HIVEMQ

MQTT Essentials

The Ultimate Guide to the MQTT
Protocol for IoT Messaging

VERSION 2.0

Abstract

In the rapidly evolving landscape of IoT, MQTT has emerged as the de facto standard protocol for data exchange. MQTT Essentials is designed to equip decision-makers, solution architects, and IoT professionals with a strategic and practical understanding of MQTT and how to execute it for scalable, reliable, and seamless data movement. Delve into how MQTT can help your organization overcome the challenges other IoT protocols cannot address with features such as persistent sessions, retained messages, Last Will and Testament (LWT), Quality of Service (QoS) levels, and more. After reading this guide, you'll be ready to use MQTT to optimize connectivity and lay the proper data foundation to enable any IoT or IIoT use case.

TABLE OF CONTENTS

Chapter 1: Introduction to MQTT

MQTT is a lightweight messaging protocol originally designed for communication in constrained networks with limited bandwidth and compute resources. Developed with simplicity and scalability in mind, MQTT is particularly well-suited for Internet of Things (IoT) applications where the variety and quantity of devices are growing exponentially.

Below is the official description of the specification:

> "MQTT is a Client Server publish/subscribe messaging transport protocol. It is lightweight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium."
>
> Citation from the official **MQTT 3.1.1 specification**
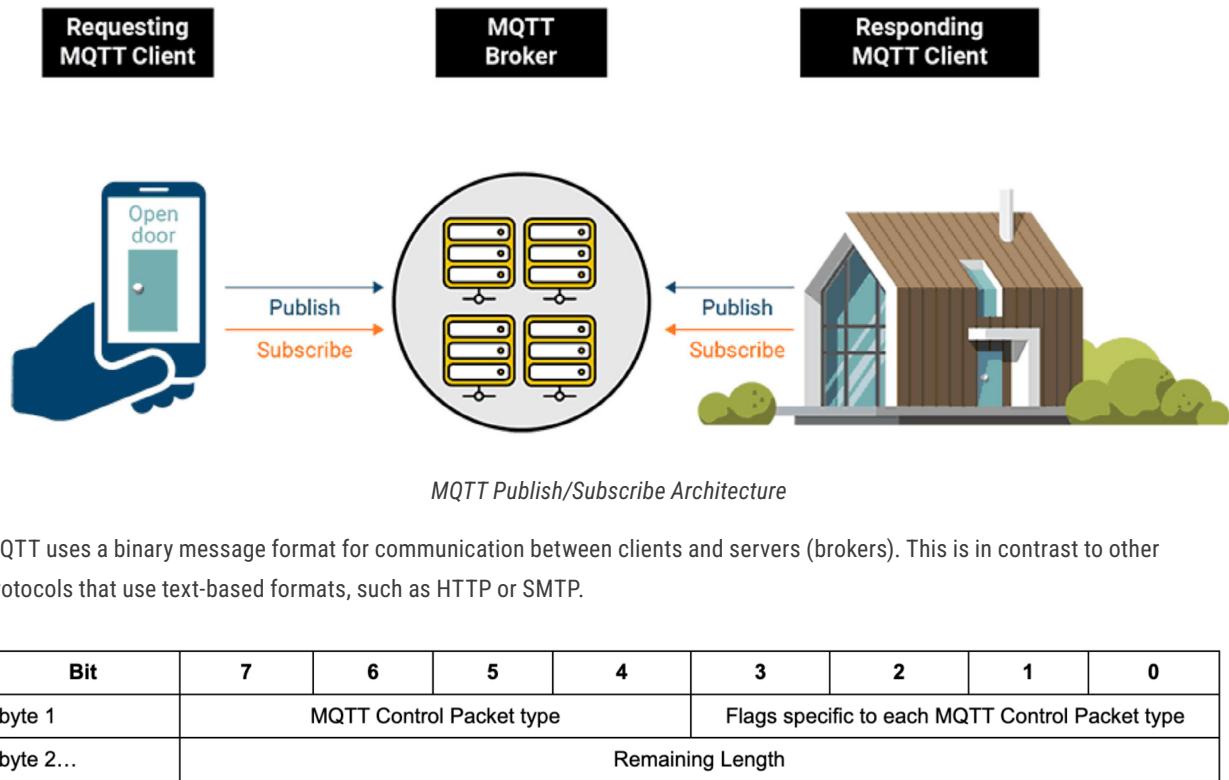
MQTT Publish/Subscribe Architecture

MQTT uses a binary message format for communication between clients and servers (brokers). This is in contrast to other protocols that use text-based formats, such as HTTP or SMTP.

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type				Flags specific to each MQTT Control Packet type			
byte 2…	Remaining Length							

MQTT Fixed Header format example.

Image Source: **http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html**

The binary format used by MQTT is designed to reduce the size of messages and increase the efficiency of communication. By using a binary format, the protocol can minimize the amount of data that needs to be transmitted and reduce the processing power required to interpret messages. This makes MQTT well-suited for use in low-bandwidth or low-power environments, such as IoT devices with limited resources. It's also used in enterprise systems, where real-time data communication is necessary.

Another important aspect of the protocol is that MQTT is extremely easy to implement on the client side. Ease of use was a key concern in the development of MQTT, and this makes it a perfect fit for constrained devices with limited resources.

Benefits of MQTT

MQTT offers several key benefits:

1. **Lightweight and efficien**t: MQTT minimizes the network bandwidth and compute resources required by clients to exchange data.

2. **Bidirectional communication:** MQTT allows devices to send and receive data from the server, allowing for bidirectional data exchange with other components in the network.

3. **Scalable:** MQTT can scale to support millions of devices or "things" in an IoT or IIoT ecosystem.

4. **Reliable message delivery:** MQTT specifies different Quality of Service (QoS) levels to ensure reliable message delivery.

5. **Message Buffering and Session Resumption:** MQTT supports persistent sessions between devices and servers, enhancing message reliability by ensuring that messages are delivered to clients even after disconnections.

6. **Security features:** MQTT supports TLS encryption for message confidentiality and authentication protocols for client verification.

Real-World Applications and Use Cases of MQTT: An Overview

MQTT is used extensively in IoT, Industrial IoT (IIoT), and M2M applications. It has been adopted by companies such as **BMW**, **Air France-KLM**, **Liberty Global**, **Mercedes Benz**, **Hytera**, **Awair**, and **Matternet**, as showcased in **HiveMQ's customer success stories**. These companies have successfully leveraged MQTT in automotive, telecommunications, energy, public safety, and connected product domains.

Here are a few examples:

1. Smart homes: MQTT is used to connect various devices in a smart home, including smart thermostats, light bulbs, security cameras, and other appliances. This allows users to control their home devices remotely using a mobile app.
2. Industrial automation: MQTT is used to connect machines and sensors in factories and other industrial settings. This allows for real-time monitoring and control of processes, which can improve efficiency and reduce downtime.
3. Agriculture: MQTT is used in precision agriculture to monitor soil moisture levels, weather conditions, and crop growth. This helps farmers optimize irrigation and other crop management practices.
4. Healthcare: MQTT is used to connect medical devices and sensors, such as glucose meters and heart rate monitors, to healthcare providers. This allows for remote monitoring of patients, which can improve patient outcomes and reduce healthcare costs.
5. Transportation: MQTT is used in connected cars and other transportation systems to enable real-time tracking and monitoring of vehicles. This can improve safety and help optimize traffic flow.

Now that we have a general understanding of what MQTT is and its characteristics, let's dive into its history and how it came to be a popular messaging protocol. We will explore some of the elements and characteristics of MQTT after learning about its origins.

The Origin and History of MQTT

In 1999, Andy Stanford-Clark of IBM and Arlen Nipper of Arcom (now Cirrus Link) developed the MQTT protocol to enable minimal battery loss and bandwidth usage when connecting with oil pipelines via satellite. The inventors specified several requirements for the protocol, including:

- Simple implementation
- Quality of Service data delivery
- Lightweight and bandwidth-efficient
- Data agnostic
- Continuous session awareness

These goals are still at the core of MQTT. However, the primary focus of the protocol has changed from proprietary embedded systems to open Internet of Things (IoT) use cases.

Over the next ten years, IBM used the protocol internally until they released MQTT 3.1 as a royalty-free version in 2010. This shift in focus from proprietary embedded systems to open IoT use cases created confusion about the acronym MQTT.

> While it formerly stood for MQ Telemetry Transport, where MQ referred to the MQ Series, a product IBM developed to support MQ telemetry transport, MQTT is no longer an acronym. It is now simply the name of the protocol.
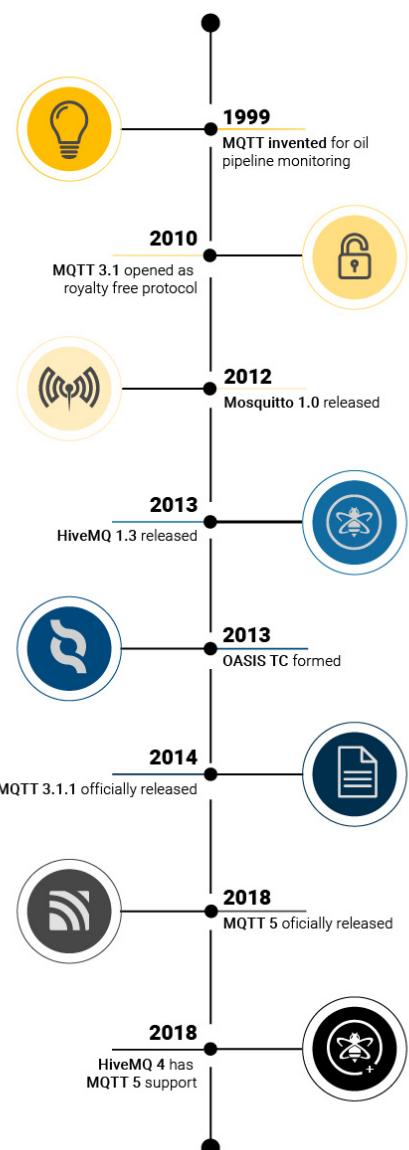
When Andy and Arlen created this protocol in 1999, they named it after the IBM product. Although many sources label MQTT as a message queue protocol, this is not entirely accurate. While it is possible to queue messages in certain cases, MQTT is not a traditional message queuing solution.

In 2011, IBM contributed MQTT client implementations to the newly founded Paho project of the **Eclipse Foundation**, an independent, non-profit corporation that provides a community for open-source software projects. This was a significant development for the protocol because it created a

more supportive ecosystem for MQTT. By contributing MQTT client implementations to an open-source project like Paho, IBM allowed developers to access the protocol and build their applications on top of it. This move helped to increase the visibility and adoption of MQTT among the developer community.

In 2012, HiveMQ became acquainted with MQTT and built the first version of their software that same year. In 2013, HiveMQ released their software to the public.

Timeline of how MQTT evolved and when HiveMQ released an earlier version of its MQTT Broker

The Role of OASIS in Standardizing MQTT

In 2014, OASIS announced that it would take over the standardization of MQTT, with the goal of making it an open and vendor-neutral protocol. Founded in 1993 as a non-profit, OASIS (Organization for the Advancement of Structured Information Standards) is an international consortium that develops open standards for the Internet and related technologies.

It has developed numerous important standards for industries such as cloud computing, security, and IoT, including AMQP, SAML, and DocBook. The standardization process took around one year, and on October 29, 2014, MQTT became an officially approved OASIS standard.

OASIS' involvement in MQTT has been critical to its success as a widely adopted IoT protocol. As a neutral, third-party organization, OASIS ensures that the protocol is maintained as an open standard that can be implemented by anyone without licensing fees or proprietary restrictions.

Additionally, OASIS provides a forum for the community to come together and collaborate on improvements to the protocol, which has resulted in the development of MQTT 5, the latest version of the protocol with new features for improved reliability and scalability.

In March 2019, OASIS ratified the new MQTT 5 specification. This version introduced new features to MQTT that are required for IoT applications deployed on cloud platforms, and cases that require more reliability and error handling to implement mission-critical messaging.

Different Versions of MQTT: MQTT 5 vs. MQTT 3

The earlier version of MQTT was 3.1.1 The latest version is MQTT 5. MQTT 5 has enhancements to improve performance, increase reliability, and provide greater control over communication between clients and servers. Some of the key enhancements include better error reporting, enhanced scalability, and improved support for offline message queueing. These improvements ensure that MQTT can handle

the ever-increasing demands of modern IoT environments, where the number of connected devices and the amount of data they generate are growing exponentially.

While MQTT 5 presents a substantial update to MQTT 3.1.1, it remains more of an evolution than a revolution, faithfully retaining all the features that have contributed to its success. Its lightweightness, push communication, unique attributes, ease of use, exceptional scalability, suitability for mobile networks, and decoupling of communication participants all endure in this latest version. However, several foundational mechanics have been added or slightly adjusted, ensuring that while the new version still has the feel of MQTT, it incorporates improvements that reinforce its standing as the most popular Internet of Things protocol to date.
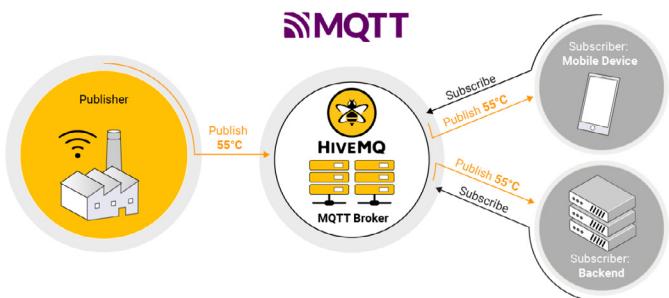
The reassuring news for those acquainted with MQTT 3.1.1 is that the fundamental principles and features remain unchanged in MQTT 5. Despite certain modifications and expansions, the heart of the MQTT you are familiar with persists in version 5. Noteworthy adjustments include slight changes to pre-existing features such as **Last Will and Testament** and the inclusion of popular HiveMQ features like TTL and **Shared Subscriptions**, which you will learn in detail in the chapters ahead.

The underlying protocol has experienced minor transformations, and an additional control packet — AUTH — has been incorporated. However, these changes do not obscure MQTT's core identity. In essence, MQTT v5 is still unambiguously MQTT, retaining its recognizable characteristics while enhancing its capabilities.

Chapter 2: Mastering the Basics of MQTT

At the core of MQTT are MQTT brokers and MQTT clients. The MQTT broker is an intermediary between senders and receivers, dispatching messages to the appropriate

recipients. MQTT clients publish messages to the broker, and other clients subscribe to specific topics to receive messages. Each MQTT message includes a topic, and clients subscribe to topics of their interest. The MQTT broker maintains a subscriber list and uses it to deliver messages to the relevant clients.

An MQTT broker can also buffer messages for disconnected clients, ensuring reliable message delivery even in unreliable network conditions. To achieve this, MQTT supports three different **Quality of Service** (QoS) levels for message delivery: 0 (at most once), 1 (at least once), and 2 (exactly once).

There are two versions of the MQTT specification: MQTT 3.1.1 and MQTT 5. While most commercial MQTT brokers now support MQTT 5, some IoT-managed cloud services still primarily support MQTT 3.1.1. We **highly recommend using MQTT 5** for new IoT deployments due to its enhanced features that focus on robustness and cloud-native scalability.

MQTT Clients

Many open source MQTT Client Libraries are available in a variety of programming languages. HiveMQ provides **MQTT Client Libraries**, which are designed to simplify the deployment and implementation of MQTT clients and offer users top-notch functionality, performance, security, and reliability. Some of the programming languages supported include C#, C++, Java, WebSockets, Python, and more.

Eclipse Paho also offers MQTT client libraries for languages like C/C++ and Python. You can find a comprehensive list of MQTT clients at **mqtt.org**.

MQTT Brokers

MQTT Brokers come in various implementations, catering to different needs, such as open-source, commercial, and managed cloud services. HiveMQ offers two commercial editions: **HiveMQ Professional and HiveMQ Enterprise**. HiveMQ also offers **HiveMQ Cloud**, a managed cloud MQTT service, and **HiveMQ Community Edition**, an open-source version. In addition, HiveMQ offers an MQTT broker embedded in HiveMQ Edge, an open-source industrial device connectivity software for IIoT use cases. For an extensive list of MQTT brokers, please visit **mqtt.org**.

If you are looking to find the ideal MQTT broker for your IoT or industry 4.0 use case, read our blog MQTT Broker Comparison – Which is the Best for Your IoT Application?

How to Make an MQTT Broker Communicate with an MQTT Client?

Here's an overview of how to make an MQTT broker communicate with an MQTT client, such as a sensor, an edge device, a PLC, etc.:

Step 1: Install an MQTT Broker of Your Choice

MQTT brokers are available in commercial, open-source, cloud-managed and general-purpose editions. To get an MQTT broker to communicate with an MQTT client, first you need to find an edition of the broker that is right for your use case.

