



Quality of Service in MQTT:

The Ultimate Guide

TABLE OF CONTENTS

Quality of Service (QoS) in MQTT	3	Best Practices	8
Good to Know MQTT Protocol Basics	3	QoS 2: Exactly Once	8
MQTT Control Packets	3	Best Practices	9
Packet Identifier (Packet ID)	4	The Message Retransmission Policy - Common QoS	
The DUP Flag	4	Misconception	9
Half-open TCP Connections	5	MQTT v3.1.1 Specification	9
MQTT Keep-alive	5	MQTT v5.0 Specification	9
PINGREQ and PINGRESP	5	Summary	10
Broker/Client Disconnects	5	Downgrade of QoS	10
Broker Disconnects the Client	5	No Per-Subscriber QoS for Publishers	11
Client Disconnects from the Broker	6	PUBLISH QoS 2 Messages	11
QoS in Detail	6	PUBLISH QoS 1 Messages	11
QoS 0: At Most Once	6	PUBLISH QoS 0 Messages	12
Best Practices	7	Summary	12
QoS 1: At Least Once	7	Conclusion	13

Quality of Service (QoS) in MQTT

Quality of Service (QoS) in MQTT refers to the level of guarantee for the delivery of a message between a publisher and the broker, or the broker and a subscriber. MQTT defines three levels of QoS, allowing different degrees of reliability based on the application's needs:

QoS 0: "At most once." Least overhead, no guarantee of delivery.

QoS 1: "At least once." Guarantees delivery but may result in duplicates.

QoS 2: "Exactly once." Guarantees exactly one delivery with the highest overhead.

The choice of the QoS level in MQTT depends on the specific requirements of the application in terms of reliability, network bandwidth, and the tolerance for message duplication. It describes the reliability and guarantee of message delivery in both directions:

1. From the publisher(s) to the MQTT broker.
2. From the MQTT broker to the subscriber(s).

It is important to note that QoS is not an **end-to-end agreement between the publisher and subscriber(s)** directly. Instead, each communication pair - publisher to broker, and broker to subscriber - is treated individually. This separation allows MQTT to manage message delivery reliability across various scenarios, while maintaining flexibility in how different QoS levels are applied. MQTT clients, whether they are publishers or subscribers, interact with the broker using the specified QoS levels when publishing a message or subscribing to a topic to ensure appropriate delivery guarantees.

But before we go into details of the different Quality of Service levels and how they work in MQTT, it is crucial to understand some MQTT protocol basics.

Good to Know MQTT Protocol Basics

MQTT Control Packets

MQTT Control Packets are the foundation of the MQTT protocol. MQTT works by exchanging a series of MQTT Control Packets in a defined way. MQTT v3.1.1 knows 14 Control Packets:

Packet Name	Description
CONNECT	Initiates a connection between the client and broker.
CONNACK	Acknowledges the connection request from the client.
PUBLISH	Used to send messages from a client to the broker (and then to subscribers).
PUBACK	Acknowledges the receipt of a message with QoS 1.
PUBREC	Acknowledges the receipt of a message with QoS 2 (first step of the QoS 2 handshake).
PUBREL	Second step in the QoS 2 handshake, indicating the message is ready to be delivered.
PUBCOMP	Final step in the QoS 2 handshake, confirming the completion of the message delivery.
SUBSCRIBE	A client sends this packet to subscribe to specific topics.
SUBACK	Acknowledges the subscription request from the client
UNSUBSCRIBE	A client sends this packet to unsubscribe from specific topics.

Packet Name	Description
UNSUBACK	Acknowledges the unsubscription request from the client.
PINGREQ	Sent by a client to check if the connection is still alive (keep-alive).
PINGRESP	Sent by the broker in response to a PINGREQ to confirm the connection is still active.
DISCONNECT	Sent by a client or broker to gracefully end the connection.

as well as

AUTH	Introduced in MQTT 5.0, this packet is used for enhanced authentication mechanisms. It supports multi-step authentication processes, allowing for more complex and secure authentication methods.
-------------	---

An MQTT Control Packet has basically the following structure:

Fixed header	Present in all MQTT Control Packets, contains information like the Control Packet type as well as flags specific to each Packet type (e.g. DUP or RETAIN)
Variable header	Present in some MQTT Control Packets, has e.g. the Packet Identifier field.
Payload	Present in some MQTT Control Packets, the final part of a message, size up to 256MB.

