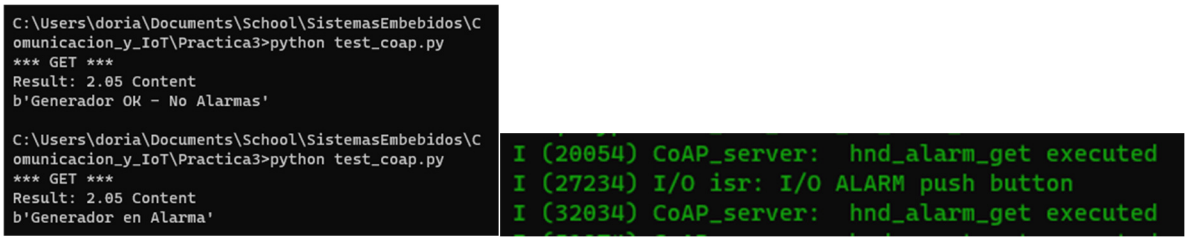

Figura 4. URI Alarm/status.

5.2 Alarm/Reset.

Partiendo de la prueba anterior, en estado de Alarma, se realizó un **Alarm/Reset PUT**, con esto se desactivo la alarma, posteriormente se realizo un **Alarm/Status GET** para confirmar.

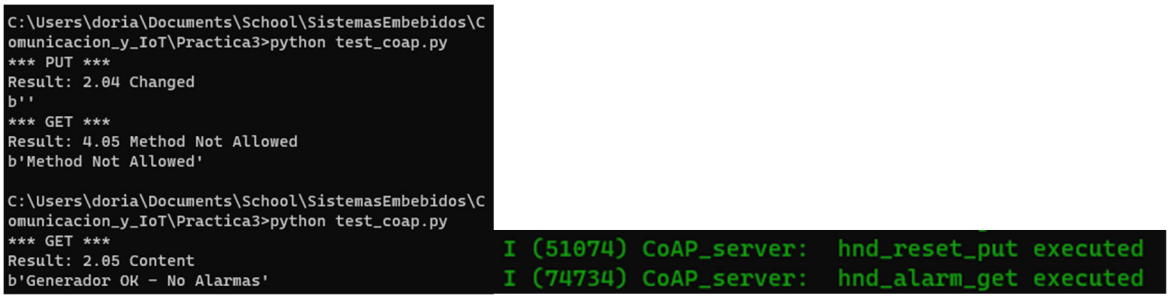

Figura 5. URI Alarm/Reset.

5.3 Status.

Partiendo de la prueba anterior, en estado de Generador Apagado, se realizó un **Status GET**, posteriormente se presionó el botón para encender el generador y se volvió a realizar realizo un **Status GET** para confirmar.

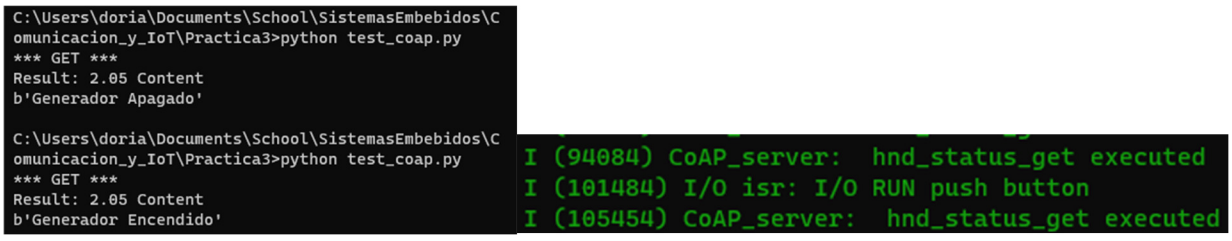

*Figura 6. URI Status **GET.***

Partiendo del paso anterior, en estado de Generador Encendido, se realizó un **Status PUT** con el **Payload ‘a’**, para apagar el generador. Se repitió el proceso, pero esta vez con un **Payload ‘e’** para Encender nuevamente el generador.