Here are a few examples of how you can install an MQTT broker in different environments:

- Use a fully-managed MQTT Broker, like HiveMQ Cloud. Here's how to get started with HiveMQ Cloud.
- Download and install an MQTT broker on a server or computer of your choice. Here's how to get started with the HiveMQ platform.
- Run an MQTT broker on Docker. Here's how to get

started with HiveMQ on-premises MQTT Broker.
- Deploy an MQTT broker on Amazon Web Services (AWS). Here's how to get started with HiveMQ on AWS.
- Deploy an MQTT Broker on Microsoft Azure. Here's how to get started with HiveMQ on Azure.

Step 2: Connect MQTT Clients to Your MQTT Broker

The method an MQTT Client uses to connect to an MQTT broker varies based on the broker's setup. Below are a few methods for establishing a connection:

Example 1: Using MQTT CLI for On-Premises MQTT Brokers

If you are unaware, the MQTT CLI is an open-source, Java-based MQTT client tool that enables you to interact quickly and easily with any MQTT broker in various ways. The MQTT CLI comes in various binary packages that can be downloaded from the documentation homepage on GitHub. Here are the steps:

1. Open a terminal window and go to the tools directory of your HiveMQ MQTT Broker. In the tools directory, go to the mqtt-cli/bin folder and enter mqtt sh to start the MQTT CLI in shell mode. The MQTT CLI starts and lists useful options and commands.
2. To connect your first MQTT client to your HiveMQ Broker on localhost and give it a custom identifier, enter con -h localhost -i testClient1. This command creates the first MQTT client with the custom identifier testClient1 and connects the client to your MQTT broker on localhost.
3. To connect another MQTT client that you can use to test your installation, open a second terminal window and enter mqtt sh (keep your original terminal window open).
4. In the second terminal window, enter con -h localhost -i testClient2. This command creates a second MQTT client with the customer identifier testClient2 and connects the client to your MQTT broker on localhost.

Example 2: Using Cloud-Managed MQTT Broker

Using a cloud-managed MQTT broker such as HiveMQ Cloud, you can set up your broker and obtain connection details through the console. These details enable MQTT Clients to connect to the broker. Here's an overview of how to connect

MQTT Client to HiveMQ Cloud.

1. Sign up for Serverless FREE HiveMQ Cloud plan.
2. Create a free HiveMQ Cloud cluster.
3. Create access credentials that your MQTT clients use to connect to the cluster.
4. Copy the Cluster url, port number, and access credentials.
5. Use the details in your MQTT client to connect to HiveMQ Cloud. For detailed steps, check out our documentation.

Step 3: Start Publishing MQTT Messages/ Subscribing to an MQTT Topic

Using MQTT CLI or via the console, you can start publishing MQTT messages and subscribing to MQTT topics. Here's an example using MQTT CLI:

1. In the terminal window of the MQTT client (testClient2), enter sub -t testTopic -s.This command subscribes testClient2 to all messages that are published with the topic testTopic.
2. In the terminal window of another MQTT client (testClient1), enter pub -t testTopic -m Hello. This command publishes the message Hello from testClient1 with the topic testTopic.The message Hello appears immediately in the terminal window of testClient2.

For detailed steps, check out our documentation.

To learn more about how an MQTT Client communicates with an MQTT Broker, read our blog MQTT Client, MQTT Broker, and MQTT Server Connection Establishment Explained.

If you are new to MQTT, read our blog post MQTT Tutorial: An Easy Guide to Getting Started with MQTT.

Mechanism of How an MQTT Client Establishes a Connection with an MQTT Broker

Here's a basic overview of the mechanism of how an MQTT client establishes a connection with an MQTT broker:

- **Client Initialization:** The MQTT client initializes its MQTT library and sets up the required parameters, such as the broker's address and port.
- **Connection to the broker:** The client establishes a TCP/IP connection with the MQTT broker. The default port for MQTT is 1883 (or 8883 for encrypted connections using TLS/SSL).
- **Handshake:** Once the TCP/IP connection is established, an MQTT handshake occurs. This involves exchanging control packets to establish the connection and deal with parameters.
- **Connect packet:** The client sends a "CONNECT" packet to the broker, indicating its intention to establish a connection. This packet includes information such as client ID, connection flags, and other settings.
- **Acknowledgment (CONNACK):** The broker responds with a "Connack" packet, indicating whether the connection request is accepted or rejected.
- **Subscriptions (optional):** If the client is a subscriber, it can then send "Subscribe" packets to the broker, indicating the topics it wants to subscribe to.
- **Publishing messages:** Once the connection is established, clients can publish messages to topics or subscribe to topics to receive messages from other clients.
- **Keep-Alive:** To maintain the connection, clients periodically exchange "Ping" packets to confirm that the connection is still active.

MQTT Brokers, like HiveMQ, offer advanced features for reliable, flexible, scalable and secure MQTT client connection.

Example Implementation of MQTT

To illustrate how MQTT works, below is a simple example that utilizes **HiveMQ Cloud**. To test this implementation on a live cluster, **sign up for HiveMQ Cloud**, which allows you to connect up to 100 IoT devices at no cost. Sign-up without
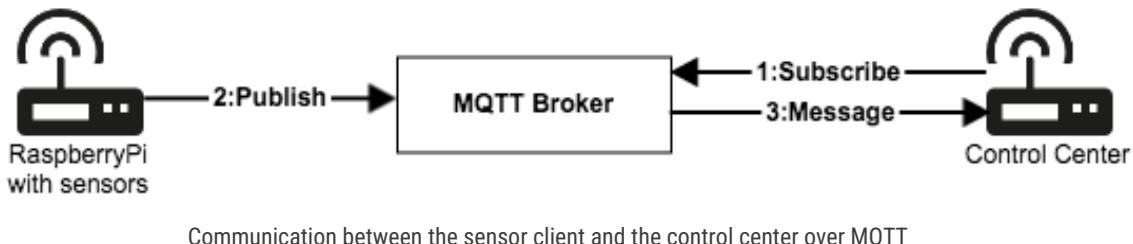
credit card information.

As a first-time user of HiveMQ Cloud, you will be automatically directed to the "Getting Started" section within the management view of your cluster. Here, you can create access credentials and obtain connection details, as you will see later in this example.

Use Case Example: An IoT Application Built Using Raspberry Pi, MQTT, and Temperature Sensor

In this example, we explore the connection of a temperature and brightness sensor to a Raspberry Pi, and then leveraging the power of MQTT to transmit the sensor data to a designated MQTT broker effortlessly. You will discover how another device, acting as a control center, can effortlessly receive and process the MQTT data, enabling efficient monitoring and control of your IoT ecosystem.

Communication between the sensor client and the control center over MQTT

Step 1 - Use the Raspberry Pi as an MQTT Client Connected to Sensors

Once you've successfully **signed up for HiveMQ Cloud**, click on Manage Cluster and head to the Connection Settings section on the Overview tab of your cluster. There, you'll discover your unique hostname/Cluster URL. Copy this hostname and replace it in the code snippet provided below.

To establish a secure connection between your MQTT client and the cluster, creating MQTT client access credentials is essential. Navigate to the Access Management tab of your HiveMQ Cloud cluster, and enter the username, password and set Permission to Publish and Subscribe. These credentials will also replace "<your_username>" and "<your_password>" within the code snippet.

```java
public class Sensor {

    public static void main(String[] args) throws InterruptedException {
        final String host = "<your_host>"; // use your host-name, it should look like
'<alphanumeric>.s2.eu.hivemq.cloud'
        final String username = "<your_username>"; // your credentials
        final String password = "<your_password>";


        // 1. create the client
        final Mqtt5Client client = Mqtt5Client.builder()
                .identifier("sensor-" + getMacAddress()) // use a unique identifier
                .serverHost(host)
                .automaticReconnectWithDefaultConfig() // the client automatically
reconnects
                .serverPort(8883) // this is the port of your cluster, for mqtt it is the
default port 8883
                .sslWithDefaultConfig() // establish a secured connection to HiveMQ Cloud
using TLS
                .build();



        // 2. connect the client
        client.toBlocking().connectWith()
                .simpleAuth() // using authentication, which is required for a secure
connection
                .username(username) // use the username and password you just created
                .password(password.getBytes(StandardCharsets.UTF_8))
                .applySimpleAuth()
                .willPublish() // the last message, before the client disconnects
                    .topic("home/will")
                    .payload("sensor gone".getBytes())
                    .applyWillPublish()
                .send();

        // 3. simulate periodic publishing of sensor data
        while (true) {
            client.toBlocking().publishWith()
                    .topic("home/brightness")
                    .payload(getBrightness())
                    .send();
```

```
            TimeUnit.MILLISECONDS.sleep(500);

            client.toBlocking().publishWith()
                    .topic("home/temperature")
                    .payload(getTemperature())
                    .send();

            TimeUnit.MILLISECONDS.sleep(500);
        }
    }
        //4. Simulate Temperature and Brightness sensor data
    private static byte[] getBrightness() {
        // simulate a brightness sensor with values between 1000lux and 10000lux
        final int brightness =
ThreadLocalRandom.current().nextInt(1_000, 10_000);
        return (brightness + "lux").getBytes(StandardCharsets.UTF_8);
    }

    private static byte[] getTemperature() {
        // simulate a temperature sensor with values between 20°C and 30°C
        final int temperature = ThreadLocalRandom.current().nextInt(20, 30);
        return (temperature + "°C").getBytes(StandardCharsets.UTF_8);
    }
}
```

Let's dissect the code snippet provided above to understand its functionality:

1. **Creating the MQTT Client:** The code initializes the MQTT client, ensuring a unique identifier is used. An automatic reconnect feature is also enabled to handle potential instability in the sensor's internet connection.
2. **Establishing Connection to "<your_host>":** The client connects to the specified host. Notably, a "will" message is set, allowing the broker to automatically publish a "sensor gone" notification if the sensor loses its connection.
3. **Periodic Publication of Simulated Sensor Data:** The code periodically publishes simulated brightness and temperature data using the methods getBrightness() and getTemperature() methods, ensuring a steady stream of information for further processing.

With this code snippet, you will have created an MQTT client, established a connection to the broker, and started regularly transmitting the Brightness and Temperature sensor data.

Now, let's move on to the next step in our implementation process:

Step 2 - Implementing the Subscribing Client

In this next step, we focus on creating the subscribing client responsible for consuming the values published on the topics home/temperature and home/brightness.

Implementing the subscribing client enables the reception of sensor data transmitted via MQTT. This functionality allows you to process and utilize the received information for various applications efficiently.

```java
public class ControlCenter {

    public static void main(String[] args) {
        final String host = "<your_host>"; // use your host-name, it should look like
'<alphanumeric>.s2.eu.hivemq.cloud'
        final String username = "<your_username>";  // your credentials
        final String password = "<your_password>";

        // 1. create the client
        final Mqtt5Client client = Mqtt5Client.builder()
                .identifier("controlcenter-" + getMacAddress()) // use a unique identifier
                .serverHost(host)
                .automaticReconnectWithDefaultConfig() // the client automatically
reconnects
                .serverPort(8883) // this is the port of your cluster, for mqtt it is the
default port 8883
                .sslWithDefaultConfig() // establish a secured connection to HiveMQ Cloud
using TLS
                .build();

        // 2. connect the client
        client.toBlocking().connectWith()
                .simpleAuth() // using authentication, which is required for a secure
connection
                .username(username) // use the username and password you just created
                .password(password.getBytes(StandardCharsets.UTF_8))
                .applySimpleAuth()
                .cleanStart(false)
                .sessionExpiryInterval(TimeUnit.HOURS.toSeconds(1)) // buffer messages
                .send();

        // 3. subscribe and consume messages
        client.toAsync().subscribeWith()
                .topicFilter("home/#")
```

```
            .callback(publish -> {
                System.out.println("Received message on topic " + publish.getTopic()
+ ": " +
                        new String(publish.getPayloadAsBytes(), StandardCharsets.
UTF_8));
            })
            .send();
    }
}
```

The code snippet above performs the following actions:

1. Creates an MQTT client instance, similar to the sensor client, with the client ID prefixed as controlcenter-.
2. Establishes a connection between the client and the specified host in <your_host>. To ensure message buffering when the control center is offline, a session expiry interval of 1 hour is set.
3. The client Subscribes to all topics starting with home using the multi-level wildcard # in the topic filter.
4. Any incoming messages with their corresponding topic and payload are printed. If the sensor loses connection, the topic home/will and the payload "sensor gone" are printed.

The code in this example was tested on Java 11. You can also find a publicly hosted example in our **GitHub repository**, along with an example tailored for our public broker.

By implementing both the sensor data publishing client and the subscribing client, you can establish a seamless MQTT communication system where sensor data is published and consumed by the control center, enabling efficient monitoring and control of devices.

## Chapter 3: MQTT Topics, Subscriptions, QoS, and Persistent Messaging
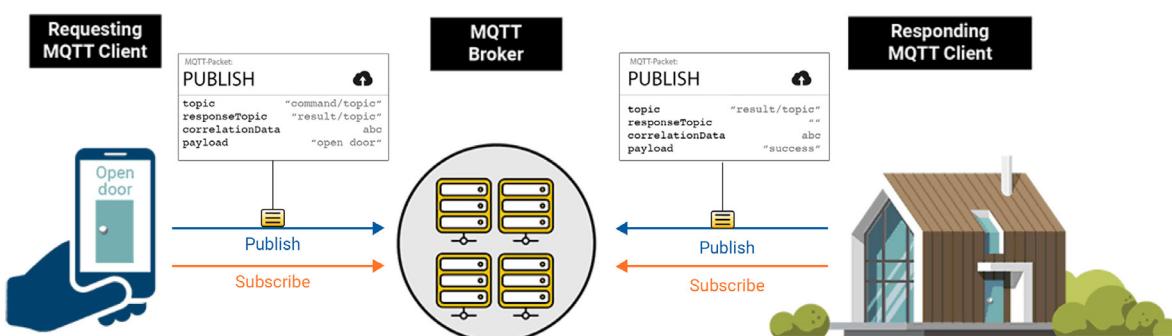
MQTT's Messaging Model: Topics and Subscriptions

MQTT's messaging model is based on topics and subscriptions. Topics are strings that messages are published to and subscribed to. Topics are hierarchical and can contain multiple levels separated by slashes, like a file path as shown below.

`myhome/kitchen/smartdishwasher`

Subscriptions are used to specify which topics a client is interested in receiving messages from.

When a client subscribes to a topic, it is essentially telling the broker that it is interested in receiving messages published to that topic. The broker then keeps track of the subscription and forwards any messages published to that topic to the subscribed client.

Example: Smart door opening with a mobile device using MQTT

It's important to note that a client can subscribe to multiple topics at once, and a topic can have multiple subscribers. This allows for a flexible and scalable messaging system.

In addition to topics and subscriptions, MQTT also supports wildcards, which can be used to subscribe to multiple topics that match a certain pattern. The two types of wildcards are the single-level wildcard (+), which matches a single level in a topic, and the multi-level wildcard (#), which matches all levels after the specified level in a topic.

Overall, MQTT's messaging model provides a flexible and scalable way to publish and subscribe to messages using topics and subscriptions. The use of wildcards adds an additional layer of flexibility, allowing for subscriptions to multiple related topics using a single subscription. Understanding MQTT's messaging model is crucial, but equally important is the quality of service (QoS) level that you choose to ensure reliable message delivery.

Understanding MQTT Quality of Service (QoS) Levels for IoT Applications

MQTT supports three levels of Quality of Service (QoS): QoS 0, QoS 1, and QoS 2. Here is the breakdown of each level:

- **QoS 0**: This level provides "at most once" delivery, where messages are sent without confirmation and may be lost. This is the lowest level of QoS and is typically used in situations where message loss is acceptable or where the message is not critical. For example, QoS 0 might be appropriate for sending sensor data where occasional data loss would not significantly impact the overall results.
- **QoS 1**: This level provides "at least once" delivery, where messages are confirmed and re-sent if necessary. With QoS 1, the publisher sends the message to the broker and waits for confirmation before proceeding. If the broker does not respond within a set time, the publisher re-sends the message. This level of QoS is typically used in situations where message loss is unacceptable, but message duplication is tolerable. For example, QoS 1 might be appropriate for sending command messages to devices, where a missed command could have serious

consequences, but duplicated commands would not.
- **QoS 2**: This level provides "exactly once" delivery, where messages are confirmed and re-sent until they are received exactly once by the subscriber. QoS 2 is the highest level of QoS and is typically used in situations where message loss or duplication is completely unacceptable. With QoS 2, the publisher and broker engage in a two-step confirmation process, where the broker stores the message until it has been received and acknowledged by the subscriber. This level of QoS is typically used for critical messages such as financial transactions or emergency alerts.

It's important to note that higher QoS levels typically require more resources and can result in increased latency and network traffic. As a result, it's important to choose the appropriate QoS level based on the specific needs of your application.

In addition to the three levels of Quality of Service, MQTT also supports message persistence, which ensures that messages are not lost in the event of a network or server failure.