Packet Identifier (Packet ID)

The Packet ID is a crucial component in MQTT that plays a key role in ensuring the reliability and order of message delivery. It is particularly important for messages with QoS levels greater than 0 and it is a unique identifier assigned to MQTT messages that require acknowledgment, specifically those with QoS 1 and QoS 2. The 2 byte Packet Identifier field is located in the variable header component of many of the MQTT Control Packet types (2 bytes/16-bit integer, ranging from 1 to 65535).

The Packet ID in MQTT must be unique for each message sent from a particular client at any given time. This uniqueness prevents confusion between different messages that might be in transit simultaneously. If a Client re-sends a particular Control Packet, then it **MUST** use the same Packet Identifier in subsequent re-sends of that packet. Once the message exchange associated with a specific Packet ID is complete, the ID can be reused for new messages. However, during the lifecycle of a message, each Packet ID must remain unique to avoid conflicts.

This design is efficient because it is unlikely that a client would have more than 65,535 active messages without completing an interaction. As a result, a larger packet identifier range is unnecessary, ensuring the system remains simple and lightweight while effectively managing message delivery and acknowledgment.

The DUP Flag

The DUP flag (Duplicate Delivery Flag) in MQTT is a key indicator used in the message header to signal whether a message is being retransmitted by the client or broker. This occurs when the original message was sent with a QoS of 1 or 2, but the sender did not receive the expected acknowledgment (PUBACK or PUBREC). As a result, the message may be retransmitted in specific circumstances with the same Packet ID and the DUP flag set to 1 to indicate it is a **potential** duplicate. The DUP flag does not mean the message is definitely a duplicate, only that it might be.

Half-open TCP Connections

One of the key features of MQTT is its reliance on TCP (Transmission Control Protocol) as the underlying transport layer protocol. When MQTT is used, TCP/IP ensures that

messages are reliably delivered from publishers to brokers and then from brokers to subscribers, which is critical for applications where message integrity is vital. TCP is a connection-oriented, reliable transport protocol that provides ordered and error-checked delivery of data between devices.

This client/broker TCP connection remains persistent and is the foundation for all subsequent MQTT communications. Thus it is assumed (and essential) that TCP ensures correct transmission of the control packets at all times, as TCP already has a built-in retry mechanism/delivery guarantee as part of its reliable data transfer process.

Still, this also introduces challenges. One major cause of problems in TCP are so called “half-open TCP connections”: one side (client or broker) “believes” the connection is still active while the other has closed it. This state can occur due to various network issues like an unexpected crash, reboot, or a device losing power.

MQTT clients and brokers need robust mechanisms to detect and handle these half-open TCP connections: the keep-alive mechanism (MQTT control packets **PINGREQ** and **PINGRESP**).

MQTT Keep-alive

The **MQTT keep-alive** mechanism is a feature designed to maintain an active connection between a client and a broker in the MQTT protocol.

The keep-alive mechanism ensures that both the client and the broker are aware that the connection is still active, even when no actual data messages are being exchanged. This helps in detecting a lost connection and enables timely reconnection or cleanup of resources.

When a client connects to the broker, it specifies a keep-alive interval in seconds. This interval is a duration agreed upon during the connection setup, indicating the maximum allowed time that can pass without the client sending any message to the broker. The MQTT specification allows this interval to be set by the client (also the broker can set a server keep-alive and include this information in the **CONNACK** packet) based on its

requirements and the network conditions.

PINGREQ and PINGRESP

If the client does not have any other message to send within the keep-alive interval, it sends a special message called **PINGREQ** (Ping Request) to the broker.

Upon receiving this, the broker responds with a **PINGRESP** (Ping Response), confirming that the connection is still alive.

This exchange of ping messages does not carry any payload but serves as a heartbeat mechanism to keep the connection open. If the broker does not receive any message (including **PINGREQ**) from the client within 1.5 times the keep-alive interval, it assumes that the connection is lost or the client is no longer active. As a result, the broker must disconnect the client and may clean up resources associated with that client (MQTT [v3.1.1](#) & [v5.0](#)).

The keep-alive interval should be chosen carefully, considering network latency, expected frequency of data exchange, and resource constraints:

- A shorter interval can detect connection issues more quickly but may consume more network resources and power, especially in battery-powered devices.
- A longer interval reduces overhead but may delay the detection of connection loss.

