SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 2nd semester

IoT Cloud Infrastructure and IoT Protocols Lab Manual

This lab manual provides a structured guide for each program listed, covering the title, aim, procedure, source code, input, and expected output.

Lab 1: MQTT Publisher

Title: MQTT Publisher: Publishing Sensor Data from NodeMCU to a Cloud MQTT Broker

Aim: To develop a program to publish sensor data (e.g., temperature, humidity) from NodeMCU to an MQTT broker hosted on a cloud platform.

Procedure:

1. **Hardware Setup:** Connect a temperature/humidity sensor (e.g., DHT11/DHT22) to your NodeMCU. Ensure proper wiring according to the sensor's datasheet.
2. **Software Setup:**
   - Install the ESP8266 board package in your Arduino IDE (Tools > Board > Boards Manager).
   - Install necessary libraries: `PubSubClient` (for MQTT) and a DHT sensor library (e.g., `DHT sensor library by Adafruit`) via Sketch > Include Library > Manage Libraries.
3. **Cloud MQTT Broker Configuration:**
   - Choose a cloud-based MQTT broker service (e.g., HiveMQ Cloud, Adafruit IO, AWS IoT Core, or a self-hosted Mosquitto instance on a cloud VM).
   - Create an account and set up an MQTT client with necessary credentials (username, password, client ID if required).
   - Note down the broker's address (hostname/IP) and port (usually 1883 for unencrypted, 8883 for SSL/TLS).
4. **Code Development (Arduino IDE):**
   - Write the NodeMCU sketch (`.ino` file).
   - Include Wi-Fi connection code to connect NodeMCU to your local network.
   - Initialize the MQTT client with the broker details and credentials.
   - Implement a function to read data from the connected sensor (e.g., `dht.readTemperature()`, `dht.readHumidity()`).
   - Create a loop that periodically reads sensor data and publishes it to a predefined MQTT topic (e.g., `your_device_id/temperature`, `your_device_id/humidity`).
   - Add reconnection logic to handle Wi-Fi and MQTT disconnections gracefully.
5. **Upload and Test:**
   - Connect your NodeMCU to your computer via USB.
   - Select the correct board and port in Arduino IDE (Tools > Board, Tools > Port).

- Upload the code to the NodeMCU.
- Open the Serial Monitor (Tools > Serial Monitor) to observe connection status and published messages.
- Use an MQTT client application (e.g., MQTT Explorer, Mosquitto_sub, or a web-based MQTT client) to subscribe to the topic you are publishing to. Verify that the sensor data is being received on the cloud broker.

Source Code:

```
#include <ESP8266WiFi.h>
#include <PubSubClient.h>
#include <DHT.h> // For DHT11/DHT22 sensor

// WiFi credentials
const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

// MQTT Broker details
const char* mqtt_server = "YOUR_MQTT_BROKER_ADDRESS"; // e.g.,
"broker.hivemq.com"
const int mqtt_port = 1883; // Or 8883 for SSL/TLS
const char* mqtt_username = "YOUR_MQTT_USERNAME"; // If required
const char* mqtt_password = "YOUR_MQTT_PASSWORD"; // If required
const char* mqtt_client_id = "NodeMCU_Publisher_123"; // Unique client ID

// MQTT Topics
const char* temperature_topic = "nodemcu/sensor/temperature";
const char* humidity_topic = "nodemcu/sensor/humidity";

// DHT Sensor details
#define DHTPIN D2     // Digital pin connected to the DHT sensor
#define DHTTYPE DHT11   // DHT 11 or DHT 22
DHT dht(DHTPIN, DHTTYPE);

WiFiClient espClient;
PubSubClient client(espClient);

long lastMsg = 0;
char msg[50];
int value = 0;

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if (client.connect(mqtt_client_id, mqtt_username, mqtt_password)) {
      Serial.println("connected");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  dht.begin();
  setup_wifi();
  client.setServer(mqtt_server, mqtt_port);
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop();

  long now = millis();
  if (now - lastMsg > 5000) { // Publish every 5 seconds
    lastMsg = now;

    // Read sensor data
    float h = dht.readHumidity();
    float t = dht.readTemperature();

    // Check if any reads failed and exit early (to try again).
    if (isnan(h) || isnan(t)) {
      Serial.println("Failed to read from DHT sensor!");
      return;
    }

    // Publish temperature
    snprintf (msg, 50, "%.2f", t);
    Serial.print("Publishing temperature: ");
    Serial.println(msg);
    client.publish(temperature_topic, msg);

    // Publish humidity
    snprintf (msg, 50, "%.2f", h);
    Serial.print("Publishing humidity: ");
    Serial.println(msg);
    client.publish(humidity_topic, msg);
  }
}
```

Input: Real-time temperature and humidity readings from the connected DHT sensor.

Expected Output:

- NodeMCU successfully connects to Wi-Fi.
- NodeMCU connects to the specified MQTT broker.
- Temperature and humidity data are periodically published to the `nodemcu/sensor/temperature` and `nodemcu/sensor/humidity` topics on the cloud MQTT broker.

- An MQTT client subscribed to these topics will display the received sensor values, e.g.:
    - `nodemcu/sensor/temperature`: `25.75`
    - `nodemcu/sensor/humidity`: `62.30`

Lab 2: MQTT Subscriber

Title: MQTT Subscriber: Subscribing to MQTT Topics on NodeMCU

Aim: To create a program to subscribe to MQTT topics on NodeMCU and perform actions based on received messages (e.g., controlling an LED).

Procedure:

1. **Hardware Setup:** Connect an LED to a digital pin on your NodeMCU (e.g., D1). Include a current-limiting resistor.
2. **Software Setup:** Install the ESP8266 board package and the `PubSubClient` library in Arduino IDE.
3. **Cloud MQTT Broker Configuration:** Use the same MQTT broker details as in Lab 1. Define a topic for receiving commands (e.g., `nodemcu/commands/led`).
4. **Code Development (Arduino IDE):**
   - Include Wi-Fi connection code.
   - Set up the MQTT client and connect to the broker.
   - Implement a `callback` function that is triggered when a message is received on a subscribed topic.
   - Inside the `callback` function, parse the incoming message and perform an action (e.g., turn LED on/off based on "ON" or "OFF" message).
   - Subscribe to the command topic in the `setup()` function after connecting to MQTT.
   - Ensure `client.loop()` is called regularly in `loop()` to process incoming messages.
5. **Upload and Test:** Upload the code to NodeMCU. Use an MQTT client to publish messages to the command topic and observe the LED's behavior.

Source Code:

```
#include <ESP8266WiFi.h>
#include <PubSubClient.h>

// WiFi credentials
const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

// MQTT Broker details
const char* mqtt_server = "YOUR_MQTT_BROKER_ADDRESS";
const int mqtt_port = 1883;
const char* mqtt_username = "YOUR_MQTT_USERNAME"; // If required
const char* mqtt_password = "YOUR_MQTT_PASSWORD"; // If required
const char* mqtt_client_id = "NodeMCU_Subscriber_456";

// MQTT Topic to subscribe to for commands
const char* command_topic = "nodemcu/commands/led";

// LED pin
const int LED_PIN = D1; // Connect LED to D1 on NodeMCU (GPIO5)

WiFiClient espClient;
PubSubClient client(espClient);

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

// Callback function to handle incoming MQTT messages
void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");
  String message = "";
  for (int i = 0; i < length; i++) {
    message += (char)payload[i];
  }
  Serial.println(message);

  // Check the topic and message content
  if (String(topic) == command_topic) {
    if (message == "ON") {
      digitalWrite(LED_PIN, HIGH); // Turn LED ON
      Serial.println("LED ON");
    } else if (message == "OFF") {
      digitalWrite(LED_PIN, LOW);  // Turn LED OFF
      Serial.println("LED OFF");
    } else {
      Serial.println("Unknown command.");
    }
  }
}

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if (client.connect(mqtt_client_id, mqtt_username, mqtt_password)) {
      Serial.println("connected");
      // Once connected, subscribe to the topic
      client.subscribe(command_topic);
      Serial.print("Subscribed to topic: ");
      Serial.println(command_topic);
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, LOW); // Ensure LED is off initially
  setup_wifi();
  client.setServer(mqtt_server, mqtt_port);
  client.setCallback(callback); // Set the callback function
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop(); // Must be called frequently to process incoming messages
}
```

Input: MQTT messages published to the `nodemcu/commands/led` topic from an external MQTT client or publisher.

- To turn LED ON: Publish `ON` to `nodemcu/commands/led`
- To turn LED OFF: Publish `OFF` to `nodemcu/commands/led`

Expected Output:

- NodeMCU successfully connects to Wi-Fi and the MQTT broker.
- NodeMCU subscribes to the `nodemcu/commands/led` topic.
- When `ON` is published to the topic, the LED connected to D1 on NodeMCU turns on.
- When `OFF` is published to the topic, the LED connected to D1 on NodeMCU turns off.
- Serial Monitor output will show "LED ON" or "LED OFF" messages.

Lab 3: HTTP Client

Title: HTTP Client: Sending Sensor Data from NodeMCU to a Cloud-Based Server

Aim: To implement an HTTP client on NodeMCU to send sensor data to a cloud-based server using HTTP GET or POST requests.

