EAST WEST UNIVERSITY

Mesh Networking with NodeMCU ESP8266 and painlessMesh

Course Title: Internet of Things

Course Code: CSE 406

Section: 01

Submitted by

Name: Akash Saha

ID: 2022-2-60-081

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

Jobayer Faisal Fahim	2022-2-60-130
Shafiqul Islam Fahim	2022-2-60-085
Mahir Faysal Dipto	2022-2-60-

Lab Report: 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.
- **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.
- **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	--	--	--	Verbose headers, TCP-based
CoAP	PUT	--	--	--	Lightweight, UDP-based
MQTT	PUBLISH	--	--	--	Small header, Pub/Sub model

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

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.