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 (in VS Code):

```
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.

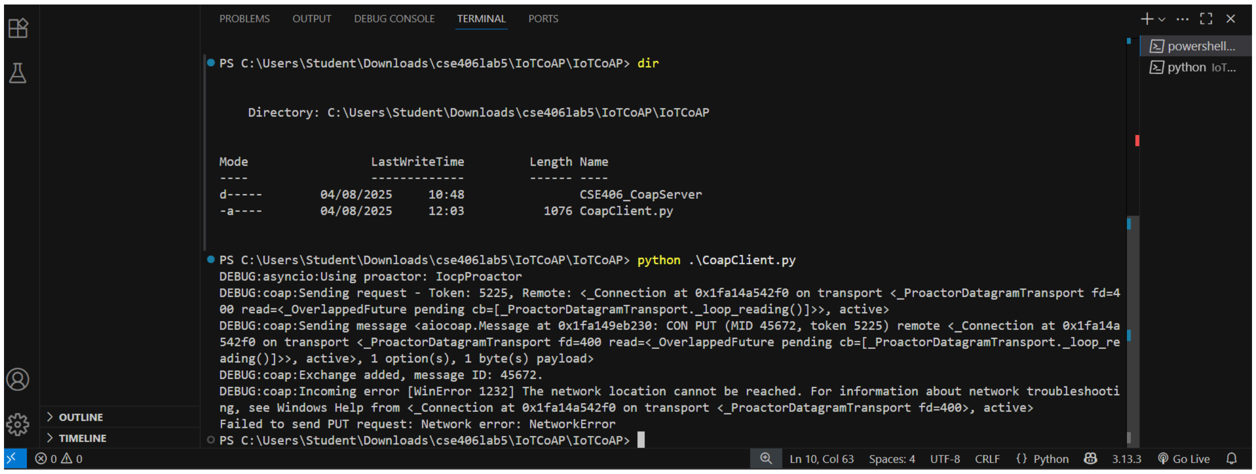
The CoAP client code in CoapClient.py applies the aiocoap library to create a client context construct a request message of PUT type to the /light endpoint address and send it to the target URI. The payload is coded into bytes and the response code and the payload downloaded by the server is printed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

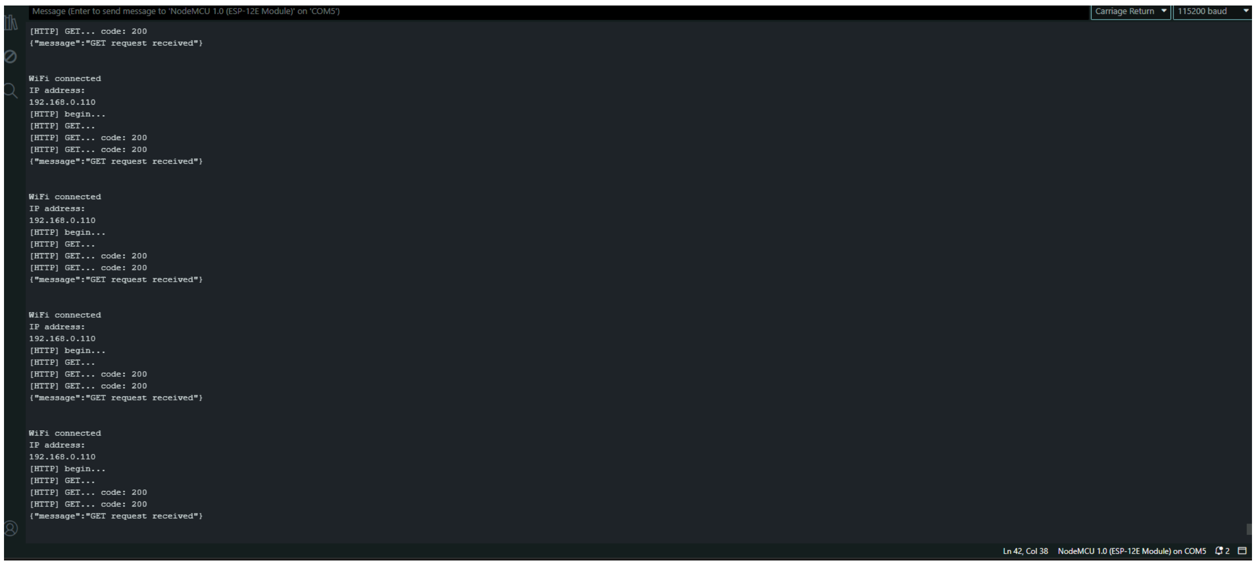


```
PS C:\Users\Student\Downloads\cse406lab5\IoTCoAP\IoTCoAP> dir


    Directory: C:\Users\Student\Downloads\cse406lab5\IoTCoAP\IoTCoAP


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        04/08/2025     10:48                CSE406_CoapServer
-a----        04/08/2025     12:03           1076 CoapClient.py

