Comparing HTTP and CoAP Protocols NodeMCU ESP8266

Course Title: *Internet of Things*

Course Code: *CSE 406*

Section: *01*

Submitted by

Name: *Shairin Akter Hashi*

ID: *2022-2-60-102*

Date: *10/08/2025*

Submitted to

Dr. Raihan Ul Islam (DRUI)

Associate Professor, Department of Computer Science and Engineering

East West University

Other Group Members

Name	***ID***
Zihad khan	*2022-2-60-107*
Nushrat Jaben Aurnima	*2022-2-60-146*
Shahrukh Hossain Shihab	*2022-1-60-372*

Table of Contents

Introduction

We compare two commonly used IoT protocols, which are **HTTP** and **CoAP** to see how much overhead they add to tiny messages on constrained devices.

- **HTTP** (Hyper Text Transfer Protocol) is a request/response, text-based protocol that runs over TCP. It's great for REST APIs and tooling, but its headers and TCP semantics can be heavy for small sensor updates due to multiple handshakes.
- **CoAP** (Constrained Application Protocol) is a lightweight, binary, REST-like protocol that runs over UDP. It keeps headers tiny and supports confirmable/non-confirmable messages, making it a better fit for low-power, lossy networks.

We capture traffic for the protocols on an ESP8266 setup and, for each one, measure **payload size**, **protocol header size**, and **total on-wire size** in Wireshark. By repeating each test and averaging results, we assess efficiency and discuss trade-offs in reliability, latency, and suitability for resource-constrained IoT deployments.

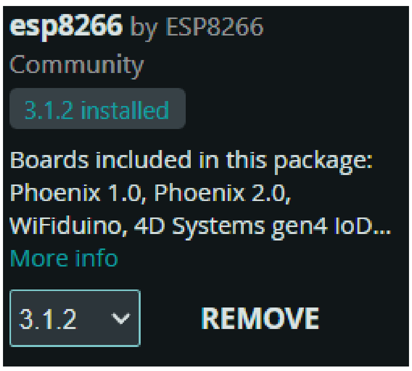
Required Materials

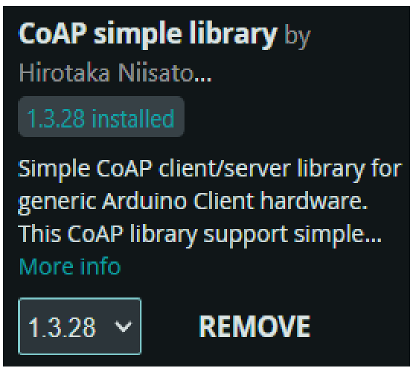
Hardware

- **Microcontroller board:** NodeMCU **ESP8266** and micro-USB cable.
- **Host machine:** Laptop running Flask (server) and Wireshark (capture)
- **Network:** 2.4 GHz Wi-Fi access point; ESP8266 **and** PC on the **same subnet**

Software & Tools

- **Arduino IDE**:
  - **ESP8266 core** installed via Boards Manager
  - **CoAP Simple** installed via Library Manager





- **Python 3** (3.10+) with packages:
  - **flask** (HTTP server)
  - **aiocoap** (CoAP client)
- **Wireshark** (GUI)

Firmware / Code we used

- **CSE406_HTTPbasicClient.ino** – ESP8266 sketch that performs an **HTTP/1.1 GET** to the Flask server
- **main.py** – Flask app serving GET /rest on port **5000** and responding with a tiny JSON body
- **CSE406_CoapServer.ino** – ESP8266 CoAP server (listens for PUT "1/0")
- **CoapClient.py** – Python CoAP client that sends **PUT** to the ESP's CoAP resource

What we captured and saved

- For **each protocol** we produced
  - **.pcapng** capture file
  - **Three screenshots:** Follow-TCP-Stream (or CoAP exchange), **request details**, **response details**

Tasks and Implementation

We built two real-world IoT exchanges to study their cost on the wire. First, set up an **HTTP** interaction where an ESP8266 issues a GET to a Flask server and receives a 200 OK. Then, configure a **CoAP** endpoint on the ESP8266 that accepts a **confirmable (CON) PUT** to toggle an onboard LED. For both cases, **Wireshark** was used to capture traffic, to measure **payload size**, **protocol header size**, and **total on-wire bytes**, so we can compare how much overhead HTTP and CoAP add to small sensor-style messages.