Broker/Client Disconnects

Broker Disconnects the Client

If the client does not send any MQTT Control packet (e.g., **PUBLISH**, **PINGREQ**, etc.) within the defined Keep Alive interval the keep-alive timeout is exceeded and the broker will assume the client is no longer active.

According to the specification:

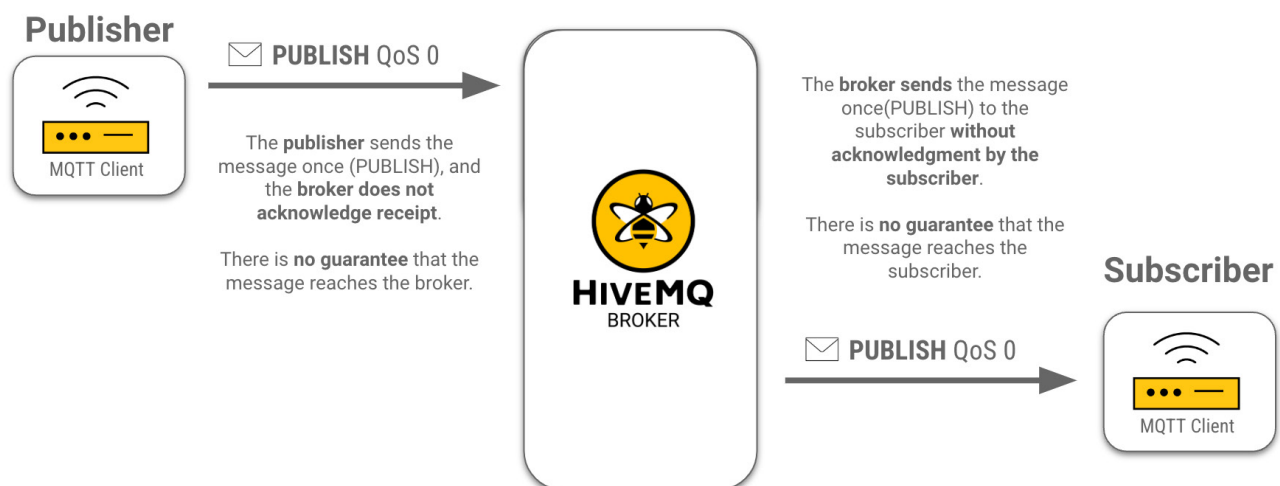
- MQTT v3.1.1: the broker may wait for 1.5 times the keep-alive period before disconnecting.
- MQTT v5.0: the broker can disconnect the client immediately after the keep-alive interval has been breached.

Client Disconnects from the Broker

On the other hand a client may disconnect from the broker when there is no PINGRESP receiver from the Broker as after the client sends a PINGREQ, it expects a PINGRESP from the broker. If the client does not receive this response, it might assume the broker is unresponsive and may decide to disconnect. The MQTT specification does not define an exact timeout for this, but the client typically waits for a reasonable time before disconnecting.

QoS in Detail

QoS 0: At Most Once



Packet ID Usage: Not used.

DUP Flag Usage: Not used, since there are no guarantees of delivery, and no acknowledgments are expected, the DUP flag is irrelevant for QoS 0 messages.

Behavior: The message is sent once and not stored by the client or broker in the context of QoS, so there's no need for further tracking.

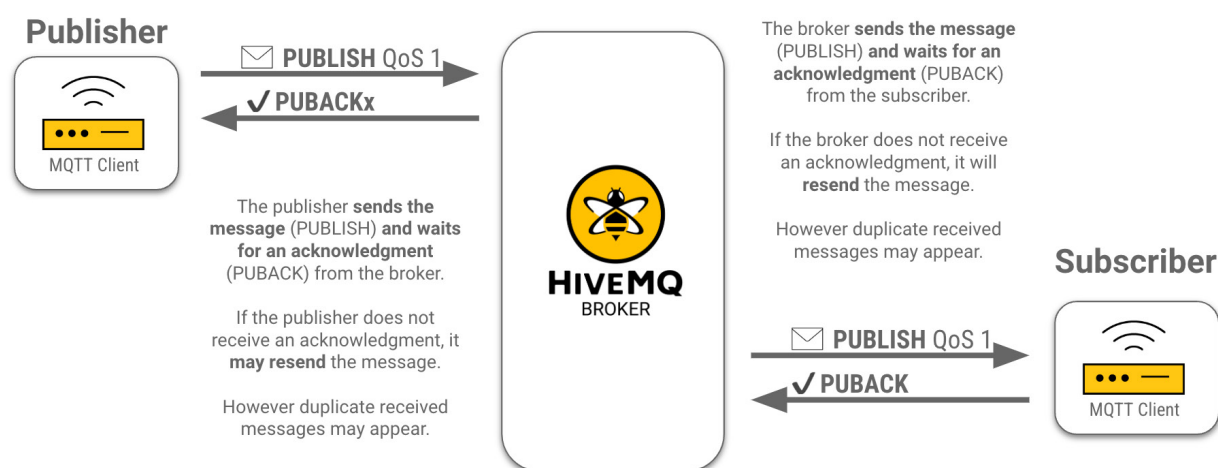
Importance No Queuing: Messages are delivered on a best-effort basis without acknowledgment. If the subscriber is offline or disconnected, the broker does not queue the message. With QoS 0 messages will be simply discarded, delivery is not ensured.