PS C:\Users\Student\Downloads\cse406lab5\IoTCoAP\IoTCoAP> python .\CoapClient.py
DEBUG:asyncio:Using proactor: IocpProactor
DEBUG:coap:Sending request - Token: 5225, Remote: <_Connection at 0x1fa14a542f0 on transport <_ProactorDatagramTransport fd=4
00 read=<_OverlappedFuture pending cb=[_ProactorDatagramTransport._loop_reading()]>>, active>
DEBUG:coap:Sending message <aiocoap.Message at 0x1fa149eb230: CON PUT (MID 45672, token 5225) remote <_Connection at 0x1fa14a
542f0 on transport <_ProactorDatagramTransport fd=400 read=<_OverlappedFuture pending cb=[_ProactorDatagramTransport._loop_re
ading()]>>, active>, 1 option(s), 1 byte(s) payload>
DEBUG:coap:Exchange added, message ID: 45672.
DEBUG:coap:Incoming error [WinError 1232] The network location cannot be reached. For information about network troubleshooti
ng, see Windows Help from <_Connection at 0x1fa14a542f0 on transport <_ProactorDatagramTransport fd=400>, active>
Failed to send PUT request: Network error: NetworkError
PS C:\Users\Student\Downloads\cse406lab5\IoTCoAP\IoTCoAP>
```



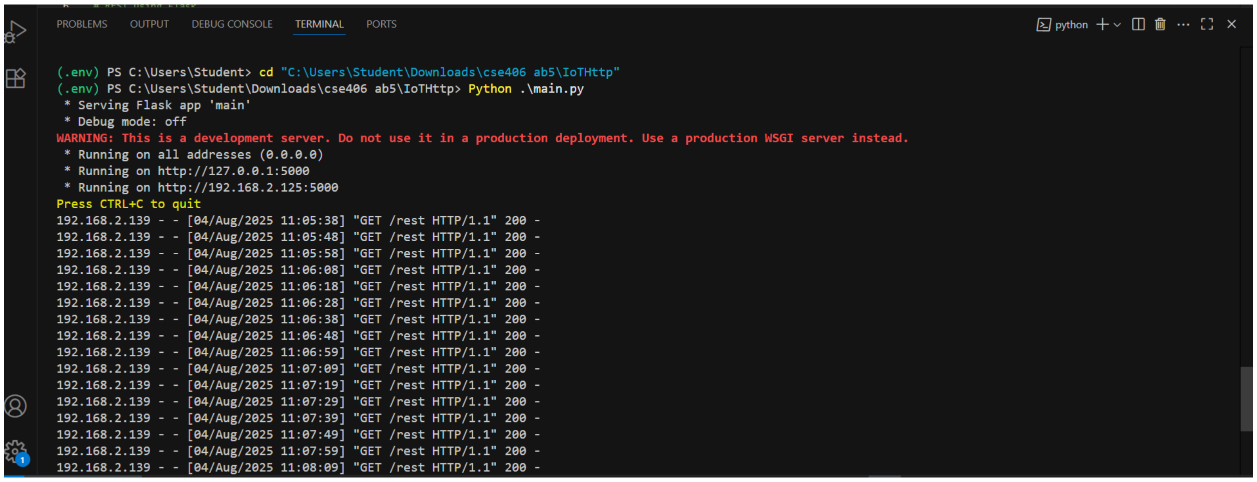
```
[HTTP] GET... code: 200
{"message":"GET request received"}


WiFi connected
IP address:
192.168.0.110
[HTTP] begin...
[HTTP] GET...
[HTTP] GET... code: 200
[HTTP] GET... code: 200
{"message":"GET request received"}


WiFi connected
IP address:
192.168.0.110
[HTTP] begin...
[HTTP] GET...
[HTTP] GET... code: 200
[HTTP] GET... code: 200
{"message":"GET request received"}


WiFi connected
IP address:
192.168.0.110
[HTTP] begin...
[HTTP] GET...
[HTTP] GET... code: 200
[HTTP] GET... code: 200
{"message":"GET request received"}


WiFi connected
IP address:
192.168.0.110
[HTTP] begin...
[HTTP] GET...
[HTTP] GET... code: 200
[HTTP] GET... code: 200
{"message":"GET request received"}
```



```
(.env) PS C:\Users\Student> cd "C:\Users\Student\Downloads\cse406 ab5\IoTHttp"
(.env) PS C:\Users\Student\Downloads\cse406 ab5\IoTHttp> Python .\main.py
 * Serving Flask app 'main'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.2.125:5000
Press CTRL+C to quit
192.168.2.139 - - [04/Aug/2025 11:05:38] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:05:48] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:05:58] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:06:08] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:06:18] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:06:28] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:06:38] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:06:48] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:06:59] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:07:09] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:07:19] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:07:29] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:07:39] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:07:49] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:07:59] "GET /rest HTTP/1.1" 200 -
192.168.2.139 - - [04/Aug/2025 11:08:09] "GET /rest HTTP/1.1" 200 -
```

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

HTTP test succeeded and demonstrated that the ESP8266 and the Flask server could connect well via the TCP protocol. Wireshark revelations showed that though the payloads were small. HTTP headers were relatively large such as a large portion of the data was overhead during transmission.

CoAP test failed after getting a network accessibility error which may have arisen because of incorrect IP address and inappropriate subnet or the firewall blocked through UDP protocol on port 5683. This made no packet capture or analytics to CoAP on this session.

In this lab effort, MQTT testing was not done and hence no results or observations could be logged against it.

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.