Task 1: Setup and Packet Capture

HTTP

Implementation

- **Server (Laptop):** Run main.py (Flask) → serves GET /rest on port **5000**, returns JSON {"message":"GET request received"}.
- **Client (ESP8266):** Flash CSE406_HTTPbasicClient.ino with SSID/PW and PC IP http://<PC-IP>:5000/rest.
- **Wireshark:** Start capture on Wi-Fi interface; applied display filter **http**.
- **Trigger:** Let ESP send one GET /rest (auto after Wi-Fi join).

Results

Clean request/response observed each run; saved as `http_2.pcapng.`

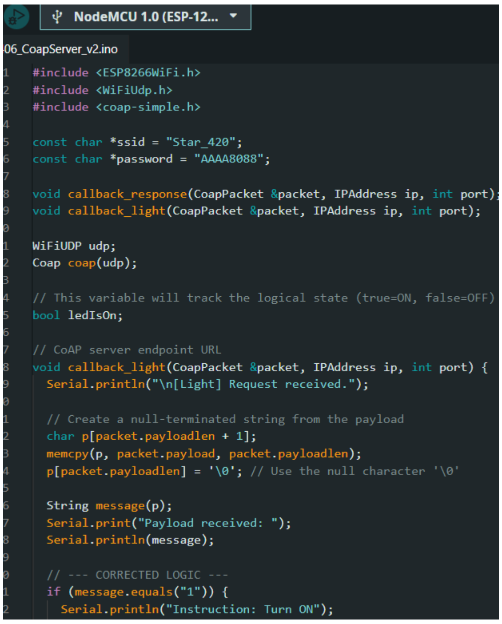

Wireshark · Follow TCP Stream (tcp.stream eq 3) · http_2.pcapng

```
GET /rest HTTP/1.1
Host: 192.168.0.101:5000
User-Agent: ESP8266HTTPClient
Accept-Encoding: identity;q=1,chunked;q=0.1,*;q=(
Connection: keep-alive
Content-Length: 0


HTTP/1.1 200 OK
Server: Werkzeug/3.1.3 Python/3.13.2
Date: Sun, 17 Aug 2025 13:56:27 GMT
Content-Type: application/json
Content-Length: 35
Connection: close

{"message":"GET request received"}
```

CoAP

Implementation

- **Board & Wi-Fi.** Connect the NodeMCU ESP8266, select the correct board/port, and enter SSID/password in the sketch.

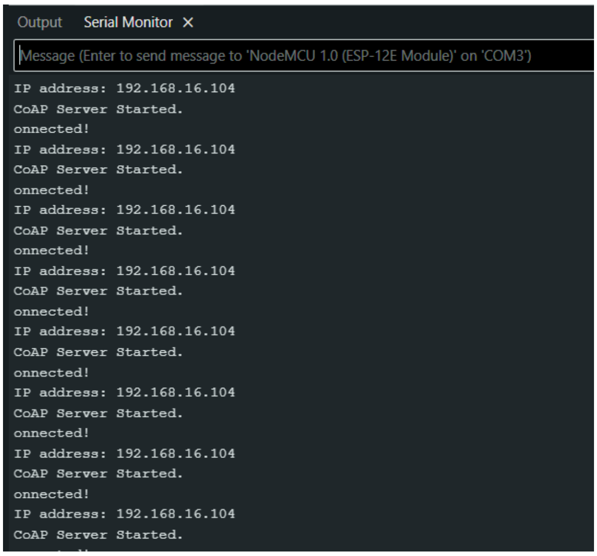



```cpp
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>
#include <coap-simple.h>

const char *ssid = "Star_420";
const char *password = "AAAA8088";

void callback_response(CoapPacket &packet, IPAddress ip, int port);
void callback_light(CoapPacket &packet, IPAddress ip, int port);

WiFiUDP udp;
Coap coap(udp);

// This variable will track the logical state (true=ON, false=OFF)
bool ledIsOn;

// CoAP server endpoint URL
void callback_light(CoapPacket &packet, IPAddress ip, int port) {
  Serial.println("\n[Light] Request received.");

  // Create a null-terminated string from the payload
  char p[packet.payloadlen + 1];
  memcpy(p, packet.payload, packet.payloadlen);
  p[packet.payloadlen] = '\0'; // Use the null character '\0'

  String message(p);
  Serial.print("Payload received: ");
  Serial.println(message);

  // --- CORRECTED LOGIC ---
  if (message.equals("1")) {
    Serial.println("Instruction: Turn ON");
```

