

Lab Report

Course Name: Internet of Things

Course Code: CSE 406

Section No: 01

Lab Exercise No: 05

Title : Comparing HTTP, CoAP, and MQTT Protocols

Submitted To:

Dr. Raihan Ul Islam

Associate Professor

Department of Computer Science & Engineering

Submitted By:

Student's Name: Jobayer Faisal Fahim

Student's ID: 2022-2-60-130

Date of submission: August 10, 2025

Teammates Name & ID

- Akash Saha (2022-2-60-081)
- Shafiqul Islam Fahim(2022-2-60-085)
- Mahir Faysal(2022-2-60-044)

Title: Comparing HTTP, CoAP, and MQTT Protocols

1. Introduction

The Internet of Things (IoT) operates under strict resource constraints, such as limited bandwidth, power, and computational capacity. To address these challenges, efficient communication protocols are crucial. In this lab, we compare three communication protocols commonly used in IoT: HTTP, CoAP, and MQTT. These protocols were implemented using NodeMCU ESP8266 microcontrollers and Python scripts. The main objective is to evaluate the performance of these protocols in terms of packet size, efficiency, transport mechanisms, and their suitability for IoT applications. Network traffic was captured and analyzed using Wireshark to measure packet sizes, header sizes, and payload details.

2. Procedure Summary

2.1. Hardware Setup:

- **NodeMCU ESP8266 Microcontrollers:** These low-cost WiFi boards were used to implement the protocols. Each board was programmed to run one of the communication protocols: HTTP, CoAP, or MQTT.
- **Wi-Fi Network:** All devices and servers were connected to a common Wi-Fi network, ensuring proper communication between the ESP8266 and the servers.
- **External Devices (optional):** Sensors or LEDs could be connected to the ESP8266, though for this experiment, the code was set up with hardcoded values (e.g., temperature, humidity) for simplicity.

2.2. Software Setup:

- **Arduino IDE:** The Arduino IDE was used for writing and uploading code to the NodeMCU ESP8266. Required libraries like ESP8266WiFi, PubSubClient, and coap-simple were included.
- **Visual Studio Code (VS Code):** Used for Python development, particularly for the Flask server (main.py) and CoAP client (CoapClient.py).
- **Python 3:** Libraries such as aiocoap (for CoAP) and flask (for HTTP) were installed via pip.

- **Wireshark:** This tool was used for packet capture and analysis. It allowed filtering and examination of the network traffic generated by the protocols.

2.3. Task Breakdown:

❖ Task 1: Setup and Packet Capture

- The software was configured to run on NodeMCU ESP8266 for all three protocols: HTTP, CoAP, and MQTT.
- The Flask server (main.py) was run on a Python machine, which accepted HTTP requests. For CoAP, the CoAP server was run on NodeMCU, while for MQTT, a local or cloud MQTT broker (e.g., HiveMQ) was used.
- Wireshark was employed to capture the network traffic, allowing analysis of request/response packets for each protocol.

❖ Task 2: Analyze Packet Details

- Captured packets were analyzed for each protocol (GET, PUT, PUBLISH) to measure the total packet size, header size, and payload size.

❖ Task 3: Comparison

- A comparative analysis was done based on packet sizes, header sizes, payload sizes, and other protocol-specific details (e.g., transport layer, overhead).

3. Tools Used:

- **Wireshark:** Essential for capturing and analyzing the network traffic generated by HTTP, CoAP, and MQTT protocols.
- **Arduino IDE:** Used for programming the NodeMCU ESP8266 boards with the required protocol implementations.
- **Visual Studio Code:** A versatile code editor used for writing Python scripts, including the Flask server (main.py) and CoAP client (CoapClient.py).
- **Python:** Used for creating the REST server and CoAP client to interact with the ESP8266 boards. Python libraries like flask and aiocoap were used for handling HTTP and CoAP protocols, respectively.

4. Setup:

- **NodeMCU ESP8266:** Configured to run HTTP, CoAP, and MQTT protocols based on the provided code examples.

- **Network Configuration:** ESP8266 was connected to the Wi-Fi network, and server IP addresses were updated in the corresponding code files (e.g., CoapClient.py, CSE406_HTTPbasicClient.ino).
- **Wireshark Capture:** The tool was used to capture network traffic for each protocol. Filters such as http, udp.port = 5683 (for CoAP), and tcp.port = 8883 (for MQTT) were applied to isolate the relevant packets.

5. Results and Analysis

5.1. Analysis

❖ HTTP Protocol (GET Request):

- HTTP packets had a larger size due to verbose headers, such as the Host, Content-Length, and Accept fields.
- The total packet size was considerably larger compared to CoAP and MQTT, reflecting the protocol's resource-heavy nature.

