



## **CME466 - Design of an Advanced Digital System**

### **MQTT and Its Implementation**

Instructor: Prof. Khan A Wahid  
TA: Omid Yaghoobian  
Winter 2024



# Copyright notice

- These slides are intended to be used in CME466 course only.
- The materials presented in this entire document are protected by copyright laws.
- You shall not reuse, publish, or distribute any of these slides without obtaining permission from the presenter and individual copyright owners.

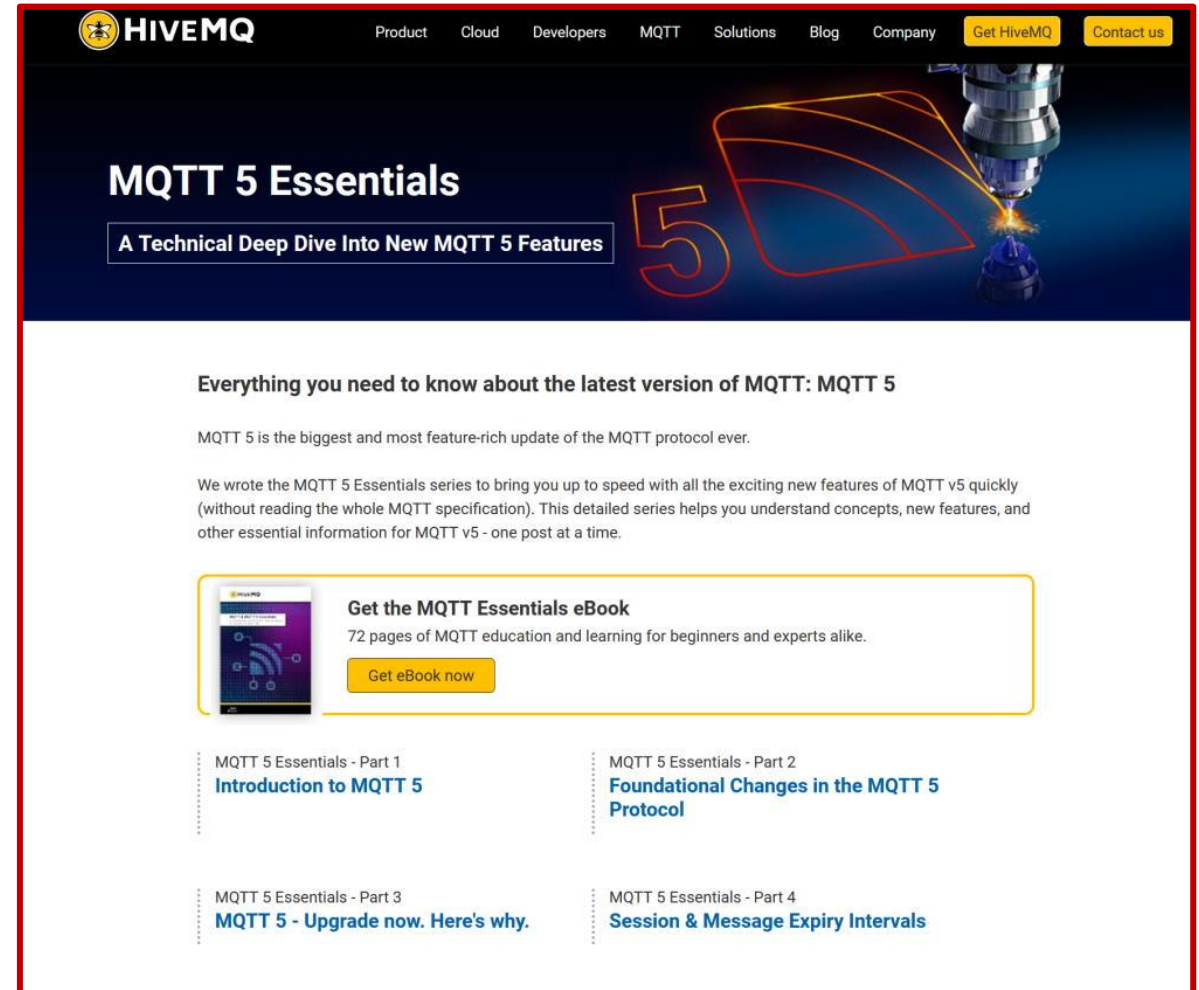


# MQTT: Download free e-book

## MQTT & MQTT 5 Essentials

A comprehensive overview of MQTT facts and features for beginners and experts alike