Understanding MQTT Message Persistence for Reliable IoT Communication

> Message persistence is an important feature in MQTT. It ensures messages are not lost in the event of a network or server failure. In MQTT, message persistence is achieved by storing messages on the server until they are delivered to the subscriber.

MQTT provides three types of message persistence options:

- **Non-persistent**: This is the default option in MQTT. In this mode, messages are not stored on the server and are lost if the server or network fails. This mode is suitable for situations where messages are not critical and can be easily regenerated.
- **Queued persistent**: In this mode, messages are stored on the server until they are delivered to the subscriber. If the subscriber is not available, messages are queued

until the subscriber reconnects. Queued persistence is useful when the subscriber is not always connected to the network, or if the subscriber needs to receive all messages, even if they are sent when the subscriber is offline.

- **Persistent with acknowledgment**: This mode provides the highest level of message persistence. In this mode, messages are stored on the server until they are delivered to the subscriber, and the subscriber must acknowledge receipt of the message. If the subscriber does not acknowledge receipt, the message is re-sent until the subscriber acknowledges receipt. This mode is useful when it is critical to ensure that messages are received and processed by the subscriber.
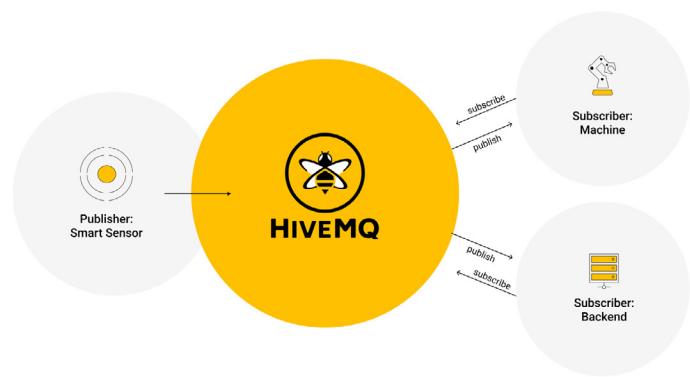
To configure message persistence in MQTT, the broker software used for handling MQTT connections must support the chosen persistence option. The configuration can be done through the broker's configuration files or through its web interface.

It is important to note that message persistence comes with a trade-off in terms of performance and storage. The more persistent the messages, the more storage and processing resources are required by the broker. Therefore, it is important to choose the appropriate persistence level based on the specific requirements of the application.

Chapter 4: MQTT Publish/Subscribe Architecture (Pub/Sub)

MQTT Pub/Sub architecture, also known as pub/sub, is a messaging pattern in software architecture. It enables communication between different components or systems in a decoupled manner. In the Pub/Sub architecture, there are publishers that generate messages and subscribers that receive those messages. However, publish-subscribe is a broader concept that can be implemented using various protocols or technologies.

MQTT is one such specific messaging protocol that follows the publish-subscribe architecture. MQTT uses a broker-based model where clients connect to a broker, and messages are published to topics. Subscribers can then subscribe to specific topics and receive the published messages.

Example of MQTT Publish / Subscribe Architecture

Decoupling Features of MQTT Pub/Sub

The Pub/Sub architecture offers a unique alternative to traditional client-server (request-response) models. In the request-response approach, the client directly communicates with the server endpoint, creating a bottleneck that slows down performance. On the other hand, the pub/sub model decouples the publisher of the message from the subscribers. The publisher and subscriber are unaware that the other exists. As a third component, a broker handles the connection between them. This decoupling produces a faster and more efficient communication process.

By eliminating the need for direct communication between publishers and subscribers, pub/sub architecture removes the exchange of IP addresses and ports. It also provides decoupling, allowing operations on both components to continue communication uninterrupted during publishing or receiving.

The pub/sub features three dimensions of decoupling for optimal efficiency:

- **Space decoupling**: Publisher and subscriber do not need to know each other (for example, no exchange of IP address and port).
- **Time decoupling**: Publisher and subscriber do not need to run at the same time.
- **Synchronization decoupling**: Operations on both components do not need to be interrupted during publishing or receiving.

> **Pub/Sub Decoupling in MQTT Protocol**
>
> MQTT decouples the publisher and subscriber spatially, meaning they only need to know the broker's hostname/IP and port to publish or receive messages. Additionally, MQTT decouples by time, allowing the broker to store messages for clients that are not online. Two conditions must be met to store messages: the client must have connected with a persistent session and subscribed to a topic with a Quality of Service greater than 0.

One of the most significant advantages of Pub/Sub software architecture is its ability to filter all incoming messages and distribute them to subscribers correctly, eliminating the need for the publisher and subscriber to know one another's existence.
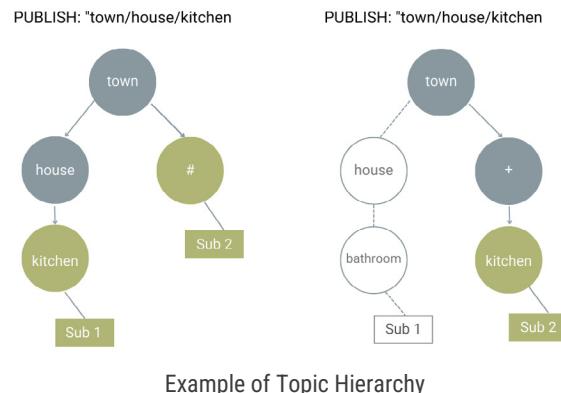
MQTT Pub/Sub Message Filtering Feature

Message filtering is a crucial aspect of the pub/sub architecture as it ensures subscribers only receive messages they are interested in. The pub/sub broker offers several filtering options, including subject-based filtering, content-based filtering, and type-based filtering.

Option 1: Subject-based Filtering of Pub/Sub Architecture

This is the most common filtering option, where the broker filters the messages based on the topic or subject. The subscribing clients indicate their interest by subscribing to specific topics, and the broker routes the messages to the appropriate subscribers based on the topic hierarchy. The topic structure is hierarchical, with levels separated by a forward slash (/), allowing subscribers to receive messages that match a specific topic level or a topic hierarchy. Here's an example of topic hierarchy.

Example of Topic Hierarchy

Benefits of Subject-based Filtering in Pub/Sub:

- Simple and easy to use
- Flexible, allowing for a hierarchical topic structure
- Efficient, as it only forwards messages to subscribers interested in a particular topic

Drawbacks of Subject-based filtering in Pub/Sub:

- Publishers and subscribers need to agree on the topic hierarchy beforehand
- Limited to filtering messages based on topic hierarchy only

Use Case of Subject-based filtering in Pub/Sub:

Subject-based filtering is best suited for use cases where messages are organized into topics and subscribers are interested in a particular subset of those topics.

For example, in a smart home system, a subscriber may be interested in receiving updates about the temperature of a specific room. The subscriber would subscribe to a topic like `"smart-home/living-room/temperature,"` and the broker would only send messages that match this topic to the subscriber.

> **Message Filtering in MQTT**
>
> MQTT uses subject-based filtering of messages. Every message contains a topic (subject) that the broker can use to determine whether a subscribing client gets the message or not. To handle the challenges of a pub/sub system, MQTT has three Quality of Service (QoS) levels. You can easily specify that a message gets successfully delivered from the client to the broker or from the broker to a client. However, there is the chance that nobody subscribes to the particular topic. If this is a problem, the broker must know how to handle the situation. For example, the **HiveMQ MQTT Broker** has an extension system that can resolve such cases. You can have the broker take action or simply log every message into a database for historical analyses. To keep the hierarchical topic tree flexible, it is important to design the topic tree very carefully and leave room for future use cases. If you follow these strategies, MQTT is perfect for production setups.

Option 2: Content-based Filtering of Pub/Sub Architecture

In this type of filtering, the broker filters messages based on their content, which is specified using a filter expression. Subscribers indicate their interest by subscribing to a specific filter expression, and the broker routes the messages to the appropriate subscribers based on the content of the messages.

> How to bring Content-based Filtering in MQTT? Even though MQTT uses subject-based filtering of messages, you can also set up content-based filtering by using the HiveMQ **MQTT Broker** and our custom **extension system.**

Benefits of Content-based Filtering in Pub/Sub:

- Provides more granular control over which messages are received

- Allows for filtering based on message content rather than just topic hierarchy
- Flexible, allowing for complex filter expressions

Drawbacks of Content-based Filtering in Pub/Sub:

- Can be more complex to use and set up than subject-based filtering
- Requires publishers to include additional metadata in the message to enable filtering
- Performance may suffer when processing large numbers of filter expressions

Use Case of Content-based Filtering in Pub/Sub:

Content-based filtering is best suited for use cases where messages are not organized into topics, and subscribers are interested in a specific subset of messages based on their content.

For example, in a logistics application, a subscriber may be interested in receiving messages only about packages with a specific tracking number. The subscriber would subscribe to a filter expression like "tracking-number = '123456'," and the broker would only send messages that match this expression to the subscriber.

Option 3: Type-based Filtering of Pub/Sub Architecture

In type-based filtering, the broker filters messages based on their type or class. This type of filtering is useful when working with object-oriented languages where messages are represented as objects. Subscribers indicate their interest by subscribing to a specific message type or class, and the broker routes messages to the appropriate subscribers based on the message type.

Benefits of Type-based Filtering in Pub/Sub:

- Allows for filtering based on message type, regardless of the topic or content
- Simple to use, especially when working with object-oriented languages
- Offers a high degree of flexibility and extensibility

Drawbacks of Type-based Filtering in Pub/Sub:

- Limited to filtering based on message type only
- Publishers and subscribers need to agree on the message type hierarchy beforehand

Use Case of Type-based Filtering in Pub/Sub:

Type-based filtering is best suited for use cases where messages are organized into a class hierarchy, and subscribers are interested in a specific type or subset of messages based on their class.

For example, in a financial application, a subscriber may be interested in receiving messages only about stock prices. The subscriber would subscribe to a message type like "stock-price," and the broker would only send messages of this type to the subscriber.

These filtering options provide flexibility and granularity in deciding which messages are sent to which subscribers. Depending on the use case, you can use one or more of these filtering options to ensure that subscribers receive only the messages they are interested in.

However, it's important to note that the pub/sub model may not be suitable for all use cases, and there are challenges to consider, such as ensuring that both the publisher and subscriber know which topics to use for subject-based filtering and dealing with instances where no subscriber reads a particular message. You need to be aware of how the published data is structured beforehand.

> For subject-based filtering, both publisher and subscriber need to know which topics to use. Also, with message delivery, the publisher can't assume somebody is listening to the messages that are sent. This is an issue because, in publish-subscribe model, the publisher sends messages to the broker without knowing who the subscribers are or whether they are currently connected to the broker. The broker is responsible for delivering messages to all connected subscribers who subscribe to the appropriate topic. However, if there are no subscribers currently

> connected to the broker who have subscribed to the topic of a particular message, that message will not be delivered to anyone. Therefore, it is vital for publishers to keep in mind that message delivery is not guaranteed and to design their systems accordingly.

MQTT Pub/Sub's Scalability Feature

Scalability is one of the significant benefits of using the Pub/Sub architecture. The traditional client-server model can limit scalability, particularly when dealing with large numbers of clients. However, with the pub/sub model, the broker can process messages in an event-driven way, enabling highly parallelized operations. This means that the system can handle a greater number of concurrent connections without sacrificing performance.

In addition to event-driven processing, message caching and intelligent message routing also contribute to improved scalability in Pub/Sub. By caching messages, the broker can quickly retrieve and deliver them to subscribers without additional processing. Intelligent routing, on the other hand, ensures that messages are delivered only to the subscribers that need them, reducing unnecessary network traffic and further improving scalability.

> As MQTT follows the pub/sub architecture, scalability comes naturally to this protocol, making it ideal for several IoT use cases. Despite its advantages, scaling up to millions of connections can still pose a challenge for Pub/Sub. In such cases, clustered broker nodes can be used to distribute the load across multiple servers, while load balancers can ensure that the traffic is evenly distributed. Check out how **HiveMQ Broker can scale to 200 million concurrent connections** using this method.

Now that we have covered the basic concepts of Publish/Subscribe architecture, let's look at the benefits of using it for IoT communication.

What Are the Key Benefits of MQTT Pub/Sub Architecture in IoT and IIoT?

The pub/sub model offers several benefits, making it a popular choice for various applications. Here are some of the key advantages of using pub/sub architecture:

Improved scalability: The pub/sub architecture is highly scalable, making it suitable for applications that handle many clients and messages. The broker acts as a central hub for all messages, allowing it to handle many clients without compromising performance.

Increased fault tolerance: The decoupled nature of pub/sub architecture also provides improved fault tolerance. In a traditional client-server model, all connected clients lose their connection if the server goes down. In contrast, in pub/sub, the broker can store messages until the client reconnects, ensuring no messages are lost.

Flexibility: The pub/sub architecture is flexible and can be used in a variety of applications, ranging from low-bandwidth, high-latency networks to high-speed, low-latency networks. The MQTT protocol, which is based on pub/sub architecture, supports various quality-of-service levels, providing the flexibility to choose the appropriate level for your application.

Common Challenges in Pub/Sub Architecture and How to Overcome Them

While pub/sub architecture offers several benefits, such as scalability, flexibility, and decoupling of components, it also presents some challenges that must be addressed to ensure a successful implementation. Below are some of the most common challenges of using pub/sub and solutions to overcome them:

1. **Message Delivery:** One challenge of using pub/sub is ensuring that messages are delivered to subscribers. In some instances, no subscribers may be available to receive a particular topic, resulting in the message being lost. To overcome this, MQTT provides quality of service (QoS) levels.

2. **Message Filtering**: Another challenge of pub/sub is filtering messages effectively so that each subscriber receives only the messages of interest. As discussed earlier, pub/sub provides three filtering options: subject-based, content-based, and type-based filtering. Each option has its benefits and drawbacks, and the choice of filtering method will depend on the use case. MQTT uses subject-based filtering of messages and every message contains a topic that the broker uses to determine whether a subscribing client receives the message or not.

3. **Security**: Security is a crucial aspect of any messaging system, and pub/sub is no exception. MQTT allows for several security options, such as user authentication, access control, and message encryption, to protect the system from unauthorized access and data breaches.

4. **Scalability**: Pub/Sub architecture must be designed with scalability in mind, as the number of subscribers can grow exponentially in a large-scale system. MQTT provides features such as multiple brokers, clustering, and load balancing to ensure that the system can handle a large number of subscribers and messages.

5. **Message Ordering**: In a pub/sub system, message ordering can be challenging to maintain. As messages are sent asynchronously, it's difficult to ensure that subscribers receive messages in the correct order. However, MQTT provides QoS levels that ensures messages are successfully delivered from the client to the broker or from the broker to a client.

   Because MQTT works asynchronously, tasks are not blocked while waiting for or publishing a message. Most client libraries are based on callbacks or a similar model, making the flow of messages usually asynchronous. In certain use cases, synchronization is desirable and possible, and some libraries have synchronous APIs to wait for a specific message.

6. **Real-time Constraints**: In some use cases, real-time constraints are critical, and pub/sub architecture may not be the best choice. For example, a request/response architecture may be a better option if low latency is essential.

You can address the challenges of using pub/sub through careful design and implementation. Developers can build scalable, secure, and efficient messaging systems by

understanding these challenges and utilizing MQTT's features effectively.

MQTT vs. Message Queues

There is a lot of confusion about the name MQTT and whether the protocol is implemented as a message queue or not. We will try to shed some light on the topic and explain the differences. We mentioned earlier that MQTT refers to the MQSeries product from IBM and has nothing to do with "message queue". Regardless of where the name comes from, it's useful to understand the differences between MQTT and a traditional message queue:

A message queue stores messages until they are consumed. When you use a message queue, each incoming message is stored in the queue until it is picked up by a client (often called a consumer). If no client picks up the message, the message remains stuck in the queue and waits to be consumed. In a message queue, it is not possible for a message not to be processed by any client, as it is in MQTT if nobody subscribes to a topic.

A message is only consumed by one client. Another big difference is that in a traditional message queue a message can be processed by one consumer only. The load is distributed between all consumers for a queue. In MQTT the behavior is quite the opposite: every subscriber that subscribes to the topic gets the message.

Queues are named and must be created explicitly. A queue is far more rigid than a topic. Before a queue can be used, the queue must be created explicitly with a separate command. Only after the queue is named and created is it possible to publish or consume messages. In contrast, MQTT topics are extremely flexible and can be created on the fly. If you can think of any other differences that we overlooked, we would love to hear from you in the comments.

To summarize, the publish/subscribe (pub/sub) architecture provides a flexible and scalable way of building distributed systems that can handle many connected clients. MQTT's lightweight and efficient pub/sub messaging characteristics

have helped it gain widespread adoption in IoT, mobile, and other distributed applications.