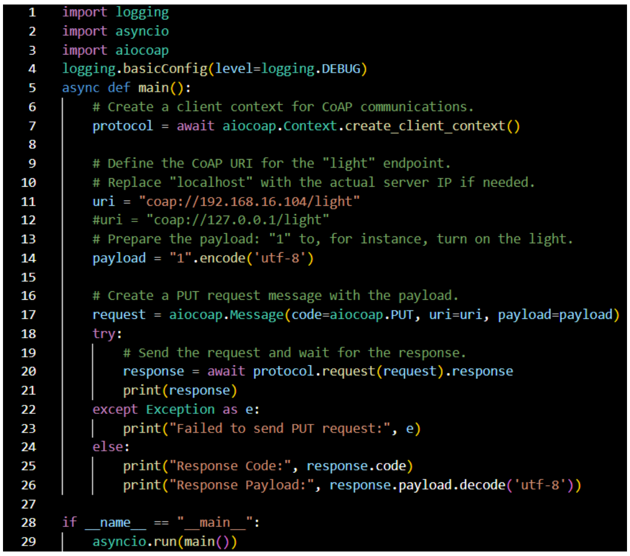

```cpp
  Serial.println(p);
}

void setup() {
  Serial.begin(115200);
  delay(100);
  Serial.println("\nBooting...");

  // Setup the built-in LED
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, HIGH); // Start with the LED OFF
  ledIsOn = false;                 // Set initial state to OFF

  // Connect to WiFi
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nWiFi connected!");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP().toString());

  // Setup CoAP server
  coap.server(callback_light, "light");
  coap.response(callback_response);
  coap.start();
  Serial.println("CoAP Server Started.");
}

void loop() {
  coap.loop();
  // We can add a small delay to prevent the loop from running too fast
  delay(10);
}
```

- **CoAP server.** Upload CoAP sketch to ESP8266. In the Serial Monitor confirm Wi-Fi connect and note the device IP (e.g., **192.168.16.104**).

- **CoAP client.** Put the ESP IP into CoapClient.py.