Procedure:

1. **Hardware Setup:** Connect a sensor (e.g., DHT11/DHT22 for temperature/humidity) to your NodeMCU.
2. **Cloud Server Setup:**
   - Set up a simple web server or API endpoint on a cloud platform that can receive HTTP requests. Examples include:
     - A simple PHP script on a shared hosting server.
     - A Python Flask/Node.js Express app deployed on Heroku, AWS EC2, or Google Cloud Run.
     - A service like Thingspeak (which has built-in HTTP API for data logging).
   - Ensure the server endpoint is accessible from the internet and can process incoming GET/POST requests, typically storing the data in a database or displaying it.
3. **Software Setup:** Install the ESP8266 board package in Arduino IDE. No additional libraries are strictly needed for basic HTTP, as `ESP8266HTTPClient` is part of the core. If using a sensor, install its library (e.g., DHT sensor library).
4. **Code Development (Arduino IDE):**
   - Include Wi-Fi connection code.
   - Include `ESP8266HTTPClient.h` and `WiFiClient.h`.
   - Read sensor data.
   - Construct the HTTP request URL (for GET) or payload (for POST) with sensor data.
   - Send the HTTP request to your cloud server endpoint.
   - Handle server responses (e.g., check HTTP status code).
   - Implement error handling and retry mechanisms.
5. **Upload and Test:** Upload the code to NodeMCU. Monitor the serial output. Check your cloud server/dashboard to verify that data is being received and logged.

Source Code:

```
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>
#include <WiFiClient.h>
#include <DHT.h> // For DHT11/DHT22 sensor

// WiFi credentials
const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

// Cloud Server/API Endpoint details
// Example for a simple GET request (e.g., Thingspeak)
const char* server_url = "http://api.thingspeak.com/update";
const char* api_key = "YOUR_THINGSPEAK_WRITE_API_KEY"; // For Thingspeak

// DHT Sensor details
#define DHTPIN D2     // Digital pin connected to the DHT sensor
#define DHTTYPE DHT11   // DHT 11 or DHT 22
DHT dht(DHTPIN, DHTTYPE);
```

```
long lastSendTime = 0;
const long sendInterval = 15000; // Send data every 15 seconds

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void setup() {
  Serial.begin(115200);
  dht.begin();
  setup_wifi();
}

void loop() {
  if (WiFi.status() == WL_CONNECTED) {
    if (millis() - lastSendTime > sendInterval) {
      lastSendTime = millis();

      // Read sensor data
      float h = dht.readHumidity();
      float t = dht.readTemperature();

      if (isnan(h) || isnan(t)) {
        Serial.println("Failed to read from DHT sensor!");
        return;
      }

      WiFiClient client;
      HTTPClient http;

      // Construct the URL for GET request (e.g., for Thingspeak)
      String url = String(server_url) + "?api_key=" + api_key + "&field1=" +
String(t) + "&field2=" + String(h);
      Serial.print("Sending data to: ");
      Serial.println(url);

      http.begin(client, url); // Specify the URL and connect

      int httpCode = http.GET(); // Send the HTTP GET request

      if (httpCode > 0) { // Check for the returning code
        String payload = http.getString();
        Serial.printf("[HTTP] GET... code: %d\n", httpCode);
        Serial.println(payload);
      } else {
        Serial.printf("[HTTP] GET... failed, error: %s\n",
http.errorToString(httpCode).c_str());
      }

      http.end(); // Free resources
    }
  } else {
    Serial.println("WiFi Disconnected. Reconnecting...");
    setup_wifi(); // Attempt to reconnect
  }
}
```

Input: Real-time temperature and humidity readings from the connected DHT sensor.

Expected Output:

- NodeMCU successfully connects to Wi-Fi.
- Periodically, HTTP GET requests are sent to the configured cloud server endpoint.
- The cloud server receives the sensor data (e.g., temperature and humidity values).
- The server logs or displays the received data on its dashboard or database.
- Serial Monitor will show HTTP status codes and server responses, indicating successful data transmission.

Lab 4: OTA Updates

Title: OTA Updates: Setting Up Over-The-Air Firmware Updates for NodeMCU

Aim: To set up Over-The-Air (OTA) firmware updates for NodeMCU to enable remote updating of firmware from a cloud server without physical access.

Procedure:

1. **Network Setup:** Ensure your NodeMCU and the computer from which you'll perform updates are on the same local network. While true cloud-based OTA involves a server pushing updates, this lab focuses on the NodeMCU's ability to receive updates over Wi-Fi. For a cloud server, you would typically host the `.bin` file on a web server and configure NodeMCU to fetch it.
2. **Software Setup (Arduino IDE):**
   - Install the ESP8266 board package.
   - Include the `ESP8266WiFi.h` and `ESP8266mDNS.h` libraries.
   - For OTA, include `ArduinoOTA.h`.
3. **Code Development (Arduino IDE):**
   - Include Wi-Fi connection code.
   - Initialize `ArduinoOTA` in `setup()`:
     - Set a hostname for your NodeMCU (e.g., `esp-updater`).
     - Optionally set an authentication password.
     - Define callback functions for `onStart`, `onEnd`, `onProgress`, and `onError` to provide feedback during the update process.
   - Call `ArduinoOTA.handle()` frequently in the `loop()` function to allow the OTA process to run.
4. **Initial Upload:** Upload the sketch with OTA enabled to your NodeMCU via USB.
5. **Performing OTA Update:**
   - After the initial upload, disconnect USB.
   - In Arduino IDE, go to Tools > Port. You should now see your NodeMCU listed under "Network ports" with the hostname you set (e.g., `esp-updater at 192.168.1.X`). Select it.
   - Make a small change to your sketch (e.g., change an LED blink interval or a Serial.print message).
   - Click the "Upload" button. Arduino IDE will now upload the new firmware over Wi-Fi.
6. **Verification:** Observe the Serial Monitor for OTA progress messages. After the update, the NodeMCU will restart with the new firmware. Verify the changes you made.

Source Code:

```
#include <ESP8266WiFi.h>
#include <ESP8266mDNS.h> // Required for mDNS (Bonjour/Zeroconf)
#include <ArduinoOTA.h> // For Over-The-Air updates

// WiFi credentials
const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void setup() {
  Serial.begin(115200);
  Serial.println("Booting...");

  setup_wifi();

  // Port defaults to 8266
  // ArduinoOTA.setPort(8266);

  // Hostname defaults to esp8266-[ChipID]
  ArduinoOTA.setHostname("my-nodemcu-device");

  // No authentication by default
  // ArduinoOTA.setPassword("admin");

  // Password can be set with it's md5 value as well
  // MD5(admin) = 21232f297a57a5a743894a0e4a801fc3
  // ArduinoOTA.setPasswordHash("21232f297a57a5a743894a0e4a801fc3");

  ArduinoOTA.onStart([]() {
    String type;
    if (ArduinoOTA.getCommand() == U_FLASH) {
      type = "sketch";
    } else { // U_SPIFFS
      type = "filesystem";
    }
    // NOTE: if updating SPIFFS this would be the place to unmount SPIFFS
using SPIFFS.end()
    Serial.println("Start updating " + type);
  });
  ArduinoOTA.onEnd([]() {
    Serial.println("\nEnd");
  });
  ArduinoOTA.onProgress([](unsigned int progress, unsigned int total) {
    Serial.printf("Progress: %u%%\r", (progress / (total / 100)));
  });
  ArduinoOTA.onError([](ota_error_t error) {
    Serial.printf("Error[%u]: ", error);
    if (error == OTA_AUTH_ERROR) {
      Serial.println("Auth Failed");
    } else if (error == OTA_BEGIN_ERROR) {
      Serial.println("Begin Failed");
    } else if (error == OTA_CONNECT_ERROR) {
      Serial.println("Connect Failed");
    } else if (error == OTA_RECEIVE_ERROR) {
      Serial.println("Receive Failed");
    } else if (error == OTA_END_ERROR) {
      Serial.println("End Failed");
    }
  });

  ArduinoOTA.begin();
  Serial.println("Ready for OTA updates!");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
}

void loop() {
  ArduinoOTA.handle(); // This must be called frequently
  // Your regular NodeMCU code goes here
  // For example, blinking an LED or reading a sensor
  delay(10);
}
```

Input:

- A new version of the NodeMCU firmware (`.ino` sketch) on your computer.
- The NodeMCU device connected to the same Wi-Fi network.

Expected Output:

- NodeMCU successfully connects to Wi-Fi and becomes discoverable for OTA updates.
- When an update is initiated from Arduino IDE (or a custom tool), the firmware is uploaded wirelessly to the NodeMCU.
- Serial Monitor shows progress updates (e.g., "Progress: 10%", "Progress: 50%").
- Upon successful completion, the NodeMCU restarts with the updated firmware.
- Any changes made in the new firmware (e.g., a different LED blink pattern) are observed on the NodeMCU.

Lab 5: Security Measures

Title: Security Measures: Securing Communication between NodeMCU and Cloud Services using TLS/SSL Encryption and Authentication

Aim: To secure communication between NodeMCU and cloud services using TLS/SSL encryption and implement authentication mechanisms (e.g., client certificates or username/password).

Procedure:

1. **Cloud Service Setup:**
   - Choose a cloud service that supports TLS/SSL for MQTT (e.g., AWS IoT Core, Google Cloud IoT Core, or a self-hosted Mosquitto with SSL configured).
   - For TLS/SSL, you will need the server's root CA certificate. For client authentication, you might need a client certificate and private key. Download these from your cloud platform.
2. **Software Setup (Arduino IDE):**
   - Install the ESP8266 board package.
   - For secure MQTT, you'll need `PubSubClient` and `WiFiClientSecure`.
   - You'll also need to include the certificates in your sketch.
3. **Code Development (Arduino IDE):**
   - Include Wi-Fi connection code.
   - Include `WiFiClientSecure.h` and `PubSubClient.h`.
   - Define the root CA certificate, client certificate, and private key as `const char*` arrays in your code.
   - Use `WiFiClientSecure` instead of `WiFiClient` for the MQTT client.
   - Set the root CA certificate using `client.setCACert()`.
   - If using client certificates for authentication, set them using `client.setCertificate()` and `client.setPrivateKey()`.
   - When connecting to MQTT, use the secure port (typically 8883).
   - Implement MQTT publish/subscribe as in previous labs.
4. **Upload and Test:** Upload the code to NodeMCU. Monitor the serial output for connection status. Verify that data is being sent/received securely. If authentication fails, debug certificate issues.

