INFORME DEL PROYECTO 1 – TELEMÁTICA

JUAN PABLO POSSO VILLEGAS

JUAN CAMILO GÓMEZ JIMÉNEZ

DOCENTE: JUAN CARLOS MONTOYA MENDOZA

UNIVERSIDAD EAFIT

MEDELLÍN

2025

Informe de desarrollo – Cliente CoAP ESP32 (Wokwi + DHT22)

El objetivo principal de este trabajo fue diseñar e implementar un cliente CoAP funcional, ejecutado sobre una placa ESP32 simulada en la plataforma Wokwi, capaz de comunicarse con un servidor remoto mediante el protocolo CoAP.

El cliente debía enviar lecturas reales de temperatura y humedad obtenidas desde un sensor DHT22 virtual, empaquetarlas en formato JSON y transmitirlas a un servidor alojado en AWS EC2, utilizando el puerto UDP 5683.

Decisiones de diseño

Desde el inicio decidimos estructurar el proyecto en módulos separados, cada uno con una responsabilidad muy específica. Esto nos ayudó a mantener el código ordenado, fácil de leer y modificar.

Modularidad y cohesión: Separamos el código en varios archivos .h y .cpp, de modo que cada módulo cumpliera una función concreta:

SensorProvider se encarga exclusivamente de leer los datos del sensor DHT22.

CoapMessage maneja la construcción del mensaje CoAP, con su encabezado, token, opciones y payload.

CoapClient gestiona el envío y recepción de paquetes por UDP, incluyendo la retransmisión y la espera de ACK.

Config.h centraliza toda la configuración, como la IP del servidor, los intervalos de envío y los puertos.

Finalmente, el archivo principal .ino coordina todo el flujo: conexión WiFi, lectura de sensor, construcción del JSON y envío periódico del POST al servidor.

Esta división nos permitió lograr alta cohesión y bajo acoplamiento.

Uso del protocolo CoAP completo: Implementamos los tipos de mensajes CON (confirmable) y ACK, respetando el modelo de fiabilidad de CoAP. Además, se incluyeron las opciones más relevantes: Uri-Path y Content-Format, usando el formato 50 (application/json).

Uso del sensor DHT22 virtual: Inicialmente el cliente enviaba datos simulados, pero después integramos el sensor DHT22 real de Wokwi para enviar valores de temperatura y humedad reales. Esto hizo la simulación mucho más cercana a un caso de uso real.

Dificultades y problemas encontrados

Durante el desarrollo surgieron varios obstáculos, tanto técnicos como de configuración:

Problemas con las librerías del DHT22: Al principio, Wokwi no reconocía las librerías DHT sensor library y Adafruit Unified Sensor, mostrando errores al compilar. La solución fue usar el archivo libraries.txt para que Wokwi las instalara automáticamente. Fue un detalle pequeño, pero crítico.

Simulación sin Serial Monitor: Una de las limitaciones de Wokwi fue que a veces el monitor serial no aparecía, por lo que no podía ver las impresiones de depuración (Serial.println). Esto me obligó a improvisar una solución visual (el parpadeo del LED) para verificar el funcionamiento.

Sincronización y tiempos de respuesta: CoAP trabaja sobre UDP, por lo que tuvimos que manejar los reintentos y los timeouts manualmente. Implementamos un mecanismo de retransmisión exponencial: si el servidor no respondía con ACK, el cliente reintentaba el envío aumentando el tiempo de espera.

Aprendizajes obtenidos

Este proyecto nos permitió afianzar varios conceptos tanto de redes como de programación estructurada, además comprendimos a fondo el funcionamiento interno del protocolo CoAP, especialmente su estructura de mensajes, los tipos de confirmación y el manejo de retransmisiones.

Aprendimos a estructurar un proyecto modular con buena separación de responsabilidades, lo cual mejora la mantenibilidad y escalabilidad del código.

Profundicé en la configuración de servidores EC2 y en temas de redes (apertura de puertos, reglas UDP, direccionamiento público).

Conclusiones

Diseñar este cliente CoAP fue un reto técnico interesante que combinó programación embebida, redes y protocolos de IoT. La simulación en Wokwi demostró ser una herramienta muy potente para validar conceptos sin hardware físico, y el hecho de haber conectado el cliente con un servidor real en la nube me permitió entender mejor cómo funcionan los sistemas distribuidos en entornos reales.

Informe de desarrollo – Servidor y Cliente CoAP

Durante el desarrollo de este proyecto implementamos un sistema de comunicación basado en el protocolo CoAP (Constrained Application Protocol), compuesto por un servidor en lenguaje C y un cliente en Python.