<https://www.hivemq.com/mqtt-5/>



**HIVEMQ** Product Cloud Developers MQTT Solutions Blog Company [Get HiveMQ](#) [Contact us](#)

## MQTT 5 Essentials

A Technical Deep Dive Into New MQTT 5 Features

Everything you need to know about the latest version of MQTT: MQTT 5

MQTT 5 is the biggest and most feature-rich update of the MQTT protocol ever.

We wrote the MQTT 5 Essentials series to bring you up to speed with all the exciting new features of MQTT v5 quickly (without reading the whole MQTT specification). This detailed series helps you understand concepts, new features, and other essential information for MQTT v5 - one post at a time.

**Get the MQTT Essentials eBook**  
72 pages of MQTT education and learning for beginners and experts alike.  
[Get eBook now](#)

MQTT 5 Essentials - Part 1  
[Introduction to MQTT 5](#)

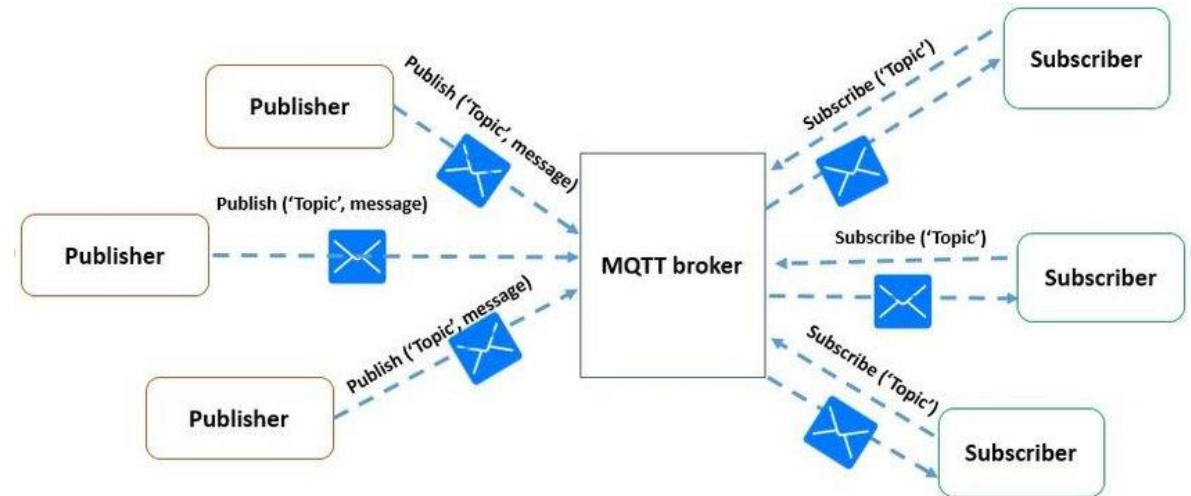
MQTT 5 Essentials - Part 2  
[Foundational Changes in the MQTT 5 Protocol](#)

MQTT 5 Essentials - Part 3  
[MQTT 5 - Upgrade now. Here's why.](#)

MQTT 5 Essentials - Part 4  
[Session & Message Expiry Intervals](#)

