Comparing HTTP, CoAP and MQTT Protocols with ESP8266

Lab-05

Name: Md. Samir Hossain
ID: 2022-3-60-161
Course: CSE406 (Internet of Things)
Section: 01
Semester: Summer-2025

Dr. Raihan Ul Islam
Associate Professor
Department of Computer Science & Engineering

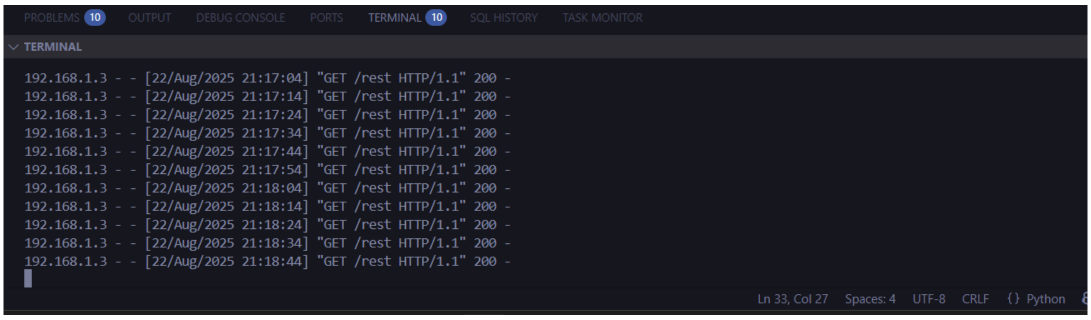
August 22, 2025

1 Introduction

The Hypertext Transfer Protocol (HTTP) is a text-based, request-response protocol that operates over TCP. It is widely used for web communications but is considered heavy-weight due to its verbose headers, making it less ideal for resource-constrained IoT devices. The Constrained Application Protocol (CoAP) is a lightweight, UDP-based protocol specifically designed for constrained environments, featuring compact binary headers and REST-like semantics for efficient data exchange. Message Queuing Telemetry Transport (MQTT) is a publish/subscribe protocol that runs over TCP (often with TLS), optimized for low-bandwidth and high-latency networks with minimal header overhead, making it suitable for scenarios like sensor data reporting. The purpose of this lab is to set up and analyze these protocols using NodeMCU ESP8266 microcontrollers and Python scripts, capture network traffic with Wireshark, and compare their efficiency, overhead, transport mechanisms, and suitability for IoT applications.

