Lecture 3

Experiment Title: Wireless Data Transmission using MQTT Protocol

Description:

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish-subscribe based messaging protocol designed for resource-constrained devices and low-bandwidth, high-latency, or unreliable networks. It is widely used in Internet of Things (IoT) applications, providing efficient communication between sensors, actuators, and other devices.

Why Is MQTT the Best Protocol for IoT?

MQTT has emerged as one of the best **IoT protocols** due to its unique features and capabilities tailored to the specific needs of IoT systems. Some of the key reasons include:

- **Lightweight:** IoT devices are often constrained in terms of processing power, memory, and energy consumption. MQTT's minimal overhead and small packet size make it ideal for these devices, as it consumes fewer resources, enabling efficient communication even with limited capabilities.

- **Reliable:** IoT networks can experience high latency or unstable connections. MQTT's support for different QoS levels, session awareness, and persistent connections ensures reliable message delivery even in challenging conditions, making it well-suited for IoT applications.

  Secure communications: Security is crucial in IoT networks as they often transmit sensitive data. MQTT supports Transport Layer Security (TLS) and Secure Sockets Layer (SSL) encryption, ensuring data confidentiality during transmission. Additionally, it provides **authentication** and **authorization** mechanisms through username/password credentials or client certificates, safeguarding access to the network and its resources.

- **Bi-directionality: MQTT's publish-subscribe model** allows for seamless bi-directional communication between devices. Clients can both publish messages to topics and subscribe

to receive messages on specific topics, enabling effective data exchange in diverse IoT ecosystems without direct coupling between devices. This model also simplifies the integration of new devices, ensuring easy scalability.

- **Large-scale IoT device support:** IoT systems often involve a large number of devices, requiring a protocol that can handle massive-scale deployments. MQTT's lightweight nature, low bandwidth consumption, and efficient use of resources make it well-suited for large-scale IoT applications. The publish-subscribe pattern allows MQTT to scale effectively, as it decouples sender and receiver, reducing network traffic and resource usage.

How Does MQTT Work?

To understand how MQTT works, you need to first master the concepts of MQTT Client, MQTT Broker, Publish-Subscribe model, Topic, and QoS:

MQTT Client

Any application or device running the **MQTT client library** is an MQTT client. For example, an instant messaging app that uses MQTT is a client, various sensors that use MQTT to report data are a client, and various **MQTT testing tools** are also a client.

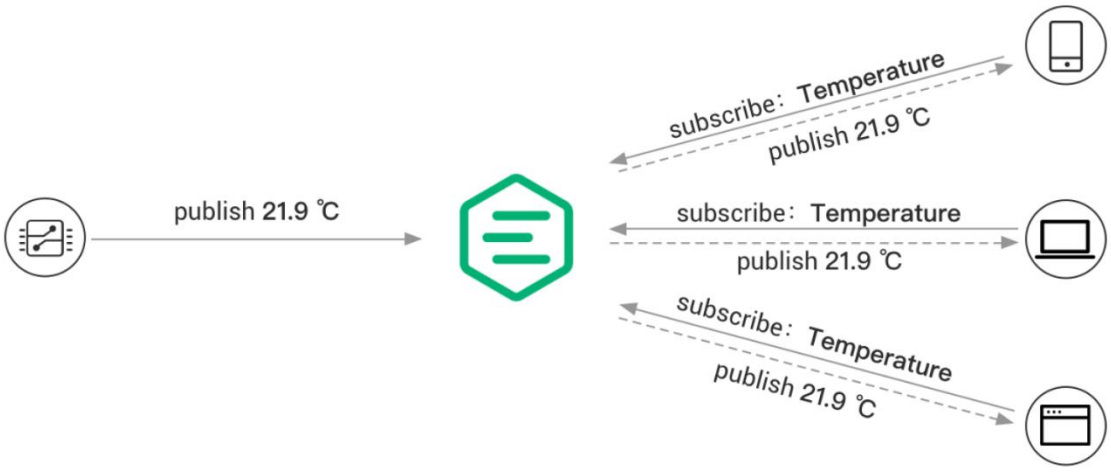
MQTT Broker

The MQTT Broker handles client connection, disconnection, subscription, and unsubscription requests, and routing messages. A powerful MQTT broker can support massive connections and million-level message throughput, helping IoT service providers focus on business and quickly create a reliable MQTT application.