Using MQTT, architecture engineers, and companies can build systems reliably and efficiently communicate data in various real-world scenarios. With its decoupling by space and time, asynchronous messaging, subject-based filtering, and Quality of Service (QoS) levels, MQTT provides a robust set of features to help developers overcome the challenges of building distributed systems. Overall, the pub/sub architecture and MQTT protocol are valuable tools for developers who want to build efficient and scalable distributed systems.
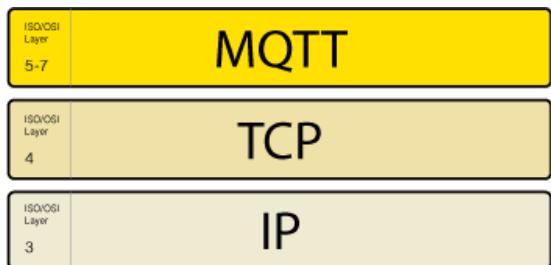
## Chapter 5: MQTT Client and MQTT Broker Connection Establishment

The two main components of the MQTT protocol are the client and the broker. An MQTT client can be any device that runs an MQTT library and connects to an MQTT broker over a network. The publisher and subscriber labels refer to whether the client is publishing or subscribed to receive messages. The MQTT broker, on the other hand, is responsible for receiving all messages, filtering them, and sending them to subscribed clients. The broker also handles client authentication and authorization and holds all clients' session data with persistent sessions. Let's dive deeper into foundational MQTT components.
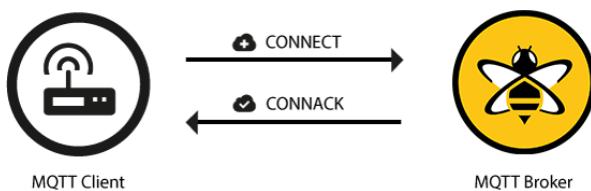
One of the key features of the MQTT protocol is its efficient and lightweight approach to exchanging messages between IoT devices. The foundation of this communication is the MQTT connection, which enables devices to securely and reliably exchange data with the MQTT broker. In this section, we will explore the process of establishing an MQTT connection and the different parameters involved. By understanding how MQTT connections work, you can optimize your IoT deployment for better performance, security, and scalability.

The MQTT protocol is based on TCP/IP, meaning the client and the broker must have a TCP/IP stack.

MQTT connections are always between one client and one broker, and clients never connect directly to other clients. To initiate a connection, the client sends a CONNECT message to the broker, which responds with a CONNACK message and a status code. Once the connection is established, the broker keeps it open until the client sends a disconnect command or the connection breaks.

This section will explore the MQTT connection through a NAT and how the MQTT client initiates a connection by sending a CONNECT message to the broker. We will delve into the details of the MQTT CONNECT command message and focus on some essential options, including ClientId, Clean Session, Username/Password, Will Message, and Keep Alive. Moreover, we will discuss the broker's response to a CONNECT message, which is a CONNACK message containing two data entries: the session present flag and a connect return code.

MQTT Connection Through a NAT

In many cases, MQTT clients live behind routers that use network address translation (NAT) to convert private network addresses (such as 192.168.x.x or 10.0.x.x) to public-facing addresses. As mentioned, the MQTT client starts the connection by sending a CONNECT message to the broker. Since the broker has a public address and maintains the connection open to enable bidirectional sending and receiving of messages (after the initial CONNECT), MQTT clients located behind NAT routers will have no difficulties.

For those unaware, NAT is a common networking technology that routers use to allow devices on a private network to access the internet through a single public IP address. NAT works by translating the IP addresses of devices on the private network to the public IP address of the router and vice versa.

> In the case of MQTT, clients behind a NAT router can still communicate with the MQTT broker because the broker has a public IP address and can connect with the client through the NAT. However, some potential issues can arise with NAT, such as configuring port forwarding or opening firewall ports to allow incoming traffic to reach the MQTT broker. Additionally, some NAT implementations may have limitations on the number of concurrent connections that can be established, which could affect the scalability of the MQTT system.

Now that we understand how MQTT clients behind a NAT establish a connection with the broker, let's take a closer look at the MQTT CONNECT command message and its contents.

How Does MQTT Client Initiate a Connection with the CONNECT Message?

Now let's examine the **MQTT CONNECT** command message, which the client sends to the broker to initiate a connection. If this message is malformed or too much time elapses between opening a network socket and sending the CONNECT message, the broker terminates the connection to deter malicious clients that can slow the broker down. In addition to other details specified in the MQTT 3.1.1 specification, a good-natured MQTT 3 client sends the following content.

Let's focus on some of the essential options:

While users of an MQTT library may find some of the information in a CONNECT message useful, certain details may be more relevant to implementers of the library. For a complete understanding of all the information contained in the message, refer to the **MQTT 3.1.1 specification**.

Let's look at some of the elements the MQTT CONNECT packet contains, such as ClientId, Clean Session, Username/Password, Will Message, Keep Alive, etc.

What is ClientId in CONNECT MQTT Packet?

The ClientId is a unique identifier that distinguishes each MQTT client connecting to a broker and enables the broker to keep track of the client's current state. To ensure uniqueness, the ClientId should be specific to each client and broker. MQTT 3.1.1 allows for an empty ClientId if no state needs to be maintained by the broker. However, this connection must have the clean session flag set to true, or the broker will reject the connection.

What is CleanSession in CONNECT MQTT Packet?

The CleanSession flag indicates whether the client wants to establish a persistent session with the broker. When CleanSession is set to false (CleanSession = false), considered a persistent session, the broker stores all subscriptions for the client and all missed messages for the client that subscribed with a **Quality of Service (QoS) level 1 or 2**. In contrast, when CleanSession is set to true (CleanSession = true), the broker doesn't retain any information for the client and discards any previous state from any persistent session.

What is Username/Password in CONNECT MQTT Packet?

MQTT provides the option to include a username and password for client authentication and authorization. However, it's important to note that sending this information in plain text poses a security risk. To mitigate this risk, we highly recommend using encryption or hashing (such as through TLS) to protect the credentials. We also recommend using a secure transport layer when transmitting sensitive data.

Alternatively, some brokers like HiveMQ offer SSL certificate authentication, eliminating the need for username and password credentials altogether. Taking these precautions ensures that your MQTT communication remains secure and protected from potential security threats.

What is Will Message in CONNECT MQTT Packet?

The MQTT Last Will and Testament (LWT) feature includes a last will message that notifies other clients when a client disconnects unexpectedly. This message can be specified by the client within the CONNECT message as an MQTT message and topic. When the client disconnects abruptly, the broker sends the LWT message on the client's behalf. Learn more about **MQTT Last Will and Testament** in Part 9 of this series.

What is Keep Alive in CONNECT MQTT Packet?

The MQTT Keep Alive feature allows the client to specify a time interval in seconds and communicate it to the broker when establishing a connection. This interval determines the longest period the broker and client can communicate without sending a message. To ensure the connection remains active, the client sends regular PING Request messages, and the broker responds with a PING response. This method allows both sides to determine if the other is still available. Learn more about **MQTT Keep Alive** functionality in Part 10 of this series.

When connecting to an MQTT broker from an MQTT 3.1.1 client, the Keep Alive interval is essential. However, some MQTT libraries have additional configuration options, such as the way queued messages are stored in a specific implementation.

MQTT Broker Response With a CONNACK Message

When a broker receives a CONNECT message, it is obligated to

MQTT-Packet:

CONNACK

contains:	Example
sessionPresent	true
returnCode	0

respond with a CONNACK message.
The CONNACK message contains two data entries:

* The session present flag
* A connect return code

What is sessionPresent Flag in a CONNACK Message?

The sessionPresent flag informs the client whether a previous session is still available on the broker. If the client has requested a clean session, the flag will always be false, indicating there is no previous session.

However, if the client requests to resume a previous session, the flag will be true if the broker still has stored session information. This flag helps clients determine whether they need to re-subscribe to topics or if the broker still has the subscriptions from the previous session.

What is returnCode Flag in a CONNACK Message?

The returnCode is a status code that informs the client about the success or failure of the connection attempt. This code can indicate various types of errors, such as invalid credentials or unsupported protocol versions.

Here are the returnCodes at a glance:

Return Code	Return Code Response
0	Connection accepted
1	Connection refused, unacceptable protocol version
2	Connection refused, identifier rejected
3	Connection refused, server unavailable
4	Connection refused, bad user name or password
5	Connection refused, not authorized

It's essential to pay attention to the connect returnCode, as it can help diagnose connection issues. For example, if the returnCode indicates an authentication failure, the client can attempt to reconnect with the correct credentials. Understanding the sessionPresent flag and connect returnCode is crucial for successful MQTT connections.

To summarize, understanding the roles of MQTT clients and the broker and the connection establishment process is essential for anyone interested in working with the MQTT protocol. MQTT client libraries make adding MQTT support to applications and devices easy without implementing the protocol from scratch. MQTT brokers are responsible for receiving, filtering, and sending messages to subscribed clients and handling client authentication and authorization. With this knowledge, you can build scalable and efficient IoT systems using MQTT.

Chapter 6: MQTT Publish, MQTT Subscribe & Unsubscribe

MQTT PUBLISH Message

In MQTT, a client can publish messages immediately when it connects to a broker. The messages are filtered based on topics, and each message must contain a topic that the broker can use to forward the message to interested clients. The payload of each message includes the data to transmit in byte format, and the sending client can choose to send any type of data, including text, numbers, images, binary data, and even full-fledged XML or JSON.

Example of MQTT Payload Format

MQTT is data-agnostic, which means the payload can be structured according to the specific use case of the client. The payload is the message's main content and is what the clients subscribe to, receive, and process.

A PUBLISH message in MQTT has several attributes that determine its behavior including the packet identifier, topic name, quality of service, retain flag, payload, and DUP flag. Let's take a look at each.

What is MQTT PacketId or Packet Identifier?

The Packet Identifier (PacketId) is an essential attribute in MQTT. It is used to identify the specific message and to ensure that messages are delivered in the order they were sent, particularly when QoS levels greater than zero are used. The Packet ID is assigned by the client and is included in the PUBLISH, PUBREL, PUBREC, and PUBCOMP messages. When the broker receives a PUBLISH message, it assigns a Packet ID to the message and sends a PUBACK message to the client containing the Packet ID of the PUBLISH message. The client uses the PUBACK message to confirm that the broker has received the message.

In the MQTT protocol, messages related to publishing and acknowledgment of messages are divided into several stages:

1. Publish (PUBLISH): This is the first stage of the process and involves an MQTT client publishing a message to the broker. The message contains a topic and a payload.
2. Publish Received (PUBREC): After receiving the PUBLISH message, the broker sends a PUBREC message to acknowledge that it has received the message. This is the second stage of the process.
3. Publish Release (PUBREL): Once the client receives the PUBREC message, it sends a PUBREL message to release the broker from the responsibility of keeping the message in memory. This is the third stage.

4. Publish Complete (PUBCOMP): The broker finally sends a PUBCOMP message to confirm that it has successfully received and processed the message. This is the fourth and final stage of the process.

These four messages are part of the MQTT protocol's Quality of Service (QoS) mechanisms, which ensure reliable message delivery. The QoS level determines the number of messages exchanged between the client and the broker.

It's worth noting that the packet identifier uniquely identifies a message as it flows between the client and broker. The packet identifier is only relevant for QoS levels greater than zero. This holds true not only for PUBLISH, but for SUBSCRIBE, UNSUBSCRIBE and CONNECT messages.

The client library and/or the broker is responsible for setting this internal MQTT identifier. When a QoS level greater than zero is used, the client must wait for a PUBACK or PUBREC message from the broker before it can send the next message. The client should also keep track of the Packet IDs it has sent and received to ensure that messages are not lost or duplicated. Overall, the Packet ID is essential to MQTT's reliability mechanism and helps ensure that messages are delivered correctly and efficiently.

What is MQTT Topic Name?

MQTT uses the topic name as a fundamental concept. It structures this name hierarchically using forward slashes as delimiters and creates a simple string. It's similar to a URL path but without the protocol and domain components. MQTT topics are used to label messages and provide a way for clients to subscribe to specific messages.

For example, a device that measures temperature might publish its readings to the topic "sensors/temperature/livingroom". A client interested in these readings can subscribe to this topic and receive updates as they're published. MQTT provides two types of wildcards to use with topic subscriptions:

- "+" (plus sign) is used to match a single level in the hierarchy. For example, a subscription to "sensors/+/livingroom" would match "sensors/temperature/livingroom" and "sensors/humidity/livingroom", but not "sensors/temperature/kitchen".
- "#" (hash sign) is used to match multiple levels in the hierarchy. For example, a subscription to "sensors/#" would match "sensors/temperature/livingroom", "sensors/humidity/kitchen", and "sensors/power/meter1".

Subscribing to a large number of topics can have a significant impact on broker performance. This is because every message that is published to a topic subscribed by a client must be delivered to that client. If many clients subscribe to many topics, this can quickly become a heavy burden on the broker.

Using wildcards to subscribe to multiple topics with a single subscription can also impact performance. When a client subscribes to a topic with a wildcard, the broker must evaluate every message published to a matching topic and determine whether to forward it to the client. If the number of matching topics is large, this can strain the broker's resources.

To avoid performance issues, it's important to use topic subscriptions efficiently. One approach is to use more specific topic filters whenever possible, rather than relying on wildcards. Another approach is to use shared subscriptions, which allow multiple clients to share a single subscription to a topic. This can help reduce the number of subscriptions and messages that the broker must handle. Finally, monitoring the broker's performance and adjusting its configuration as necessary is important to ensure optimal performance.

What is Quality of Service (QoS) in MQTT?

We touched upon Quality of Service Level (QoS) of an MQTT message earlier. To refresh, QoS is indicated by a number that ranges from 0 to 2. Each level provides a different level of reliability and assurance for message delivery.

- QoS 0 (at most once): This level provides no guarantee that a message will be delivered. The message is sent once, and if it is lost or not received by the recipient, it will not be resent.
- QoS 1 (at least once): This level ensures that a message is delivered at least once, but it may be delivered multiple times in the case of network issues or failures.
- QoS 2 (exactly once): This level provides the highest level of assurance for message delivery. The message is guaranteed to be delivered exactly once, but this level requires more communication between the sender and receiver, which can increase latency and network traffic.

Choosing the appropriate QoS level depends on the specific use case. For example, QoS 0 might be suitable for non-critical data, while QoS 2 might be necessary for critical data requiring high-reliability levels.

It's important to note that the QoS level can impact the performance of the broker and network, so it's recommended to use the appropriate level for the specific use case.

What is MQTT Retain Flag?

The Retained Flag is an important feature that determines whether the broker saves a message as the last known good value for a specified topic. When the retained flag is set to true, the broker will save the most recent message that matches the specified topic, regardless of whether there are any subscribed clients.

When a new client subscribes to a topic with a retained message, the broker sends the last retained message (on that topic) to the client. This allows clients to receive the most recent and relevant information even if they have not subscribed to that topic before.

It's important to note that, like many of the other elements, the use of retained messages can also impact the broker's performance, especially if there are many retained messages. Additionally, if a retained message is updated frequently, it can result in increased network traffic and potentially affect the network's performance.

What is MQTT Payload?

The payload is the actual content of the message and can contain any kind of data. MQTT is data-agnostic, meaning it can handle different data types, including images, text in any encoding, encrypted data, and binary data. However, it's important to note that the payload size can impact network performance and memory usage on the client and broker. Therefore, keeping payloads as small as possible is recommended, especially when publishing messages with a high frequency.

What is MQTT DUP Flag?

The MQTT DUP Flag indicates that a message is a duplicate and has been resent because the intended recipient (client or broker) did not acknowledge the original message. It is only relevant for messages with QoS greater than 0. When a client or broker receives a message with the DUP flag set, it should ignore the message if it has already received a message with the same message ID. The client or broker should process the message normally if they have not previously received it.

The MQTT protocol (MQTT client library or broker) handles resend and duplicate mechanism automatically, but it's important to note that this can impact network performance and increase network traffic.
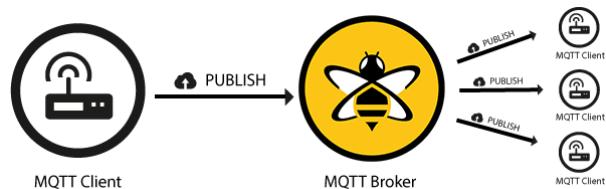
How do MQTT brokers handle messages from clients?

When a client publishes a message to an MQTT broker, the broker performs several tasks to ensure the message is delivered according to the QoS level specified by the client. Here's what happens:

1. Message reception: The broker reads the message sent by the client and verifies its syntax and format.
2. Acknowledgement: The broker sends an acknowledgment message to the client to confirm receipt of the message. The level of acknowledgment depends on the QoS level requested by the client.
3. Processing: The broker determines which clients have subscribed to the topic of the message and sends a copy of the message to each of them. The broker may also retain the message as the last known good value for that topic, depending on the value of the Retained flag.