Best Practices

Use QoS 0 when:

- You have a completely or mostly stable connection between sender and receiver. A classic use case for QoS 0 is connecting a test client or a front end application to an MQTT broker over a wired connection.
- You don't mind if a few messages are lost occasionally. The loss of some messages can be acceptable if the data is not that important or when data is sent at short intervals
- You don't need message queuing. Messages are only queued for disconnected clients if they have QoS 1 or 2 and a persistent session.

QoS 1: At Least Once



Packet ID Usage: Used.

DUP Flag Usage: Used.

- When a publisher (client or broker) sends a message for the first time, the DUP flag is set to 0.
- If the publisher does not receive an acknowledgment (PUBACK), it may retransmit the message on reconnect with the DUP flag set to 1 (see section "The Message Retransmission Policy").
- This indicates to the receiver that the message might be a duplicate, although the Packet ID remains the same.

Behavior:

- When a publisher sends a message, it assigns a unique Packet ID.
- The client or broker, upon receiving the message, acknowledges it using the same Packet ID (in PUBACK).
- If the publisher does not receive the acknowledgment, it re-sends the message after reconnecting with the same Packet ID until an acknowledgment is received.

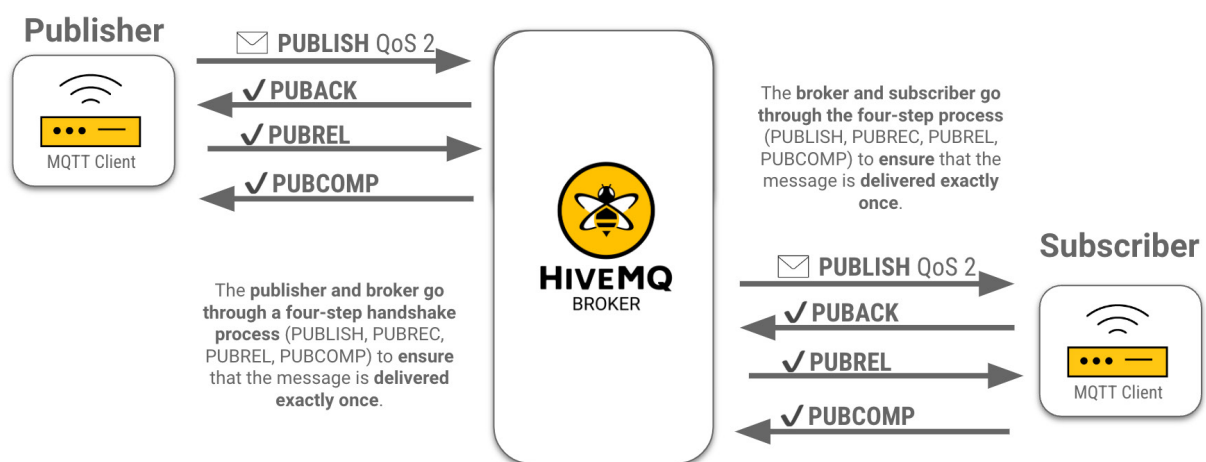
Importance: Messages are queued with acknowledgment: If the subscriber is offline, the broker queues the message and attempts to deliver it when the QoS 1 subscriber reconnects. This requires a persistent session, so the broker stores the message until it can be delivered and acknowledged by the QoS 1 subscriber. QoS 1 ensures the message is delivered at least once but may result in duplicate messages.