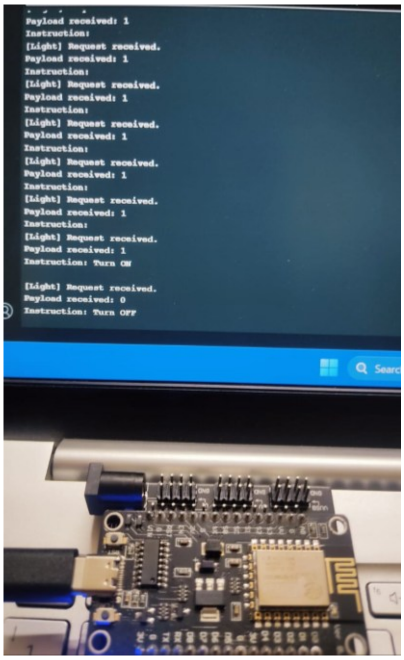

```
IP address: 192.168.16.104
CoAP Server Started.
onnected!
IP address: 192.168.16.104
CoAP Server Started.
onnected!
IP address: 192.168.16.104
CoAP Server Started.
onnected!
IP address: 192.168.16.104
CoAP Server Started.
onnected!
IP address: 192.168.16.104
CoAP Server Started.
onnected!
IP address: 192.168.16.104
CoAP Server Started.
onnected!
IP address: 192.168.16.104
CoAP Server Started.
onnected!
IP address: 192.168.16.104
CoAP Server Started.
```



```python
import logging
import asyncio
import aiocoap
logging.basicConfig(level=logging.DEBUG)
async def main():
    # Create a client context for CoAP communications.
    protocol = await aiocoap.Context.create_client_context()

    # Define the CoAP URI for the "light" endpoint.
    # Replace "localhost" with the actual server IP if needed.
    uri = "coap://192.168.16.104/light"
    #uri = "coap://127.0.0.1/light"
    # Prepare the payload: "1" to, for instance, turn on the light.
    payload = "1".encode('utf-8')

    # Create a PUT request message with the payload.
    request = aiocoap.Message(code=aiocoap.PUT, uri=uri, payload=payload)
    try:
        # Send the request and wait for the response.
        response = await protocol.request(request).response
        print(response)
    except Exception as e:
        print("Failed to send PUT request:", e)
    else:
        print("Response Code:", response.code)
        print("Response Payload:", response.payload.decode('utf-8'))

if __name__ == "__main__":
    asyncio.run(main())
```

- **Generate traffic**

    - Send **PUT** with payload **“0”** → server replies **2.04 Changed**, LED **OFF**.

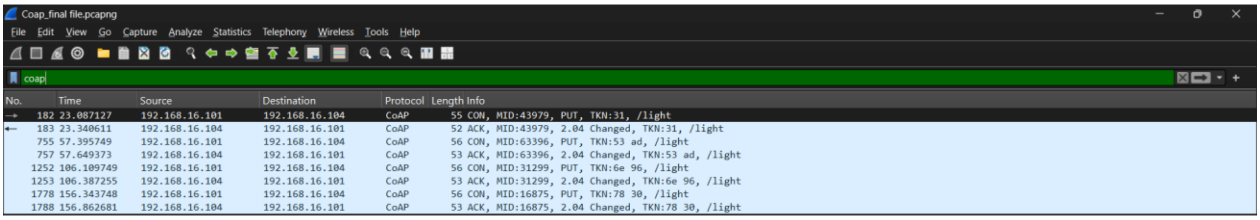


    - Send **PUT** with payload **“1”** → server replies **2.04 Changed**, LED **ON**.

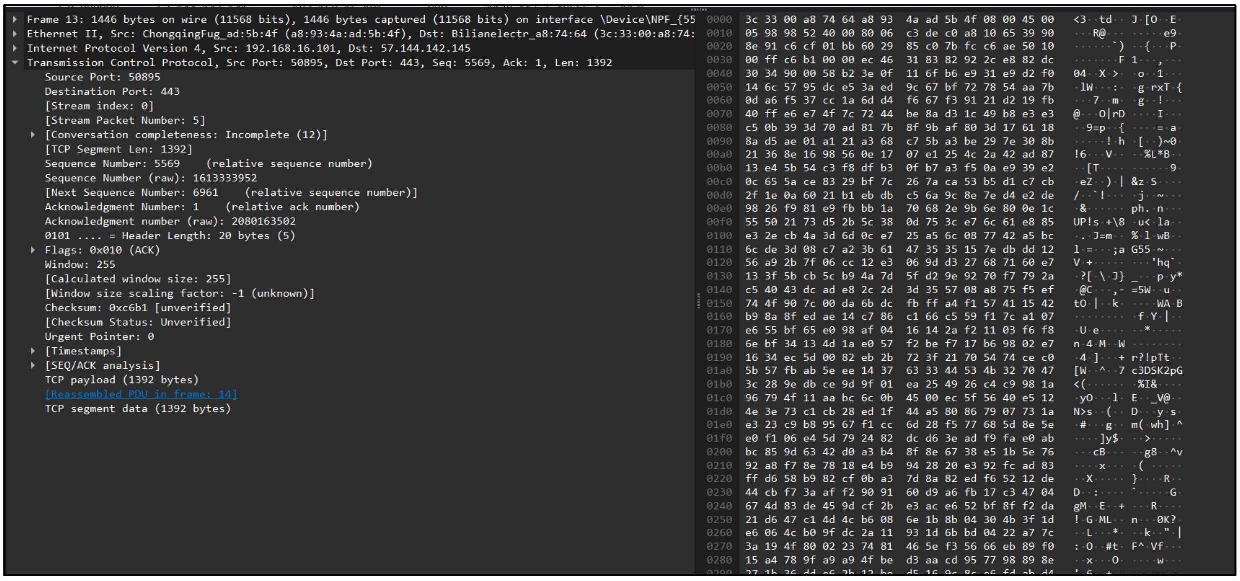
- **Capture** filter using Wireshark.