El objetivo fue simular la interacción entre un sensor IoT (ESP32) y un servidor remoto mediante sockets UDP Berkeley, cumpliendo con los requisitos del protocolo, los tipos de mensajes (CON, NON, ACK, RST) y los métodos básicos (GET, POST, PUT, DELETE).

Decisiones de diseño

Desde el inicio decidimos estructurar el servidor de forma modular, separando claramente las responsabilidades: manejo de red, almacenamiento de datos, registro de eventos y construcción de mensajes CoAP. Esto nos permitió mantener el código legible y fácil de depurar.

Lenguaje C para el servidor: Escogimos C porque el proyecto exigía utilizar la API de sockets Berkeley directamente, sin librerías externas. Esto nos obligó a trabajar a bajo nivel con recvfrom() y sendto() para implementar la comunicación UDP, y a manejar manualmente la estructura de los mensajes CoAP (encabezado, token, opciones y payload).

Cliente en Python: Implementamos el cliente en Python para facilitar las pruebas.

El cliente procesa la respuesta del servidor, mostrando el código CoAP (por ejemplo, 2.04 Changed o 4.04 Not Found) y el contenido del payload.

Logger integrado: Agregamos un logger en el servidor que registra cada petición entrante y su respuesta, tanto en la terminal como en un archivo de log (server.log), con la fecha y hora exactas.

Este logger fue muy útil para validar el flujo del protocolo, especialmente durante las pruebas con múltiples clientes y cuando algo fallaba.

Concurrencia: En sistemas Linux y en AWS EC2 el servidor utiliza hilos (pthread) para manejar varios clientes de forma simultánea. En Windows, por simplicidad, las peticiones se procesan secuencialmente. Esta decisión mantuvo la compatibilidad cruzada sin perder funcionalidad.

Almacenamiento en memoria: Diseñamos una estructura tipo tabla (arreglo de pares clave/valor) donde cada recurso (por ejemplo, /sensor) guarda el último JSON recibido con los datos del sensor. Esto nos permitió soportar los métodos POST, PUT y DELETE sin necesidad de base de datos externa.

Dificultades encontradas

Compatibilidad entre Windows y Linux: Tuvimos que adaptar el código para que funcionara tanto en entornos Windows (MSYS2, MinGW) como en Linux (AWS). En especial, la gestión de sockets (SOCKET vs int) y funciones como closesocket() o close() nos generaron varios errores de compilación al inicio.

Estructura del protocolo CoAP: Entender cómo se codifican los mensajes CoAP fue una de las partes más complejas. Tuvimos que estudiar el formato de encabezado (versión, tipo, token length, código, message ID) y las opciones URI-Path. Implementar manualmente el parseo y la reconstrucción de cada campo nos tomó tiempo y muchas pruebas.

Mensajes “Not Found” (404): Al principio el método GET no devolvía correctamente los datos. Descubrimos que estábamos interpretando mal el campo de opciones (URI-Path), lo que hacía que las claves almacenadas no coincidieran con las consultadas.

Después de ajustar la función parse_options() y build_path(), logramos que los recursos se identificaran correctamente por su nombre.

Sin librerías externas: Trabajar sin librerías como libcoap o microcoap significó que todo, desde el parseo de bytes hasta la generación del token, lo tuviéramos que hacer manualmente. Esto aumentó la carga de trabajo, pero nos permitió entender a fondo cómo funciona CoAP a nivel de protocolo.

Aprendizajes obtenidos

Este proyecto fue una experiencia muy completa. Más allá de cumplir los requisitos, logramos entender cómo se construyen los protocolos de comunicación desde cero.

Comprendimos en detalle la estructura de los mensajes CoAP, cómo se relacionan los tokens entre cliente y servidor y cómo se define un flujo de petición/respuesta eficiente.

Aprendimos a utilizar correctamente sockets UDP Berkeley, manejando tanto la recepción y envío de datos como las estructuras sockaddr_in y el control de errores.

Mejoramos nuestras habilidades de debugging y diseño modular, separando responsabilidades en funciones pequeñas y específicas.

Finalmente, pudimos ejecutar exitosamente el servidor y el cliente, observar las peticiones en la terminal y almacenar los datos de los sensores simulados.

Este proyecto nos dejó una comprensión mucho más profunda sobre la comunicación en entornos IoT y sobre cómo los protocolos como CoAP logran eficiencia en dispositivos con recursos limitados.