Output:

```
WiFi connected
IP address:
192.168.0.109
[HTTP] begin...
[HTTP] GET...
[HTTP] GET... code: 200
[HTTP] GET... code: 200
{"message":"GET request received FROM JOBAYER FAISAL FAHIM With his all friends"}
```

```
WiFi connected
IP address:
192.168.0.109
[HTTP] begin...
[HTTP] GET...
[HTTP] GET... code: 200
[HTTP] GET... code: 200
{"message":"GET request received FROM JOBAYER FAISAL FAHIM With his all friends"}
```

170	16.504409	192.168.0.109	192.168.0.106	TCP	62 55083 → 5000 [SYN] Seq=0 Win=2144 Len=0 MSS=536 SACK_PERM
171	16.504507	192.168.0.106	192.168.0.109	TCP	62 5000 → 55083 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM
172	16.500098	192.168.0.109	192.168.0.106	TCP	54 55083 → 5000 [ACK] Seq=1 Ack=1 Win=2144 Len=0
173	16.510472	192.168.0.109	192.168.0.106	HTTP	222 6310 → 63210 [GET] 112/11
174	16.511439	192.168.0.106	192.168.0.109	TCP	219 5000 → 55083 [PSH, ACK] Seq=1 Ack=174 Win=65362 Len=165 [TCP PDU reassembled in 175]
175	16.529244	192.168.0.106	192.168.0.109	HTTP/1.1	115 HTTP/1.1 200 OK , JSON (application/json)
176	16.541513	192.168.0.109	192.168.0.106	TCP	54 55083 → 5000 [ACK] Seq=174 Ack=228 Win=1917 Len=0
177	16.541513	192.168.0.109	192.168.0.106	TCP	54 55083 → 5000 [FIN, ACK] Seq=174 Ack=228 Win=1917 Len=0
178	16.542008	192.168.0.106	192.168.0.109	TCP	54 5000 → 55083 [ACK] Seq=228 Ack=175 Win=65362 Len=0
179	19.689822	192.168.0.106	104.16.185.241	UDP	540 63210 → 443 Len=498
180	19.692564	104.16.185.241	192.168.0.106	UDP	67 443 → 63210 Len=25
181	19.781399	104.16.185.241	192.168.0.106	UDP	223 443 → 63210 Len=181

Wi-Fi

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

Packet list String Header

Options: Narrow & Wide Case sensitive Backwards Multiple occurrences

No.	Time	Source	Destination	Protocol	Length	Info
592	54.451733	192.168.0.109	192.168.0.106	TCP	62	52088 → 5000 [SYN] Seq=0 Win=2144 Len=0 MSS=536 SACK_PERM
593	54.451855	192.168.0.106	192.168.0.109	TCP	62	5000 → 52088 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM
594	54.462330	192.168.0.109	192.168.0.106	TCP	54	52088 → 5000 [ACK] Seq=1 Ack=1 Win=2144 Len=0
595	54.462330	192.168.0.109	192.168.0.106	HTTP	227	GET /rest HTTP/1.1
596	54.465181	192.168.0.106	192.168.0.109	TCP	219	5000 → 52088 [PSH, ACK] Seq=1 Ack=174 Win=65362 Len=165 [TCP PDU reassembled in 597]
597	54.476559	192.168.0.106	192.168.0.109	HTTP/1.1	136	HTTP/1.1 200 OK, JSON (application/json)
598	54.480648	192.168.0.109	192.168.0.106	TCP	54	52088 → 5000 [ACK] Seq=174 Ack=249 Win=1896 Len=0
599	54.484713	192.168.0.109	192.168.0.106	TCP	54	52088 → 5000 [FIN, ACK] Seq=174 Ack=249 Win=1896 Len=0
600	54.484757	192.168.0.106	192.168.0.109	TCP	54	5000 → 52088 [ACK] Seq=249 Ack=175 Win=65362 Len=0
601	55.632640	192.168.0.106	104.16.185.241	UDP	545	63210 → 443 Len=503

Frame 597: 136 bytes on wire (1088 bits), 136 bytes captured (1088 bits) on interface \Device\NPF_{D0D2B4E8-001-...} Ethernet II, Src: Intel_Bc:04:9c:(c4:23:60:8c:04:9c), Dst: Espressif_56:3f:20 (34:5f:45:56:3f:20) Internet Protocol Version 4, Src: 192.168.0.106, Dst: 192.168.0.109 Transmission Control Protocol, Src Port: 5000, Dst Port: 52088, Seq: 166, Ack: 174, Len: 82