Results

- Successful **Confirmable (CON)** CoAP **PUT** exchanges observed with responses **2.04 Changed**, matching the LED state. We can see the states of light (ON or OFF) in wire shark after applying **coap** filter.



- Captures saved with CoAP filter applied showing request/response and logs.



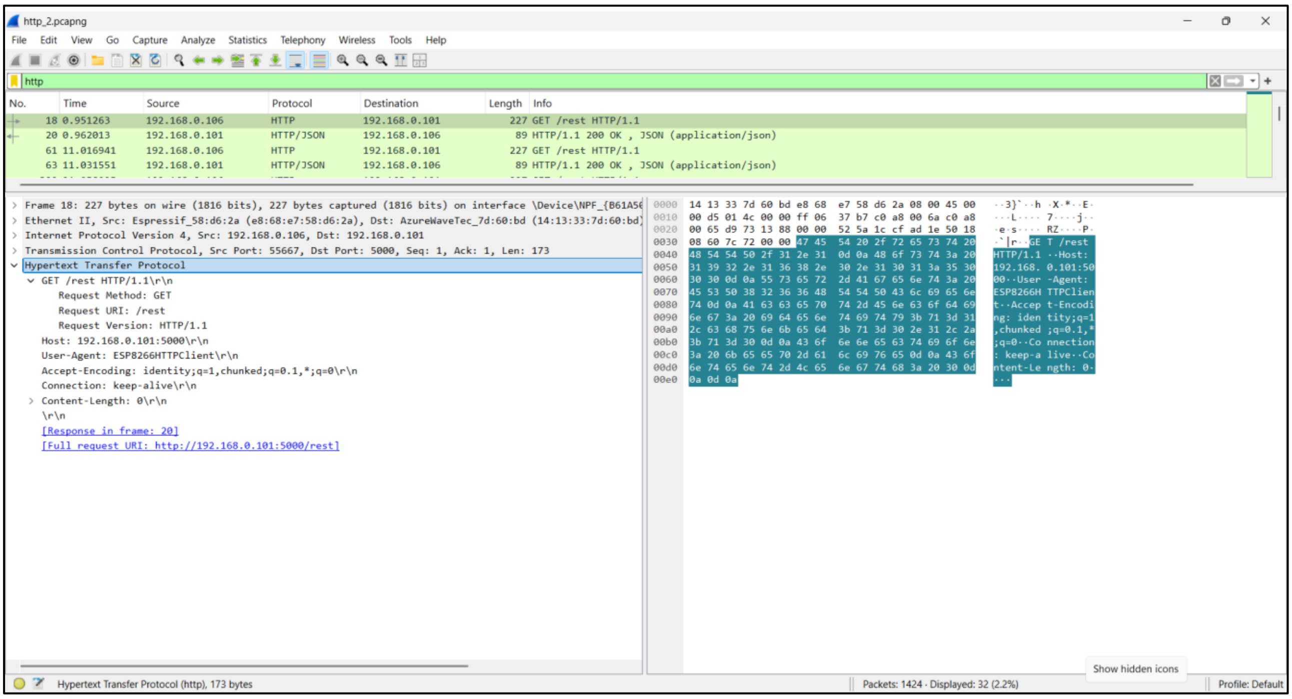
Task 2: Analyze Packet Details

Here, we measured **payload size**, **HTTP header sizes**, and **total on-wire bytes** for a single **HTTP GET → 200 