Source Code:

```
#include <ESP8266WiFi.h>
#include <WiFiClientSecure.h> // For TLS/SSL
#include <PubSubClient.h>

// WiFi credentials
const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

// MQTT Broker details for secure connection (e.g., AWS IoT Core)
const char* mqtt_server = "YOUR_AWS_IOT_ENDPOINT.iot.your-
region.amazonaws.com";
const int mqtt_port = 8883; // Secure MQTT port
const char* mqtt_client_id = "NodeMCU_Secure_Client";

// AWS IoT Core requires client certificates and a root CA
// Replace with your actual certificates and private key
const char* aws_iot_root_ca =
"-----BEGIN CERTIFICATE-----\n"
"YOUR_AWS_IOT_ROOT_CA_CERTIFICATE\n"
```

```
"-----END CERTIFICATE-----\n";

const char* client_certificate =
"-----BEGIN CERTIFICATE-----\n"
"YOUR_DEVICE_CERTIFICATE\n"
"-----END CERTIFICATE-----\n";

const char* private_key =
"-----BEGIN RSA PRIVATE KEY-----\n"
"YOUR_DEVICE_PRIVATE_KEY\n"
"-----END RSA PRIVATE KEY-----\n";

WiFiClientSecure espClient;
PubSubClient client(espClient);

long lastMsg = 0;
char msg[50];
int value = 0;

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect with client ID, username, password (if applicable)
    // For AWS IoT, client certs are used for authentication, so
username/password might not be needed here.
    if (client.connect(mqtt_client_id)) {
      Serial.println("connected");
      // Subscribe to a topic if needed
      client.subscribe("nodemcu/secure/commands");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  setup_wifi();

  // Set certificates for secure connection
  espClient.setCACert(aws_iot_root_ca);
  espClient.setCertificate(client_certificate);
  espClient.setPrivateKey(private_key);

  client.setServer(mqtt_server, mqtt_port);
  // Optional: Set a callback for incoming messages if subscribing
  // client.setCallback(callback);
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop(); // Keep the MQTT client alive

  long now = millis();
  if (now - lastMsg > 10000) { // Publish every 10 seconds
    lastMsg = now;
    value++;
    snprintf (msg, 50, "hello world #%ld", value);
    Serial.print("Publishing message: ");
    Serial.println(msg);
    client.publish("nodemcu/secure/data", msg);
  }
}
```

Input:

- Valid Root CA certificate, client certificate, and private key obtained from the cloud IoT platform.
- Correct MQTT broker endpoint and port for secure communication.

Expected Output:

- NodeMCU successfully connects to Wi-Fi.
- NodeMCU establishes a secure TLS/SSL connection to the cloud MQTT broker.
- Data is published to the specified topic (e.g., `nodemcu/secure/data`) using the secure connection.
- An MQTT client (configured for secure connection) subscribed to the topic will receive the data.
- Serial Monitor will show "connected" messages and confirmation of published messages, indicating successful secure communication. If there are certificate issues, errors will be printed.

Lab 6: Cloud-triggered Actions

Title: Cloud-triggered Actions: Controlling Devices on NodeMCU from a Cloud IoT Platform

Aim: To create a program on NodeMCU to perform specific actions (e.g., turn on/off an LED) based on commands received from a cloud-based IoT platform.

Procedure:

1. **Hardware Setup:** Connect an LED to a digital pin on your NodeMCU (e.g., D1).
2. **Cloud IoT Platform Setup:**
   - Choose a cloud IoT platform (e.g., AWS IoT Core, Google Cloud IoT Core, Azure IoT Hub, or Adafruit IO).
   - Create a "device" in the platform and configure its MQTT endpoint.
   - Define a topic for commands (e.g., `your_device_id/commands`).
   - Set up a mechanism to publish messages to this command topic from the cloud (e.g., using the platform's console, a rule engine, or an API call).
3. **Software Setup (Arduino IDE):** Install ESP8266 board package and `PubSubClient` library. If using TLS/SSL, include `WiFiClientSecure`.
4. **Code Development (Arduino IDE):**
   - Include Wi-Fi connection code.
   - Set up the MQTT client to connect to the cloud IoT platform's MQTT broker (securely if possible, as in Lab 5).
   - Subscribe to the command topic (e.g., `your_device_id/commands`).
   - Implement the `callback` function to process incoming messages.
   - Inside `callback`, parse the message (e.g., "ON", "OFF") and control the LED accordingly using `digitalWrite()`.
5. **Upload and Test:** Upload the code to NodeMCU. Use the cloud IoT platform's console or API to send commands to the device's command topic. Observe the LED's response.

Source Code:

```
#include <ESP8266WiFi.h>
#include <PubSubClient.h>
// #include <WiFiClientSecure.h> // Uncomment and configure for TLS/SSL if
your platform requires it

// WiFi credentials
const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

// MQTT Broker details (adjust for your cloud platform, e.g., Adafruit IO,
AWS IoT)
const char* mqtt_server = "YOUR_MQTT_BROKER_ADDRESS"; // e.g.,
"io.adafruit.com"
const int mqtt_port = 1883; // Or 8883 for SSL/TLS
const char* mqtt_username = "YOUR_MQTT_USERNAME"; // e.g., Adafruit IO
username
const char* mqtt_password = "YOUR_MQTT_KEY";     // e.g., Adafruit IO AIO Key
const char* mqtt_client_id = "NodeMCU_Cloud_Action_Client";

// Topic to subscribe to for commands
const char* command_topic = "YOUR_MQTT_USERNAME/feeds/led-control"; // e.g.,
for Adafruit IO

// LED pin
const int LED_PIN = D1; // Connect LED to D1 on NodeMCU (GPIO5)
```

```
// WiFiClient espClient; // Use this for insecure connection
// WiFiClientSecure espClient; // Use this for secure connection (uncomment
above and configure certs)

WiFiClient espClient; // Using regular client for simplicity, but secure is
recommended for production
PubSubClient client(espClient);

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

// Callback function to handle incoming MQTT messages
void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");
  String message = "";
  for (int i = 0; i < length; i++) {
    message += (char)payload[i];
  }
  Serial.println(message);

  if (String(topic) == command_topic) {
    if (message == "ON") {
      digitalWrite(LED_PIN, HIGH); // Turn LED ON
      Serial.println("LED ON");
    } else if (message == "OFF") {
      digitalWrite(LED_PIN, LOW);  // Turn LED OFF
      Serial.println("LED OFF");
    } else {
      Serial.println("Unknown command: " + message);
    }
  }
}

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect with client ID, username, password
    if (client.connect(mqtt_client_id, mqtt_username, mqtt_password)) {
      Serial.println("connected");
      // Once connected, subscribe to the command topic
      client.subscribe(command_topic);
      Serial.print("Subscribed to topic: ");
      Serial.println(command_topic);
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, LOW); // Ensure LED is off initially
  setup_wifi();

  client.setServer(mqtt_server, mqtt_port);
  client.setCallback(callback); // Set the callback function
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop(); // Must be called frequently to process incoming messages
}
```

Input:

- Commands published from the cloud IoT platform's console or API to the subscribed command topic (e.g., `ON` or `OFF`).

Expected Output:

- NodeMCU successfully connects to Wi-Fi and the cloud IoT platform's MQTT broker.
- NodeMCU subscribes to the designated command topic.
- When a message (e.g., "ON") is sent from the cloud platform to the command topic, the LED connected to NodeMCU turns on.
- When "OFF" is sent, the LED turns off.
- Serial Monitor will display incoming messages and the corresponding action (e.g., "LED ON", "LED OFF").

Lab 7: Real-time Data Streaming

Title: Real-time Data Streaming: Streaming Sensor Data from NodeMCU to a Cloud-Based Database

Aim: To develop a program to stream real-time sensor data from NodeMCU to a cloud-based database (e.g., Firebase Realtime Database, AWS DynamoDB).

Procedure:

1. **Hardware Setup:** Connect a sensor (e.g., DHT11/DHT22) to your NodeMCU.
2. **Cloud Database Setup:**
   - Choose a cloud-based database service (e.g., Google Firebase Realtime Database, AWS DynamoDB, MongoDB Atlas).
   - Create a new project/table and configure access rules to allow writes from your NodeMCU (securely, if possible).
   - Obtain necessary credentials (e.g., Firebase API Key, Database URL, AWS Access Key/Secret Key).
3. **Software Setup (Arduino IDE):**
   - Install ESP8266 board package.
   - For Firebase, install the `Firebase-ESP8266` library. For other databases, you might use HTTP client libraries (e.g., `ESP8266HTTPClient`) to interact with their REST APIs.
4. **Code Development (Arduino IDE):**
   - Include Wi-Fi connection code.
   - Initialize the database client with credentials.
   - Read sensor data.
   - Construct the data payload (e.g., JSON object) with sensor readings and a timestamp.
   - Send the data to the cloud database using the appropriate library functions (e.g., `Firebase.set()`, `Firebase.push()`) or HTTP POST requests.
   - Implement error handling for database operations.
5. **Upload and Test:** Upload the code to NodeMCU. Monitor the serial output. Check your cloud database console to verify that new sensor data entries are appearing in real-time.