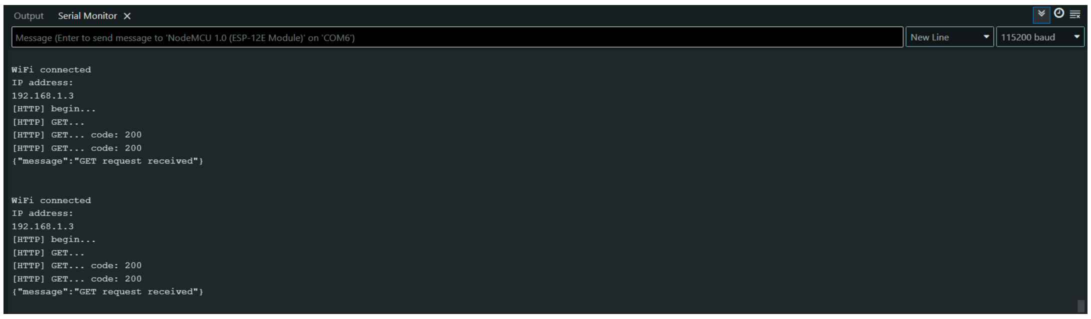
2 Procedure Summary

The setup followed the lab instructions with minor modifications to the provided code files. For HTTP, the Arduino code (CSE406_HTTPbasicClient.ino) was updated with WiFi credentials (”ZTE_2.4G_fRbXhk”, ”4GAXxtCA”) and the Flask server IP (192.168.1.2:5000). The Python Flask server (main.py) was modified to handle only the REST endpoint for GET requests, returning a JSON message. No hardware sensors were used; hardcoded values were assumed if needed. For CoAP, the provided CSE406_CoapServer.ino was used on the ESP8266, and CoapClient.py was run on the Python side to send PUT requests with payload 1” or 0” to the /light resource. The ESP8266 IP was updated in CoapClient.py. MQTT setup was prepared but not executed; the CSE406_mqtt.ino file was reviewed, but no captures were performed. Packet captures were done using Wireshark on a computer in the same WiFi network. For HTTP, traffic was filtered with http” after running the Flask server and uploading the Arduino code. For CoAP, the filter udp.port==5683” was used while running CoapClient.py. Captures were saved as .pcapng files. Multiple runs ensured consistent measurements, and connectivity was tested beforehand.



```
192.168.1.3 - - [22/Aug/2025 21:17:04] "GET /rest HTTP/1.1" 200 -
192.168.1.3 - - [22/Aug/2025 21:17:14] "GET /rest HTTP/1.1" 200 -
192.168.1.3 - - [22/Aug/2025 21:17:24] "GET /rest HTTP/1.1" 200 -
192.168.1.3 - - [22/Aug/2025 21:17:34] "GET /rest HTTP/1.1" 200 -
192.168.1.3 - - [22/Aug/2025 21:17:44] "GET /rest HTTP/1.1" 200 -
192.168.1.3 - - [22/Aug/2025 21:17:54] "GET /rest HTTP/1.1" 200 -
192.168.1.3 - - [22/Aug/2025 21:18:04] "GET /rest HTTP/1.1" 200 -
192.168.1.3 - - [22/Aug/2025 21:18:14] "GET /rest HTTP/1.1" 200 -
192.168.1.3 - - [22/Aug/2025 21:18:24] "GET /rest HTTP/1.1" 200 -
192.168.1.3 - - [22/Aug/2025 21:18:34] "GET /rest HTTP/1.1" 200 -
192.168.1.3 - - [22/Aug/2025 21:18:44] "GET /rest HTTP/1.1" 200 -
```

Figure 1: Python HTTP Flask Server Output

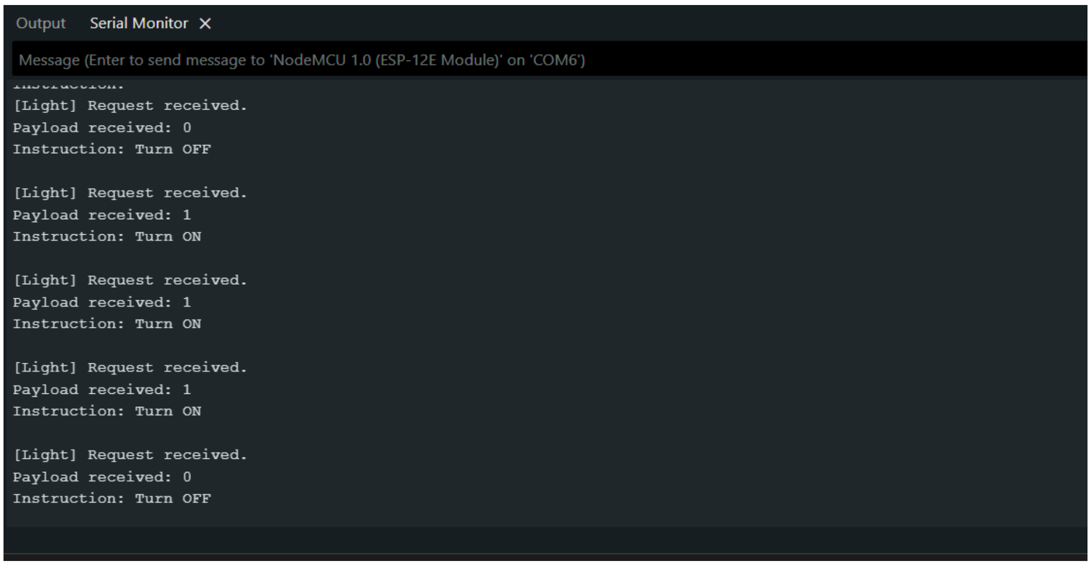


```
Message (Enter to send message to 'NodeMCU 1.0 (ESP-12E Module)' on 'COM6')

WiFi connected
IP address:
192.168.1.3
[HTTP] begin...
[HTTP] GET...
[HTTP] GET... code: 200
[HTTP] GET... code: 200
{"message":"GET request received"}


WiFi connected
IP address:
192.168.1.3
[HTTP] begin...
[HTTP] GET...
[HTTP] GET... code: 200
[HTTP] GET... code: 200
{"message":"GET request received"}
```

