Lab 6: IoT Communications: MQTT

1. What is MQTT, and why is it commonly used in IoT applications?

Answer:
MQTT (Message Queue Telemetry Transport) is a **lightweight publish/subscribe messaging protocol** designed for resource-constrained environments. It is widely used in IoT because:

- It minimizes **bandwidth usage** with small packet overhead.
- It uses a **broker** to decouple publishers and subscribers.
- It handles **asynchronous** communication, which is ideal for devices that may connect intermittently.

---

2. Why do we need an MQTT broker such as Mosquitto, and how does it differ from an MQTT client?

Answer:

- The **MQTT broker** (e.g., Mosquitto) is the central server that:
    - Listens for incoming messages on specified topics from publishers.
    - Routes those messages to subscribers who have registered interest in those topics.
- An **MQTT client** can be a device or software that:
    - **Publishes** messages to a topic on the broker.
    - **Subscribes** to a topic to receive relevant messages.

They differ in roles: the broker is the mediator, while the client either sends or receives (or both).

---

3. How do you install and configure Mosquitto on a Raspberry Pi, and what does the line listener 1883 in the configuration file mean?

Answer:

1. **Install Mosquitto** using:

sql

Copy

sudo apt update

sudo apt install mosquitto

2. **Configure** Mosquitto by editing /etc/mosquitto/mosquitto.conf:
    - listener 1883 tells Mosquitto to **listen** for MQTT connections on port 1883 (the default MQTT port).
    - allow_anonymous true allows connections without username/password (not recommended for production).
3. **Start** the broker manually:

bash

Copy

sudo mosquitto -c /etc/mosquitto/mosquitto.conf

---

4. What are some security considerations when using allow_anonymous true in Mosquitto?

Answer:

- It **permits anyone** to connect to the broker without authentication, potentially exposing it to unauthorized publishers/subscribers.
- For secure environments, you should:
    - Disable anonymous access.
    - Use **username/password** authentication.
    - Enable **TLS/SSL** encryption for sensitive data transmissions.

---

5. How can you enable Mosquitto to start automatically on boot?

Answer:
Using **systemd**:

bash

Copy

sudo systemctl enable mosquitto

sudo systemctl start mosquitto

This ensures Mosquitto launches whenever the Raspberry Pi boots up. You can also manage it with:

bash

Copy

sudo systemctl disable mosquitto

sudo systemctl stop mosquitto

to turn off auto-start and stop the service.

---

6. In MQTT terminology, what is a "topic," and how do wildcard topics work?

Answer:

- A **topic** is a hierarchical string (e.g., sensor/temperature/room1) that classifies messages.
- Wildcards:
    - **+** matches a single level (e.g., sensor/+/room1 matches sensor/temperature/room1, sensor/humidity/room1, etc.).
    - **#** matches multiple levels (e.g., sensor/# matches everything under sensor/, such as sensor/temperature/room1/ceiling).

---

7. What does the Python Paho MQTT client library do, and how do you install it?

Answer:

- **Paho MQTT** is a Python library that provides methods to **connect** to an MQTT broker, **publish** messages to a topic, and **subscribe** to receive messages.
- It's installed via:

bash

Copy

pip install paho-mqtt

- You typically create a mqtt.Client() object, set callbacks, and then connect to the broker.

---

8. How do the Python scripts mqtt_publisher.py and mqtt_subscriber.py communicate with each other?

Answer:

1. **mqtt_publisher.py**:
    - Connects to the broker.
    - Publishes messages to a specific topic (e.g., test/topic).
2. **mqtt_subscriber.py**:
    - Connects to the same broker.
    - Subscribes to the same topic (test/topic).
    - Receives and processes messages from the publisher.

Because they share the same broker and topic, the subscriber receives messages as soon as they are published.

---

9. Why do we typically use two separate terminals to run the subscriber and publisher scripts?

Answer:

- Each script is a standalone client that **blocks** in its main loop (the subscriber runs client.loop_forever(), while the publisher uses time.sleep() in an infinite loop).
- Running them in separate terminals allows both scripts to run **concurrently**, simulating two independent IoT devices.

