Experiment No 1

AIM: To simulate IoT device communication using MQTT or HTTP protocols.

Objective: Demonstrate message publishing and subscribing using `paho-mqtt` in Google Colab.

Tools Used: Google Colab, Python libraries (`paho-mqtt` or `requests`).

Theory: The Internet of Things (IoT) refers to the network of interconnected devices that communicate and exchange data over the internet or other communication networks. These devices range from simple sensors and actuators to complex machines. For these devices to work seamlessly, communication protocols are essential for enabling data exchange. Two of the most widely used protocols for communication in IoT are MQTT (Message Queuing Telemetry Transport) and HTTP (Hypertext Transfer Protocol).

MQTT Protocol:

MQTT is a lightweight, publish-subscribe messaging protocol designed for low-bandwidth, high-latency, and unreliable networks, which are common in IoT environments. It uses a "broker" to handle communication between devices. In MQTT, devices (known as "clients") connect to a central broker, and each client can either publish messages to a topic or subscribe to receive messages from a topic.

The publish-subscribe model decouples message producers (publishers) from message consumers (subscribers), making it an ideal solution for IoT applications. A device can publish sensor data to a specific topic, and other devices that are subscribed to that topic will receive the data. This mechanism allows real-time communication with minimal overhead, which is crucial for applications that require rapid responses, such as remote monitoring and control systems.

Key advantages of MQTT include:

1. **Low Bandwidth**: MQTT's small message headers and low overhead make it suitable for low-bandwidth environments.
2. **Quality of Service Levels**: MQTT supports three levels of Quality of Service (QoS) to ensure reliable message delivery.
3. **Retained Messages**: The broker can store the last message sent on a topic and send it to any new subscribers.
4. **Last Will and Testament (LWT)**: Clients can specify a message to be sent if they disconnect unexpectedly.

HTTP Protocol:

While MQTT is optimal for IoT, HTTP remains one of the most widely used protocols, especially in web services. It works on a request-response model, where a client sends an HTTP request to a server, and the server returns a response. HTTP is more suited for IoT applications where devices need to request data from servers or submit data in a stateless manner.

However, HTTP can be less efficient in IoT settings because each request requires a new connection to the server, which may increase bandwidth usage and latency. This makes MQTT preferable for real-time, continuous communication, whereas HTTP is often used for periodic data retrieval or interaction with centralized systems.

Program Code:

```
import paho.mqtt.client as mqtt

# Define callback functions
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    client.subscribe("iot/test")

def on_message(client, userdata, msg):
    print(f"Message received: {msg.payload.decode()}")

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("broker.hivemq.com", 1883, 60)
client.loop_start()

client.publish("iot/test", "Hello IoT")
client.loop_stop()
```

Explanation and logic of Code

This Python code demonstrates how to use the Paho-MQTT library to connect to an MQTT broker, subscribe to a topic, and publish a message to that topic. Let's go through the code line by line:

1. `import paho.mqtt.client as mqtt`

- This imports the `paho.mqtt.client` module and gives it the alias `mqtt`. This module provides functions for MQTT client operations, such as connecting to a broker, subscribing to topics, and publishing messages.

2. `# Define callback functions`

- This is a comment indicating that the following functions will define callbacks for handling events when certain actions (like connecting or receiving messages) occur.

3. `def on_connect(client, userdata, flags, rc):`

- This defines a function called `on_connect` which will be executed when the client successfully connects to the MQTT broker. The function has parameters:
  - `client`: the MQTT client object.
  - `userdata`: a user-defined object that is passed by the client (not used in this case).
  - `flags`: flags returned by the broker (not used in this case).
  - `rc`: result code indicating the connection status (0 for successful connection).

4. `print("Connected with result code "+str(rc))`

- This prints a message indicating whether the connection was successful or not. The result code (`rc`) is converted to a string and appended to the message. A result code of `0` means the connection was successful.

5. `client.subscribe("iot/test")`

- This instructs the client to subscribe to the MQTT topic `"iot/test"`. Once subscribed, the client will start receiving messages published to that topic.

6. `def on_message(client, userdata, msg):`

- This defines a function called `on_message` that will be executed whenever a message is received from the broker. The function parameters are:
  - `client`: the MQTT client object.
  - `userdata`: user-defined data passed by the client (not used here).
  - `msg`: the message object containing the topic and the payload (message content).

7. `print(f"Message received: {msg.payload.decode()}")`

- This prints the payload (message content) of the received message after decoding it. MQTT message payloads are typically in byte format, so the `decode()` method is used to convert it to a string.

8. `client = mqtt.Client()`

- This creates an MQTT client instance. The `Client()` constructor initializes a new MQTT client object that will be used for connecting to the broker, subscribing to topics, and publishing messages.

9. `client.on_connect = on_connect`

- This assigns the previously defined `on_connect` function as the callback to handle connection events. This means that whenever the client connects to the broker, the `on_connect` function will be invoked.

10. `client.on_message = on_message`

- This assigns the previously defined `on_message` function as the callback to handle received messages. This means that whenever the client receives a message, the `on_message` function will be triggered.

11. `client.connect("broker.hivemq.com", 1883, 60)`

- This line connects the MQTT client to the broker at the specified address (`broker.hivemq.com`) and port (`1883`). The `60` represents the keep-alive interval in seconds. The broker address is an MQTT broker that allows public connections for experimentation and testing.

12. `client.loop_start()`

- This starts a new thread for the MQTT client's network loop. The loop allows the client to manage network traffic (connect, publish, subscribe, and receive messages). This non-blocking call enables the program to continue running while the client is connected to the broker and performing tasks.

13. `client.publish("iot/test", "Hello IoT")`

- This publishes a message to the topic `"iot/test"`. The message content is `"Hello IoT"`. When the message is successfully sent, all clients that are subscribed to this topic will receive it. In this case, since the client itself is subscribed to the topic, it will also receive the message.

14. `client.loop_stop()`

- This stops the MQTT client's network loop that was started with `loop_start()`. Since the client has already published the message and is no longer actively listening for messages, the loop is stopped to end the program.

Logic:

- The program demonstrates the basic publish-subscribe mechanism in MQTT. The MQTT client connects to the broker, subscribes to a topic, publishes a message to that topic, and receives its own message since it is subscribed to the same topic. The `on_connect` and `on_message` callbacks handle connection events and message reception, respectively.
- The message flow is as follows:
    1. The client connects to the broker.
    2. Upon connection, it subscribes to the topic `"iot/test"`.
    3. It publishes the message `"Hello IoT"` to the topic.
    4. The client receives the message through the `on_message` callback and prints it.

The use of `client.loop_start()` and `client.loop_stop()` allows the MQTT client to run asynchronously without blocking the rest of the code.

Observation Table:

Published Message	Received Message
Hello IoT	Hello IoT

Conclusion : We have learned to set up basic MQTT communication, publish messages, and subscribe to topics.

Home Work Assigned:

Expand this experiment for another two topics Temperature and Pressure for publishing and subscribing using MQTT.