4. Feedback: The publishing client receives a confirmation message from the broker indicating that the message was published successfully. However, the client does not receive feedback on how many subscribers received the message or whether anyone is interested in it.

The client that initially publishes the message is only concerned about delivering the PUBLISH message to the
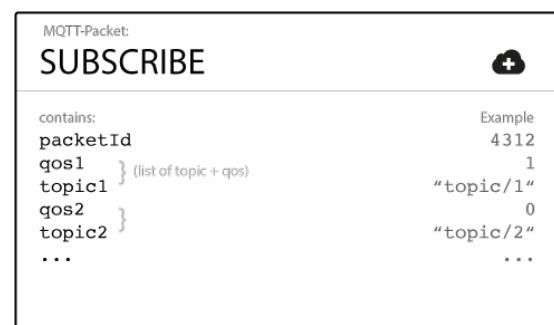
How MQTT PUBLISH works

broker. Once the broker receives the PUBLISH message, it is the responsibility of the broker to deliver the message to all subscribers. The publishing client does not get any feedback about whether anyone is interested in the published message or how many clients received the message from the broker.

How to Subscribe to MQTT Topics?

Publishing a message doesn't make sense if no one ever receives it. This is where subscribing comes into play. Once a client publishes a message to an MQTT broker, the message must be delivered to interested clients. Clients that want to receive messages on topics of interest send a **SUBSCRIBE** message to the broker. The SUBSCRIBE message is simple and contains a unique packet identifier and a list of subscriptions.

Example of MQTT SUBSCRIBE Packet

Packet Identifier: The packet identifier is unique and identifies a message as it flows between the client and broker. The client library or the broker is responsible for setting this internal MQTT identifier.

List of Subscriptions: A SUBSCRIBE message can contain multiple subscriptions for a client. Each subscription includes a topic and a QoS level. The topic in the SUBSCRIBE message can contain wildcards that make it possible to subscribe to a topic pattern instead of a specific topic. If there are overlapping subscriptions for one client, the broker delivers the message with the highest QoS level for that topic.

Overall, MQTT allows clients to subscribe to specific topics, receive messages published to those topics, and process the payloads according to their specific use case. The packet identifier and QoS level in the SUBSCRIBE message ensure that messages are delivered reliably and with the appropriate level of quality.

Once a client sends a SUBSCRIBE message with the list of desired topics and QoS levels to an MQTT broker, the broker responds with a SUBACK message that confirms the subscription and indicates the maximum QoS level that the broker will deliver. Let's look deeper into SUBACK.

What is MQTT Suback?

Once the client sends a SUBSCRIBE message to the broker with the topics and corresponding QoS levels, the broker acknowledges the subscription request by sending a **SUBACK** message to the client. The SUBACK message confirms the receipt of the SUBSCRIBE message and indicates whether the broker has accepted or rejected each subscription.

Example of MQTT SUBACK Packet

Packet Identifier: The SUBACK message includes the same packet identifier that the client included in the SUBSCRIBE message, which enables the client to match the acknowledgment to the original request.

Return Code: The SUBACK message also includes one return code for each topic/QoS-pair specified in the SUBSCRIBE message. The return codes are binary values that indicate whether the broker has granted or rejected the subscription request for each topic.

The return codes for QoS levels are as follows:

- QoS 0: This means the subscription request has been granted at QoS 0. The broker delivers the messages to the client as soon as they are available and with no quality guarantees.
- QoS 1: This means the subscription request has been granted at QoS 1. The broker delivers messages at least once, meaning that the broker sends the message to the client at least one time. The client sends a PUBACK message back to the broker after receiving the message, which acts as an acknowledgment.
- QoS 2: This means the subscription request has been granted at QoS 2. The broker delivers messages exactly once, which means that the broker guarantees that the message is delivered once and only once to the client. The client sends a PUBREC message back to the broker after receiving the message, which acts as an acknowledgment. The broker sends a PUBREL message to the client after receiving the PUBREC message, and the client sends a PUBCOMP message to the broker after receiving the PUBREL message.

If the broker rejects any of the subscriptions in the SUBSCRIBE message, the SUBACK message contains a failure return code for that specific topic. The reason for the failure could be that the client has insufficient permission to subscribe to the topic, the topic is malformed, or another reason.

The failure return code is represented by 0x80 and indicates that the subscription is not accepted by the broker. This can happen if the client does not have sufficient permission to
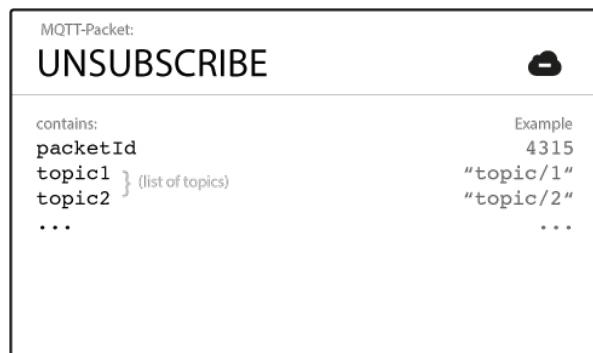
subscribe to the topic, the topic is malformed, or there is another issue with the subscription request. When a client receives a failure return code, it should retry the subscription with a different topic or QoS level or take appropriate action to address the issue with the subscription request.

The SUBACK message is an acknowledgment message from
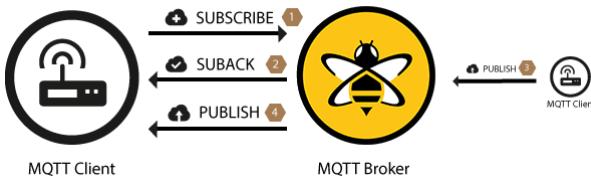
Return Code	Return Code Response
0	Success - Maximum QoS 0
1	Success - Maximum QoS 1
2	Success - Maximum QoS 2
128	Failure

How MQTT SUBSCRIBE, SUBACK, and PUBLISH work

the broker to the client to confirm the subscriptions that have been granted or rejected. The packet identifier enables the client to match the acknowledgment to the original request, while the return codes indicate the QoS levels at which the broker has granted the subscriptions.

After a client has subscribed to topics of interest and received messages published to those topics, it may eventually need to unsubscribe. Let's now explore the counterpart of the SUBSCRIBE message, the UNSUBSCRIBE message, and the corresponding UNSUBACK message that confirms the unsubscription.

How to Use Unsubscribe in MQTT to Revoke Subscriptions?

In MQTT, clients can unsubscribe from the topics they have subscribed to by sending an **UNSUBSCRIBE** message to the broker. Similar to SUBSCRIBE, this message includes a packet identifier to uniquely identify it and a list of topics to unsubscribe from.


Example of MQTT UNSUBSCRIBE Packet
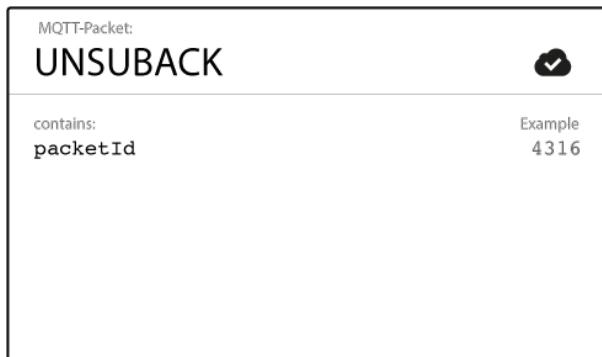
Packet Identifier: Similar to the SUBSCRIBE message, the packet identifier in the UNSUBSCRIBE message serves as an internal MQTT identifier for message flow between the client and broker. It ensures that the client and broker can keep track of the message and its corresponding acknowledgment messages.

List of Topics: The list of topics in the UNSUBSCRIBE message can contain one or multiple topics from which the client wants to unsubscribe. It is not necessary to specify the QoS level since the broker will unsubscribe the topic regardless of the QoS level with which it was originally subscribed.

What is MQTT Unsuback?

After receiving the UNSUBSCRIBE message, the broker sends an **UNSUBACK** acknowledgment message to confirm the removal of the client's subscriptions. This message includes the packet identifier of the UNSUBSCRIBE message and serves as an acknowledgment that the broker has successfully removed the topics from the client's subscription list.
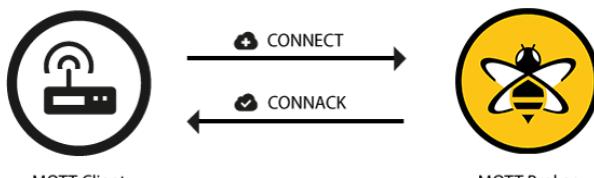
Example of MQTT UNSUBACK Packet

Packet Identifier: The packet identifier in the UNSUBACK message is the same as the one in the corresponding UNSUBSCRIBE message. This ensures that the client can identify the acknowledgment message and correlate it to the original UNSUBSCRIBE message.

Return Codes: The UNSUBACK message includes a list of return codes for each topic/QoS-pair that was unsubscribed. A return code of 0 indicates a successful removal, while a return code of 17 indicates an unsuccessful removal due to an invalid or malformed topic. Other return codes may also be specified for different error scenarios.

How MQTT UNSUBACK works

After receiving the UNSUBACK from the broker, the client can assume that the subscriptions in the UNSUBSCRIBE message are deleted.

These details provide a comprehensive understanding of how clients can unsubscribe from topics and how brokers confirm the removal of those subscriptions through the UNSUBSCRIBE and UNSUBACK messages, respectively.

To summarize, MQTT provides a flexible and data-agnostic approach to publishing messages between clients and brokers. By using topics to filter messages, clients can quickly and easily subscribe to the content that interests them. The payload of each message can be customized to meet each client's specific needs, and MQTT's support for various data types makes it a versatile solution for many use cases. Additionally, understanding the attributes of a PUBLISH message, such as the QoS level and retain flag, can help clients and brokers ensure that messages are delivered efficiently and reliably.

Chapter 7: MQTT Topics and Wildcards

What Are MQTT Topics and Their Role in MQTT Message Filtering?

In MQTT, Topic refers to a UTF-8 string that filters messages for a connected client. A topic consists of one or more levels

Learn MQTT Topic Basics

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

Here's some examples of MQTT Topics:

1. `myhome/groundfloor/livingroom/temperature:` This topic represents the temperature in the living room of a home located on the ground floor.
2. `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.

3. `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.
4. `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.

Best Practices for Using MQTT Topics

Here're some best practices for using MQTT Topics:

- Each topic must contain at least one character.
- Topic strings can include empty spaces to allow for more readable or descriptive topics.
- Topics are case-sensitive, meaning "myhome/temperature" and "MyHome/Temperature" are considered as two different topics.
- The forward slash alone is a valid topic and can be used to represent a broad topic or serve as a wildcard for subscribing to multiple topics simultaneously.

> MQTT topics are key in establishing communication between MQTT clients and brokers. They enable efficient filtering and routing of messages based on their content. Properly defining and structuring topics is crucial in ensuring effective data exchange and handling within MQTT-based systems.

MQTT Wildcards and How to Use Them With Topic Subscriptions?

In MQTT, wildcards provide a powerful mechanism for subscribing to multiple topics simultaneously. When a client subscribes to a topic, it can either subscribe to the exact topic of a published message or utilize wildcards to broaden its subscription. It's important to note that wildcards can only be used for subscription and not for publishing messages. There are two types of wildcards: single-level and multi-level.

MQTT Wildcard – Single Level: +

The single-level wildcard is represented by the plus symbol (+) and allows the replacement of a single topic level. By subscribing to a topic with a single-level wildcard, any topic that contains an arbitrary string in place of the wildcard will be matched.

Example of how to use MQTT Wildcard Single Level +

For example, a subscription to myhome/groundfloor/+/temperature can produce the following results:

MQTT Wildcard – Multi Level:

The multi-level wildcard covers multiple topic levels. It is represented by the hash symbol (#) and must be placed as the last character in the topic, preceded by a forward slash. When a client subscribes to a topic with a multi-level wildcard,

Example of how to use MQTT Wildcard Multi Level #

it receives all messages of a topic that begins with the pattern before the wildcard character, regardless of the length or depth of the topic. If the topic is specified as "#" alone, the client receives all messages sent to the MQTT broker.

MQTT Topic wildcard hash example

However, it's important to consider that subscribing with a multi-level wildcard alone can be an anti-pattern if high throughput is expected. Subscribing to a broad topic can result in a large volume of messages being delivered to the client, potentially impacting system performance and bandwidth usage. Follow best practices to optimize topic subscriptions and avoid unnecessary message overload.

MQTT Topics Beginning with $

In MQTT, topic naming flexibility is vast, allowing you to choose names that suit your needs. However, there is one important exception to be aware of: topics that start with a $ symbol have a distinct purpose. These topics are not included in the subscription when using the multi-level wildcard (#) as a topic. Instead, topics beginning with $ are reserved for internal statistics of the MQTT broker, providing valuable insights into its operation.

Publishing messages to topics starting with $ is not permitted, as these topics serve as a means for the MQTT broker to expose internal information and statistics to clients. While there is currently no official standardization for these topics, it is common to use the prefix $SYS/ to denote such information, although specific implementations of brokers may vary.

One recommended resource for understanding $SYS topics is available in the **MQTT GitHub wiki**.

Here are a few examples of $SYS topics and the information they can provide:

1. $SYS/broker/clients/connected: Indicates the number of clients currently connected to the MQTT broker.
2. $SYS/broker/clients/disconnected: Shows the number of clients that have disconnected from the MQTT broker.

3. $SYS/broker/clients/total: Represents the total count of clients, both connected and disconnected, that have interacted with the MQTT broker.
4. $SYS/broker/messages/sent: Provides the count of messages sent by the MQTT broker.
5. $SYS/broker/uptime: Reflects the duration the MQTT broker has been running.

> These $SYS topics offer valuable insights into the internal workings and performance of the MQTT broker, enabling administrators and developers to monitor and analyze crucial statistics.

By understanding the purpose and significance of topics starting with $, you can effectively leverage this convention to gain deeper visibility into the behavior and performance of their MQTT infrastructure.

Exploring the Dynamic Nature of MQTT Topics

These are the basics of MQTT message topics. As you can see, MQTT topics are dynamic and provide great flexibility. When you use wildcards in real-world applications, there are some challenges you should be aware of. We have collected the best practices that we have learned from working extensively with MQTT in various projects and are always open to suggestions or a discussion about these practices. Use the comments to start a conversation, Let us know your best practices or if you disagree with one of ours!

MQTT Best Practices

Avoid Leading Forward Slash

While MQTT allows a leading forward slash in topics (e.g., /myhome/groundfloor/livingroom), it introduces an unnecessary topic level with a zero character at the front. This can cause confusion (having a zero character at the front) without providing any benefit. Hence, it's recommended to exclude the leading forward slash.

Never use spaces in an MQTT Topic

A space is the natural enemy of every programmer. Spaces in topics can hinder readability and debugging efforts, particularly

during troubleshooting scenarios. Moreover, UTF-8 has many different white space types. We advise against using spaces and uncommon characters altogether in MQTT topics.

Keep MQTT topics short and concise

Remember that each topic is included in every message in which it is used. To optimize network traffic and conserve valuable resources, strive to make your topics concise. This is especially crucial when dealing with resource-constrained devices, where every byte counts.

Use only ASCII characters, and avoid non-printable characters

To ensure consistent and accurate representation of topics, it's advisable to stick to ASCII characters. Non-ASCII UTF-8 characters may display incorrectly, making identifying typos or character set-related issues challenging. Unless essential, refrain from using non-ASCII characters in your MQTT topics.

Embed a unique identifier or the Client Id in topics

To enhance message identification and enforce authorization, consider embedding a unique identifier or the client ID of the publishing client in the topic. This allows you to determine the message sender and control publishing permissions. For example, a client with the client1 ID can publish to client1/status but not to client2/status.

Avoid Subscribing to Wildcards (#)

Sometimes, it is necessary to subscribe to all messages that are transferred over the broker. For example, to persist all messages into a database. Do not subscribe to all messages on a broker by using an MQTT client and subscribing to a multi-level wildcard. Frequently, the subscribing client is not able to process the load of messages that results from this method (especially if you have a massive throughput). Our recommendation is to implement an extension in the MQTT broker. For example, with the **HiveMQ extensions**, you can hook into the behavior of HiveMQ and add an asynchronous routine to process each incoming message and persist it to a database.

Embrace Extensibility

Topics in MQTT provide inherent flexibility, allowing for future expansion and new features. Consider how your topic structure can accommodate future enhancements or the addition of new sensors or functionalities. Design your topics to facilitate extensibility without substantially changing the overall topic hierarchy. For example, if your smart-home solution adds new sensors, it should be possible to add these to your topic tree without changing the whole topic hierarchy.

Use specific topics, not general ones

Differentiate your topics to reflect specific data streams or entities. Avoid the temptation to use a single topic for multiple types of messages. For instance, if you have three sensors in your living room, create topics like myhome/livingroom/temperature, myhome/livingroom/brightness, and myhome/livingroom/humidity instead of using a generic topic like myhome/livingroom. This practice promotes clarity and enables the utilization of advanced MQTT features such as retained messages.