---

10. If the subscriber doesn't receive messages, what troubleshooting steps should you take?

Answer:

- Check if both publisher and subscriber are using the **same IP address** or hostname for the broker (not localhost if the broker is on another machine).
- Confirm the **topic name** matches exactly (case-sensitive).
- Verify **Mosquitto** is running and listening on port **1883**.
- Look for firewall rules blocking port 1883.

- Use mosquitto_sub and mosquitto_pub CLI tools for quick local debugging.

---

11. How might you modify the code to capture an image with a webcam on receiving a specific message?

Answer:

- In the **subscriber** script, add logic in the on_message callback:

python

Copy

```
if message.topic == "camera/capture" and message.payload.decode() == "capture_image":
    # Access the webcam using OpenCV (cv2.VideoCapture)
    # Read a frame and save it to disk or memory
```

- When a publisher sends "capture_image" to camera/capture, the subscriber triggers a webcam capture.

---

12. How could you send an image via MQTT once captured?

Answer:

- Convert the image to a **byte stream** (e.g., JPEG format) using OpenCV or PIL:

python

Copy

```
_, buffer = cv2.imencode('.jpg', frame)
image_bytes = buffer.tobytes()
```

- **Publish** the byte array to a topic:

python

Copy

```
client.publish("camera/image", image_bytes)
```

- On the receiving subscriber side, decode the bytes back into an image.

---

13. What are some best practices for transmitting large payloads, such as images, over MQTT?

Answer:

- **Compress** or resize images to reduce payload size.
- Consider **QoS levels** (Quality of Service) to ensure reliable delivery if needed.
- If images are very large, consider **chunking** them into multiple messages or using an alternative file transfer method.
- Be mindful of broker **message size limits** and memory usage.

---

14. Why might you choose to run the MQTT broker on a separate machine rather than on the same Raspberry Pi?

Answer:

- Offloading the broker to another machine can reduce **CPU/memory** load on the Pi, freeing it to handle sensor data or computations.
- A dedicated broker server may have better **network** throughput or reliability.
- For distributed systems, it's common to have a central broker that many devices connect to.

---

15. What are the optional lab assignment goals involving MQTT and a webcam, and how do they integrate IoT principles?

Answer:

- **Goal**: Combine image capture with MQTT messaging so that:
    - A subscriber triggers a webcam capture upon receiving a particular message.
    - The captured image is then published to another MQTT topic.
- **Integration**: This merges **device control** (subscriber triggers camera) and **data transfer** (publishing images), illustrating a typical IoT flow where sensors (camera) respond to commands and send data back to the network.

---

16. If you wanted to secure your MQTT communication, what are some potential methods?

Answer:

- **Username/Password**: Configure Mosquitto to require authentication.
- **TLS/SSL**: Encrypt data in transit to prevent eavesdropping.
- **Access Control Lists (ACLs)**: Restrict which clients can publish or subscribe to specific topics.
- **Firewalls/VPNs**: Isolate broker ports and networks to limit unauthorized access.

---

17. How can you verify that your MQTT broker is actually receiving and sending messages?

Answer:

- Use **CLI tools**:

bash

Copy

mosquitto_sub -t "test/topic" -v

mosquitto_pub -t "test/topic" -m "Hello"

The subscriber should see "Hello".

- Check **Mosquitto logs** in /var/log/mosquitto/mosquitto.log or use journalctl -u mosquitto.service for systemd logs.
- Observe **publisher** and **subscriber** script outputs.

---

18. What are the MQTT Quality of Service (QoS) levels, and why might you choose a higher QoS?

Answer:
MQTT defines three QoS levels for message delivery guarantees:

- **QoS 0 (At most once):** The message is delivered **once**, with no confirmation. Fast but can lose messages.
- **QoS 1 (At least once):** The message is delivered **at least once**, requiring an acknowledgment from the broker. Possible duplicates but more reliable.
- **QoS 2 (Exactly once):** The message is delivered **exactly once** by using a two-phase handshake. Highest reliability but the most overhead.