# MQTT (MESSAGE QUEUING TELEMETRY TRANSPORT)

- MQTT, designed in 1999, is a lightweight machine to machine (M2M) communication protocol that supports the publish/subscribe architecture with minimal bandwidth requirements, power consumption, and message data overhead
  - An MQTT client publishes messages to a broker through an address known as Topic. Another client can receive the messages by subscribing to that Topic. Clients can subscribe to multiple topics.
  - MQTT publish/subscribe protocol provides a scalable and reliable way to connect devices over the Internet. Today, MQTT is used by many companies to connect millions of devices to the Internet
- <https://mosquitto.org/>
  - [www.mqtt.org/](http://www.mqtt.org/)





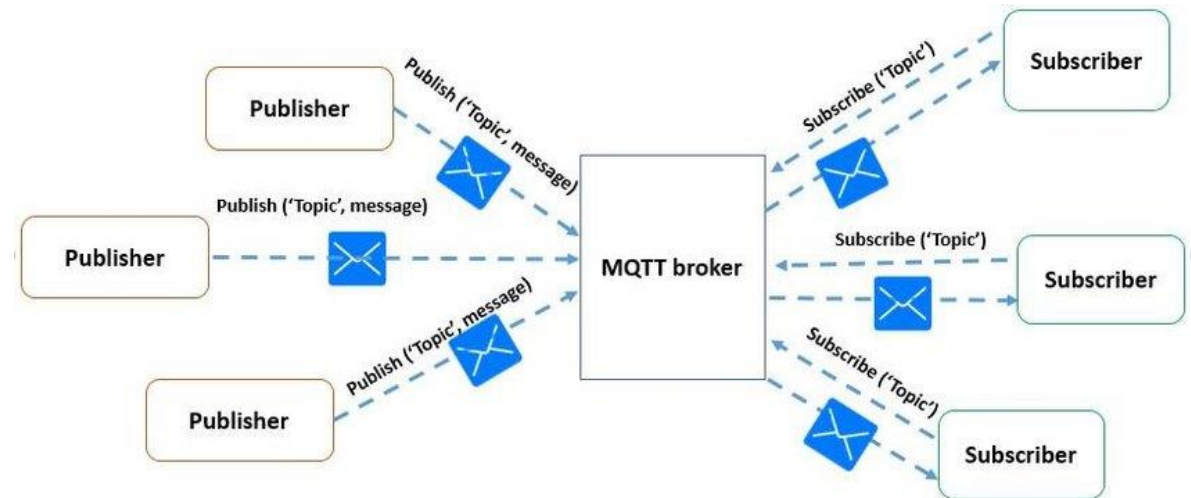
# MQTT (MESSAGE QUEUING TELEMETRY TRANSPORT)

## MQTT Clients

- MQTT clients publish a message to an MQTT broker and other MQTT clients subscribe to messages they want to receive
- Implementation of MQTT clients typically requires a minimal footprint, so it is well suited for deployment on small constrained devices and are very efficient in their bandwidth requirements

## MQTT Broker

- MQTT brokers receive published messages and dispatch the message to the subscribing MQTT clients
- An MQTT message contains a message topic that MQTT clients subscribe to and MQTT brokers use these subscription lists for determining the MQTT clients to receive the message



# MQTT Implementation in Python - Publish

- Install MQTT package
  - *pip install paho-mqtt*
  - More info here: <https://pypi.org/project/paho-mqtt/>
- Import paho library
  - *import paho.mqtt.client as mqtt*
- Use a public broker, declare objects/methods
  - *mqttBroker = "broker.hivemq.com" // public broker*
  - *client = mqtt.Client("fake\_temp1") // your name as client*
  - *client.connect(mqttBroker)*
- Publish a message
  - *client.publish("room\_temp", randN) // topic is room\_temp*

# MQTT Implementation in Python - Subscribe

- Import paho library
  - *import paho.mqtt.client as mqtt*
- Use a public broker, declare objects/methods
  - *mqttBroker = "broker.hivemq.com"*
  - *client = mqtt.Client("fake\_temp2") // your name as client*
  - *client.connect(mqttBroker)*
- Subscribe to receive a message
  - *client.loop\_start()*
  - *client.subscribe("room\_temp")*
  - *client.on\_message = on\_message // call a function*
- Callback function
  - *def on\_message(client, userdata, message):*
    - *b = message.payload.decode("utf-8")*
    - *print ("message received", b)*

# Quality of Service Levels (QoS)

- The Quality of Service (QoS) level is an agreement between the sender of a message and the receiver of a message that defines the guarantee of delivery for a specific message.
- These QoS levels allow for more reliable IoT applications since the underlying messaging infrastructure can adapt to unreliable network conditions.
- MQTT implements 3 levels of Quality of Service
  1. At most once (0),
  2. At least once (1),
  3. Exactly once (2).