Source Port: 5000
Destination Port: 52088
[Stream index: 31]
[Stream Packet Number: 6]
[Conversation completeness: Complete, WITH DATA [31]]
[TCP Segment Len: 82]
Sequence Number: 166 (relative sequence number)
Sequence Number (raw): 200769635
[Next Sequence Number: 249 (relative sequence number)]
Acknowledgment Number: 174 (relative ack number)
Acknowledgment number (raw): 9345
0101 = Header Length: 20 bytes (5)
Flags: 0x019 (FIN, PSH, ACK)
Window: 65362
[calculated window size: 65362]
[Window size scaling factor: -2 (no window scaling used)]
Checksum: 0x8294 [unverified]
[Checksum status: Unverified]
Urgent Pointer: 0
[Timestamps]
[SEQ/ACK analysis]
TCP payload (82 bytes)
TCP segment data (82 bytes)

Frame (136 bytes) Reassembled TCP (247 bytes)

Internet Protocol Version 4 (ip), 20 bytes

Packets: 631 · Dropped: 0 (0.0%)

Profile: Default

Wi-Fi

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

Packet list String Header

Options: Narrow & Wide Case sensitive Backwards Multiple occurrences

No.	Time	Source	Destination	Protocol	Length	Info
592	54.451733	192.168.0.109	192.168.0.106	TCP	62	52088 → 5000 [SYN] Seq=0 Win=2144 Len=0 MSS=536 SACK_PERM
593	54.451855	192.168.0.106	192.168.0.109	TCP	62	5000 → 52088 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM
594	54.462330	192.168.0.109	192.168.0.106	TCP	54	52088 → 5000 [ACK] Seq=1 Ack=1 Win=2144 Len=0
595	54.462330	192.168.0.109	192.168.0.106	HTTP	227	GET /rest HTTP/1.1
596	54.465181	192.168.0.106	192.168.0.109	TCP	219	5000 → 52088 [PSH, ACK] Seq=1 Ack=174 Win=65362 Len=165 [TCP PDU reassembled in 597]
597	54.476559	192.168.0.106	192.168.0.109	HTTP/1.1	136	HTTP/1.1 200 OK, JSON (application/json)
598	54.480648	192.168.0.109	192.168.0.106	TCP	54	52088 → 5000 [ACK] Seq=174 Ack=249 Win=1896 Len=0
599	54.484713	192.168.0.109	192.168.0.106	TCP	54	52088 → 5000 [FIN, ACK] Seq=174 Ack=249 Win=1896 Len=0
600	54.484757	192.168.0.106	192.168.0.109	TCP	54	5000 → 52088 [ACK] Seq=249 Ack=175 Win=65362 Len=0
601	55.632640	192.168.0.106	104.16.185.241	UDP	545	63210 → 443 Len=503

Frame 597: 136 bytes on wire (1088 bits), 136 bytes captured (1088 bits) on interface \Device\NPF_{D0D2B4E8-001-...} Ethernet II, Src: Intel_Bc:04:9c:(c4:23:60:8c:04:9c), Dst: Espressif_56:3f:20 (34:5f:45:56:3f:20) Internet Protocol Version 4, Src: 192.168.0.106, Dst: 192.168.0.109

0100 = Version: 4
... 0101 = Header Length: 20 bytes (5)
Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 122
Identification: 0xc506 (56438)
010 = Flags: 0x2, Don't Fragment
... 0000 0000 0000 = Fragment Offset: 0
Time to Live: 128
Protocol: TCP (6)
Header Checksum: 0x0000 [validation disabled]
[Header checksum status: Unverified]
Source Address: 192.168.0.106
Destination Address: 192.168.0.109
[Stream index: 8]
Transmission Control Protocol, Src Port: 5000, Dst Port: 52088, Seq: 166, Ack: 174, Len: 82

[2] Reassembled TCP Segments (247 bytes): #596(165), #597(82)
Hypertext Transfer Protocol
JavaScript Object Notation: application/json

Frame (136 bytes) Reassembled TCP (247 bytes)

Internet Protocol Version 4 (ip), 20 bytes

Packets: 631 · Dropped: 0 (0.0%)

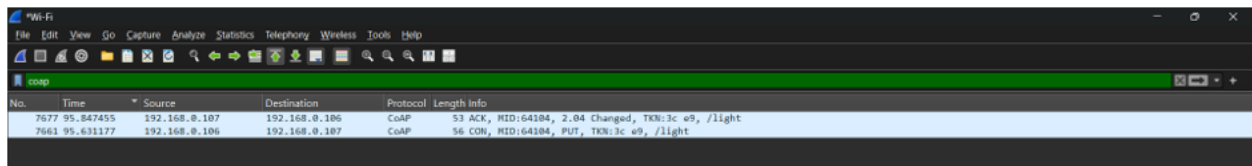
Profile: Default

❖ CoAP Protocol (PUT Request):

- CoAP packets were much more compact, using binary headers and a lightweight structure. The fixed header size (4 bytes) and minimal options reduced the total packet size.