You might choose **QoS 1 or 2** if message loss or duplication is unacceptable (e.g., for critical sensor data), accepting some extra latency or complexity.

---

19. What is the Last Will and Testament (LWT) feature in MQTT, and how is it configured?

Answer:
LWT is a mechanism that lets a client specify a **"last message"** to be published by the broker if the client disconnects unexpectedly. This is useful for detecting node failures. In Python Paho, it's set before connect():

python

Copy

client.will_set(topic="device/status", payload="offline", qos=1, retain=True)

client.connect("broker_address", 1883)

If the client disconnects ungracefully, the broker automatically publishes "offline" to device/status.

---

20. How do retained messages differ from normal messages in MQTT?

Answer:
A **retained message** is one that the broker stores and immediately sends to any new subscribers of the topic, even if the message was published before they subscribed. It's a way to keep a **"last known good"** message available.
To publish a retained message, you set the retain flag:

python

Copy

client.publish("sensor/temperature", "22.5", retain=True)

New subscribers instantly receive 22.5 upon subscribing.

---

21. What is the difference between a "clean session" and a "persistent session" in MQTT?

Answer:

- **Clean session:** The broker does not retain subscription information or queued messages when the client disconnects. Everything is fresh on reconnect.

- **Persistent session:** The broker **stores** subscription info and undelivered messages for the client. On reconnect, any messages published while the client was offline are delivered.
  Persistent sessions are useful for ensuring clients don't miss messages during temporary disconnections.

---

22. How might you handle a scenario where you have two different MQTT brokers on separate networks and want to exchange messages?

Answer:
You can set up an MQTT **bridge** between the two brokers. A bridge is a special configuration that subscribes to certain topics on one broker and republishes them to the other. This allows messages to flow seamlessly between networks without manually connecting clients to both brokers.

---

23. What does client.loop_start() do in the Python Paho MQTT client, and how does it differ from client.loop_forever()?

Answer:

- **loop_start()**: Runs the network loop in a **separate thread**, allowing your main program to continue executing other tasks concurrently. You must eventually call loop_stop() to end it.
- **loop_forever()**: Blocks indefinitely, handling network traffic in the main thread. Your script won't proceed beyond this call unless an exception occurs or you stop it.

---

24. If your on_message callback in a subscriber does heavy processing, how can you avoid blocking the MQTT network loop?

Answer:

- Offload the work to a **separate thread** or **process**.
- Use queue.Queue or another concurrency mechanism to pass messages from the callback to a worker thread.
  This keeps the main loop responsive, preventing timeouts or missed messages due to lengthy computations.

---

25. How do you handle large image or binary payloads in MQTT without overwhelming the Raspberry Pi's memory?

Answer:

- **Chunk** the data into smaller segments to avoid single huge messages.
- Use **QoS** to ensure reliable transfer or consider an alternative transfer protocol if extremely large data is frequent.
- Compress or resize images before publishing.
- Adjust the broker's **maximum message size** settings, if needed, and ensure the Pi's memory usage is monitored.

---

26. Why might you use wildcard topics, and can you give an example of single-level vs. multi-level wildcards?

Answer:
Wildcards allow a subscriber to receive messages from multiple related topics without explicitly subscribing to each one. For example:

- **Single-level (+)**: building/+/temperature catches building/floor1/temperature and building/floor2/temperature, but not deeper levels.
- **Multi-level (#)**: building/floor1/# matches building/floor1/temperature, building/floor1/humidity/ceiling, etc.

---

27. What are some debugging or monitoring techniques to ensure your MQTT system runs smoothly on the Raspberry Pi?

Answer:

- **Check logs**: Mosquitto logs in /var/log/mosquitto/mosquitto.log or via journalctl -u mosquitto.
- **CLI tools**: mosquitto_pub and mosquitto_sub for quick local tests.
- **Network monitoring**: Tools like netstat or ss to confirm port 1883 is open.
- **System resource usage**: Use top or htop to watch CPU and memory usage, ensuring the Pi isn't overloaded.