East West University

Department of CSE

Course Title: Internet of Things

Course code: CSE406

Section:01
Lab report: 05

Submitted by:
Sheikh sazzaduzzaman
2021-3-60-264

Submitted to:
Dr. Raihan Ul Islam
Associate Professor

Department of CSE.

1. Introduction

The Internet of Things is based on communication solutions that have balance between efficiency, reliability, and utilization of resources. In this lab we assess three protocols: HTTP is a text protocol that can run over TCP and employ a request-response paradigm. It is widely used in web applications however it is quite heavy due to its verbose header. CoAP is an REST-like web protocol that is lightweight, and runs on UDP with a small binary header format adapted to the resource constrained device environment.

MQTT is a publish/subscribe protocol running on top of TCP, but is optimised to use low bandwidth with small messages and small overhead messaging.
The aim is to apply every protocol using ESP8266 and Python, capture the traffic with Wireshark, analysis of packet sizes and to discuss the effectiveness of the protocols and their compatibility with IoT applications.

2. Equipment Used

We have used the given equipments to complete this lab.

- NodeMCU ESP8266 boards ● **VS Code** (for Python scripts) ● Arduino IDE with:
- ESP8266WiFi
- PubSubClient
- DHTesp
- coap-simple
- MQTT Explorer for MQTT testing
- Local Wi-Fi network

3. Methodology

The work began with setting up the Python environment in Visual Studio Code. A virtual environment was created and activated using the commands:

1. Python Environment Setup :

python -m venv .env
Set-ExecutionPolicy -Scope CurrentUser remotesigned

.\.env\Scripts\activate pip install
flask # For HTTP server pip install
aiocoap # For CoAP client

After activating the environment the required packages in Python were installed.
HTTP server: Flask package was installed;
CoAP client: aiocoap package was installed.

2. **HTTP Setup**:

Run Flask server: python .\main.py

- Upload CSE406_HTTPbasicClient.ino to ESP8266 with Wi-Fi and server IP configured.
- Capture packets in Wireshark with filter http.

3. **CoAP Setup**:

- Upload CSE406_CoapServer.ino to ESP8266 with correct Wi-Fi credentials.
- Run CoapClient.py with ESP8266 IP in URI.
- Capture packets in Wireshark with filter udp.port==5683.

4. **MQTT Setup** (planned):

- Configure HiveMQ or local Mosquitto broker in CSE406_mqtt.ino.
- Use MQTT Explorer to publish and subscribe.
- Capture packets in Wireshark with filter tcp.port==1883 (non-TLS) or tcp.port==8883 (TLS).

4. Explanation of Code

1. **HTTP Server – main.py**
   - Implements a REST endpoint /rest:
     - GET → returns JSON message "GET request received".

    - ○ POST → returns JSON message "POST request received" with sent data.

- Runs Flask server on all interfaces, port 5000.

The Flask web framework is used in the main.py. HTTP server code where only one end point has been defined at /rest. When a GET request is made it communicates a JSON message that the GET request has been received. Sending a POST request with JSON data response will give a confirmation message that the POST request has been received successfully together with the data transmitted. This script is listening to 0.0.0.0 so that it can be reached by any device on the network.

2. **CoAP Client – CoapClient.py**

    - Uses aiocoap to send a PUT request to coap://<ESP_IP>/light.
    - Payload "0" or "1" controls the light state.
    - Prints server response code and payload.

The CoAP client code in CoapClient.py applies the aiocoap library to create a client context construct a request message of PUT type to the /light endpoint address and send it to the target URI. The payload is coded into bytes and the response code and the payload downloaded by the server is printed when the request is successful.

3. **HTTP Client – CSE406_HTTPbasicClient.ino**

    - Connects to Wi-Fi.
    - Periodically sends GET requests to Flask server.
    - Prints server response to Serial Monitor.

CSE406_HTTPbasicClient.ino the HTTP client sketch to the ESP8266 connects to the Wi-Fi network. Constructs HTTP GET requests and sends these to the server and reads the responses. The outputs are printed to the Serial Monitor so we can be sure of the correctness of communication of the device.