# Quality of Service Levels (QoS)

- At most once (0) (fire and forget)
  - QoS0, no guarantee of delivery.
  - The recipient does not acknowledge receipt of the message and the message is not stored and re-transmitted by the sender.
  - provides the same guarantee as the underlying TCP protocol.
- At least once (1)
  - QoS1 guarantees that a message is delivered at least one time to the receiver. The sender stores the message until it gets a PUBACK packet from the receiver that acknowledges receipt of the message.
  - It is possible for a message to be sent or delivered multiple times.
- Exactly once (2)
  - QoS 2 mode guarantees that the message is delivered exactly once. MQTT has a facility for a variable length header. It does not provide any MQTT ACK response, but its default transport protocol (TCP) provides TCP ACK for each packet sent.

# Retained Messages

- MQTT clients that subscribe to a new topic have no insight into when to expect the first message they will receive.
- However, an MQTT broker can store a retained message that can be sent immediately upon a new MQTT subscription.
- In this case, the MQTT client will receive at least one message upon subscribing to the topic.

## Publish:

```
client.publish("room_temp", randN, qos = 0, retain = True)
```

*# you need to send a null message to cancel the retained message flag*

```
client.publish("room_temp", payload = None, retain = True)
```

## Subscribe:

```
print ("topic", message.topic, "retained flag?", message.retain)
```

# Last Will and Testament

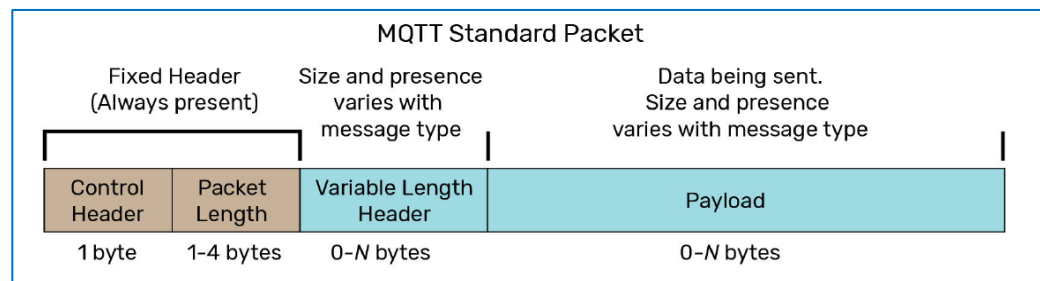
An MQTT client can specify to an MQTT broker a message, called the last will, that will be sent if the MQTT client ungracefully disconnects. This allows for a more graceful system wide notification that a client has been disconnected.

# Persistent Sessions

MQTT allows for a persistent session between the client and the broker. This allows for sessions to persist even if the network is disconnected. Once the network is reconnected, the information to reconnect the client to the broker still exists. This is one of the key features that makes the MQTT protocol more efficient than HTTP for use over unreliable cellular networks.