Source Code (Example with Firebase Realtime Database):

```
#include <ESP8266WiFi.h>
#include <FirebaseESP8266.h> // For Firebase Realtime Database
#include <DHT.h> // For DHT11/DHT22 sensor

// WiFi credentials
const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

// Firebase project details
#define FIREBASE_HOST "YOUR_FIREBASE_PROJECT_ID.firebaseio.com" // e.g.,
"your-project-id.firebaseio.com"
#define FIREBASE_AUTH "YOUR_FIREBASE_DATABASE_SECRET" // Legacy database
secret or service account private key (less secure for device)

// Recommended: Use Firebase Authentication with Email/Password or Anonymous
for device
// For simplicity, using database secret here. For production, use Firebase
Auth.
// If using API Key for Web API access, you'd use that instead of
FIREBASE_AUTH for some operations.
```

```
// DHT Sensor details
#define DHTPIN D2     // Digital pin connected to the DHT sensor
#define DHTTYPE DHT11   // DHT 11 or DHT 22
DHT dht(DHTPIN, DHTTYPE);

FirebaseData firebaseData;
FirebaseJson json;

long lastSendTime = 0;
const long sendInterval = 15000; // Send data every 15 seconds

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void setup() {
  Serial.begin(115200);
  dht.begin();
  setup_wifi();

  // Connect to Firebase
  Firebase.begin(FIREBASE_HOST, FIREBASE_AUTH);
  Firebase.reconnectWiFi(true); // Automatically reconnect WiFi if
disconnected
  Serial.println("Firebase initialized.");
}

void loop() {
  if (WiFi.status() == WL_CONNECTED) {
    if (millis() - lastSendTime > sendInterval) {
      lastSendTime = millis();

      // Read sensor data
      float h = dht.readHumidity();
      float t = dht.readTemperature();

      if (isnan(h) || isnan(t)) {
        Serial.println("Failed to read from DHT sensor!");
        return;
      }

      // Create JSON object for sensor data
      json.set("temperature", String(t, 2)); // 2 decimal places
      json.set("humidity", String(h, 2));
      json.set("timestamp", Firebase.getCurrentTimestamp()); // Add timestamp

      // Push data to Firebase (creates a unique key)
      // Path: /sensor_data
      if (Firebase.pushJSON(firebaseData, "/sensor_data", json)) {
        Serial.println("PASSED: Data pushed to Firebase");
        Serial.println("PATH: " + firebaseData.dataPath());
        Serial.println("PUSH NAME: " + firebaseData.pushName());
      } else {
        Serial.println("FAILED: Failed to push data to Firebase");
        Serial.println("REASON: " + firebaseData.errorReason());
      }
    }
  } else {
    Serial.println("WiFi Disconnected. Reconnecting...");
    setup_wifi(); // Attempt to reconnect
  }
}
```

Input: Real-time temperature and humidity readings from the connected DHT sensor.

Expected Output:

- NodeMCU successfully connects to Wi-Fi and authenticates with Firebase.
- Periodically, new JSON objects containing temperature, humidity, and a timestamp are added as new entries under the `/sensor_data` path in your Firebase Realtime Database.
- The Firebase console will show the streaming data being updated in real-time.
- Serial Monitor will confirm successful data pushes or report any errors.

Lab 8: Data Visualization

Title: Data Visualization: Interfacing NodeMCU with a Cloud-Based Data Visualization Platform

Aim: To interface NodeMCU with a cloud-based data visualization platform (e.g., ThingSpeak, Grafana) to visualize sensor data in real-time.

Procedure:

1. **Hardware Setup:** Connect a sensor (e.g., DHT11/DHT22) to your NodeMCU.
2. **Cloud Visualization Platform Setup:**
   - Choose a platform (e.g., ThingSpeak, Grafana Cloud, Ubidots, or a custom dashboard using data from Firebase/DynamoDB).
   - For ThingSpeak: Create a channel, define fields (e.g., temperature, humidity), and note down the Write API Key. ThingSpeak automatically generates charts.
   - For Grafana: Set up a Grafana instance (cloud or self-hosted), configure a data source (e.g., InfluxDB, Prometheus, or a database where your data is stored), and create dashboards with panels to visualize your data.
3. **Software Setup (Arduino IDE):** Install ESP8266 board package. For ThingSpeak, install the `ThingSpeak` library. For other platforms, you might use `ESP8266HTTPClient` or MQTT libraries.
4. **Code Development (Arduino IDE):**
   - Include Wi-Fi connection code.
   - Initialize the ThingSpeak client with the channel ID and Write API Key.
   - Read sensor data.
   - Set the sensor values to ThingSpeak fields.
   - Write the data to the ThingSpeak channel.
   - Implement error checking for ThingSpeak writes.
5. **Upload and Test:** Upload the code to NodeMCU. Monitor the serial output. Open your ThingSpeak channel's private view or Grafana dashboard. Observe the real-time charts updating with your sensor data.

Source Code (Example with ThingSpeak):

```
#include <ESP8266WiFi.h>
#include <ThingSpeak.h> // For ThingSpeak integration
#include <DHT.h> // For DHT11/DHT22 sensor

// WiFi credentials
const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

// ThingSpeak Channel details
unsigned long myChannelNumber = YOUR_THINGSPEAK_CHANNEL_ID; // e.g., 123456
const char* myWriteAPIKey = "YOUR_THINGSPEAK_WRITE_API_KEY"; // e.g.,
"ABCDEF1234567890"

// DHT Sensor details
#define DHTPIN D2     // Digital pin connected to the DHT sensor
#define DHTTYPE DHT11   // DHT 11 or DHT 22
DHT dht(DHTPIN, DHTTYPE);

WiFiClient client;

long lastSendTime = 0;
const long sendInterval = 20000; // Send data every 20 seconds
```

```
void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void setup() {
  Serial.begin(115200);
  dht.begin();
  setup_wifi();

  ThingSpeak.begin(client); // Initialize ThingSpeak
  Serial.println("ThingSpeak initialized.");
}

void loop() {
  if (WiFi.status() == WL_CONNECTED) {
    if (millis() - lastSendTime > sendInterval) {
      lastSendTime = millis();

      // Read sensor data
      float h = dht.readHumidity();
      float t = dht.readTemperature();

      if (isnan(h) || isnan(t)) {
        Serial.println("Failed to read from DHT sensor!");
        return;
      }

      // Set the fields with the sensor data
      ThingSpeak.setField(1, t); // Field 1 for Temperature
      ThingSpeak.setField(2, h); // Field 2 for Humidity

      // Write the data to the ThingSpeak channel
      int x = ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
      if (x == 200) {
        Serial.println("Channel update successful.");
      } else {
        Serial.println("Problem updating channel. HTTP error code " +
String(x));
      }
    }
  } else {
    Serial.println("WiFi Disconnected. Reconnecting...");
    setup_wifi(); // Attempt to reconnect
  }
}
```

Input: Real-time temperature and humidity readings from the connected DHT sensor.

Expected Output:

- NodeMCU successfully connects to Wi-Fi and ThingSpeak.
- Periodically, temperature and humidity data are sent to your ThingSpeak channel.
- On the ThingSpeak channel's private view, the charts for Field 1 (Temperature) and Field 2 (Humidity) will update in real-time, displaying the incoming sensor data.
- Serial Monitor will show "Channel update successful" messages.

Lab 9: Device Shadowing

Title: Device Shadowing: Synchronizing Device States and Configurations with a Cloud-Based IoT Platform

Aim: To implement device shadowing functionality on NodeMCU to synchronize device states and configurations with a cloud-based IoT platform.

Procedure:

1. **Cloud IoT Platform Setup:**
   - Choose a cloud IoT platform that supports device shadowing (e.g., AWS IoT Core, Google Cloud IoT Core).
   - Create a "thing" (device) in the platform. The platform automatically creates a "device shadow" for it.
   - Note down the MQTT endpoint for device shadows.
   - Configure policies/permissions to allow your device to update and get its shadow.
2. **Software Setup (Arduino IDE):** Install ESP8266 board package, `PubSubClient`, and `WiFiClientSecure`. You'll also need a JSON library (e.g., `ArduinoJson`) to parse and construct shadow messages.
3. **Code Development (Arduino IDE):**
   - Include Wi-Fi connection code.
   - Set up secure MQTT client using `WiFiClientSecure` and certificates (as in Lab 5) to connect to the IoT platform's MQTT broker.
   - Subscribe to the device shadow topics for updates (`$aws/things/YOUR_THING_NAME/shadow/update/accepted`, `$aws/things/YOUR_THING_NAME/shadow/get/accepted`).
   - Implement a `callback` function to process incoming shadow messages. This function will parse the `desired` state and update the device's physical state (e.g., LED).
   - Periodically publish the device's `reported` state to the shadow update topic (`$aws/things/YOUR_THING_NAME/shadow/update`).
   - Implement logic to request the current shadow state on boot (`$aws/things/YOUR_THING_NAME/shadow/get`).
4. **Upload and Test:** Upload the code to NodeMCU. Use the cloud platform's console to update the device's "desired" state in the shadow. Observe the NodeMCU's response. Also, verify that the "reported" state from NodeMCU is updated in the shadow.