Figure 2: HTTP Client Arduino Serial Monitor Output



```
Message (Enter to send message to 'NodeMCU 1.0 (ESP-12E Module)' on 'COM6')
[Light] Request received.
Payload received: 0
Instruction: Turn OFF

[Light] Request received.
Payload received: 1
Instruction: Turn ON

[Light] Request received.
Payload received: 1
Instruction: Turn ON

[Light] Request received.
Payload received: 1
Instruction: Turn ON

[Light] Request received.
Payload received: 0
Instruction: Turn OFF
```

Figure 3: CoAP Server Arduino Serial Monitor Output

3 Task Results

3.1 Task 1: Setup and Packet Capture

The environment was set up by installing required Arduino libraries (ESP8266WiFi, ESP8266HTTPClient) and Python packages (flask, aiocoap). WiFi credentials were configured in the Arduino sketches. For HTTP, the Flask server was started on port 5000, and the ESP8266 acted as a client sending GET requests every 10 seconds. The setup used one NodeMCU ESP8266 board per protocol. No additional hardware like LEDs or sensors was connected; references in code were commented out or ignored. For CoAP, the ESP8266 ran as the server, and the Python client sent PUT requests to toggle a virtual light state. Captures were performed as follows:

HTTP: Uploaded the modified CSE406_HTTPbasicClient.ino, started main.py, and captured GET requests/responses. CoAP: Uploaded CSE406_CoapServer.ino, ran CoapClient.py with payload ”1”, and captured PUT requests/responses.

Code changes included updating IPs, ports, and WiFi details. Captures were run multiple times for accuracy.

3.2 Task 2: Analyze Packet Details

For each protocol (HTTP and CoAP), key packets were examined in Wireshark. **HTTP:**

Request Packet: Identified as a GET /rest HTTP/1.1 from 192.168.1.3 to 192.168.1.2. Key fields include the method (GET), Host header (192.168.1.2:5000), and no payload (empty request body). The packet shows standard HTTP headers like User-Agent and Accept. Response Packet: 200 OK with JSON payload "message": "GET request received". Key fields: Status code (200), Content-Type (application/json), Content-Length (32), and the JSON payload. Sizes: Request total packet size 225 bytes (including Ethernet 14, IP 20, TCP 20, HTTP 171). HTTP header size approximately 171 bytes (request line + headers; no payload). Response total 219 bytes, header ∼54 bytes, payload 32 bytes. Wireshark Screenshot Annotation: (Description: Highlight the HTTP request line GET /rest HTTP/1.1", Host header, and lack of payload. For response, annotate status 200 OK", Content-Length, and JSON data.)

CoAP:

Request Packet: CON PUT /light with token 5b 1f and payload 1". Key fields: Version (1), Type (Confirmable), Code (0.03 PUT), Options (Uri-Path: light"), Token, and 1-byte payload. Response Packet: ACK 2.04 Changed with payload "1". Key fields: Type (Acknowledgement), Code (2.04), Token matching request. Sizes: Request total packet size 56 bytes (Ethernet 14, IP 20, UDP 8, CoAP header/options ∼13, payload 1). CoAP header size 4 bytes fixed + options (∼8 bytes). Response total 53 bytes, header/options ∼11 bytes, payload 1 byte. Wireshark Screenshot Annotation: (Description: Highlight CoAP fixed header (Ver, T, TKL, Code), Uri-Path option, Token, and payload "1" in request. For response, annotate Code 2.04 and matching Token.)