# MQTT: Packet

- Unlike HTTP, MQTT was designed for machine-to-machine communication. While HTTP is famously heavyweight, with a long list of message headers used to describe and respond to resources, MQTT is data-agnostic, with a streamlined on-the-wire footprint that can be processed efficiently by devices with limited power and processing capabilities.
- It uses a simple byte array payload with a fixed 2-byte header and variable-length header fields (up to a few additional bytes) to indicate packet length or control codes. A packet can be up to 256 MB in size



The image shows a Wireshark network traffic capture of MQTT messages. The packet list pane shows several MQTT packets, with packet 62 selected. The packet details pane shows the structure of the selected packet: Frame 62: 71 bytes on wire (568 bits), 71 bytes captured (568 bits) on interface \Device\NPF\_{C2FDF6B2-36E8-46F3-BC9...}. The packet is an Ethernet II frame from HitronTe\_ac:75:22 to Dell\_a1:c9:b0. It is an Internet Protocol Version 4 packet from 18.193.126.219 to 192.168.0.46. The transport layer is Transmission Control Protocol, port 1883 to 60847. The application layer is the MQTT Telemetry Transport Protocol, Publish Message. The message details are: Header Flags: 0x30, Message Type: Publish Message, QoS Level: At most once delivery (Fire and Forget), Msg Len: 15, Topic Length: 9, Topic: room\_temp, Message: 32322e35. The packet bytes pane shows the raw data in hexadecimal and ASCII.

No.	Time	Source	Destination	Protocol	Length	Info
21	0.594609	18.193.126.219	192.168.0.46	MQTT	71	Publish Message [room_temp]
31	1.723784	192.168.0.46	18.193.126.219	MQTT	75	Connect Command
33	1.862568	192.168.0.46	18.193.126.219	MQTT	70	Subscribe Request (id=1) [room_temp]
34	1.869243	18.193.126.219	192.168.0.46	MQTT	60	Connect Ack
42	1.999658	18.193.126.219	192.168.0.46	MQTT	60	Subscribe Ack (id=1)
62	2.688308	18.193.126.219	192.168.0.46	MQTT	71	Publish Message [room_temp]
116	4.922080	18.193.126.219	192.168.0.46	MQTT	71	Publish Message [room_temp]
142	6.945012	18.193.126.219	192.168.0.46	MQTT	71	Publish Message [room_temp]
178	8.970933	18.193.126.219	192.168.0.46	MQTT	71	Publish Message [room_temp]
240	11.077713	18.193.126.219	192.168.0.46	MQTT	71	Publish Message [room_temp]

> Frame 62: 71 bytes on wire (568 bits), 71 bytes captured (568 bits) on interface \Device\NPF\_{C2FDF6B2-36E8-46F3-BC9...}

> Ethernet II, Src: HitronTe\_ac:75:22 (1c:ab:c0:ac:75:22), Dst: Dell\_a1:c9:b0 (b8:85:84:a1:c9:b0)

> Internet Protocol Version 4, Src: 18.193.126.219, Dst: 192.168.0.46

> Transmission Control Protocol, Src Port: 1883, Dst Port: 60847, Seq: 10, Ack: 38, Len: 17

> MQ Telemetry Transport Protocol, Publish Message

> Header Flags: 0x30, Message Type: Publish Message, QoS Level: At most once delivery (Fire and Forget)

Msg Len: 15

Topic Length: 9

Topic: room\_temp

Message: 32322e35

0000 b8 85 84 a1 c9 b0 1c ab c0 ac 75 22 08 00 45 00 .....u"·E·

0010 00 39 b1 63 40 00 f0 06 86 e8 12 c1 7e db c0 a8 ·9·c@·...·

0020 00 2e 07 5b ed af d9 1c d7 16 77 5b f7 d2 50 18 ··[·...·w[·P·

0030 00 6a 9a 3a 00 00 30 0f 00 09 72 6f 6f 6d 5f 74 ·j·:·:0·...room\_t

0040 65 6d 70 32 32 2e 35 emp22.5

# MQTT strengths

- It requires minimal resources since it is lightweight and efficient
  - Support bi-directional messaging between device and cloud
  - Can scale to millions of connected devices
  - Support reliable message delivery through 3 QoS levels
  - Works well over unreliable networks
  - Security enabled, so it works with TLS and common authentication protocols
- 
- Note: you cannot identify the clients (pub or sub) using this protocol

# FAQs on MQTT Payload Format Description and Content Type

- Format of MQTT payload:
  - The format of the MQTT payload is flexible and can contain any data or information that you want to send. MQTT treats the payload as a raw sequence of bytes and does not impose any specific format or structure on it. It can be any arbitrary data, such as strings, JSON, binary data, or even custom formats specific to your application.
- Max payload size:
  - The maximum allowable size of a packet sent by the client or server, before any MQTTS or other framing is added, is 256MB.
- Can we use JSON?
  - Yes, you can use JSON as the payload format in MQTT. MQTT treats the payload as a raw sequence of bytes, allowing you to send any data format you prefer, including JSON.