Documentation

Maintain comprehensive documentation detailing your MQTT topics, including their purpose, expected message payload, and any associated conventions or guidelines. This aids in onboarding new team members and fosters better collaboration.

Continuous Improvement

Regularly review and optimize your topic structure based on evolving requirements and feedback from your MQTT ecosystem. Embrace a continuous improvement mindset to ensure efficient and scalable MQTT communication.

Security Considerations

Ensure that your topic structure and naming conventions don't inadvertently expose sensitive information. Implement proper access controls and authentication mechanisms to protect your MQTT communications.

By following these best practices, you can enhance your MQTT infrastructure's readability, maintainability, and security.

To summarize, MQTT topics serve as the backbone of flexible and efficient message communication in MQTT. By understanding the intricacies and applying best practices, you can optimize your MQTT implementations for maximum performance and scalability.

Chapter 8: MQTT Quality of Service (QoS) 0,1, & 2

MQTT Quality of Service (QoS) is an agreement between the message sender and receiver that defines the level of delivery guarantee for a specific message.

MQTT provides three levels of QoS:

- At most once (QoS 0)
- At least once (QoS 1)
- Exactly once (QoS 2)

How to Examine Message Delivery in MQTT?

When discussing QoS in MQTT, it's important to consider message delivery from the publishing client to the broker and from the broker to the subscribing client. These two aspects of message delivery have subtle differences.

The client that publishes a message to the broker defines the QoS level for the message during transmission. The broker then transmits the message to subscribing clients using the QoS level defined by each subscribing client during the subscription process. If the subscribing client defines a lower QoS level than the publishing client, the broker will transmit the message with the lower QoS level.

Understanding how message delivery works in MQTT sets the foundation for appreciating the significance of Quality of Service (QoS) levels in ensuring reliable communication between the publishing client, broker, and subscribing client.

Why is Quality of Service (QoS) Important?

Quality of Service (QoS) is crucial in MQTT due to its role in

providing the client with the ability to select a service level that aligns with both the network reliability and the application's requirements. MQTT's inherent capability to handle message re-transmission and ensure delivery, even in unreliable network conditions, makes QoS essential for facilitating seamless communication in such challenging environments. By offering different QoS levels, MQTT empowers clients to optimize their network usage and achieve the desired balance between reliability and efficiency.

Now that we understand the significance of Quality of Service (QoS) in MQTT, let's delve into the inner workings of QoS and explore how it operates to ensure reliable message delivery in varying network conditions.

How does QoS 0 work in MQTT?

At the lowest level, QoS 0 in MQTT offers a best-effort delivery mechanism where the sender does not expect an acknowledgment or guarantee of message delivery. This means that the recipient does not acknowledge receiving the message, and the sender does not store or re-transmit it. QoS 0, commonly called "fire and forget," functions akin to the underlying TCP protocol, where the message is sent without further follow-up or confirmation.

Quality of Service level 0: delivery at most once

How does QoS 1 work in MQTT?
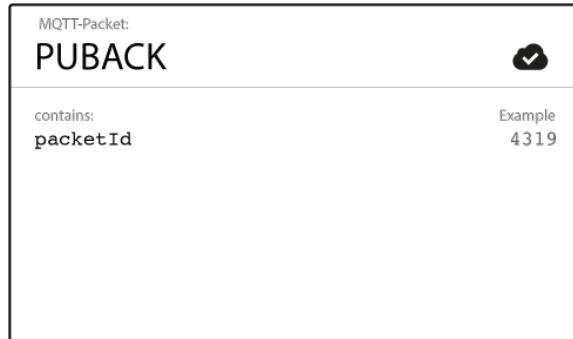
In QoS 1 of MQTT, the focus is on ensuring message delivery at least once to the receiver. When a message is published with QoS 1, the sender keeps a copy of the message until it receives a **PUBACK** packet from the receiver, confirming the successful receipt. If the sender doesn't receive the **PUBACK** packet within a reasonable time frame, it re-transmits the message to ensure its delivery.

Upon receiving the message, the receiver can process it
Quality of Service level 1: delivery at least once

immediately. For example, if the receiver is an MQTT broker, it distributes the message to all subscribing clients and responds with a PUBACK packet to acknowledge the receipt of the message.

It's important to note that in QoS 1, if the publishing client

MQTT PUBACK packet

sends the same message again, it sets a duplicate (DUP) flag. However, this flag is used for internal purposes and is not processed by the broker or client. Regardless of the DUP flag, the receiver still sends a PUBACK packet to acknowledge the receipt of the message, ensuring the sender is aware of the successful delivery.

This approach of QoS 1 strikes a balance between reliability and efficiency, guaranteeing that the message reaches the receiver at least once while allowing for potential duplicates to be handled appropriately.
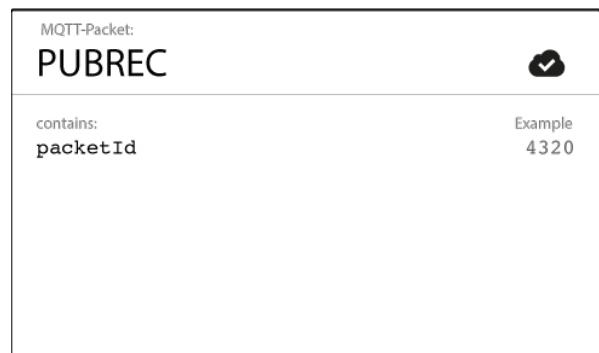
How does QoS 2 work in MQTT?

QoS 2 offers the highest level of service in MQTT, ensuring that each message is delivered exactly once to the intended recipients. To achieve this, QoS 2 involves a four-part handshake between the sender and receiver.
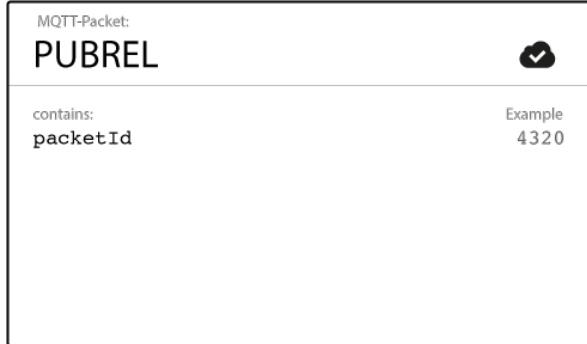
MQTT Quality of Service level 2: delivery exactly once

When a receiver gets a QoS 2 PUBLISH packet from a sender, it processes the publish message accordingly and replies to the sender with a **PUBREC** packet that acknowledges the PUBLISH packet. If the sender does not get a PUBREC packet from the receiver, it sends the PUBLISH packet again with a duplicate (DUP) flag until it receives an acknowledgment.

MQTT PUBREC Packet

Once the sender receives a PUBREC packet from the receiver, the sender can safely discard the initial PUBLISH packet. The sender stores the PUBREC packet from the receiver and responds with a **PUBREL** packet, the receiver discards all stored states and replies with a PUBCOMP packet.
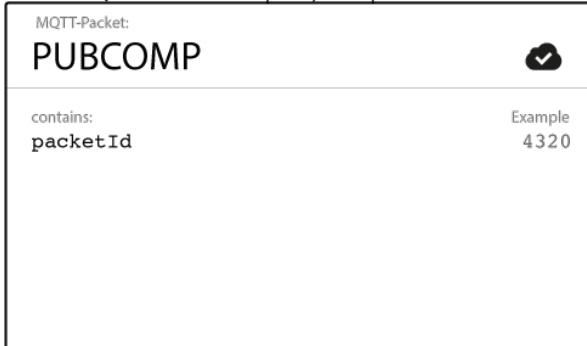
MQTT PUBREL Packet

After the receiver gets the PUBREL packet, it can discard all stored states and answer with a **PUBCOMP** packet (the same is true when the sender receives the PUBCOMP). Until the receiver completes processing and sends the PUBCOMP packet back to the sender, the receiver stores a reference to the packet identifier of the original PUBLISH packet. This step is important to avoid processing the message a second time.

After the sender receives the PUBCOMP packet, the packet identifier of the published message becomes available for reuse.

When the QoS 2 flow is complete, both parties are sure that the

MQTT PUBCOMP Packet

message is delivered and the sender has confirmation of the delivery.

If a packet gets lost along the way, the sender is responsible to retransmit the message within a reasonable amount of time. This is equally true if the sender is an MQTT client or an **MQTT broker**. The recipient has the responsibility to respond to each command message accordingly.

Key Considerations for QoS in MQTT

While understanding QoS in MQTT, there are several important aspects to keep in mind:

Downgrade of QoS

The QoS levels defined by the sender and receiver can differ. The client sending the message to the broker defines the QoS level, while the broker uses the QoS defined by the receiver during subscription. For example, if the sender uses QoS 2 and the receiver subscribes with QoS 1, the broker delivers the message to the receiver with QoS 1. This can result in multiple deliveries of the same message to the receiver.

Packet identifiers are unique per client

Packet identifiers used for QoS 1 and 2 are unique between a specific client and a broker within an interaction. However, they are not unique across all clients. Once a flow is complete, the packet identifier becomes available for reuse. This is why the packet identifier does not need to exceed 65535 , as it is unrealistic for a client to send more messages than that without completing an interaction

Best Practices While Using MQTT Quality of Service (QoS) 0,1, & 2

We are often asked for advice about how to choose the correct QoS level. Selecting the appropriate QoS level depends on your specific use case. Here are some guidelines to help you make an informed decision:

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.**

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. QoS 1 delivers messages much faster than QoS 2.

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.

Queuing of QoS 1 and 2 Messages

All messages sent with QoS 1 and 2 are queued for offline clients until the client is available again. However, this queuing is only possible if the client has a **persistent session**.

Chapter 9: MQTT Persistent Sessions and Clean Sessions

Persistent sessions in MQTT allow a client to maintain its subscription and message state across multiple connections. When a client establishes a persistent session with an MQTT broker, the broker stores the client's subscription information and any undelivered messages intended for the client. This way, if the client disconnects and reconnects later, it can resume communication seamlessly. This chapter will dive deep into how persistent sessions in MQTT enhance QoS by enabling reliable message delivery, providing QoS level guarantees, and facilitating efficient reconnection for clients, even in the presence of intermittent connectivity or client disconnections.

What are Persistent Sessions?

To receive messages from an **MQTT broker**, a client establishes a connection and **creates subscriptions to the desired topics**. In a non-persistent session, if the connection between the client and broker is interrupted, the client loses its subscriptions and needs to re-subscribe upon reconnection. This can be burdensome for resource-constrained clients. To address this issue, clients can request a persistent session when connecting to the broker.

> Persistent sessions store all relevant client information on the broker, ensuring that subscriptions and messages are retained even when the client is offline. The session is identified by the clientId provided by the client during the connection establishment process.

What's Stored in a Persistent Session?

In a persistent session, the broker stores the following information (even if the client is offline). This information becomes immediately available to the client upon reconnection:

- Session existence (even if there are no subscriptions): The broker retains information about the existence of the session, allowing the client to resume its previous state upon reconnection.
- All client's subscriptions: The broker stores the list of topics to which the client has subscribed. This ensures the client does not need to re-subscribe to the same topics every time it reconnects, saving valuable time and resources.
- Flow of all messages in a Quality of Service (QoS) 1 or 2 where the client has not yet confirmed: The broker keeps track of unacknowledged messages sent to the client at QoS 1 or QoS 2 levels. These messages are stored in the broker's message queue and will be delivered to the client upon reconnection, ensuring reliable message delivery.
- All new QoS 1 or 2 messages that the client missed while offline: If the client was offline when QoS 1 or QoS 2 messages were published to subscribed topics, the

broker stores these missed messages. Once the client reconnects, it receives the queued messages, preventing any loss of important information.

- All QoS 2 messages received from the client that are awaiting complete acknowledgment: For QoS 2 messages sent by the client, the broker keeps track of their acknowledgment status. If any of these messages are not fully acknowledged, the broker holds them until the acknowledgment is complete.

Clean Sessions in MQTT and How to Use Them to Start or End a Persistent Session

When establishing a connection to the broker, clients can enable or disable a persistent session by setting the value of the cleanSession flag. Here's how it works: When the cleanSession flag is set to true, the client explicitly requests a non-persistent session. In this scenario, if the client disconnects from the broker, all queued information and messages from the previous persistent session are discarded. The client starts with a clean slate upon reconnection.

> On the other hand, when the cleanSession flag is set to false, the broker creates a persistent session for the client. This means that the broker preserves all relevant information and messages even if the client goes offline. The session remains intact until the client explicitly requests a clean session. If the cleanSession flag is set to false and the broker already has a session available for the client, it will utilize the existing session and deliver any previously queued messages to the client upon reconnection.

How Does A Client Know if a Session is Already Stored?

In MQTT version 3.1.1 and beyond, the CONNACK message sent by the broker in response to the client's connection request includes a session present flag. This flag serves as an indicator for the client, informing it whether a previously established session is still available on the broker. By examining the session present flag, the client can determine whether to establish a new session or reconnect to an existing one.

Persistent Session on the Client Side: Ensuring Local Message Persistence and Acknowledgment

In addition to the broker storing a persistent session, each MQTT client also plays a role in maintaining session continuity. When a client requests the server to retain session data, it assumes the responsibility and must store the following information:

- Flow of all messages in a QoS 1 or 2 where the broker has not yet confirmed: The client keeps track of messages it has sent to the broker at QoS 1 or QoS 2 levels. These messages are stored locally until the broker acknowledges their receipt or completion. By maintaining these unconfirmed messages, the client ensures that it can retransmit any messages if necessary and achieve the desired level of reliability.

- All QoS 2 messages received from the broker awaiting complete acknowledgment: When the broker sends QoS 2 messages to the client, they are received and processed by the client. However, until the client acknowledges the successful processing of these messages, the client stores them locally. This ensures that the client can handle any interruptions or disconnections while maintaining the reliability and integrity of message delivery.

- By storing these message-related details on the client side, MQTT clients can actively maintain session persistence and ensure the successful processing of messages even in challenging network conditions.

MQTT Session Management Best Practices: Enhancing Message Delivery and Resource Efficiency

When working with MQTT, it's essential to consider the best practices for session management to optimize your implementation. Here are some guidelines to help you determine when to use a persistent session or a clean session:

Best Practices for MQTT Persistent Sessions

- Ensure Message Reliability: If your client needs to receive all messages from a specific topic, even when they are offline, a persistent session is the way to go. This ensures that the broker queues the messages for the client and delivers them promptly when the client reconnects.

- Resource Optimization: If your client has limited resources, leveraging a persistent session is beneficial. Storing the client's subscription information on the broker facilitates a quick restoration of interrupted communication, reducing the burden on constrained clients.

- Resume Publishes: A persistent session is necessary if your client needs to resume publishing Quality of Service (QoS) 1 and 2 messages after reconnecting. The broker retains these messages until the client returns online, ensuring their reliable delivery.

Best Practices for MQTT Clean Session

- Publish-Only Clients: A clean session is suitable if your client only needs to publish messages and does not require subscriptions to topics. In such cases, the broker does not need to store session information or attempt to transmit QoS 1 and 2 messages, simplifying the session management process.

- Avoid Offline Message Retrieval: A clean session suffices if your client does not need to receive missed messages while offline. It eliminates the overhead of storing and delivering messages that the client did not subscribe to during its offline period.

MQTT Message Storage Duration: How Long Does the MQTT Broker Store Messages?

People often ask how long the broker stores session information and messages. The answer depends on various factors and considerations:

- Memory Limit Constraint: Typically, the primary constraint on message storage is the memory limit of the operating system hosting the broker. Monitoring and allocating sufficient resources to handle the expected message volume is crucial.

- Use Case Specifics: The appropriate solution for managing message storage duration varies based on your use case. Consider factors such as the importance of retaining messages, message expiration policies, and regulatory or compliance requirements.

By following these best practices and considering the above-mentioned factors, you can effectively manage MQTT sessions, optimize message persistence, and ensure reliable communication in your MQTT-based solutions.

Remember to incorporate these guidelines into your implementation based on your unique requirements and application needs.

To summarize, understanding and effectively utilizing persistent sessions, queuing mechanisms, and proper session management practices, we can harness the full potential of MQTT and build robust, scalable, and reliable IoT and messaging applications.

Chapter 10: MQTT Retained Messages

A retained message is a normal MQTT message with the retained flag set to true. The broker stores the last retained message and the corresponding QoS for that topic. Each client that subscribes to a topic pattern that matches the topic of the retained message receives the retained message immediately after they subscribe. The broker stores only one retained message per topic.

How are Retained Messages Different from Normal MQTT Messages?

> Retained messages are a valuable feature in MQTT that mitigates uncertainty regarding message publication. By enabling the retention of the most recent message on a topic, subscribers can stay informed about the current state, even during periods of inactivity.