Output:

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



The image shows a Wireshark packet capture window titled "Wi-Fi". The packet list pane shows two captured packets, both of type CoAP. The first packet (No. 7677) is an ACK with RID:64104, 2.04 Changed, TKN:3c e9, /light. The second packet (No. 7661) is a CON with RID:64104, PUT, TKN:3c e9, /light. The packet details pane shows the CoAP protocol structure for the selected packet.

No.	Time	Source	Destination	Protocol	Length	Info
7677	95.847455	192.168.0.107	192.168.0.106	CoAP	53	ACK, RID:64104, 2.04 Changed, TKN:3c e9, /light
7661	95.631177	192.168.0.106	192.168.0.107	CoAP	56	CON, RID:64104, PUT, TKN:3c e9, /light

▼ Frame 7677: 53 bytes on wire (424 bits), 53 bytes captured (424 bits) on interface \Device\NPF_{DDD2B4EB-0D1C-42DD-A7BE-6E4B4FDE874F}

Section number: 1

- ▶ Interface id: 0 (\Device\NPF_{DDD2B4EB-0D1C-42DD-A7BE-6E4B4FDE874F})
- Encapsulation type: Ethernet (1)
- Arrival Time: Aug 17, 2025 23:02:38.276243000 Bangladesh Standard Time
- UTC Arrival Time: Aug 17, 2025 17:02:38.276243000 UTC
- Epoch Arrival Time: 1755450158.276243000
- [Time shift for this packet: 0.000000000 seconds]
- [Time delta from previous captured frame: 0.003975000 seconds]
- [Time delta from previous displayed frame: 0.216278000 seconds]
- [Time since reference or first frame: 95.847455000 seconds]
- Frame Number: 7677
- Frame Length: 53 bytes (424 bits)
- Capture Length: 53 bytes (424 bits)
- [Frame is marked: False]
- [Frame is ignored: False]
- [Protocols in frame: eth:ethertype:ip:udp:coap:data]
- [Coloring Rule Name: UDP]
- [Coloring Rule String: udp]

▼ Ethernet II, Src: Espressif_56:3f:20 (34:5f:45:56:3f:20), Dst: Intel_8c:04:9c (c4:23:60:8c:04:9c)

- ▶ Destination: Intel_8c:04:9c (c4:23:60:8c:04:9c)
- ▶ Source: Espressif_56:3f:20 (34:5f:45:56:3f:20)
- Type: IPv4 (0x0800)
- [Stream index: 14]

▼ Internet Protocol Version 4, Src: 192.168.0.107, Dst: 192.168.0.106

0100 = Version: 4

.... 0101 = Header Length: 20 bytes (5)

- ▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
- Total Length: 39
- Identification: 0x0003 (3)
- ▶ 000. = Flags: 0x0
- ...0 0000 0000 0000 = Fragment Offset: 0
- Time to Live: 255
- Protocol: UDP (17)
- Header Checksum: 0x399d [validation disabled]
- [Header checksum status: Unverified]
- Source Address: 192.168.0.107
- Destination Address: 192.168.0.106
- [Stream index: 23]

▼ User Datagram Protocol, Src Port: 5683, Dst Port: 53371

Source Port: 5683

Destination Port: 53371

Length: 19

Checksum: 0xdf5c [unverified]

[Checksum Status: Unverified]

[Stream index: 50]

[Stream Packet Number: 2]

- ▶ [Timestamps]
- UDP payload (11 bytes)

▼ Constrained Application Protocol, Acknowledgement, 2.04 Changed, MID:64104

01.. = Version: 1

..10 = Type: Acknowledgement (2)

.... 0010 = Token Length: 2

Code: 2.04 Changed (68)

Message ID: 64104

Token: 3ce9

- ▶ Opt Name: #1: Content-Format: application/octet-stream
- End of options marker: 255
- ▶ Payload: Payload Content-Format: application/octet-stream, Length: 1
- [Uri-Path: /light]
- [Request In: 7661]
- [Response Time: 0.216278000 seconds]

▼ Data (1 byte)

Data: 31

[Length: 1]

❖ MQTT Protocol (PUBLISH):

- MQTT's packet sizes were smaller than HTTP but larger than CoAP. MQTT's fixed header size (1–2 bytes) and its minimalistic approach made it highly efficient for constrained environments.