# Private MQTT Broker on RPI

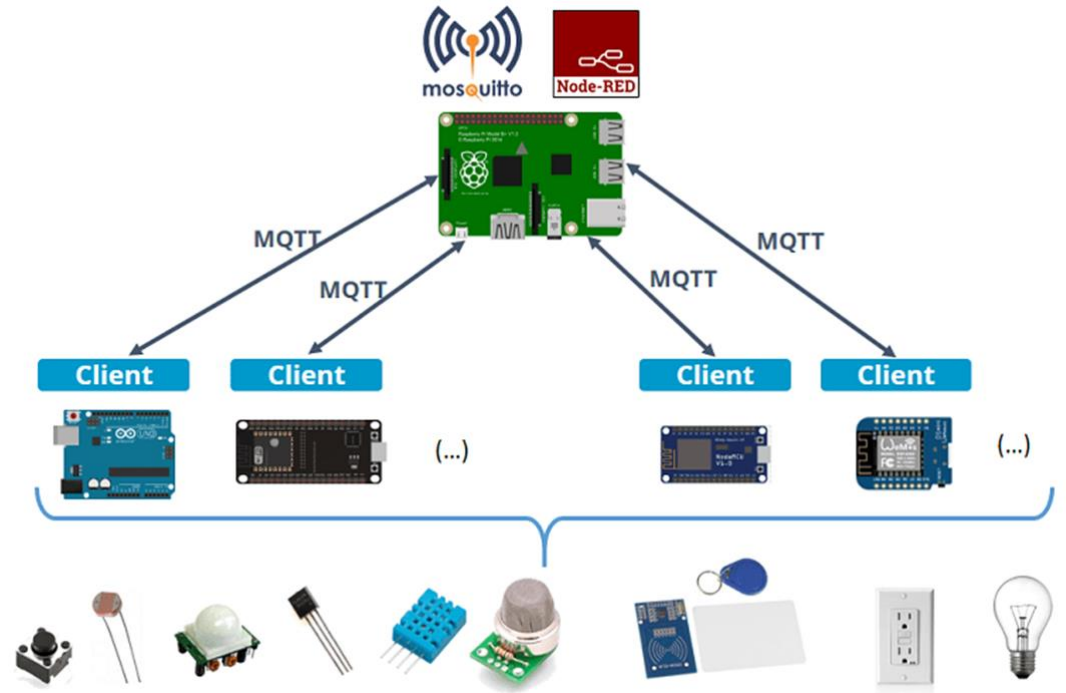
Instead of using public MQTT brokers, you setup a Broker on your PC or RPI.

```
sudo apt update  
sudo apt install -y mosquitto mosquitto-clients
```

```
sudo systemctl enable mosquitto.service
```

```
mosquitto -v
```