Source Code (Example with AWS IoT Core Device Shadow):

```
#include <ESP8266WiFi.h>
#include <WiFiClientSecure.h>
#include <PubSubClient.h>
#include <ArduinoJson.h> // For JSON parsing and creation

// WiFi credentials
const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

// AWS IoT Core details
const char* aws_iot_endpoint = "YOUR_AWS_IOT_ENDPOINT.iot.your-
region.amazonaws.com";
const int mqtt_port = 8883; // Secure MQTT port
const char* thing_name = "YourNodeMCUThingName"; // Must match your AWS IoT
Thing name
```

```
// AWS IoT Core requires client certificates and a root CA
// Replace with your actual certificates and private key
const char* aws_iot_root_ca =
"-----BEGIN CERTIFICATE-----\n"
"YOUR_AWS_IOT_ROOT_CA_CERTIFICATE\n"
"-----END CERTIFICATE-----\n";

const char* client_certificate =
"-----BEGIN CERTIFICATE-----\n"
"YOUR_DEVICE_CERTIFICATE\n"
"-----END CERTIFICATE-----\n";

const char* private_key =
"-----BEGIN RSA PRIVATE KEY-----\n"
"YOUR_DEVICE_PRIVATE_KEY\n"
"-----END RSA PRIVATE KEY-----\n";

// MQTT Topics for Device Shadow
String shadow_update_topic = "$aws/things/" + String(thing_name) +
"/shadow/update";
String shadow_update_accepted_topic = "$aws/things/" + String(thing_name) +
"/shadow/update/accepted";
String shadow_get_topic = "$aws/things/" + String(thing_name) +
"/shadow/get";
String shadow_get_accepted_topic = "$aws/things/" + String(thing_name) +
"/shadow/get/accepted";

// Device state variables
bool ledState = false; // Initial LED state
const int LED_PIN = D1; // Connect LED to D1 on NodeMCU (GPIO5)

WiFiClientSecure espClient;
PubSubClient client(espClient);

long lastReportedTime = 0;
const long reportInterval = 30000; // Report state every 30 seconds

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

// Function to update the LED based on state
void updateLed() {
  digitalWrite(LED_PIN, ledState ? HIGH : LOW);
  Serial.print("LED is now: ");
  Serial.println(ledState ? "ON" : "OFF");
}

// Function to publish the device's reported state to the shadow
void publishReportedState() {
  DynamicJsonDocument doc(256);
  JsonObject state = doc.createNestedObject("state");
  JsonObject reported = state.createNestedObject("reported");
  reported["led"] = ledState ? "ON" : "OFF";

  String payload;
  serializeJson(doc, payload);

  Serial.print("Publishing reported state: ");
  Serial.println(payload);
  client.publish(shadow_update_topic.c_str(), payload.c_str());
}

// Callback function to handle incoming MQTT messages (shadow updates)
void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");

  String message = "";
  for (int i = 0; i < length; i++) {
    message += (char)payload[i];
  }
  Serial.println(message);

  DynamicJsonDocument doc(512); // Adjust size as needed
  DeserializationError error = deserializeJson(doc, message);

  if (error) {
    Serial.print(F("deserializeJson() failed: "));
    Serial.println(error.f_str());
    return;
  }

  // Handle shadow update/accepted messages
  if (String(topic) == shadow_update_accepted_topic || String(topic) ==
shadow_get_accepted_topic) {
    // Check for desired state changes
    if (doc["state"]["desired"].containsKey("led")) {
      String desiredLedState = doc["state"]["desired"]["led"].as<String>();
      if (desiredLedState == "ON") {
        ledState = true;
      } else if (desiredLedState == "OFF") {
        ledState = false;
      }
      updateLed();
      // After processing desired state, report the new state
      publishReportedState();
    }
    // Also, if there's a reported state in the incoming message, update our
internal state if necessary
    // (useful for initial sync on boot or if another source updates the
shadow)
    if (doc["state"]["reported"].containsKey("led")) {
      String reportedLedState = doc["state"]["reported"]["led"].as<String>();
      bool currentLedState = (reportedLedState == "ON");
      if (currentLedState != ledState) {
        ledState = currentLedState;
        updateLed();
      }
    }
  }
}

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect(thing_name)) { // Use thing name as client ID
      Serial.println("connected");
      // Subscribe to shadow topics
      client.subscribe(shadow_update_accepted_topic.c_str());
      client.subscribe(shadow_get_accepted_topic.c_str());
      Serial.println("Subscribed to shadow topics.");

      // Request current shadow state on connect/reconnect
      Serial.println("Requesting current shadow state...");
      client.publish(shadow_get_topic.c_str(), "");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, LOW); // Ensure LED is off initially
  setup_wifi();

  // Set certificates for secure connection
  espClient.setCACert(aws_iot_root_ca);
  espClient.setCertificate(client_certificate);
  espClient.setPrivateKey(private_key);

  client.setServer(aws_iot_endpoint, mqtt_port);
  client.setCallback(callback);
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop(); // Keep the MQTT client alive

  long now = millis();
  if (now - lastReportedTime > reportInterval) {
    lastReportedTime = now;
    publishReportedState(); // Periodically report current state
  }
}
```

Input:

- Updates to the "desired" state of the device shadow from the AWS IoT Core console or API (e.g., setting `{"state":{"desired":{"led":"ON"}}}`).

Expected Output:

- NodeMCU connects securely to AWS IoT Core.
- NodeMCU subscribes to the device shadow update and get accepted topics.
- When the "desired" state of the LED in the device shadow is changed (e.g., to "ON"), the NodeMCU receives the update, turns the physical LED on, and then publishes its "reported" state back to the shadow, confirming the change.
- The AWS IoT Core console will show the "reported" state of the LED reflecting the physical device's state.

- Serial Monitor will display incoming shadow messages, LED state changes, and reported state publications.

Lab 10: Integration with IoT Platforms

Title: Integration with IoT Platforms: Leveraging Cloud IoT Services for IoT Applications

Aim: To integrate NodeMCU with popular cloud-based IoT platforms (e.g., AWS IoT, Google Cloud IoT Core) to leverage their services for IoT applications. This lab builds upon previous labs, focusing on using the platform's full capabilities.

Procedure:

1. **Cloud IoT Platform Setup:**
    - Choose an IoT platform (e.g., AWS IoT Core).
    - Create a "thing" (device) and attach policies for MQTT publish/subscribe.
    - Generate and download device certificates and private key.
    - Set up Rules Engine to process incoming data (e.g., store in DynamoDB, trigger Lambda functions, send notifications).
    - Set up Device Defender (optional) for security monitoring.
2. **Software Setup (Arduino IDE):** Install ESP8266 board package, `PubSubClient`, `WiFiClientSecure`, and a JSON library (`ArduinoJson`).
3. **Code Development (Arduino IDE):**
    - **Secure Connection:** Implement secure Wi-Fi and MQTT connection to the platform using TLS/SSL and device certificates (as in Lab 5).
    - **Publish Sensor Data:** Publish sensor data (e.g., temperature, humidity) to a specific topic (e.g., `iot/data/sensor`) that a platform rule can process.
    - **Subscribe to Commands:** Subscribe to a command topic (e.g., `iot/commands/led`) and implement a callback to control an LED or other actuator based on messages from the platform (as in Lab 6).
    - **Device Shadow (Optional but Recommended):** Integrate device shadowing to maintain synchronized state between the device and the cloud (as in Lab 9).
4. **Platform Configuration:**
    - **Rules Engine:** Create a rule that triggers on your sensor data topic.
        - Action 1: Send data to a database (e.g., DynamoDB table).
        - Action 2: Publish a message to another topic (e.g., for alerts).
    - **Device Management:** Use the platform's device management features to monitor device status.
    - **Authentication & Authorization:** Understand and apply appropriate IAM policies/roles for secure interactions.
5. **Upload and Test:** Upload the code to NodeMCU. Verify data flow from NodeMCU to the cloud platform and its services. Test sending commands from the platform to NodeMCU.