Figure 4: Wireshark Screenshot for HTTP and CoAP Packets

3.3 Task 3: Compare Protocols

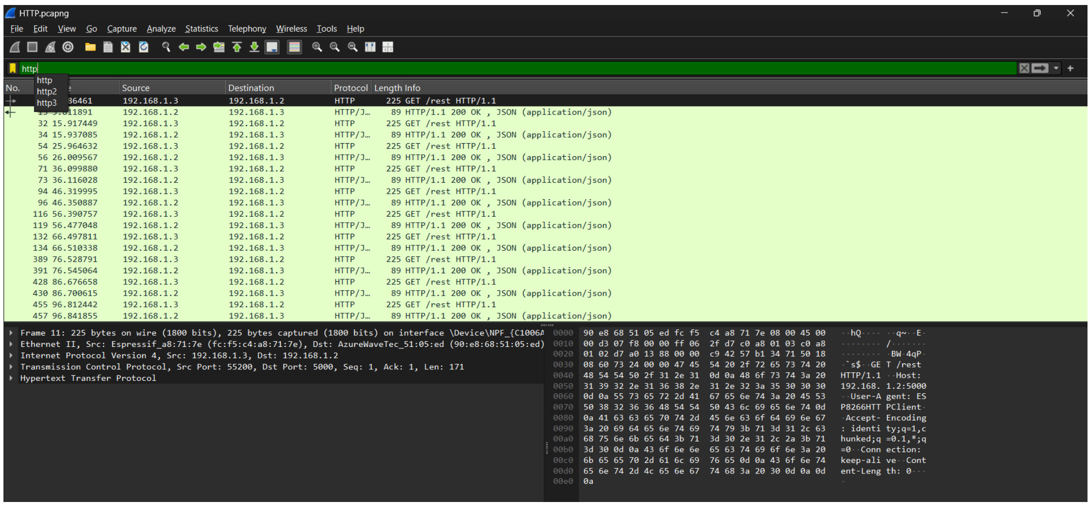
The comparison is based on one sample request/response pair per protocol: HTTP GET, CoAP PUT, MQTT PUBLISH. **Analysis and Discussion:**

Protocol	Request Type	Total Size (bytes)	Header Size (bytes)	Payload Size (bytes)	Key Details
HTTP	GET	225 (req) / 219 (res)	171 (req) / 168 (res)	0 (req) / 32 (res)	GET, 200 OK, Host, Content-Length, JSON {"message": "GET request"}
CoAP	PUT	56 (req) / 53 (res)	12 (req) / 11 (res)	1 (req) / 1 (res)	0.03 PUT / 2.04 Changed, Uri-Path "light", Payload: "1"

Table 1: Protocol Comparison Table

Efficiency: CoAP's compact binary format results in significantly smaller packets (e.g., 56 bytes for PUT) compared to HTTP's text-based headers (225 bytes for GET), making CoAP more efficient for constrained devices. **Overhead:** For HTTP, header-to-total ratio is high (∼76**Transport:** HTTP and MQTT use TCP for reliability (connection-oriented, retransmissions), ensuring delivery but adding overhead like handshakes. CoAP uses UDP for speed and lower overhead, but lacks built-in reliability (though confirmable messages add ACKs). **IoT Suitability:** CoAP is highly suitable for resource-constrained devices due to its lightweight REST model and UDP efficiency, e.g., for simple device control like lights. HTTP is less suitable due to high resource demands but offers familiarity. MQTT's pub/sub is ideal for sensor networks with many-to-many communication, supporting QoS for reliability in unreliable networks.

Include Wireshark screenshots with annotations: For HTTP, highlight headers and payload; for CoAP, options and code fields.