OK** pair, **and** identified **CoAP** message structure (Version/Type/Code/Token/Options). Also measured **payload**, **header**, and **on-wire sizes** for a single **confirmable PUT → 2.04 Changed** exchange.

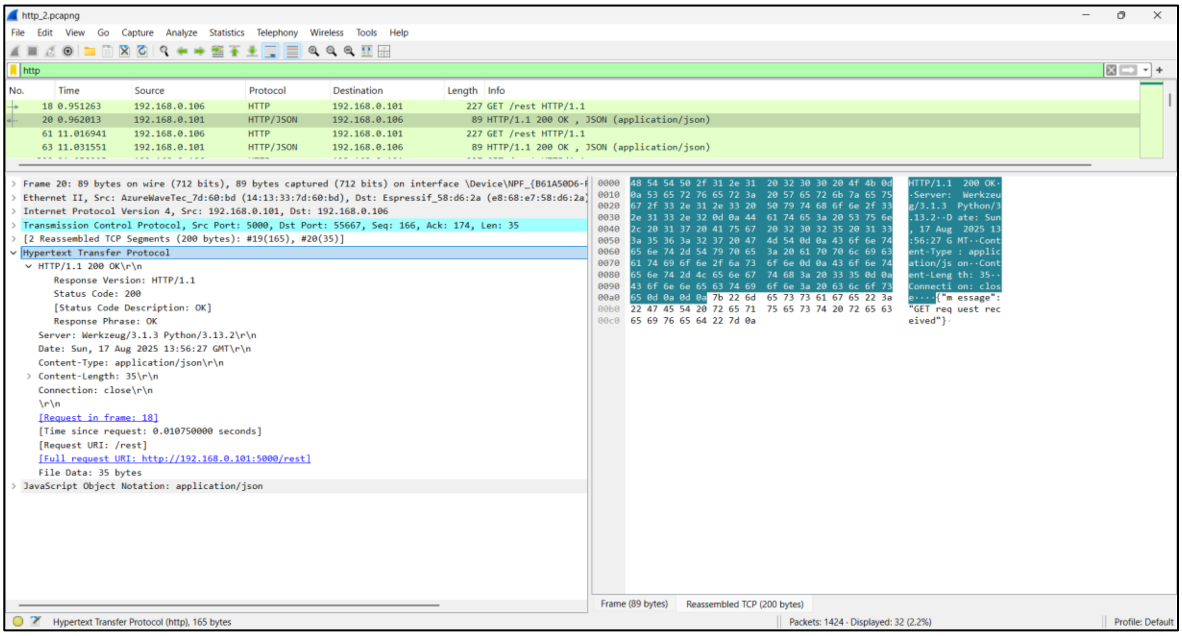
HTTP

Measurements

- **Request header (GET): 173 bytes**.
  - **GET frame Length (on wire): 227 bytes**.



- **Response header (200 OK): 165 bytes**.
  - **Payload (from Content-Length): 35 bytes** (JSON).
  - **200-OK body frame Length: 89 bytes**.