Source Code (Conceptual - combines elements from Lab 5, 6, 7, 9):

```
#include <ESP8266WiFi.h>
#include <WiFiClientSecure.h>
#include <PubSubClient.h>
#include <ArduinoJson.h> // For JSON payload creation/parsing
#include <DHT.h> // For DHT11/DHT22 sensor

// WiFi credentials
const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

// Cloud IoT Platform (e.g., AWS IoT Core) details
const char* iot_endpoint = "YOUR_AWS_IOT_ENDPOINT.iot.your-
region.amazonaws.com";
```

```
const int mqtt_port = 8883;
const char* device_id = "NodeMCU_MyIoTDevice"; // Unique device ID/Thing Name

// Certificates and Private Key (from your IoT platform)
const char* iot_root_ca =
"-----BEGIN CERTIFICATE-----\n"
"YOUR_IOT_ROOT_CA_CERTIFICATE\n"
"-----END CERTIFICATE-----\n";

const char* device_certificate =
"-----BEGIN CERTIFICATE-----\n"
"YOUR_DEVICE_CERTIFICATE\n"
"-----END CERTIFICATE-----\n";

const char* device_private_key =
"-----BEGIN RSA PRIVATE KEY-----\n"
"YOUR_DEVICE_PRIVATE_KEY\n"
"-----END RSA PRIVATE KEY-----\n";

// MQTT Topics
const char* publish_topic = "iot/data/sensor"; // Topic for publishing sensor
data
const char* subscribe_topic = "iot/commands/led"; // Topic for receiving
commands

// Device Shadow Topics (optional, but good for robust integration)
String shadow_update_topic = "$aws/things/" + String(device_id) +
"/shadow/update";
String shadow_update_accepted_topic = "$aws/things/" + String(device_id) +
"/shadow/update/accepted";
String shadow_get_topic = "$aws/things/" + String(device_id) + "/shadow/get";
String shadow_get_accepted_topic = "$aws/things/" + String(device_id) +
"/shadow/get/accepted";

// Sensor and Actuator
#define DHTPIN D2
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);
const int LED_PIN = D1;
bool ledState = false;

WiFiClientSecure espClient;
PubSubClient client(espClient);

long lastSensorReadTime = 0;
const long sensorReadInterval = 10000; // Read and publish sensor data every
10 seconds

long lastShadowReportTime = 0;
const long shadowReportInterval = 30000; // Report shadow state every 30
seconds

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

// Function to update the LED based on state
void updateLed() {
  digitalWrite(LED_PIN, ledState ? HIGH : LOW);
  Serial.print("LED is now: ");
  Serial.println(ledState ? "ON" : "OFF");
}

// Function to publish the device's reported state to the shadow
void publishReportedShadowState() {
  DynamicJsonDocument doc(256);
  JsonObject state = doc.createNestedObject("state");
  JsonObject reported = state.createNestedObject("reported");
  reported["led"] = ledState ? "ON" : "OFF";

  String payload;
  serializeJson(doc, payload);

  Serial.print("Publishing reported shadow state: ");
  Serial.println(payload);
  client.publish(shadow_update_topic.c_str(), payload.c_str());
}

// Callback function for incoming MQTT messages
void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");
  String message = "";
  for (int i = 0; i < length; i++) {
    message += (char)payload[i];
  }
  Serial.println(message);

  // Handle commands from cloud
  if (String(topic) == subscribe_topic) {
    if (message == "ON") {
      ledState = true;
      updateLed();
      publishReportedShadowState(); // Report change to shadow
    } else if (message == "OFF") {
      ledState = false;
      updateLed();
      publishReportedShadowState(); // Report change to shadow
    }
  }
  // Handle device shadow messages
  else if (String(topic) == shadow_update_accepted_topic || String(topic) ==
shadow_get_accepted_topic) {
    DynamicJsonDocument doc(512);
    DeserializationError error = deserializeJson(doc, message);

    if (error) {
      Serial.print(F("deserializeJson() failed: "));
      Serial.println(error.f_str());
      return;
    }

    if (doc["state"]["desired"].containsKey("led")) {
      String desiredLedState = doc["state"]["desired"]["led"].as<String>();
      if (desiredLedState == "ON") {
        ledState = true;
      } else if (desiredLedState == "OFF") {
        ledState = false;
      }
      updateLed();
      publishReportedShadowState(); // Report new state after desired change
    }
  }
}

void reconnect() {
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect(device_id)) {
      Serial.println("connected");
      // Subscribe to command topic
      client.subscribe(subscribe_topic);
      Serial.print("Subscribed to command topic: ");
      Serial.println(subscribe_topic);

      // Subscribe to shadow topics
      client.subscribe(shadow_update_accepted_topic.c_str());
      client.subscribe(shadow_get_accepted_topic.c_str());
      Serial.println("Subscribed to shadow topics.");

      // Request current shadow state on connect/reconnect
      Serial.println("Requesting current shadow state...");
      client.publish(shadow_get_topic.c_str(), "");

    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  dht.begin();
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, LOW);
  setup_wifi();

  espClient.setCACert(iot_root_ca);
  espClient.setCertificate(device_certificate);
  espClient.setPrivateKey(device_private_key);

  client.setServer(iot_endpoint, mqtt_port);
  client.setCallback(callback);
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop();

  long now = millis();

  // Publish sensor data
  if (now - lastSensorReadTime > sensorReadInterval) {
    lastSensorReadTime = now;
    float h = dht.readHumidity();
    float t = dht.readTemperature();
    if (isnan(h) || isnan(t)) {
      Serial.println("Failed to read from DHT sensor!");
      return;
    }

    DynamicJsonDocument doc(256);
    doc["temperature"] = String(t, 2);
    doc["humidity"] = String(h, 2);
    doc["timestamp"] = now;

    String payload;
    serializeJson(doc, payload);

    Serial.print("Publishing sensor data: ");
    Serial.println(payload);
    client.publish(publish_topic, payload.c_str());
  }

  // Periodically report shadow state (even if no change, for robustness)
  if (now - lastShadowReportTime > shadowReportInterval) {
    lastShadowReportTime = now;
    publishReportedShadowState();
  }
}
```

Input:

- Sensor readings from NodeMCU.
- Commands sent from the cloud IoT platform (e.g., via console, rules, or other services).
- Updates to the device shadow's "desired" state from the cloud.

Expected Output:

- NodeMCU securely connects to the cloud IoT platform.
- Sensor data is published to the designated topic and processed by platform rules (e.g., stored in a database, triggering alerts).
- Commands sent from the cloud platform are received by NodeMCU, triggering actions (e.g., LED control).
- The device shadow accurately reflects the device's reported state, and the device updates its physical state based on desired shadow changes.
- The cloud IoT platform's console will show active devices, data streams, and shadow synchronization.

Lab 11: MQTT Publisher (Raspberry Pi)

Title: MQTT Publisher: Publishing Sensor Data from Raspberry Pi to a Cloud MQTT Broker

Aim: To develop a Python script on Raspberry Pi to publish data from sensors connected to GPIO pins to an MQTT broker hosted on a cloud platform.

Procedure:

1. **Hardware Setup:**
   - Set up your Raspberry Pi with Raspbian OS.
   - Connect a sensor (e.g., DHT11/DHT22) to the Raspberry Pi's GPIO pins.
2. **Software Setup (Raspberry Pi):**
   - Ensure Python 3 is installed.
   - Install necessary Python libraries: `paho-mqtt` (for MQTT client) and a library for your sensor (e.g., `Adafruit_DHT` for DHT sensors).
   - `sudo apt-get update`
   - `sudo apt-get install python3-pip`
   - `pip3 install paho-mqtt`
   - `# For DHT sensor:`
   - `sudo apt-get install libgpiod2`
   - `pip3 install Adafruit_DHT`
3. **Cloud MQTT Broker Configuration:** Use the same cloud MQTT broker details (address, port, credentials) as in Lab 1.
4. **Code Development (Python):**
   - Write a Python script.
   - Import `paho.mqtt.client` and your sensor library.
   - Configure MQTT client with broker details.
   - Define a callback for `on_connect`.
   - Read sensor data from GPIO pins.
   - Publish the sensor data to a specific MQTT topic (e.g., `raspberrypi/sensor/temperature`).
   - Implement a loop to periodically read and publish data.
   - Handle potential errors in sensor reading or MQTT connection.
5. **Execution and Test:** Run the Python script on Raspberry Pi. Use an MQTT client to subscribe to the topic and verify data reception on the cloud broker.

Source Code:

```
import paho.mqtt.client as mqtt
import time
import Adafruit_DHT # For DHT11/DHT22 sensor

# MQTT Broker details
MQTT_BROKER = "YOUR_MQTT_BROKER_ADDRESS" # e.g., "broker.hivemq.com"
MQTT_PORT = 1883 # Or 8883 for SSL/TLS
MQTT_USERNAME = "YOUR_MQTT_USERNAME" # If required
MQTT_PASSWORD = "YOUR_MQTT_PASSWORD" # If required
MQTT_CLIENT_ID = "RaspberryPi_Publisher_789"

# MQTT Topics
TEMPERATURE_TOPIC = "raspberrypi/sensor/temperature"
HUMIDITY_TOPIC = "raspberrypi/sensor/humidity"

# DHT Sensor details
DHT_SENSOR = Adafruit_DHT.DHT11 # Or Adafruit_DHT.DHT22
```

```
DHT_PIN = 4 # GPIO pin connected to the DHT sensor (e.g., GPIO 4)

# The callback for when the client receives a CONNACK response from the
server.
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connected to MQTT Broker!")
    else:
        print(f"Failed to connect, return code {rc}\n")

client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION1, MQTT_CLIENT_ID) #
Specify API version
client.username_pw_set(MQTT_USERNAME, MQTT_PASSWORD) # Set username and
password if required
client.on_connect = on_connect

try:
    client.connect(MQTT_BROKER, MQTT_PORT, 60)
except Exception as e:
    print(f"Could not connect to MQTT broker: {e}")
    exit()

client.loop_start() # Start a non-blocking loop for network traffic

while True:
    try:
        humidity, temperature = Adafruit_DHT.read_retry(DHT_SENSOR, DHT_PIN)

        if humidity is not None and temperature is not None:
            print(f"Temp={temperature:.2f}C Humidity={humidity:.2f}%")

            # Publish temperature
            client.publish(TEMPERATURE_TOPIC, f"{temperature:.2f}")
            print(f"Published temperature to {TEMPERATURE_TOPIC}")

            # Publish humidity
            client.publish(HUMIDITY_TOPIC, f"{humidity:.2f}")
            print(f"Published humidity to {HUMIDITY_TOPIC}")
        else:
            print("Failed to retrieve data from humidity sensor.")

    except RuntimeError as error:
        # Errors happen when DHT is not connected or not working
        print(error.args[0])
    except Exception as e:
        print(f"An error occurred: {e}")

    time.sleep(5) # Publish every 5 seconds
```

Input: Real-time temperature and humidity readings from the connected DHT sensor on Raspberry Pi.

Expected Output:

- Raspberry Pi successfully connects to the MQTT broker.
- Temperature and humidity data are periodically published to the `raspberrypi/sensor/temperature` and `raspberrypi/sensor/humidity` topics on the cloud MQTT broker.
- An MQTT client subscribed to these topics will display the received sensor values, e.g.:
    - o `raspberrypi/sensor/temperature`: `26.15`
    - o `raspberrypi/sensor/humidity`: `58.90`

- Console output on Raspberry Pi will show "Connected to MQTT Broker!" and "Published..." messages.

Lab 12: MQTT Subscriber (Raspberry Pi)

Title: MQTT Subscriber: Subscribing to MQTT Topics on Raspberry Pi and Taking Actions

Aim: To create a Python script to subscribe to MQTT topics on Raspberry Pi and take actions based on messages received from a cloud-based MQTT broker (e.g., controlling a GPIO pin for an LED).

Procedure:

1. **Hardware Setup:**
   - Set up your Raspberry Pi.
   - Connect an LED to a GPIO pin on the Raspberry Pi (e.g., GPIO 17). Include a current-limiting resistor.
2. **Software Setup (Raspberry Pi):**
   - Ensure Python 3 is installed.
   - Install `paho-mqtt` and `RPi.GPIO` (for GPIO control).
   - `pip3 install paho-mqtt RPi.GPIO`

3. **Cloud MQTT Broker Configuration:** Use the same MQTT broker details as in previous labs. Define a topic for receiving commands (e.g., `raspberrypi/commands/led`).
4. **Code Development (Python):**
   - Write a Python script.
   - Import `paho.mqtt.client` and `RPi.GPIO`.
   - Configure GPIO pin for output.
   - Configure MQTT client and define `on_connect` and `on_message` callbacks.
   - In `on_connect`, subscribe to the command topic.
   - In `on_message`, parse the incoming message (e.g., "ON", "OFF") and control the LED using `GPIO.output()`.
   - Start the MQTT client loop.
5. **Execution and Test:** Run the Python script on Raspberry Pi. Use an MQTT client to publish messages to the command topic and observe the LED's behavior.