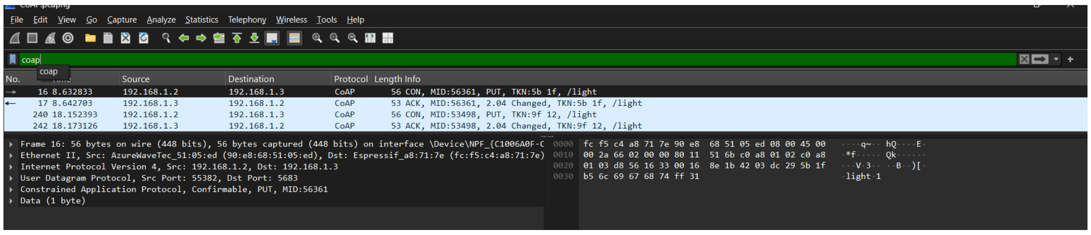

HTTP.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
http
http
http2
http3

```
No.                Source          Destination     Protocol Length Info
          36461    192.168.1.3     192.168.1.2     HTTP      225 GET /rest HTTP/1.1
          .811891  192.168.1.2     192.168.1.3     HTTP/J…    89 HTTP/1.1 200 OK , JSON (application/json)
     32 15.917449  192.168.1.3     192.168.1.2     HTTP      225 GET /rest HTTP/1.1
     34 15.937085  192.168.1.2     192.168.1.3     HTTP/J…    89 HTTP/1.1 200 OK , JSON (application/json)
     54 25.964632  192.168.1.3     192.168.1.2     HTTP      225 GET /rest HTTP/1.1
     56 26.009567  192.168.1.2     192.168.1.3     HTTP/J…    89 HTTP/1.1 200 OK , JSON (application/json)
     71 36.099880  192.168.1.3     192.168.1.2     HTTP      225 GET /rest HTTP/1.1
     73 36.116028  192.168.1.2     192.168.1.3     HTTP/J…    89 HTTP/1.1 200 OK , JSON (application/json)
     94 46.319995  192.168.1.3     192.168.1.2     HTTP      225 GET /rest HTTP/1.1
     96 46.350887  192.168.1.2     192.168.1.3     HTTP/J…    89 HTTP/1.1 200 OK , JSON (application/json)
    116 56.390757  192.168.1.3     192.168.1.2     HTTP      225 GET /rest HTTP/1.1
    119 56.477048  192.168.1.2     192.168.1.3     HTTP/J…    89 HTTP/1.1 200 OK , JSON (application/json)
    132 66.497811  192.168.1.3     192.168.1.2     HTTP      225 GET /rest HTTP/1.1
    134 66.510338  192.168.1.2     192.168.1.3     HTTP/J…    89 HTTP/1.1 200 OK , JSON (application/json)
    389 76.528791  192.168.1.3     192.168.1.2     HTTP      225 GET /rest HTTP/1.1
    391 76.545064  192.168.1.2     192.168.1.3     HTTP/J…    89 HTTP/1.1 200 OK , JSON (application/json)
    428 86.676658  192.168.1.3     192.168.1.2     HTTP      225 GET /rest HTTP/1.1
    430 86.700615  192.168.1.2     192.168.1.3     HTTP/J…    89 HTTP/1.1 200 OK , JSON (application/json)
    455 96.812442  192.168.1.3     192.168.1.2     HTTP      225 GET /rest HTTP/1.1
    457 96.841855  192.168.1.2     192.168.1.3     HTTP/J…    89 HTTP/1.1 200 OK , JSON (application/json)
```

```
› Frame 11: 225 bytes on wire (1800 bits), 225 bytes captured (1800 bits) on interface \Device\NPF_{C1006A
› Ethernet II, Src: Espressif_a8:71:7e (fc:f5:c4:a8:71:7e), Dst: AzureWaveTec_51:05:ed (90:e8:68:51:05:ed)
› Internet Protocol Version 4, Src: 192.168.1.3, Dst: 192.168.1.2
› Transmission Control Protocol, Src Port: 55200, Dst Port: 5000, Seq: 1, Ack: 1, Len: 171
› Hypertext Transfer Protocol
```

```
0000  90 e8 68 51 05 ed fc f5  c4 a8 71 7e 08 00 45 00   ··hQ····  ··q~··E·
0010  00 d3 07 f8 00 00 ff 06  2f d7 c0 a8 01 03 c0 a8   ········  /·······
0020  01 02 d7 a0 13 88 00 00  c9 42 57 b1 34 71 50 18   ········  ·BW·4qP·
0030  08 60 73 24 00 00 47 45  54 20 2f 72 65 73 74 20   ·`s$··GE  T /rest 
0040  48 54 54 50 2f 31 2e 31  0d 0a 48 6f 73 74 3a 20   HTTP/1.1  ··Host: 
0050  31 39 32 2e 31 36 38 2e  31 2e 32 3a 35 30 30 30   192.168.  1.2:5000
0060  0d 0a 55 73 65 72 2d 41  67 65 6e 74 3a 20 45 53   ··User-A  gent: ES
0070  50 38 32 36 36 48 54 54  50 43 6c 69 65 6e 74 0d   P8266HTT  PClient·
0080  0a 41 63 63 65 70 74 2d  45 6e 63 6f 64 69 6e 67   ·Accept-  Encoding
0090  3a 20 69 64 65 6e 74 69  74 79 3b 71 3d 31 2c 63   : identi  ty;q=1,c
00a0  68 75 6e 6b 65 64 3b 71  3d 30 2e 31 2c 2a 3b 71   hunked;q  =0.1,*;q
00b0  3d 30 0d 0a 43 6f 6e 6e  65 63 74 69 6f 6e 3a 20   =0··Conn  ection: 
00c0  6b 65 65 70 2d 61 6c 69  76 65 0d 0a 43 6f 6e 74   keep-ali  ve··Cont
00d0  65 6e 74 2d 4c 65 6e 67  74 68 3a 20 30 0d 0a 0d   ent-Leng  th: 0···
00e0  0a                                                 ·
```