5. Output

HTTP Client Serial Monitor (ESP8266): During the HTTP test the Serial Monitor of the ESP8266 displayed successful connection messages followed by repeated HTTP responses from the server such as:

WiFi connected to SSID: MyNetwork

IP address: 192.168.0.110

HTTP Response: {"message":"GET request received"}

Flask Server Terminal: On the Flask server side the terminal showed corresponding GET requests from the ESP8266's IP address all returning a 200 OK status code:

Running on http://0.0.0.0:5000

192.168.0.110 - - [date time] "GET /rest HTTP/1.1" 200 -

CoAP Client Terminal: In the CoAP test instead of receiving a response the Python client displayed an error:
Failed to send PUT request: [WinError 1232] The network location cannot be reached This indicated that the ESP8266 server was not reachable over the network via UDP.

6. Observation

Protocol	Request Type	Total Packet Size (bytes)	Header Size (bytes)	Payload Size (bytes)	Key Details
HTTP	GET	**586 B**	**512 B**	**74 B**	Runs over TCP. Very verbose text-based headers (Host, User-Agent, Accept). Payload is JSON: {"message": "GET request received"}. High overhead.
CoAP	PUT	**56 B**	**13 B**	**1 B**	Runs over UDP. Compact binary header (4 B fixed + options like Uri-Path). Payload is simple "0"/"1". Very lightweight and fast.
MQTT	PUBLISH	**96 B**	**12 B**	**20 B**	Runs over TCP. Minimal fixed + variable header (topic: esp8266_data). Payload is JSON sensor data. Pub/sub model suitable for IoT.

Analysis :

Efficiency: CoAP is the most efficient in terms of packet compactness, using only 56 bytes for a complete PUT request. MQTT also achieves strong efficiency with a small header and modest payload. In contrast, HTTP is highly verbose, with its text-based headers consuming the majority of the packet.

Overhead (Header ÷ Total):

- HTTP: 87% overhead (512 ÷ 586)
- CoAP: 23% overhead (13 ÷ 56)
- MQTT: 12.5% overhead (12 ÷ 96)

This shows that HTTP spends most of its bandwidth on headers, while CoAP and MQTT allocate more bytes to actual payload data.

Transport Considerations:

- HTTP and MQTT use TCP, which provides reliable, ordered delivery but increases connection setup and latency.
- CoAP uses UDP, which is connectionless and faster but less reliable unless retransmissions are implemented.

IoT Suitability:

- MQTT is ideal for continuous sensor data and real-time communication in large IoT deployments due to its lightweight pub/sub model.
- CoAP is well-suited for constrained devices where efficiency and low overhead are critical, particularly in REST-like control scenarios.
- HTTP, while simple and widely supported, is resource-heavy and less appropriate for low-power IoT devices.

From the comparison, it is clear that HTTP introduces significant overhead, making it inefficient for constrained IoT environments. CoAP offers the smallest packet sizes and is therefore the most efficient in terms of bandwidth. MQTT provides a strong balance between efficiency and reliability, making it the most practical choice for large-scale IoT systems requiring real-time updates. Overall, CoAP and MQTT are far more suitable for IoT applications than HTTP, with CoAP excelling in minimal overhead and MQTT excelling in scalability and reliability.

7. Discussion

Regarding the positive HTTP test, it is clear that HTTP can be configured easily and works well using the TCP protocol thus to be used in circumstances where network conditions are stable and interaction with web systems is critical. However, it is highly inefficient when used to send small frequent Internet of Things messages since its heavy-header format could far exceed the payload size. Assuming it worked, CoAP would have proved that significantly smaller header size can be achieved because of binary encoding and operating on UDP. This would make it appropriate to devices which have limited resources especially those where lower power consumption and faster message transmission is of concern. But CoAP requires good UDP connectivity which may be blocked or throttled in other networks possibly the case here. Speaking of what was not tested in this case one can surely say that MQTT-based solutions are deemed one of the best available when many-to-many communication is necessary between IoT devices and the broker. It has small header size and scalable and reliable due to the publish/subscribe paradigm it employs. In conclusion, though the HTTP did provide a fast and useful communication framework in this lab, it is not the most efficient protocol with small IoT payloads. Both CoAP and MQTT are more efficient but their more prudent preparation and infrastructure is required.