Source Code:

```
import paho.mqtt.client as mqtt
import RPi.GPIO as GPIO # For controlling GPIO pins
import time

# MQTT Broker details
MQTT_BROKER = "YOUR_MQTT_BROKER_ADDRESS"
MQTT_PORT = 1883
MQTT_USERNAME = "YOUR_MQTT_USERNAME" # If required
MQTT_PASSWORD = "YOUR_MQTT_PASSWORD" # If required
MQTT_CLIENT_ID = "RaspberryPi_Subscriber_012"

# MQTT Topic to subscribe to for commands
COMMAND_TOPIC = "raspberrypi/commands/led"

# GPIO pin for LED
LED_PIN = 17 # GPIO 17 (physical pin 11)

# GPIO Setup
GPIO.setmode(GPIO.BCM) # Use Broadcom pin-numbering scheme
GPIO.setup(LED_PIN, GPIO.OUT) # Set LED pin as output
GPIO.output(LED_PIN, GPIO.LOW) # Ensure LED is off initially
```

```
# The callback for when the client receives a CONNACK response from the
server.
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connected to MQTT Broker!")
        client.subscribe(COMMAND_TOPIC)
        print(f"Subscribed to topic: {COMMAND_TOPIC}")
    else:
        print(f"Failed to connect, return code {rc}\n")

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    print(f"Message received [{msg.topic}]: {msg.payload.decode()}")
    message = msg.payload.decode().strip().upper() # Convert to uppercase for
consistent comparison

    if msg.topic == COMMAND_TOPIC:
        if message == "ON":
            GPIO.output(LED_PIN, GPIO.HIGH)
            print("LED ON")
        elif message == "OFF":
            GPIO.output(LED_PIN, GPIO.LOW)
            print("LED OFF")
        else:
            print("Unknown command.")

client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION1, MQTT_CLIENT_ID)
client.username_pw_set(MQTT_USERNAME, MQTT_PASSWORD)
client.on_connect = on_connect
client.on_message = on_message

try:
    client.connect(MQTT_BROKER, MQTT_PORT, 60)
except Exception as e:
    print(f"Could not connect to MQTT broker: {e}")
    GPIO.cleanup() # Clean up GPIO on exit
    exit()

client.loop_forever() # Blocks and handles network traffic, callbacks

# Clean up GPIO on exit (if loop_forever is exited, e.g., by
KeyboardInterrupt)
GPIO.cleanup()
print("GPIO cleanup done.")
```

Input: MQTT messages published to the `raspberrypi/commands/led` topic from an external MQTT client or publisher.

- To turn LED ON: Publish `ON` to `raspberrypi/commands/led`
- To turn LED OFF: Publish `OFF` to `raspberrypi/commands/led`

Expected Output:

- Raspberry Pi successfully connects to the MQTT broker and subscribes to the command topic.
- When `ON` is published to the topic, the LED connected to the specified GPIO pin on Raspberry Pi turns on.
- When `OFF` is published, the LED turns off.
- Console output on Raspberry Pi will show "Connected to MQTT Broker!", "Subscribed...", "Message received...", and "LED ON" or "LED OFF" messages.

Lab 13: HTTP Server

Title: HTTP Server: Implementing an HTTP Server on Raspberry Pi to Receive Sensor Data

Aim: To implement an HTTP server on Raspberry Pi to receive sensor data from NodeMCU or other IoT devices and store it in a cloud-based database.

Procedure:

1. **Hardware Setup:** Set up your Raspberry Pi.
2. **Cloud Database Setup:** Choose a cloud-based database (e.g., Firebase Realtime Database, MongoDB Atlas, PostgreSQL on AWS RDS). Configure access credentials.
3. **Software Setup (Raspberry Pi):**
   - Install Python 3.
   - Install a web framework (e.g., Flask) and a database client library (e.g., `firebase-admin` for Firebase, `pymongo` for MongoDB).
   - `pip3 install Flask firebase-admin # Or other database client`
   - For external access, configure port forwarding on your router or use a tunnel service like ngrok if your Raspberry Pi is behind a NAT. For production, deploy to a cloud VM.
4. **Code Development (Python Flask):**
   - Write a Flask application.
   - Initialize the cloud database client.
   - Define an HTTP route (e.g., `/sensor_data`) that accepts POST requests.
   - In the route handler, parse the incoming JSON sensor data from the request body.
   - Store the received data in your cloud database.
   - Return an appropriate HTTP response (e.g., 200 OK).
5. **Execution and Test:** Run the Flask server on Raspberry Pi. From a NodeMCU (using HTTP Client from Lab 3) or a tool like Postman, send HTTP POST requests with JSON sensor data to the Raspberry Pi's IP address and the defined route. Verify that data appears in your cloud database.

Source Code (Example with Flask and Firebase):

```
from flask import Flask, request, jsonify
import firebase_admin
from firebase_admin import credentials, db
import datetime

app = Flask(__name__)

# Initialize Firebase Admin SDK
# Download your Firebase service account key JSON file and place it in the
same directory
# as this script, or provide the full path.
try:
    cred = credentials.Certificate("path/to/your/serviceAccountKey.json")
    firebase_admin.initialize_app(cred, {
        'databaseURL': 'https://YOUR_FIREBASE_PROJECT_ID.firebaseio.com/'
    })
    print("Firebase Admin SDK initialized successfully.")
except Exception as e:
    print(f"Error initializing Firebase Admin SDK: {e}")
    print("Please ensure 'serviceAccountKey.json' is correctly configured and
accessible.")
    exit()
```

```
@app.route('/sensor_data', methods=['POST'])
def receive_sensor_data():
    if request.is_json:
        sensor_data = request.get_json()
        print(f"Received sensor data: {sensor_data}")

        # Add a timestamp
        sensor_data['timestamp'] = datetime.datetime.now().isoformat()

        # Store data in Firebase Realtime Database
        try:
            ref = db.reference('/sensor_readings')
            new_reading_ref = ref.push(sensor_data)
            print(f"Data stored in Firebase with key: {new_reading_ref.key}")
            return jsonify({"message": "Data received and stored 
successfully", "key": new_reading_ref.key}), 200
        except Exception as e:
            print(f"Error storing data in Firebase: {e}")
            return jsonify({"error": "Failed to store data in database"}),
500
    else:
        return jsonify({"error": "Request must be JSON"}), 400

if __name__ == '__main__':
    # Run the Flask app
    # To make it accessible from other devices on the network, use
host='0.0.0.0'
    # Ensure port 5000 is open in your Raspberry Pi's firewall if applicable.
    print("Starting Flask server on http://0.0.0.0:5000")
    app.run(host='0.0.0.0', port=5000)
```

Input: HTTP POST requests with JSON sensor data sent from NodeMCU or other IoT devices to the Raspberry Pi's IP address and `/sensor_data` endpoint. Example JSON payload:

```
{
    "temperature": 25.5,
    "humidity": 60.2,
    "device_id": "nodemcu_001"
}
```

Expected Output:

- The Flask HTTP server starts successfully on Raspberry Pi.
- When a POST request is received at `/sensor_data`, the server parses the JSON data.
- The sensor data, along with a timestamp, is pushed as a new entry to the `/sensor_readings` path in your Firebase Realtime Database.
- The server responds with a 200 OK status and a success message.
- Console output on Raspberry Pi will show received data and Firebase storage confirmation.
- The Firebase console will show new sensor data entries appearing.

Lab 14: Data Logging to Cloud Storage

Title: Data Logging to Cloud Storage: Logging Sensor Data from Raspberry Pi to a Cloud-Based Storage Service

Aim: To write a Python script to log sensor data (e.g., temperature, humidity) from Raspberry Pi to a cloud-based storage service (e.g., Google Cloud Storage, AWS S3).

Procedure:

1. **Hardware Setup:** Set up your Raspberry Pi and connect a sensor (e.g., DHT11/DHT22).
2. **Cloud Storage Setup:**
   - Choose a cloud storage service (e.g., Google Cloud Storage, AWS S3).
   - Create a storage bucket.
   - Configure access credentials (e.g., service account key for Google Cloud, IAM user credentials for AWS S3) and ensure they have write permissions to the bucket.
3. **Software Setup (Raspberry Pi):**
   - Install Python 3.
   - Install the appropriate cloud SDK for your chosen service (e.g., `google-cloud-storage` for GCS, `boto3` for AWS S3).
   - `pip3 install google-cloud-storage # For GCS`
   - `# pip3 install boto3 # For AWS S3`
   - For Google Cloud Storage, ensure your `GOOGLE_APPLICATION_CREDENTIALS` environment variable is set to the path of your service account key file, or authenticate programmatically.
4. **Code Development (Python):**
   - Write a Python script.
   - Import necessary libraries (sensor library, cloud storage client).
   - Initialize the cloud storage client.
   - Read sensor data.
   - Format the data (e.g., CSV string, JSON string).
   - Create a unique filename (e.g., with timestamp).
   - Upload the data as a new file (or append to an existing file) to your cloud storage bucket.
   - Implement a loop for periodic logging.
   - Handle potential errors during sensor reading or cloud storage operations.
5. **Execution and Test:** Run the Python script on Raspberry Pi. Check your cloud storage bucket via the web console to verify that new data files are being created or updated.