In MQTT, it is important to understand that the publisher of a message cannot guarantee that the subscribing client will receive the message. The publisher's responsibility lies in ensuring the safe delivery of the message to the broker. Similarly, the subscribing client cannot determine when the publishing client will send a new message related to their subscribed topics. The time interval between messages can vary significantly, ranging from a few seconds to several minutes or even hours. As a result, the subscribing client remains unaware of the current topic status until a new message is published. This is where retained messages play a vital role.

Retained messages provide a solution to the challenge mentioned above. When a client publishes a message with the "retained" flag set to true, the broker retains the message. Consequently, any client subscribing to the corresponding topic will receive the most recent retained message, even if no recent publications have occurred.

In essence, retained messages offer subscribers a snapshot of the last-known state of a topic, ensuring access to the latest relevant information regardless of publishing frequency.

Structure of a Retained Message in MQTT

An MQTT retained message is a standard message that includes the retained flag set to true, signifying its importance. When a client publishes a retained message, the broker stores it along with the corresponding Quality of Service (QoS) for a specific topic. Subscribing clients immediately receive the retained message when they subscribe to a topic pattern that matches the retained message's topic.

It is worth noting that the broker retains only one message per topic.

Even if a subscribing client uses wildcards in their topic pattern, they can still receive a retained message that may not be an exact match for the topic.

For instance, if Client A publishes a retained message to myhome/livingroom/temperature and later Client B subscribes

to myhome/#, Client B will receive the retained message for myhome/livingroom/temperature immediately after subscribing to myhome/#. By recognizing the retained flag set to true, the subscribing client can process the retained messages according to its requirements.

Retained messages are crucial in providing newly-subscribed clients with immediate status updates upon subscribing to a topic, eliminating the need to wait for subsequent updates from publishing clients. Essentially, a retained message represents the last known valid value for a particular topic. It does not necessarily have to be the latest value, but it must be the most recent message with the retained flag set to true.

Something else important to emphasize, retained messages operate independently of **persistent sessions**, which we discussed in Part 7 of the series. Once the broker stores a retained message, there's only one way to remove it. We will discuss that shortly in an upcoming section.

Now that you understand the structure of retained messages, let's learn more about managing them.

How to Send a Retained Message in MQTT?

As a developer, sending a retained message is a straightforward process. To mark a message as retained, all you need to do is set the retained flag of your **MQTT publish message** to true. This flag signals the broker to retain the message and make it available to subscribers. The good news is that most MQTT client libraries provide a convenient and user-friendly way to enable this flag, streamlining the process. By leveraging this feature, developers can ensure that critical information persists and remains accessible to subscribers, even if they join the network later or experience temporary connectivity issues. Retained messages offer a powerful mechanism for sharing important data and enabling seamless communication in MQTT-based systems.

How to Delete Retained Messages in MQTT?

There is only one way to delete the retained message of a topic. To achieve this, simply publish a retained message with

a zero-byte payload to the topic where the retained message is stored. When the broker receives this special retained message, it identifies it as a request for deletion and promptly removes the retained message associated with that topic. As a result, new subscribers will no longer receive the previously retained message for that particular topic.

It's worth noting that in many cases, explicitly deleting retained messages may not be necessary. This is because each new retained message automatically overwrites the previous one for the same topic. Therefore, if you publish a new retained message on a topic, it will replace and supersede any existing retained message, effectively achieving the same outcome as deleting the previous message. This behavior ensures that subscribers receive the most up-to-date and relevant information, eliminating the need for manual deletion in most scenarios.

Why and When Should You Use Retained Messages?

Retained messages offer valuable benefits in various scenarios, particularly when you need newly-connected subscribers to receive messages promptly without waiting for the next message publication.

This is particularly advantageous for delivering real-time status updates of components or devices on specific topics. For instance, let's consider the example of a device named device1, whose status is published on the topic "myhome/devices/device1/status". By utilizing retained messages, new subscribers to this topic instantly receive the device's status (such as online or offline) immediately after subscribing.

Similarly, this applies to clients that transmit data periodically, such as temperature readings, GPS coordinates, and other relevant information. Without retained messages, newly-subscribed clients would remain unaware of the latest updates between message intervals. By leveraging retained messages, you can seamlessly provide connecting clients with the most recent and accurate value, ensuring they have immediate access to critical information.

Closing the Loop: The Lasting Impression of Retained Messages in MQTT's Ecosystem

As you can see, retained messages play a crucial role in MQTT communication by addressing the challenge of uncertain message delivery and providing immediate access to the last-known state of a topic. By enabling the retention of the most recent message on a topic, subscribers can stay informed about the current status, even during periods of inactivity.

Retained messages are beneficial for providing status updates, ensuring newly-subscribed clients receive relevant information without having to wait for the subsequent message publication. By leveraging retained messages, MQTT empowers efficient and reliable communication between clients and brokers, enhancing the overall effectiveness of IoT and messaging applications.

Chapter 11. MQTT Last Will and Testament (LWT)

Last Will and Testament (LWT) is a powerful feature in MQTT that allows clients to specify a message that will be automatically published by the broker on their behalf, if or when an unexpected disconnection occurs. It provides a reliable means of communication and ensures that clients can gracefully handle disconnections without leaving topics in an inconsistent state. This feature is particularly valuable when clients must notify others of their unavailability or convey important information upon an unexpected disconnection.

What is the Purpose of Last Will and Testament (LWT) in MQTT?

In scenarios where unreliable networks are prevalent, it is common for MQTT clients to experience occasional unintended breaks, which can happen due to loss of connection or depleted batteries. Understanding the type of disconnection (graceful - with a disconnect message, or ungraceful - without a disconnect message) is crucial for taking appropriate actions.

> The Last Will and Testament feature in MQTT offers a solution for clients to respond effectively to ungraceful disconnects and ensure proper handling of such events.

The LWT allows clients to notify others about their unexpected disconnections. When a client connects to a broker, it can specify a last-will message. This message follows the structure of a regular MQTT message structure, including a topic, retained message flag, Quality of Service (QoS), and payload. The broker stores this message until it detects an ungraceful disconnect from the client. Upon detecting the disconnection, the broker broadcasts the last will message to all subscribed clients of the corresponding topic. The broker discards the stored LWT message if the client disconnects gracefully using the DISCONNECT message.

DISCONNECT MQTT Packet

By utilizing LWT, you can implement various strategies to handle client disconnections and inform other clients about the offline status.

How to Configure a Last Will and Testament (LWT) Message for an MQTT Client?

To specify an LWT message for an MQTT client, you include it in the CONNECT message, which is used to initiate the connection between the client and the broker.

CONNECT MQTT Packet

When does the MQTT Broker Send the LWT Message?

According to the **MQTT 3.1.1 specification**, the broker sends a client's Last Will and Testament (LWT) message in the following situations:

1. I/O error or network failure: If the broker detects any issues with the input/output or network connection, it will distribute the LWT message.
2. Failed communication within Keep Alive period: If the client fails to communicate with the broker within the specified Keep Alive period, the LWT message is sent. In Part-10 of our MQTT Essentials, we will explore the concept of **MQTT Keep Alive** time and delve into its significance.
3. Client closes connection without DISCONNECT: When the client terminates the network connection without sending a DISCONNECT packet, the broker ensures the LWT message is distributed.
4. Broker closes connection due to protocol error: If the broker closes the network connection due to a protocol error, it will send the LWT message.

Understanding when and why the broker sends the Last Will and Testament (LWT) messages lays the groundwork for implementing best practices in leveraging this feature, which we will delve into in the next section.

When to Use Last Will and Testament (LWT) in MQTT?

LWT proves invaluable for alerting subscribed clients about an abrupt disconnection of a client. It becomes a powerful tool for storing and communicating client state on specific topics when combined with retained messages.

For instance, by setting a lastWillMessage with Offline payload, enabling the lastWillRetain flag, and specifying the lastWillTopic as client1/status, followed by publishing an Online retained message to the same topic, client1 can keep newly-subscribed clients informed about its online status. Should client1 disconnect unexpectedly, the broker publishes the LWT message with Offline payload as the new **retained message**, ensuring that clients subscribing to the topic while client1 is offline receive the LWT message and stay up to date on its current status.

LWT not only notifies subscribed clients about unexpected disconnections but also assists in maintaining the system's integrity by providing valuable information on client states. Combining LWT with retained messages allows you to create a robust solution that stores and communicates the latest client state on specific topics, ensuring reliable updates for all subscribers. This approach enables seamless integration and synchronization between clients, enhancing the overall resilience and functionality of the MQTT network.

The Importance of Last Will and Testament in MQTT

The Last Will and Testament (LWT) feature in MQTT is crucial in ensuring efficient communication and maintaining system integrity in the event of unexpected client disconnections. By combining LWT with retained messages, developers can store and communicate client state on specific topics, providing valuable information to subscribed clients. LWT empowers MQTT networks with enhanced resilience, seamless integration, and reliable updates, making it a powerful tool for various applications. By understanding the benefits and best practices of LWT, you can leverage this feature to create robust and effective MQTT solutions.

Chapter 12: MQTT Keep Alive and Client Take-Over

Keep Alive is a feature of the MQTT protocol that allows an MQTT client to maintain its connection with a broker by sending regular control packets called PINGREQ to the broker. Let's dive into MQTT Keep Alive's critical role in mobile networks, and in maintaining a robust and efficient MQTT connection.

What is MQTT Keep Alive and Why It's Important?

The Keep Alive mechanism in MQTT ensures the connection's liveliness and provides a way for the broker to detect if a client becomes unresponsive or disconnected.

When a client establishes a connection with an MQTT broker, it negotiates a Keep Alive value, which is a time interval expressed in seconds. The client must send a PINGREQ packet to the broker at least once within this interval to indicate its presence and keep the connection alive. Upon receiving a PINGREQ packet, the broker responds with a PINGRESP packet, confirming that the connection is still active.

> The MQTT Keep Alive mechanism is important for connection monitoring, efficient resource utilization, detecting network failures, graceful disconnection, etc.

Now, let's establish why the kept alive feature is so important by delving into the issue of half-open TCP connections and how it poses a challenge within MQTT, particularly in mobile networks.

The Problem of Half-Open TCP Connections in MQTT

The problem of half-open TCP connections arises within **MQTT**, **which relies on the Transmission Control Protocol (TCP)** to ensure **"reliable, ordered, and error-checked"** packet transfer over the internet. Despite TCP's robustness, there are instances where the synchronization between communicating parties can falter due to crashes or transmission errors.

In TCP, this state of an incomplete connection is referred to as a **half-open connection**, where one side of the communication remains unaware of the other side's failure. The connected side persistently attempts to send messages while eagerly awaiting acknowledgments.

Andy Stanford-Clark, the inventor of the MQTT protocol, highlights that the issue of half-open connections becomes more pronounced in mobile networks. While TCP/IP theoretically notifies users when a socket breaks, practical scenarios, particularly on mobile and satellite links, involve the "faking" of TCP over the air with added headers at each end. This practice can lead to a phenomenon known as a "black hole" TCP session, where the connection appears open but, in reality, discards any transmitted data.

> "Although TCP/IP, in theory, notifies you when a socket breaks, in practice, particularly on things like mobile and satellite links, which often "fake" TCP over the air and put headers back on at each end, it's quite possible for a TCP session to "black hole," i.e. it appears to be open still, but is just dumping anything you write to it onto the floor."
>
> *Andy Stanford-Clark on the topic "Why is the keep-alive needed?"* **Source**

How Does MQTT Keep Alive Work?

To address the challenge of half-open connections and enable continuous assessment of connection status, MQTT incorporates a vital feature called Keep Alive. This mechanism guarantees that the connection between the MQTT broker and client remains active and that both parties know their connection status.

When a client establishes a connection with the broker, it specifies a time interval in seconds known as the Keep Alive duration. This duration sets the maximum allowed time gap during which the broker and client may not exchange any communication. According to the MQTT specification, the Keep Alive interval is defined as follows:

> "The Keep Alive ... is the maximum time interval permitted to elapse between the point at which the Client finishes transmitting one Control Packet and the point it starts sending the next. It is the responsibility of the Client to ensure that the interval between Control Packets being sent does not exceed the Keep Alive value. In the absence of sending any other Control Packets, the Client MUST send a PINGREQ Packet."

As long as messages are transmitted frequently within the Keep Alive interval, there is no need to send an additional message to verify the connection status. However, if the client remains inactive during the Keep Alive period, it must send a PINGREQ packet to the broker as a confirmation of its availability and to ensure that the broker is still accessible.

If a client fails to send any messages or a PINGREQ packet within one and a half times the Keep Alive interval, the broker is responsible for disconnecting the client. Likewise, the client should close the connection if it does not receive a response from the broker within a reasonable timeframe.

By employing the Keep Alive mechanism, MQTT enhances connection stability, mitigates the risks associated with half-open connections, and facilitates efficient communication between brokers and clients in various network conditions.

How Does Keep Alive Ensure Connection Vitality?

Let's examine the flow of Keep Alive messages to gain a deeper understanding of the Keep Alive mechanism. The Keep Alive feature utilizes two packets: PINGREQ and PINGRESP.

What is PINGREQ in MQTT Keep Alive?

MQTT-Packet:
PINGREQ
no payload

When a client wants to signal its continued presence and activity to the broker, it sends a PINGREQ packet. This packet is a "heartbeat" message indicating that the client is still alive. If the client has not sent any other type of packet, such as a PUBLISH or SUBSCRIBE packet, it must send a PINGREQ packet to the broker. The client can choose to send a PINGREQ packet at any time to verify the ongoing vitality of the network connection. Notably, the PINGREQ packet does not contain any payload.

What is PINGRESP in MQTT Keep Alive?

```
MQTT-Packet:
PINGRESP                                    ✅

no payload
```

Upon receiving a PINGREQ packet from a client, the broker is obligated to respond with a PINGRESP packet. The PINGRESP packet serves as an acknowledgment from the broker to the client, confirming its availability and continued connection. Like the PINGREQ packet, the PINGRESP packet does not include a payload.

How Can I Customize Keep Alive Settings for Optimal Performance?

• If the broker does not receive a PINGREQ packet or any other packet from a client within the expected time frame, the broker will close the connection and dispatch the **Last Will and Testament message (LWT)** message if the client has specified one.
• The MQTT client is responsible for setting an appropriate Keep Alive value. For instance, the client can adjust the keep-alive interval based on its current signal strength, optimizing the connection for its specific circumstances.
• Importantly, the maximum Keep Alive interval is 18 hours,

12 minutes, and 15 seconds. Conversely, setting the Keep Alive interval to 0 effectively deactivates the Keep Alive mechanism, removing its influence on the connection's stability and management.

What is Client Take-Over in MQTT?

In the MQTT protocol, when a client becomes disconnected, it typically attempts to reconnect. However, there are instances when the broker still maintains a half-open connection for that client. In such cases, when a client, which the MQTT broker perceives as online, initiates a reconnection and performs a `client take-over, the broker takes necessary action. It promptly terminates the previous connection associated with the same client (identified by the client identifier) and establishes a fresh connection with the client. This intelligent behavior ensures that half-open connections do not impede the disconnected client from successfully re-establishing its connection. By seamlessly managing client take-over, MQTT guarantees smooth connectivity and resilient communication in the face of intermittent network interruptions.

How Does Keep Alive and Client Take-Over Enhance MQTT Performance?

The Keep Alive feature and client take-over mechanism are vital components of MQTT that ensure reliable and efficient communication in various scenarios. By implementing Keep Alive messages through PINGREQ and PINGRESP packets, MQTT enables clients to actively signal their presence and verify network connectivity. This mechanism prevents half-open connections and allows for timely detection of inactive or lost connections.
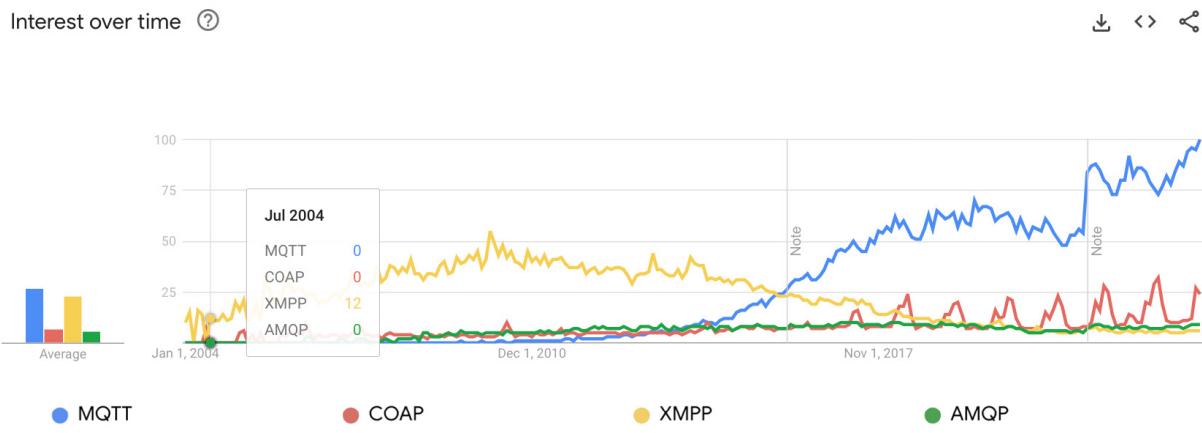
Furthermore, client take-over facilitates seamless reconnection for disconnected clients. When a client initiates a reconnection, the MQTT broker intelligently closes any existing half-open connection associated with that client and establishes a fresh connection. This process ensures that disconnections do not hinder the client's ability to regain connectivity and resume communication smoothly.