Best Practices

Use QoS 1 when:

- You need to get every message and your use case can handle duplicates. QoS level 1 is the most frequently used service level because it guarantees the message arrives at least once but allows for multiple deliveries. Of course, your application must tolerate duplicates and be able to process them accordingly.
- You can't bear the overhead of QoS 2 as QoS 1 delivers messages much faster than QoS 2.

QoS 2: Exactly Once



Packet ID Usage: Used extensively.

DUP Flag Usage: Used, similar to QoS 1, when a client sends a message for the first time, the DUP flag is set to 0.

- If the message needs to be retransmitted due to a lack of acknowledgment by PUBREC (if publisher does not receive the PUBCOMP it just resends the PUBREL not the PUBLISH, otherwise exactly once will not be guaranteed), the DUP flag of the PUBLISH is set to 1.
- The message goes through the four-step handshake process, and the DUP flag helps the receiver identify and manage duplicates during this process.
- The broker uses the DUP flag to retransmit unacknowledged messages in the first phase of the QoS 2 handshake (PUBLISH => PUBREC).
- Internally each subscriber has its own outgoing message queue and the broker keeps track of the message flow. After delivering the PUBLISH to the client, and receiving of PUBREC, the PUBLISH is replaced in the subscribers queue with a PUBREL which is then sent to the subscribing client. After receipt of PUBCOMP the PUBREL is removed from the subscribers queue. This is how exactly once on QoS 2 is ensured and retransmissions does not result in multiple message deliveries to the client

Behavior:

- The publisher assigns a Packet ID to the message and sends it to the broker.
- The broker responds with a PUBREC (Acknowledgment Receipt) using the same Packet ID.
- The publisher sends a PUBREL (Release) message with the Packet ID to the broker.
- The broker responds with PUBCOMP (Completion) using the Packet ID to confirm that the message was delivered exactly once.

Importance: Reliable Queuing with Handshake: Messages are delivered exactly once. If the QoS 2 subscriber is offline, the broker queues the message and ensures delivery using a four-step handshake process (PUBLISH, PUBREC, PUBREL, PUBCOMP) when the subscriber reconnects. Similar to QoS 1, QoS 2 relies on a persistent session, the broker first stores the PUBLISH until the PUBREC is received and then it stores the PUBREL until the PUBCOMP is received, guaranteeing that the message is delivered exactly once.

Best Practices

Use QoS 2 when:

- It is critical to your application to receive all messages exactly once. This is often the case if a duplicate delivery can harm application users or subscribing clients.
- Be aware of the overhead and that the QoS 2 interaction takes more time to complete.

The Message Retransmission Policy - Common QoS Misconception

First of all it is very important to understand: There is no timeout for message retransmission in MQTT QoS. Message retransmission only happens on client reconnection!

“After the timeout expires, the sender retransmits the original message and resets the timer, waiting again for the PUBACK (QoS 1+2) and PUBREC, PUBREL, PUBCOMP (QoS 2) . This process continues until the awaited acknowledgment is received, ensuring that the message is delivered, even though duplicates might occur.”

This is a very widespread misunderstanding of how QoS with MQTT works, and this is not right at all!

Following the MQTT specification **there is no timeout on missing QoS 1/2 acknowledgment** control packets (PUBACK,

PUBREC, PUBREL, PUBCOMP) and not retransmission in an ongoing client session.

Again, retransmission of unacknowledged PUBLISH packets is only required by client or broker on client reconnects (MQTT v3.1.1+5.0). MQTT v5.0 is even much stricter by saying: Clients and Servers **MUST NOT** resend messages at any other time than on reconnects.

MQTT v3.1.1 Specification

The MQTT v3.1.1 about retransmission of messages:

4.4 Message delivery retry

*When a Client reconnects with CleanSession set to 0, both the Client and Server **MUST** re-send any unacknowledged PUBLISH Packets (where QoS > 0) and PUBREL Packets using their original Packet Identifiers [MQTT-4.4.0-1]. **This is the only circumstance where a Client or Server is REQUIRED to redeliver messages.***

Non normative comment

Historically retransmission of Control Packets was required to overcome data loss on some older TCP networks. This might remain a concern where MQTT 3.1.1 implementations are to be deployed in such environments.