Calculations

- Per-frame link/IP/TCP **overhead** = **89 − 35 = 54 B**
- 200-OK ***header* frame Length = 165 + 54 = 219 B**
- Protocol header total **(request + response) = 173 + 165 = 338 B**
- **Headers-only** on-wire **= 227 + 219 = 446 B**
- **Full** on-wire **= 446 + 89 = 535 B**

CoAP

Message interpretation

- **Version:** 1
- **Type: CON** (Confirmable)
- **Code: 0.03 (PUT)**; server replies **2.04 (Changed)**
- **Message ID:** unique per request
- **Token:** present (token length **4 bytes**), matches request/response
- **Options:** Uri-Path: "light"

- **Payload:** ASCII **“0”** or **“1”** (toggles LED)

```
▾ Constrained Application Protocol, Confirmable, PUT, MID:43979
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0001 = Token Length: 1
    Code: PUT (3)
    Message ID: 43979
    Token: 31
  ▸ Opt Name: #1: Uri-Path: light
    End of options marker: 255
  ▸ Payload: Payload Content-Format: application/octet-stream (no Content-Format), Length: 1
    [Uri-Path: /light]
    [Response In: 183]
▾ Data (1 byte)
    Data: 31
    [Length: 1]
```

Measurements

- **UDP payload (CoAP message length): 14 bytes**.

```
▾ User Datagram Protocol, Src Port: 56339, Dst Port: 5683
    Source Port: 56339
    Destination Port: 5683
    Length: 22
    Checksum: 0xba05 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 42]
    [Stream Packet Number: 1]
  ▸ [Timestamps]
    UDP payload (14 bytes)
```

- **Total header size (reported): 42 bytes**.

```
▾ Internet Protocol Version 4, Src: 192.168.16.101, Dst: 192.168.16.104
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 42
    Identification: 0xc24a (49738)
  ▸ 000. .... = Flags: 0x0
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 128
    Protocol: UDP (17)
    Header Checksum: 0xd65a [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 192.168.16.101
    Destination Address: 192.168.16.104
    [Stream index: 13]
▾ User Datagram Protocol, Src Port: 56339, Dst Port: 5683
```

Calculations

- **On-wire per packet =** 14 + 42 = 56 B

- **Full on-wire (request + response) =** 56 + 56 = 112 B

- **Application payload total =** 1 B (request) + 0 B (response) = 1 B

- **On-wire (headers-only)** = 112 − 1 = 111 B
- **CoAP protocol bytes (inside UDP payload):**
  - Request: 14 − 1 = 13 B
  - Response: 14 − 0 = 14 B
  - **Total CoAP:** 13 + 14 = 27 B

Task 3: Compare Protocols

Protocol	Request/Response	Total on-wire (full)	Total on-wire (headers-only)	Header bytes	Payload bytes	Notes
HTTP	GET /rest → 200 OK	**535 B**	**446 B**	**338 B**	**35 B**	TCP; text headers (Host, User-Agent, Content-Length); JSON reply
CoAP	PUT /light "1" → 2.04 Changed	**112B**	**111B**	**27B**	**1B**	UDP; CoAP v1 CON; token; Uri-Path "light"; response empty payload

Conclusion

This lab highlighted the differences between the protocols clearly. By the two tiny IoT exchanges on an ESP8266 and looked at the packets in Wireshark: an **HTTP GET → 200 OK** to a Flask server, and a **CoAP CON PUT → 2.04 Changed** to toggle an LED. Seeing the raw bytes helped more than just reading theory.

The numbers were very distinct. On one side, HTTP exchange moved about **535 B** on the wire to deliver a **35 B** JSON message, and **338 B** of that was just HTTP headers whereas CoAP exchange, doing the same job for the LED, used only **~112 B** end-to-end, with **~27 B** of CoAP message bytes (header + token + options. The main reason is HTTP rides on TCP(more header overhead, connection semantics), while CoAP is compact and runs over UDP.

If we're sending tiny, frequent updates on constrained devices, **CoAP is the better fit**—lighter, faster to get on and off the air, and still reliable when we use **confirmable (CON)** messages. If we need easy integration with web tools, browsers, proxies, and JSON APIs, **HTTP is still the most convenient** option even though it costs more bytes.