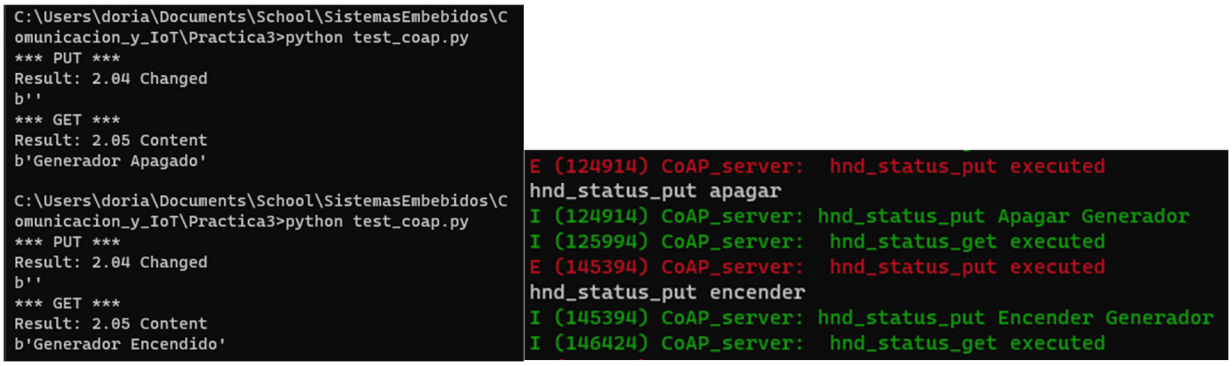

*Figura 7. URI Status **PUT**.*

En el caso del handler PUT, solo se esta comparando el primer carácter de la cadena del payload, por lo cual enviar un “encender” o un “apagar” (o cualquier palabra que inicie con ‘a’ o ‘e’) tendrá el mismo efecto. Si el payload no empieza con dichas vocales el se regresa un mensaje de error en el LOG del ESP32, cabe destacar que esto no interrumpe el funcionamiento ya que es meramente informativo.

5.4 Battery.

Partiendo de la prueba anterior, se realizaron una serie de **Battery GET**, para obtener los distintos voltajes simulados de la batería.

Figura 8. URI Battery.

5.5 ESP32 Log Completo.

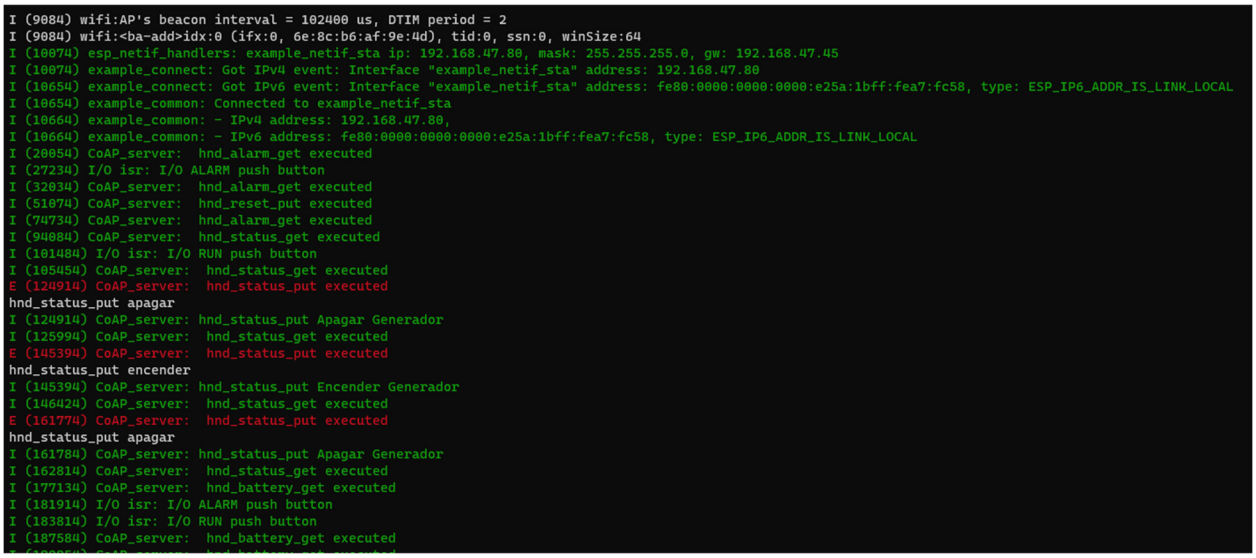

Figura 9. Log ESP32 completo.

6. Conclusiones

Esta práctica fue sencilla, pero a la vez retadora. La parte de los protocolos y su implementación fue bastante rápida y fluida. Lo que hizo que la practica fuera retadora, e incluso estresante, fueron las limitantes del entorno de desarrollo del ESP32. Tuve muchísimos problemas con la plataforma:

- El ejemplo de MDNS ya no existe en la versión 5 de los ejemplos. Esto se pudo solucionar hasta que el profesor nos ayudó en clase.
- Una vez que el MDNS estaba funcionando y que podía ver el broadcast en Bonjour, el script de Python fallaba diciendo que la IP no era válida, después de muchos intentos, encontré que utilizar el puerto 80 solucionaba el problema.
- El no tener un "debugger" como tal, complica las cosas. Tuve problemas haciendo la identificación de Payloads del PUT, utilicé *strcmp* y revisé que el tamaño de los datos recibidos coincidiera con la longitud de cadena que esperaba, ambos coincidían, más sin embargo *strcmp* indicaba que no era así. El *printf* no imprimía consistentemente, supongo que es por que FreeRTOS le da la prioridad más baja. De momento no imprimía nada y después imprimía varias líneas. Comencé a utilizar el ESP_LOGI el cual fue mucho más consistente, el único inconveniente es que no permite la impresión de variables que no sean del tipo "int". Al final del día, después de hacer muchas pruebas con métodos diferentes "a ciegas", termine haciendo una comparación de un solo carácter, lo cual funciono y soluciono el problema.
- El agregar GPIO con interrupciones fue mucho mas sencillo de lo que esperaba, punto para el ESP32

En general, me gustó mucho esta práctica. Creo que CoAP es mucho más sencillo de utilizar que MQTT, aunque también parece que es menos robusto. Me hubiera gustado que la practica fuera entre dos dispositivos ESP32 en vez de mediante una PC.