MQTT v5.0 Specification

The MQTT v5.0 about retransmission of messages:

4.4 Message delivery retry

*When a Client reconnects with Clean Start set to 0 and a session is present, both the Client and Server **MUST** resend any unacknowledged PUBLISH packets (where QoS > 0) and PUBREL packets using their original Packet Identifiers. **This is the only circumstance where a Client or Server is REQUIRED to resend messages. Clients and Servers MUST NOT** resend messages at any other time [MQTT-4.4.0-1].*

*If PUBACK or PUBREC is received containing a Reason Code of 0x80 or greater the corresponding PUBLISH packet is treated as acknowledged, and **MUST NOT** be retransmitted [MQTT-4.4.0-2].*

Please note: A MQTT broker will always **initiate a session for every client**, and this is how it manages the state in MQTT. MQTT is **stateful** because it requires the broker to maintain the client's session information, which includes the state of subscriptions, message queues, and other session-related data. However a persistent session is crucial in order to queue undelivered QoS 1 + 2 messages, to be retransmitted on reconnect:

The client explicitly requests a **persistent session** (MQTT v3.1.1: `cleanSession=false`, MQTT v5.0: `cleanStart=false + sessionExpiry>0`): The broker retains both unacknowledged messages and the client's subscription state. When the client reconnects, all undelivered messages (QoS 1 and 2) are retransmitted, and the session picks up where it left off.

The client explicitly requests a **non-persistent session**:

MQTT v3.1.1: `cleanSession=true`, the broker does not retain the session after disconnection, meaning previous subscriptions, queued messages (QoS 1 and 2), and other session state are discarded.

MQTT v5.0: `cleanStart=true + sessionExpiry=0`, with `cleanStart=true`, a new session is initiated with no previous state, and `sessionExpiry=0` ensures that the session is **deleted immediately** when the client **disconnects**. In this case, undelivered QoS 1 and 2 messages are discarded.

See <https://www.hivemq.com/blog/mqtt-essentials-part-7-persistent-session-queuing-messages/>

Summary

Unlike TCP, where retransmission is driven by timeouts at the transport layer, MQTT handles retransmissions upon the re-establishment of a connection. The session persistence of MQTT plays an important role in this: **session persistence must be enabled**, in order to store unacknowledged messages (QoS 1 and QoS 2) by both the client and broker, and retransmissions occur automatically when the client reconnects. QoS 0 doesn't support retransmission or queuing of undelivered messages at

all.

Downgrade of QoS

In a nutshell: No Persistent Session => No State => No retransmission (on reconnect)

MQTT is an asynchronous one-to-many protocol. Publishers as well as subscribers are free to choose their QoS level. It is entirely possible to have a number of subscribers with different Quality of Service levels. The difference in QoS levels between the publisher and the subscriber provides flexibility in MQTT communication. It allows the sender to define the reliability of message delivery to the broker independently of how the message is ultimately delivered to the receiver. This is particularly useful in scenarios where different devices or applications have varying reliability and bandwidth requirements.

But it is important to understand that in MQTT the QoS is NOT an end-to-end delivery guarantee, as publishers can only specify a single QoS level for the messages they published, and this QoS level does not systematically apply to subscribers of the topic. This QoS level applies to the delivery of that message from the publisher to the broker. The broker then forwards the message to all subscribers according to the QoS individual level granted to each subscriber, which might be the same, lower or higher than the QoS level used by the publisher.

The broker plays a crucial role in managing these different QoS levels. Even if the message is published with a higher QoS, the broker will downgrade it as necessary to match the QoS level granted to each subscriber. Conversely, if the message is published with a lower QoS, the broker cannot upgrade it beyond that level, even if a subscriber requests a higher QoS.