Publish–subscribe pattern

The publish-subscribe pattern differs from the client-server pattern in that it separates the client that sends messages (publisher) from the client that receives messages (subscriber). Publishers and subscribers do not need to establish a direct connection, and the MQTT Broker is responsible for routing and distributing all messages.

The following diagram shows the MQTT publish/subscribe process. The temperature sensor connects to the MQTT server as a client and publishes temperature data to a topic (e.g., Temperature), and the server receives the message and forwards it to the client subscribed to the Temperature topic.



Topic

In MQTT, Topic refers to a UTF-8 string that filters messages for a connected client. A topic consists of one or more levels separated by a forward slash (topic level separator).

In comparison to a message queue, MQTT topics are very lightweight. The client does not need to create the desired topic before they publish or subscribe to it. The broker accepts each valid topic without any prior initialization.

Examples of MQTT Topics

- Myhome/groundfloor/livingroom/temperature: This topic represents the temperature in the living room of a home located on the ground floor.
- USA/California/San Francisco/Silicon Valley: This topic hierarchy can track or exchange information about events or data related to the Silicon Valley area in San Francisco, California, within the United States.

- 5ff4a2ce-e485-40f4-826c-b1a5d81be9b6/status: This topic could be used to monitor the status of a specific device or system identified by its unique identifier.

- Germany/Bavaria/car/2382340923453/latitude: This topic structure could be utilized to share the latitude coordinates of a particular car in the region of Bavaria, Germany.

Quality of Service (QoS)

MQTT provides three kinds of Quality of Service and guarantees messaging reliability in different network environments.

- QoS 0: The message is delivered at most once. If the client is not available currently, it will lose this message.
- QoS 1: The message is delivered at least once.
- QoS 2: The message is delivered only once.

The MQTT Workflow

1. **Clients initiate a connection** to the broker using TCP/IP, with optional TLS/SSL encryption for secure communication. Clients provide authentication credentials and specify a clean or persistent session.
2. **Clients either publish messages to specific topics or subscribe to topics** to receive messages. Publishing clients send messages to the broker, while subscribing clients express interest in receiving messages on particular topics.
3. **The broker receives published messages** and forwards them to all clients subscribed to the relevant topics. It ensures reliable message delivery according to the specified Quality of Service (QoS) level and manages message storage for disconnected clients based on session type.

MQTT on ESP8266

- **Hardware**:
  - 1 x NodeMCU ESP8266 development board
- **Software**:
  - Arduino IDE
  - **MQTTX client** (or other MQTT client)
  - We will use the **free public MQTT broker** provided by EMQX, based on the EMQX Platform. The broker access information is as follows:

- Broker: broker.emqx.io
- TCP Port: 1883
- TLS Port: 8883
- Websocket Port: 8083
- Websockets Port: 8084
-

Connecting ESP8266 to an MQTT Broker

Installing Support for the ESP8266 Board

In the Arduino IDE, select "Preferences" from the "File" menu. In the dialog box that appears, find "Additional Board Manager URLs" and add the URL for ESP8266: http://arduino.esp8266.com/stable/package_esp8266com_index.json. Then, search and install ESP8266 from "Board Manager" under the "Tools" menu.

Installing the PubSubClient Library

We also need to install the PubSubClient library in the Arduino IDE, which is used to connect to the MQTT broker. You can find and install this library through the Library Manager in the IDE.

Initializing the Wi-Fi Connection

Before connecting to the MQTT broker, we first need to initialize the Wi-Fi connection. This can be done using the ESP8266WiFi library.

```
#include <ESP8266WiFi.h>

#include <PubSubClient.h>



// WiFi settings

const char *ssid = "WIFI_SSID";            // Replace with your WiFi name

const char *password = "WIFI_PASSWORD";   // Replace with your WiFi password
```

Setting MQTT Broker Connection Parameters

```
// MQTT Broker settings
const char *mqtt_broker = "broker.emqx.io";  // EMQX broker endpoint
const char *mqtt_topic = "emqx/esp8266/led";  // MQTT topic
const char *mqtt_username = "emqx";  // MQTT username for authentication
const char *mqtt_password = "public";  // MQTT password for authentication
const int mqtt_port = 1883;  // MQTT port (TCP)
```

Initializing the WIFI and MQTT Client

```
WiFiClient espClient;
PubSubClient mqtt_client(espClient);
```

Connecting to WIFI

```
void connectToWiFi() {
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi");
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("\nConnected to the WiFi network");
}
```

Connecting to the MQTT Broker and Subscribing to a Topic

This section will detail how to use the ESP8266 to connect to an MQTT broker via TCP and subscribe to a topic. The connection method discussed here is based on unencrypted TCP communication. While TCP connections are sufficient for most basic applications, if your project involves sensitive data or requires higher security, we recommend using TLS encryption.