Source Code (Example with Google Cloud Storage):

```
import time
import datetime
import Adafruit_DHT # For DHT11/DHT22 sensor
from google.cloud import storage # For Google Cloud Storage

# Google Cloud Storage details
BUCKET_NAME = "your-gcs-bucket-name" # Replace with your bucket name
# Ensure GOOGLE_APPLICATION_CREDENTIALS environment variable is set
# or provide credentials explicitly (e.g., from a JSON key file)
# Example:
storage.Client.from_service_account_json('path/to/your/service_account_key.js
on')
```

```
# DHT Sensor details
DHT_SENSOR = Adafruit_DHT.DHT11 # Or Adafruit_DHT.DHT22
DHT_PIN = 4 # GPIO pin connected to the DHT sensor (e.g., GPIO 4)

# Interval for logging data (in seconds)
LOG_INTERVAL = 30

def upload_to_gcs(data_string, filename):
    """Uploads a string to the bucket."""
    try:
        storage_client = storage.Client()
        bucket = storage_client.bucket(BUCKET_NAME)
        blob = bucket.blob(filename)

        blob.upload_from_string(data_string, content_type="text/csv") # Or
"application/json"
        print(f"Uploaded {filename} to {BUCKET_NAME}.")
        return True
    except Exception as e:
        print(f"Error uploading to GCS: {e}")
        return False

if __name__ == "__main__":
    print("Starting Raspberry Pi data logger to Google Cloud Storage...")

    while True:
        humidity, temperature = Adafruit_DHT.read_retry(DHT_SENSOR, DHT_PIN)

        if humidity is not None and temperature is not None:
            current_time = datetime.datetime.now()
            timestamp_str = current_time.isoformat()

            # Format data as CSV string
            log_data = f"{timestamp_str},{temperature:.2f},{humidity:.2f}\n"
            print(f"Logging: {log_data.strip()}")

            # Define filename (e.g., daily log file)
            filename = f"sensor_logs_{current_time.strftime('%Y-%m-%d')}.csv"

            # To append to an existing file, you'd need to download, append,
then re-upload.
            # For simplicity, this example overwrites or creates a new file
for each interval.
            # For continuous logging, consider appending to a single file
daily/hourly.
            # Or creating unique files per reading.

            # Simple approach: create a new file for each reading (less
efficient for many readings)
            # filename = f"sensor_data/{timestamp_str.replace(':', '-')}.csv"
            # upload_to_gcs(log_data, filename)

            # More efficient: append to a daily file (requires reading
existing content)
            try:
                storage_client = storage.Client()
                bucket = storage_client.bucket(BUCKET_NAME)
                blob = bucket.blob(filename)

                existing_content = ""
                if blob.exists():
                    existing_content = blob.download_as_string().decode('utf-
8')

                new_content = existing_content + log_data
                blob.upload_from_string(new_content, content_type="text/csv")
                print(f"Appended data to {filename} in {BUCKET_NAME}.")

            except Exception as e:
                print(f"Error appending to GCS file: {e}")

        else:
            print("Failed to retrieve data from humidity sensor.")

        time.sleep(LOG_INTERVAL)
```

Input: Real-time temperature and humidity readings from the connected DHT sensor on Raspberry Pi.

Expected Output:

- Raspberry Pi successfully connects to Google Cloud Storage.
- Periodically, sensor data (e.g., CSV formatted lines) is appended to a daily log file (e.g., `sensor_logs_YYYY-MM-DD.csv`) in your specified GCS bucket.
- The Google Cloud Storage console will show the updated files with new content.
- Console output on Raspberry Pi will show "Logging:" messages and confirmation of successful uploads/appends.

Lab 15: Real-time Data Analytics

Title: Real-time Data Analytics: Performing Real-time Analytics on Sensor Data and Sending Alerts

Aim: To develop a Python script to perform real-time analytics on sensor data received from NodeMCU or other devices and send alerts or notifications based on predefined thresholds.

Procedure:

1. **Data Source Setup:**
   - Ensure you have a stream of sensor data available (e.g., from Lab 11 - MQTT Publisher from Raspberry Pi, or Lab 13 - HTTP Server on Raspberry Pi receiving data).
   - This lab assumes data is coming in via MQTT.
2. **Notification Service Setup:**
   - Choose a notification service (e.g., Email via SMTP, Twilio for SMS, Telegram Bot API, or a cloud-based notification service like AWS SNS).
   - Configure credentials for the chosen service.
3. **Software Setup (Raspberry Pi):**
   - Install Python 3.
   - Install `paho-mqtt` (to subscribe to sensor data).
   - Install libraries for your chosen notification service (e.g., `smtplib` for email, `python-telegram-bot` for Telegram, `boto3` for AWS SNS).
   - `pip3 install paho-mqtt`
   - `# pip3 install python-telegram-bot # For Telegram`
   - `# pip3 install boto3 # For AWS SNS`

4. **Code Development (Python):**
   - Write a Python script.
   - Configure an MQTT client to subscribe to the sensor data topic (e.g., `raspberrypi/sensor/temperature`).
   - Implement the `on_message` callback function.
   - Inside `on_message`, parse the incoming sensor data.
   - Apply real-time analytics logic: Check if the sensor value exceeds or falls below predefined thresholds.
   - If a threshold is crossed, trigger a notification using the configured service. Implement a cooldown period to prevent excessive alerts.
5. **Execution and Test:** Run the Python script on Raspberry Pi. Simulate sensor data (e.g., by publishing values manually from an MQTT client or letting your sensor device run). Observe if alerts are triggered when thresholds are crossed.

Source Code (Example with MQTT and Email Alert):

```
import paho.mqtt.client as mqtt
import json
import smtplib
from email.mime.text import MIMEText
import time

# MQTT Broker details (where sensor data is published)
MQTT_BROKER = "YOUR_MQTT_BROKER_ADDRESS"
MQTT_PORT = 1883
MQTT_USERNAME = "YOUR_MQTT_USERNAME"
MQTT_PASSWORD = "YOUR_MQTT_PASSWORD"
```

```
SENSOR_DATA_TOPIC = "raspberrypi/sensor/temperature" # Topic to subscribe to

# Email Alert Configuration
SENDER_EMAIL = "your_email@example.com"
SENDER_PASSWORD = "your_email_password" # Use app password for Gmail
RECEIVER_EMAIL = "recipient_email@example.com"
SMTP_SERVER = "smtp.gmail.com" # e.g., "smtp.mail.yahoo.com"
SMTP_PORT = 587 # Or 465 for SSL

# Thresholds for alerts
TEMP_HIGH_THRESHOLD = 30.0 # Celsius
TEMP_LOW_THRESHOLD = 10.0 # Celsius

# Cooldown period for alerts (in seconds)
ALERT_COOLDOWN = 600 # 10 minutes
last_alert_time = 0

def send_email_alert(subject, body):
    """Sends an email alert."""
    try:
        msg = MIMEText(body)
        msg['Subject'] = subject
        msg['From'] = SENDER_EMAIL
        msg['To'] = RECEIVER_EMAIL

        with smtplib.SMTP(SMTP_SERVER, SMTP_PORT) as server:
            server.starttls() # Enable TLS encryption
            server.login(SENDER_EMAIL, SENDER_PASSWORD)
            server.send_message(msg)
        print(f"Email alert sent: '{subject}'")
        return True
    except Exception as e:
        print(f"Failed to send email: {e}")
        return False

# The callback for when the client receives a CONNACK response from the
server.
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connected to MQTT Broker!")
        client.subscribe(SENSOR_DATA_TOPIC)
        print(f"Subscribed to topic: {SENSOR_DATA_TOPIC}")
    else:
        print(f"Failed to connect, return code {rc}\n")

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    global last_alert_time
    print(f"Message received [{msg.topic}]: {msg.payload.decode()}")

    if msg.topic == SENSOR_DATA_TOPIC:
        try:
            temperature = float(msg.payload.decode())
            print(f"Parsed temperature: {temperature:.2f}C")

            current_time = time.time()

            if temperature > TEMP_HIGH_THRESHOLD:
                if (current_time - last_alert_time) > ALERT_COOLDOWN:
                    subject = "ALERT: High Temperature Detected!"
                    body = f"Temperature exceeded threshold:
{temperature:.2f}C (Threshold: {TEMP_HIGH_THRESHOLD:.2f}C)"
                    if send_email_alert(subject, body):
                        last_alert_time = current_time
            elif temperature < TEMP_LOW_THRESHOLD:
                if (current_time - last_alert_time) > ALERT_COOLDOWN:
                    subject = "ALERT: Low Temperature Detected!"
                    body = f"Temperature dropped below threshold: 
{temperature:.2f}C (Threshold: {TEMP_LOW_THRESHOLD:.2f}C)"
                    if send_email_alert(subject, body):
                        last_alert_time = current_time

        except ValueError:
            print("Received non-numeric temperature data.")
        except Exception as e:
            print(f"Error processing message: {e}")

client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION1)
client.username_pw_set(MQTT_USERNAME, MQTT_PASSWORD)
client.on_connect = on_connect
client.on_message = on_message

try:
    client.connect(MQTT_BROKER, MQTT_PORT, 60)
except Exception as e:
    print(f"Could not connect to MQTT broker: {e}")
    exit()

client.loop_forever() # Blocks and handles network traffic, callbacks
```

Input: Real-time temperature data published to the `raspberrypi/sensor/temperature` MQTT topic (e.g., from Lab 11).

Expected Output:

- Raspberry Pi successfully connects to the MQTT broker and subscribes to the sensor data topic.
- The script continuously monitors incoming temperature data.
- If the temperature value crosses the `TEMP_HIGH_THRESHOLD` or falls below `TEMP_LOW_THRESHOLD`, an email alert is sent to the `RECEIVER_EMAIL` (respecting the `ALERT_COOLDOWN` period).
- Console output on Raspberry Pi will show received messages, parsed temperature, and confirmation of email alerts sent.
- The recipient will receive email notifications when thresholds are breached.