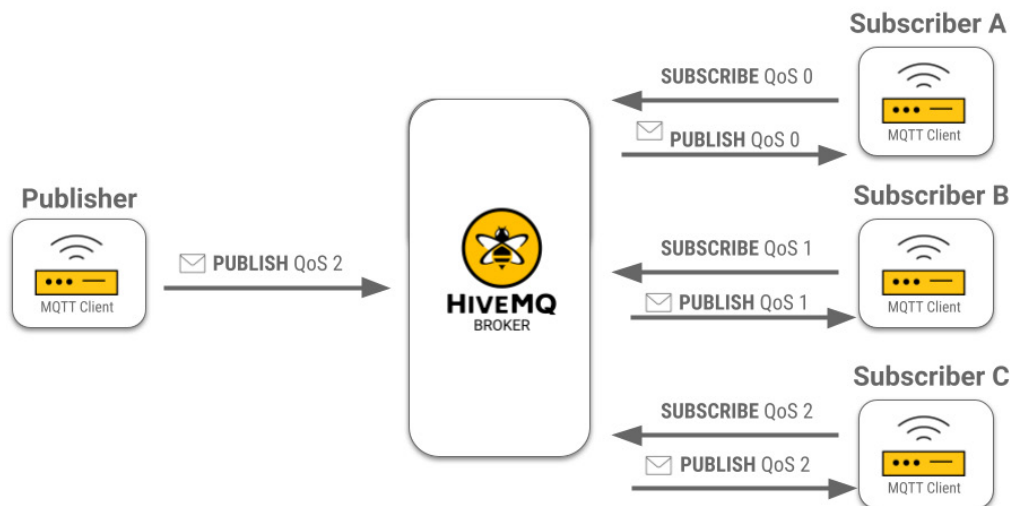
No Per-Subscriber QoS for Publishers

Given each subscriber to a particular topic can specify its desired QoS level independently when it subscribes to the topic:

- Subscriber A subscribes with QoS 0
- Subscriber B subscribes with QoS 1
- Subscriber C subscribes with QoS 2

- Given a Publisher sends a message with a QoS of 2:
- **Subscriber A** will receive the message with QoS 0
- **Subscriber B** will receive the message with QoS 1
- **Subscriber C** will receive the message with QoS 2

F



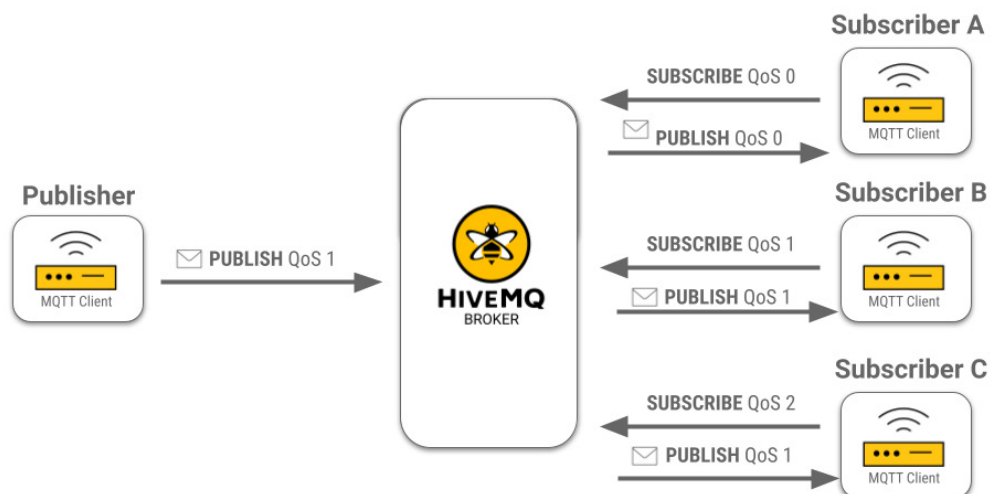
PUBLISH QoS 1 Messages

Given a Publisher sends a message with a QoS of 1:

Subscriber A will receive the message with QoS 0

Subscriber B will receive the message with QoS 1

Subscriber C will receive the message with QoS 1



PUBLISH QoS 0 Messages

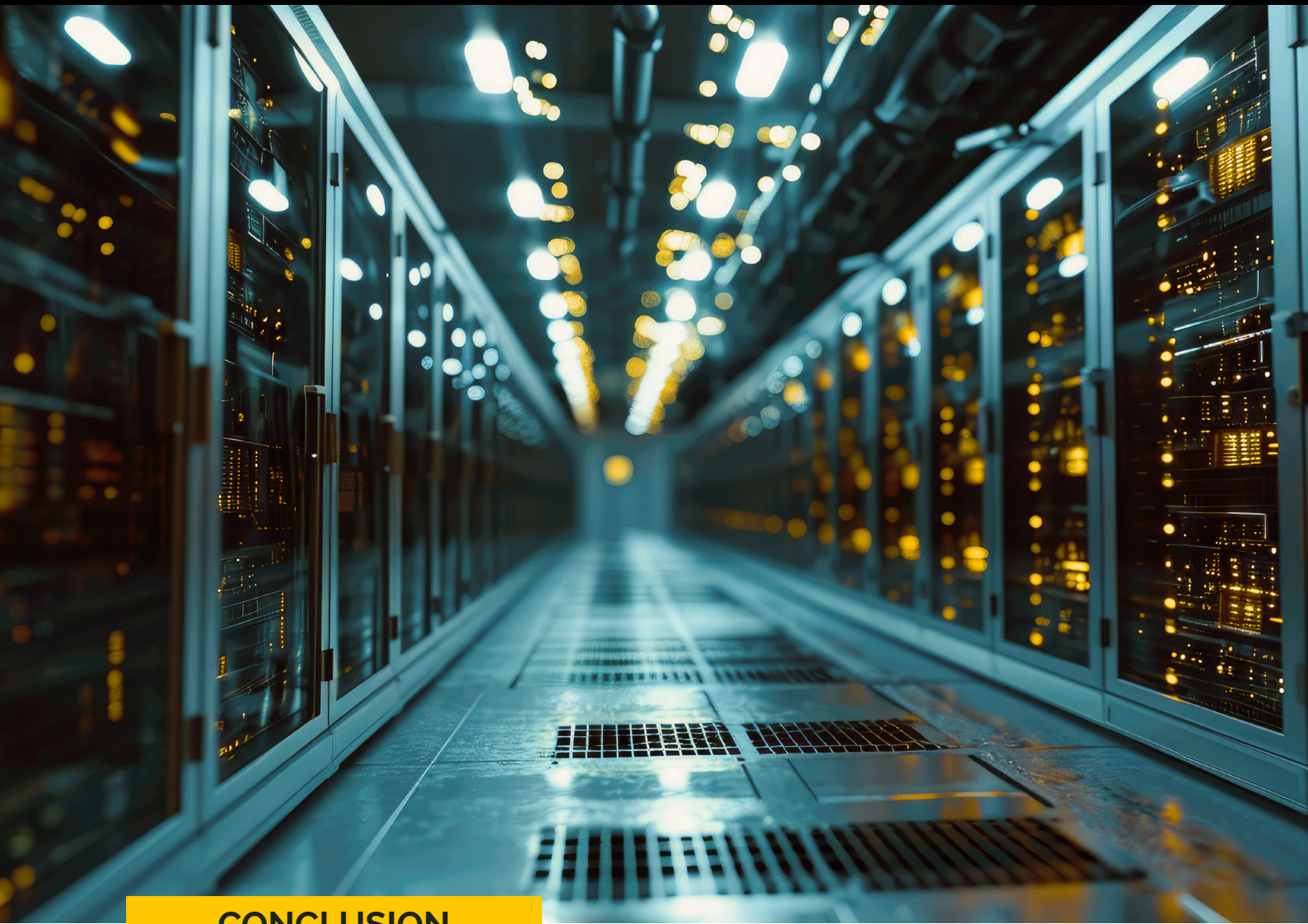
Given a Publisher sends a message with a QoS of 0:

- **Subscriber A** will receive the message with QoS 0
- **Subscriber B** will receive the message with QoS 0
- **Subscriber C** will receive the message with QoS 0



Summary

Publisher - Broker (publish QoS)	Broker - Subscriber (subscribed QoS)	Effective End to End QoS
0	0/1/2	0
1	0	0
1	1 or 2	1
2	0	0
2	1	1
2	2	2



CONCLUSION

Understanding the QoS levels in MQTT is crucial for implementing efficient and reliable IoT communications. Whether your application demands high reliability or can tolerate some degree of message loss, MQTT's flexible QoS options provide the necessary tools to tailor message delivery to your specific needs.

For those looking to deepen their understanding of MQTT and its practical applications, we encourage you to explore HiveMQ's [MQTT Essentials series](#).

About HiveMQ

HiveMQ empowers businesses to unlock the full potential of their data with the most trusted edge-to-cloud IoT data streaming platform. Built on MQTT's publish/subscribe architecture for seamless and flexible integration across operational technology (OT) assets and information technology (IT) applications, HiveMQ ensures businesses can efficiently connect, stream, and govern their data in real-time. With a focus on reliability, scalability, and security, HiveMQ helps organizations get their data AI-ready—enabling advanced analytics, predictive maintenance, and digital transformation. Leading brands like Audi, BMW, Liberty Global, Mercedes-Benz, Siemens, and Eli Lilly trust HiveMQ to modernize their operations, accelerate innovation, and create smarter, data-driven experiences

Visit hivemq.com to learn more

BRANDS THAT TRUST HIVEMQ: _____



AIRFRANCEKLM
GROUP



SIEMENS