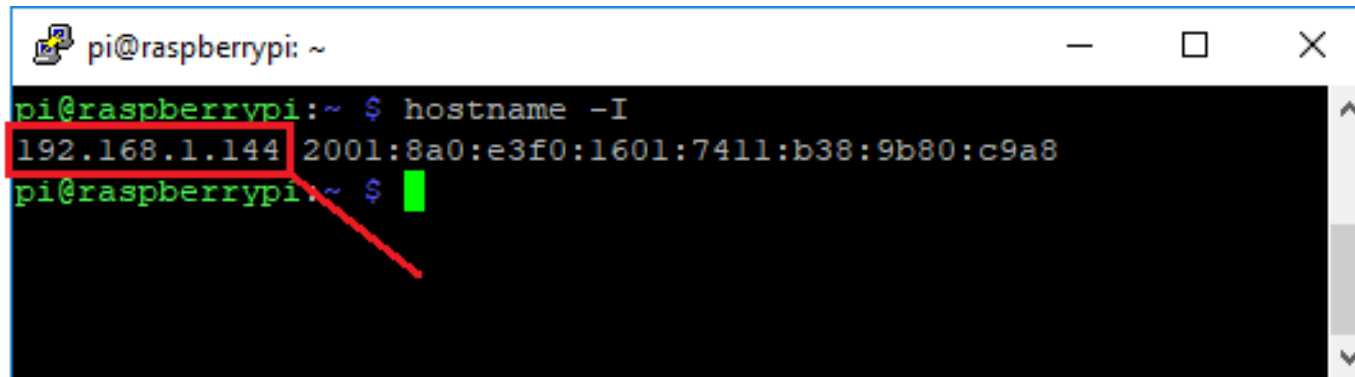
```
pi@raspberrypi: ~  
pi@raspberrypi:~ $ mosquitto -v  
1531840823: mosquitto version 1.4.10 (build date Fri, 22 Dec 2017 08:19:25 +0000) starting  
1531840823: Using default config.  
1531840823: Opening ipv4 listen socket on port 1883.  
1531840823: Error: Address already in use  
pi@raspberrypi:~ $
```



# Private MQTT Broker on RPI

To check the RPI IP address use the following command.  
You will use the RPI IP address as the broker IP.

hostname -I



```
pi@raspberrypi: ~  
pi@raspberrypi:~ $ hostname -I  
192.168.1.144 2001:8a0:e3f0:1601:7411:b38:9b80:c9a8  
pi@raspberrypi:~ $
```

# Secure MQTT

- Authentication with Username and Password

- The MQTT protocol provides username and password for authentication on the application level
- <https://www.hivemq.com/blog/mqtt-security-fundamentals-authentication-username-password/>
- `username_pw_set(username, password=None)` from <https://pypi.org/project/paho-mqtt/#publishing>
- Mosquitto 2.0 and up offers three choices for authentication: password files, authentication plugins, and unauthorised/anonymous access
- <https://mosquitto.org/documentation/authentication-methods/>

- Encrypting the MQTT payload

- The advantage is that the data is encrypted end to end and not just between the broker and the client (which is the data link)
- Use python libraries such as [cryptography](#), etc.
- `from cryptography.fernet import Fernet`
- `cipher_key = Fernet.generate_key()`
- `cipher = Fernet(cipher_key)`
- `encrypted_message = cipher.encrypt(randN)`
- `out_message = encrypted_message.decode()`
- `client.publish("room_temp", out_message)`

# How to send large files using MQTT

- There are many ways we can publish larger file, like an image or excel file.
- For example, for publishing an image file:

```
#''  
# open image in binary read format as file  
with open("./image.jpg", 'rb') as file:  
    filecontent = file.read()  
    byteArr = bytearray(filecontent)  
    print(byteArr)  
    print (len(byteArr))  
  
#''  
  
client.publish("room_temp", byteArr)  
print(f"just published byteArr")
```

# How to send large files using MQTT

- Then subscribing the same image file:

```
# ""  
f = open('image_r.jpg', 'wb')  
# open a file for the image and create a name for the received image  
f.write(message.payload)  
# write the received bytes to the opened file named "f"  
f.close()  
# ""
```

- To view the received image:

```
""  
  
import cv2  
from matplotlib  
import pyplot as plt  
%matplotlib inline  
#img = cv2.imread("image_r.jpg")  
# the above will show BGR image, not typical RGB). below is the fix  
img = cv2.imread("image_r.jpg")[...,:-1]  
plt.imshow(img)  
plt.show()  
""
```

# AMQP Implementation

See instructions posted in canvas

# CoAP Implementation

See instructions posted in canvas