TCP Connection

```
void connectToMQTTBroker() {
  while (!mqtt_client.connected()) {
    String client_id = "esp8266-client-" + String(WiFi.macAddress());
    Serial.printf("Connecting to MQTT Broker as %s.....\n", client_id.c_str());
    if (mqtt_client.connect(client_id.c_str(), mqtt_username, mqtt_password)) {
      Serial.println("Connected to MQTT broker");
      mqtt_client.subscribe(mqtt_topic);
      // Publish message upon successful connection
      mqtt_client.publish(mqtt_topic, "Hi EMQX I'm ESP8266 ^^");
    } else {
      Serial.print("Failed to connect to MQTT broker, rc=");
      Serial.print(mqtt_client.state());
      Serial.println(" try again in 5 seconds");
      delay(5000);
    }
```

```
  }

}
```

TLS Connection

Additional settings are required if you need to connect to the MQTT broker via TLS. TLS connections provide encrypted data transfer, ensuring secure communication. To implement a TLS connection, you need to:

1. **CA Certificate**: Obtain and load the MQTT broker's CA certificate. This certificate is used to verify the broker's identity, ensuring that you are connecting to the correct broker.
2. **NTP Synchronization**: The ESP8266 device's time must be synchronized with global standard time. TLS connections require accurate system time to ensure the validity of TLS certificates. You can use an NTP (Network Time Protocol) client library for synchronization.

A complete TLS connection example and related code can be found on the following GitHub link: **ESP8266 MQTT TLS Example**. This example demonstrates how to configure the ESP8266 to use a TLS connection to the MQTT broker, including loading a CA certificate and setting up NTP synchronization.

Writing the Callback Function

In MQTT communication, message reception is handled through a callback function. We need to define a callback function that is triggered when the ESP8266 receives a message from the MQTT broker. Our example will demonstrate how to receive and print message content within the callback function.

```
void mqttCallback(char *topic, byte *payload, unsigned int length) {

  Serial.print("Message received on topic: ");

  Serial.println(topic);

  Serial.print("Message:");

  for (unsigned int i = 0; i < length; i++) {

    Serial.print((char) payload[i]);

  }

  Serial.println();

  Serial.println("-----------------------");

}
```

Full Code

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

// WiFi settings
const char *ssid = "WIFI_SSID";              // Replace with your WiFi name
const char *password = "WIFI_PASSWORD";    // Replace with your WiFi password

// MQTT Broker settings
const char *mqtt_broker = "broker.emqx.io";  // EMQX broker endpoint
const char *mqtt_topic = "emqx/esp8266";     // MQTT topic
const char *mqtt_username = "emqx";  // MQTT username for authentication
const char *mqtt_password = "public";  // MQTT password for authentication
const int mqtt_port = 1883;  // MQTT port (TCP)

WiFiClient espClient;
PubSubClient mqtt_client(espClient);

void connectToWiFi();

void connectToMQTTBroker();

void mqttCallback(char *topic, byte *payload, unsigned int length);

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

```
    connectToWiFi();
    mqtt_client.setServer(mqtt_broker, mqtt_port);
    mqtt_client.setCallback(mqttCallback);
    connectToMQTTBroker();
}

void connectToWiFi() {
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi");
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("\nConnected to the WiFi network");
}

void connectToMQTTBroker() {
    while (!mqtt_client.connected()) {
        String client_id = "esp8266-client-" + String(WiFi.macAddress());
        Serial.printf("Connecting to MQTT Broker as %s.....\n",
client_id.c_str());
        if (mqtt_client.connect(client_id.c_str(), mqtt_username,
mqtt_password)) {
            Serial.println("Connected to MQTT broker");
            mqtt_client.subscribe(mqtt_topic);
            // Publish message upon successful connection
            mqtt_client.publish(mqtt_topic, "Hi EMQX I'm ESP8266 ^^");
        } else {
```

```
            Serial.print("Failed to connect to MQTT broker, rc=");
            Serial.print(mqtt_client.state());
            Serial.println(" try again in 5 seconds");
            delay(5000);
        }
    }
}

void mqttCallback(char *topic, byte *payload, unsigned int length) {
    Serial.print("Message received on topic: ");
    Serial.println(topic);
    Serial.print("Message:");
    for (unsigned int i = 0; i < length; i++) {
        Serial.print((char) payload[i]);
    }
    Serial.println();
    Serial.println("-----------------------");
}

void loop() {
    if (!mqtt_client.connected()) {
        connectToMQTTBroker();
    }
    mqtt_client.loop();
}
```