Figure 5: HTTP WireShark Results


CoAP.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
coap
coap

```
No.                Source          Destination     Protocol Length Info
     16 8.632833   192.168.1.2     192.168.1.3     CoAP       56 CON, MID:56361, PUT, TKN:5b 1f, /light
     17 8.642703   192.168.1.3     192.168.1.2     CoAP       53 ACK, MID:56361, 2.04 Changed, TKN:5b 1f, /light
    240 18.152393  192.168.1.2     192.168.1.3     CoAP       56 CON, MID:53498, PUT, TKN:9f 12, /light
    242 18.173126  192.168.1.3     192.168.1.2     CoAP       53 ACK, MID:53498, 2.04 Changed, TKN:9f 12, /light
```

```
› Frame 16: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{C1006A0F-C
› Ethernet II, Src: AzureWaveTec_51:05:ed (90:e8:68:51:05:ed), Dst: Espressif_a8:71:7e (fc:f5:c4:a8:71:7e)
› Internet Protocol Version 4, Src: 192.168.1.2, Dst: 192.168.1.3
› User Datagram Protocol, Src Port: 55382, Dst Port: 5683
› Constrained Application Protocol, Confirmable, PUT, MID:56361
› Data (1 byte)
```

```
0000  fc f5 c4 a8 71 7e 90 e8  68 51 05 ed 08 00 45 00   ····q~··  hQ····E·
0010  00 2a 66 02 00 00 80 11  51 6b c0 a8 01 02 c0 a8   ·*f·····  Qk······
0020  01 03 d8 56 16 33 00 16  8e 1b 42 03 dc 29 5b 1f   ···V·3··  ··B··)[·
0030  b5 6c 69 67 68 74 ff 31                            ·light·1
```

Figure 6: CoAP WireShark Results

4 Conclusions

In summary, CoAP and MQTT offer lightweight designs with low overhead, making them preferable for IoT over HTTP's verbose, resource-heavy nature. Pros of CoAP include speed and efficiency on UDP; cons are potential packet loss without confirmables. HTTP's pros are universality and reliability via TCP; cons are high bandwidth use. MQTT pros include scalability for pub/sub; cons are TCP overhead. Potential applications: MQTT for large-scale sensor networks (e.g., environmental monitoring), CoAP for battery-powered devices (e.g., smart locks), and HTTP for integration with web services where resources allow.