5.2. Comparative Analysis

Protocol	Request Type	Total Packet Size (bytes)	Header Size (bytes)	Payload Size (bytes)	Key Details
HTTP	GET	136	20	82	Verbose headers, TCP-based
CoAP	PUT	56	8	14	Lightweight, UDP-based
MQTT	PUBLISH	--	--	--	Small header, Pub/Sub model

Reference:

HTTP

```

Frame 597: 136 bytes on wire (1088 bits), 136 bytes captured (1088 bits) on interface \Device\NPF_{D002B4E8-0...}
Ethernet II, Src: Intel_8c:04:9c (c4:23:60:8c:04:9c), Dst: Espressif_56:3f:20 (34:5f:45:56:3f:20)
Internet Protocol Version 4, Src: 192.168.0.106, Dst: 192.168.0.109
Transmission Control Protocol, Src Port: 5000, Dst Port: 52088, Seq: 166, Ack: 174, Len: 82
  Source Port: 5000
  Destination Port: 52088
  [Stream Index: 31]
  [Stream Packet Number: 6]
  [Conversation completeness: Complete, WITH_DATA (31)]
  [TCP Segment Len: 82]
  Sequence Number: 166 (relative sequence number)
  Sequence Number (raw): 2007096635
  [Next Sequence Number: 249 (relative sequence number)]
  Acknowledgment Number: 174 (relative ack number)
  Acknowledgment number (raw): 9345
  0101 .... = Header Length: 20 bytes (5)
  Flags: 0x019 (FIN, PSH, ACK)
  Window: 65362
  [Calculated window size: 65362]
  [Window size scaling factor: -2 (no window scaling used)]
  Checksum: 0x8294 [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  [Timestamps]
  [SEQ/ACK analysis]
  TCP payload (82 bytes)
  TCP segment data (82 bytes)
Internet Protocol Version 4 (IPv4) 20 bytes

```

CoAP

```

Frame 7661: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{D002B4E8-0...}
Ethernet II, Src: Intel_8c:04:9c (c4:23:60:8c:04:9c), Dst: Espressif_56:3f:20 (34:5f:45:56:3f:20)
Internet Protocol Version 4, Src: 192.168.0.106, Dst: 192.168.0.107
User Datagram Protocol, Src Port: 53371, Dst Port: 5683
  Source Port: 53371
  Destination Port: 5683
  Length: 22
  Checksum: 0x824d [unverified]
  [Checksum Status: Unverified]
  [Stream index: 50]
  [Stream Packet Number: 1]
  [Timestamps]
  UDP payload (14 bytes)
  Constrained Application Protocol, Confirmable, PUT, MID:64104
  Data (1 byte)

```


6. Pros and Cons of Each Protocol:

6.1. HTTP:

❖ Pros:

- Well-established and widely used in web communication.
- Simple request/response model.
- Mature libraries and tools.

❖ Cons:

- Large header size increases overhead.
- TCP-based, making it slower compared to UDP.
- Not ideal for constrained devices or networks.

6.2. CoAP:

❖ Pros:

- Lightweight, using binary headers, and efficient for low-power, low-bandwidth devices.
- UDP-based, providing faster transmission with less overhead.
- Well-suited for simple IoT applications (e.g., device control).

❖ Cons:

- UDP lacks built-in reliability, which may cause data loss in some cases.
- Fewer tools and libraries compared to HTTP.

6.3. MQTT:

❖ Pros:

- Efficient, minimal header size.

- Suitable for large-scale IoT networks with many devices.
- Pub/Sub model allows easy scalability and management of many-to-many communication.

❖ **Cons:**

- Requires a central broker, introducing a potential single point of failure.
- Overhead due to TCP, though smaller compared to HTTP.

7. Conclusion

The lab successfully compared three communication protocols - HTTP, CoAP, and MQTT - focusing on their efficiency and suitability for IoT applications. While HTTP is reliable and widely used, it is not the most efficient for resource-constrained devices. CoAP and MQTT offer more efficient alternatives, with CoAP excelling in constrained environments due to its lightweight design and UDP transport, while MQTT shines in scalable, large-scale networks due to its pub/sub model and low overhead. The choice of protocol depends on specific use cases, with MQTT being ideal for sensor networks and CoAP suited for simple device control.