Connection and Testing

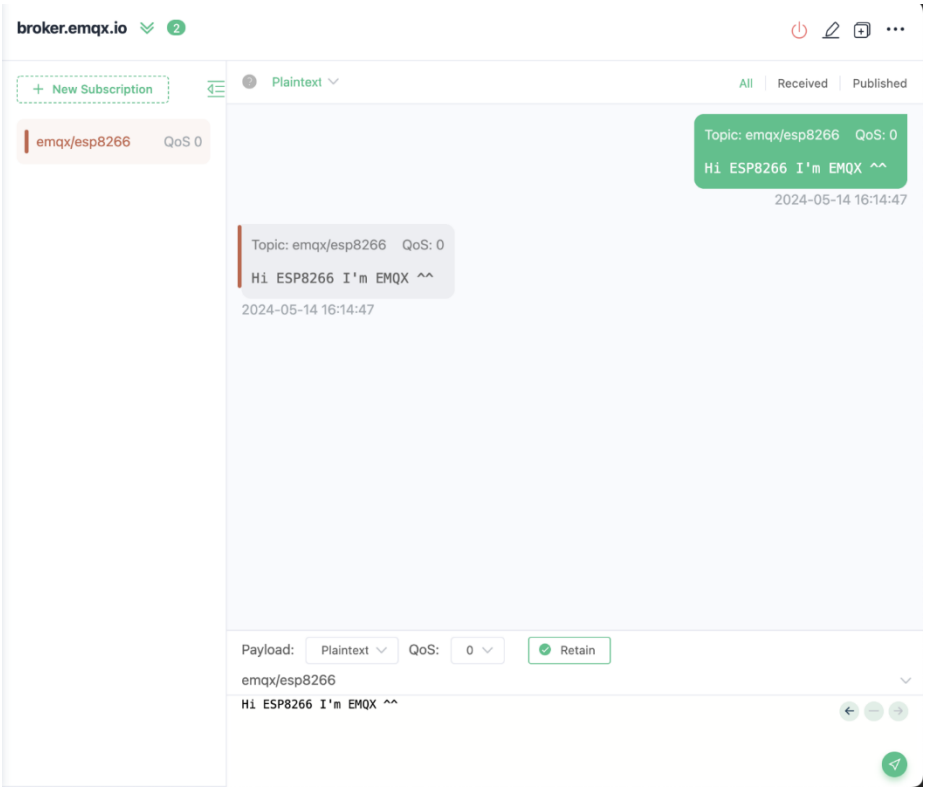
Uploading the Code to ESP8266

Copy the full code into the Arduino IDE, then upload it to your ESP8266 development board. Open the serial monitor, and you can see the ESP8266 board connecting to the Wi-Fi network and then to the MQTT broker.

```
10:50:52.626 -> {l$ѡ|@l@|  @$@b<@@ @ @;@b@ c@@o'@$go@@@ c x@@ls${lx@g@  @ d@@ bo@| @ @ @b@@'g@lD̃l` @ o'$
10:50:59.793 -> Connected to the WiFi network
10:50:59.793 -> Connecting to MQTT Broker as esp8266-client-9C:9C:1F:45:6A:EF.....
10:51:00.708 -> Connected to MQTT broker
```

Using MQTTX to Send Messages to ESP8266

To test this functionality, you can use any **MQTT client** software (like MQTTX) to connect to the same MQTT broker and send messages to the topic subscribed by ESP8266.



Then, you can see these messages being correctly received and displayed in the ESP8266's serial output, which is a good way to check if the communication is successful.

```
10:51:01.100 -> ------------------------
10:54:41.065 -> Message received on topic: emqx/esp8266
10:54:41.065 -> Message:Hi ESP8266 I'm EMQX ^^
10:54:41.065 -> ------------------------
```