A logo for a university

AI-generated content may be incorrect.

**IoT Lab: Comparing HTTP, CoAP Protocols with ESP8266**

**Course: CSE 406**

**Section: 1**

**Lab: 5**

**Submitted To: Dr. Raihan Ul Islam**

**Associate Professor, CSE Dept.**

**Submitted by: Nawreen Islam**

**ID: 2021-3-60-052**

**1. Introduction**

The Internet of Things (IoT) involves resource-constrained devices that must communicate efficiently despite limited processing power, memory, and network bandwidth. Therefore, choosing a communication protocol is critical.

In this lab, we explore and compare two IoT communication protocols using NodeMCU ESP8266 boards:

* **HTTP (HyperText Transfer Protocol)**
  + A text-based, request–response protocol operating over TCP.
  + Advantages: highly standardized, universally supported, and easily integrated with web servers.
  + Limitations: verbose headers and higher overhead, which make it less efficient for constrained IoT devices.
* **CoAP (Constrained Application Protocol)**
  + A lightweight, binary protocol operating over UDP, designed specifically for IoT.
  + Advantages: compact header size, REST-like methods, low power consumption.
  + Limitations: lacks TCP’s reliability (requires retransmission mechanisms if needed).

**Objective of this lab**:

* Implement HTTP and CoAP communication using ESP8266 microcontrollers.
* Capture and analyze packet structures using Wireshark.
* Compare both protocols in terms of packet size, header overhead, and suitability for IoT applications.

**HTTP Implementation Details (from uploaded files)**

**Client (CSE406\_HTTPbasicClient.ino)**

* Written for **ESP8266 (NodeMCU)**.
* Connects to WiFi (SSID and password provided by user).
* Send an **HTTP GET request** to the Flask REST server at /rest.
* Wait for server response and print it to the serial monitor.

**Key functions in the sketch:**

* WiFi.begin(ssid, password) → connects ESP8266 to WiFi.
* client.connect(server, port) → establishes TCP connection to Flask server.
* client.println("GET /rest HTTP/1.1") → sends HTTP GET request.
* Reads back the response from the server and prints it.

A screenshot of a computer

AI-generated content may be incorrect.

**CoAp Materials Required**

**Hardware**

* 1 × NodeMCU ESP8266 board

**Software**

* Arduino IDE with libraries: ESP8266WiFi, coap-simple
* Python 3 with aiocoap package
* WiFi network for both ESP8266 and client machine

**Files Used**

* CSE406\_CoapServer\_v2.ino (ESP8266 CoAP server)
* CoapClient.py (Python CoAP client)
* *(For comparison)* CSE406\_HTTPbasicClient.ino and main.py (Flask HTTP server)

**3. Procedure**

**3.1 CoAP Server Setup (CSE406\_CoapServer\_v2.ino)**

* ESP8266 runs as a **CoAP server**.
* Exposes a resource at the URI path **/light**.
* Accepts **PUT requests** with payloads "0" or "1".
* On receiving "1", the server can toggle a device state (e.g., turn a light ON). On receiving "0", it can turn the device OFF.
* Responds with a CoAP acknowledgment and the code **2.04 Changed**, confirming the resource state has been updated.

**3.2 CoAP Client Setup (CoapClient.py)**

* Python script using the **aiocoap** library.
* Defines the URI: coap://<ESP8266\_IP>/light (e.g., coap://192.168.137.221/light).
* Sends a **PUT request** with a payload ("0" or "1").
* Waits for the response and prints:
  + Response code (e.g., 2.04 Changed)
  + Response payload (if any).
  + A screenshot of a computer

    AI-generated content may be incorrect.

**4. Results and Analysis**

CoAP Results

CoAP Results (Serial Monitor Output)

When the CoAP server sketch (CSE406\_CoapServer\_v2.ino) was uploaded to the ESP8266, the serial monitor displayed the following output:

* The ESP8266 connected to the WiFi network.
* Assigned IP address: 192.168.0.109
* The CoAP server successfully started and was ready to handle requests.

**Observations:**

* CoAP messages are compact (binary format).
* Very efficient for simple device control.
* UDP transport minimizes overhead but uses ACKs for reliability.

5. Protocol Comparison

| **Feature** | **HTTP (GET)** | **CoAP (PUT)** |
| --- | --- | --- |
| **Transport** | TCP (reliable, connection-oriented) | UDP (lightweight, connectionless) |
| **Message Size** | Large due to verbose headers | Very small (binary compact headers) |
| **Response** | JSON structured response (200 OK) | Simple acknowledgment (2.04 Changed) |
| **Efficiency** | Heavy overhead for small payloads | Extremely efficient for short messages |
| **Reliability** | Guaranteed by TCP | Provided via confirmable messages + ACKs |
| **Suitability** | Web integration, REST APIs | IoT control, constrained environments |
|  |  |  |

**Wireshark Analysis**

**Objective:**  
To capture and analyze network packets to observe the communication between devices.

**Procedure:**  
Wireshark was used to monitor the network while the devices exchanged messages. Relevant protocol filters (e.g., CoAP) were applied to focus on the required traffic.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

**Observations:**  
Captured packets showed the request and response flow between the client and server, including message types, source and destination IPs, and payload content.

**Conclusion:**  
Wireshark effectively visualized the network communication, allowing verification of message exchange and protocol behavior.