MQTT clients must set appropriate Keep Alive values, considering factors such as signal strength and network conditions. This allows for optimal management of the Keep Alive mechanism and ensures efficient resource utilization.

Understanding the intricacies of the Keep Alive feature and client take-over in MQTT empowers developers to build robust and resilient MQTT applications. By leveraging these capabilities, MQTT facilitates the creation of reliable, real-time IoT and messaging solutions that thrive even in challenging network environments.

Chapter 13: Introduction to MQTT 5 Protocol

MQTT has solidified its footprint by connecting numerous constrained devices across multiple deployments, establishing a vast network of connected systems and standalone devices. From **connected cars** and manufacturing systems, **logistics** to enterprise chat applications, and mobile apps, MQTT's widespread use has spurred demands for its evolution. MQTT 5 answers this call, promising an extensive array of exciting new features and improvements.

In this chapter, we provide an in-depth exploration of MQTT 5. We discuss topics ranging from foundational changes in the protocol to user properties, shared subscriptions, payload format description, request-response pattern, topic alias, enhanced authentication, and flow control.

Design Goals of MQTT 5

The OASIS Technical Committee (TC), tasked with defining and standardizing MQTT, faced the complex challenge of adding long-desired features without increasing overhead or decreasing ease of use. They sought to improve performance and scalability without inducing unnecessary complexity. After much deliberation, the TC decided on a host of functional objectives for MQTT 5, aimed at enhancing scalability, formalizing common patterns such as capability discovery and request-response, and enabling extensibility mechanisms like user properties.

Unlocking the Power of IoT with MQTT 5: New Features and Improvements

One of the significant improvements in MQTT 5 is the introduction of enhanced authentication mechanisms. These provide a more robust security framework critical in today's world, where the risk of cyber-attacks is always looming. With MQTT 5, users have more options for securing their devices and data, including using more sophisticated encryption algorithms and key management techniques.

In addition, MQTT 5 introduced Shared Subscriptions, a feature that allows load balancing of messages across multiple client instances. This ensures that people can manage many messages effectively without overloading individual clients. The shared subscriptions feature is practical in scenarios where many devices are transmitting data concurrently.

Further, MQTT 5 also introduces the concept of message properties, allowing additional metadata to be included with each message. This is useful for providing context about the transmitted data, such as timestamps, location information, or device status.

As described, the transition from MQTT 3.1.1 to MQTT 5 was not a simple version number update but rather a significant leap in the protocol's capabilities, addressing several areas of improvement. The result is a more robust, reliable, and scalable protocol better suited to modern IoT applications' needs.

It's important to understand these changes and improvements when considering MQTT for IoT applications, as they can significantly affect the performance and reliability of your IoT system. So, whether you're a developer, a systems integrator, or an end user, the step to MQTT 5 is a key advancement in evolving IoT communication protocols.

> While MQTT 5 brings numerous benefits, it also demands careful implementation. Like any technology upgrade, it is critical to evaluate the implications on your specific use case and ensure the transition is controlled and managed. Understanding the benefits and potential challenges of MQTT 5 will help ensure a successful implementation, maximizing the potential benefits of this powerful IoT communication protocol.

As we look to the future, MQTT remains a vital player in IoT. With MQTT 5, developers now have even more flexibility in designing and implementing solutions that can handle the diverse requirements of modern IoT systems.

The evolution of MQTT doesn't stop at version 5. The MQTT Technical Committee is already working on further enhancements and features that will continue to advance the protocol's capabilities. This ongoing work demonstrates the strong commitment of the MQTT community to the continued growth and advancement of this technology, ensuring that MQTT will remain a relevant and powerful tool for IoT

communication in the foreseeable future.

As technology continues to evolve, and as the needs of IoT systems grow and change, so will MQTT. It's critical for anyone working in the IoT space to stay abreast of these developments to ensure they are leveraging the full power of MQTT and other related technologies to deliver the most effective, reliable, and secure IoT solutions possible.

What Properties Are Available in MQTT 5 Header & Reason Codes?

Undoubtedly, one of the most transformative and flexible features introduced in MQTT 5 is the ability to incorporate custom **key-value properties in the MQTT header**. This capability can potentially revolutionize many deployments, and is of such significance that it warrants an entire chapter dedicated to it. Like protocols such as HTTP, MQTT clients and brokers can add an unlimited number of custom or pre-defined headers to carry metadata. This metadata is adaptable and can be tailored to specific application data, with pre-defined headers predominantly employed in the execution of most new MQTT features.

Additionally, MQTT packets now include Reason Codes, signifying the occurrence of a pre-defined protocol error. Traditionally found on acknowledgment packets, these Reason Codes facilitate error interpretation and potentially aid in devising remedies by both clients and brokers.

Occasionally referred to as Negative Acknowledgments, these Reason Codes can be found on a range of MQTT packets, such as:

- CONNACK
- PUBACK
- PUBREC
- PUBREL
- PUBCOMP
- SUBACK
- UNSUBACK
- AUTH
- DISCONNECT

The range of Reason Codes for Negative Acknowledgments varies widely, spanning from "Quota Exceeded" to "Protocol Error". It is the joint responsibility of clients and brokers to decode and comprehend these newly introduced Reason Codes, further enriching the overall MQTT experience.

Having established the custom key-value properties and Reason Codes, let's explore how MQTT handles unsupported features and uses CONNACK return codes.

Responding to Unsupported Features with CONNACK Return Codes

As MQTT's popularity surged, various MQTT implementations began to emerge, proposed by a diverse range of companies. However, not all these implementations adhere strictly to the complete MQTT specification. Certain features, such as **QoS 2, retained messages or persistent sessions** may not consistently be implemented. In contrast, HiveMQ remains

wholly conformed to the MQTT specification, comprehensively supporting all features. Pivoting toward the subject of unsupported features, MQTT 5 presents an ingenious resolution. It enables those implementations that are not entirely complete, often seen in SaaS offerings, to indicate the broker's inability to support certain features. The onus is then placed upon the client to ensure these unsupported features are not utilized. The broker employs pre-defined headers in the CONNACK packet, sent in response to the client's CONNECT packet, to communicate the lack of support for specific features. These headers can also be leveraged to notify the client that they are not permitted to use certain features.

What are the Pre-Defined Headers Available in MQTT 5?

Here are the pre-defined headers available in MQTT v5 for indicating unimplemented features or features not authorized for client use:

Pre-Defined Header	Data Type	Description
Retain Available	Boolean	Are retained messages available?
Maximum QoS	Number	The maximum QoS the client is allowed to use for publishing messages or subscribing to topics
Wildcard available	Boolean	If Wildcards can be used for topic subscriptions
Subscription identifiers available	Boolean	If Subscription Identifiers are available for the MQTT client
Shared Subscriptions available	Boolean	If **Shared Subscriptions** are available for the MQTT client
Maximum Message Size	Number	Defines the maximum message size a MQTT client can use
Server Keep Alive	Number	**The Keep Alive Interval** the server supports for the individual client

These return codes represent a significant stride forward in expressing the permissions of individual MQTT clients. However, this new capability carries an inherent trade-off; MQTT clients must implement the interpretation of these codes independently, and must ensure that application developers avoid using features unsupported by the broker, or those the client lacks permission for.

> As a note of assurance, HiveMQ is fully compliant with all MQTT 5 features, ensuring these custom headers would only be utilized at the administrator's discretion for setting permissions in deployments.

Enhanced Session Management: From Clean Session to Clean Start in MQTT 5

The concept of "Clean Session," a prominent feature in MQTT 3.1.1, is now succeeded by "Clean Start" in MQTT v5. The Clean Session feature was highly used in MQTT 3.1.1 by clients that either had temporary connections or didn't subscribe to messages. Upon connecting to the broker, the client was required to send a **CONNECT packet** with either the Clean Session flag enabled or disabled. If enabled, it indicated to the broker that all client data should be discarded when the underlying TCP connection was severed or upon the client's decision to disconnect from the broker. Moreover, if a previous session was associated with the client identifier on the broker, a Clean Session CONNECT packet compelled the broker to discard the prior data.

MQTT 5 introduces the Clean Start option, signaled by the Clean Start flag in the CONNECT message. With this flag, the broker dismisses any prior session data, and the client initiates a new session. However, the session isn't automatically cleared when the TCP connection is closed between the client and the server. To prompt the deletion of the session post-disconnection, a new header field, **Session Expiry Interval**, must be set to 0.

This revised Clean Start functionality enhances and simplifies MQTT's session handling, offering more flexibility and easier implementation compared to the former Clean Session/

persistent session concept. In MQTT 5, all sessions persist unless the "Session Expiry Interval" is set to 0. Session deletion takes place either after the interval timeout or when the client reconnects using Clean Start.

AUTH Packet in MQTT 5?

MQTT 5 brings forth a new packet type: the AUTH packet. Serving as a vital tool in implementing **non-trivial authentication mechanisms**, we anticipate this packet to be integral in production environments. It's crucial to understand that this novel packet can be dispatched by both brokers and clients post-connection establishment. It enables the use of intricate challenge/response authentication methods, such as SCRAM or Kerberos as outlined in the **SASL framework** and is also compatible with cutting-edge IoT authentication methods like **OAuth**. Importantly, the AUTH packet allows for the re-authentication of MQTT clients without requiring the termination of the connection.

MQTT 5: Enriching MQTT with UTF-8 String Pairs

To accommodate the custom headers, a new data type was necessitated, and thus, the UTF-8 string pairs were introduced. Essentially, a string pair is a key-value structure wherein both the key and value are of the String data type. At present, this data type is primarily used for custom headers.

This novel addition expands the spectrum of MQTT data types utilized on the wire to a total of seven:

1. Bit
2. Two Byte Integer
3. Four Byte Integer
4. UTF-8 Encoded String
5. Variable Byte Integer
6. Binary Data
7. UTF-8 String Pair

> For most application users, Binary Data and UTF-8 Encoded Strings remain the go-to data types within their MQTT library APIs. However, with the onset of MQTT 5, the UTF-8 String Pairs are anticipated to gain frequent usage.

While the remaining data types might not be directly visible to users, they are leveraged on the wire to construct valid MQTT packets by MQTT client libraries and brokers.

How Does MQTT 5 Enhance Communication with Bi-directional DISCONNECT Packets?

Under the MQTT 3.1.1 framework, clients could gracefully terminate their connection by dispatching a DISCONNECT packet before closing the underlying TCP connection. However, there was no provision for the MQTT broker to inform the client of a potential issue that necessitates closing the TCP connection. This shortcoming has been addressed in the new protocol version.

In the enhanced MQTT, the broker is authorized to transmit a DISCONNECT packet before severing the socket connection. This provision empowers the client to understand the cause behind the disconnection and devise a suitable response accordingly. While the broker is not mandated to disclose the exact reason (for security purposes, for instance), this feature benefits developers as it provides valuable insight into why the broker terminated a connection.

Another valuable addition is that DISCONNECT packets can now carry Reason Codes, simplifying the process of revealing the disconnection's rationale, such as in the case of invalid permissions.

Revamping QoS 1 and 2 in MQTT 5: Eliminating Retries for Healthy Connections

MQTT employs persistent TCP connections (or similar protocols with identical assurances) for the underlying transport. A robust TCP connection ensures bi-directional connectivity with exactly-once and in-order guarantees, meaning all MQTT packets dispatched by clients or brokers will be received at the other end. When the TCP connection is disrupted while the message is in transit, **QoS 1 and 2** ensure message delivery across multiple TCP connections.

Under the MQTT 3.1.1 protocol, re-delivery of MQTT messages was permissible even when the TCP connection was healthy. This often proved detrimental in practice, risking overloading

already burdened MQTT clients. Consider a scenario where an MQTT client takes 11 seconds to process a message received from a broker to acknowledge the packet after processing. There is no tangible advantage if the broker retransmits the message after a 10-second timeout. This approach merely consumes valuable bandwidth and further overwhelms the MQTT client.

With the advent of MQTT 5, retransmission of MQTT messages for healthy TCP connections is disallowed for both brokers and clients. However, brokers and clients must resend unacknowledged packets when the TCP connection has been severed. Thus, the QoS 1 and 2 guarantees remain as vital as they were in MQTT 3.1.1.

If your use case depends on retransmission packets (for instance, if your implementation fails to acknowledge packets under certain circumstances), we recommend reevaluating this strategy before upgrading to MQTT v5.

How Does MQTT 5 Simplify Authentication?

In the MQTT 3.1.1 protocol, MQTT clients needed to provide a username when utilizing a password in the CONNECT packet. This proved inconvenient for some use cases where a username was unnecessary. One prominent example includes using **OAuth**, which leverages a JSON web token for authentication and authorization information. With MQTT 3.1.1, static usernames were often utilized given that the critical information was contained within the password field.

With the advent of MQTT 5, the protocol has introduced more refined ways to handle tokens, for instance, via the AUTH packet. However, using the password field of the CONNECT packet is still feasible. The major improvement here is that users can simply leverage the password field without the obligation to fill out the username. This adjustment offers a more streamlined and straightforward approach to authentication in specific scenarios.

While the core of the MQTT protocol remains relatively unchanged, subtle refinements have been implemented under the surface, laying the groundwork for many new features

in version 5 of this widely adopted IoT protocol. As a user leveraging MQTT libraries, these modifications may seem minor and do not radically alter the way MQTT is utilized. However, for developers working on MQTT libraries and brokers, these changes, particularly those related to protocol nuances, are critical and demand attention.

Some shifts in specified behavior, such as message retransmission, necessitate revisiting deployment design decisions during the transition to MQTT 5.

Chapter 14: Key Reasons to Upgrade to MQTT 5 from MQTT 3.1.1

The Internet of Things (IoT) has evolved rapidly over the last decade, with MQTT emerging as the de facto protocol for the seamless and efficient communication of IoT devices. As the scale and complexity of IoT systems grow, the MQTT protocol is also evolving to meet these demands. In this light, the MQTT 5 upgrade presents significant improvements catering to modern IoT applications' expanding needs.

Now, let's explore why your organization or development team should consider upgrading to MQTT 5 from MQTT 3.1.1.

Why Upgrade to MQTT 5 from MQTT 3?

MQTT 5 represents a substantial enhancement to the MQTT protocol, developed with valuable insights from MQTT users. It incorporates features essential for contemporary IoT applications, tailor-made for cloud-based deployments. These advancements promote resilience, dependable error management for executing crucial messaging, and facilitate the straightforward integration of MQTT messages into existing computational frameworks.

In this chapter, we present seven reasons why your organization or development team should consider upgrading to MQTT 5; they include:

1. Better Error Handling for More Robust Systems
2. More Scalability for Cloud Native Computing
3. Greater Flexibility and Easier Integration
4. The Rise of a Single IoT Standard
5. Future-Proofing

6. Streamlined Migration
7. Improved features, including
    1. Negative Acknowledgments
    2. Topic Aliases for Efficiency
    3. User Properties for Customization
    4. Payload Format Indicators

Here is the rationale behind how and why MQTT 5 is better compared to MQTT 3.

1. How MQTT 5 is Better Than MQTT 3 in Error Handling

MQTT 5 significantly enhances error handling mechanisms, contributing to the robustness and reliability of the system. One of the notable introductions is the **session and message expiry** feature. This allows developers to set a defined time limit for each message and session. If a message isn't delivered within this time frame, it's automatically deleted. This feature is particularly useful in ensuring that network latency or outages do not lead to delivering outdated or irrelevant commands to IoT devices.

2. How MQTT 5 is Better Than MQTT 3 in Offering Greater Scalability

Scalability is a crucial requirement in the modern world of growing IoT networks. MQTT 5 addresses this by standardizing shared subscriptions. This allows multiple MQTT client instances to share the same subscription on the broker. It's a powerful feature for load-balancing MQTT clients deployed on a cloud cluster. This scalability makes MQTT 5 a robust choice for enterprises with large deployments and those planning to scale up their IoT systems.

3. How MQTT 5 is Better Than MQTT 3 in Offering Greater Flexibility and Easier Integration

Taking customization to a new level, MQTT 5 introduces User Properties, a feature that allows for adding key-value properties to the headers of MQTT messages. This capacity means that application-specific information can be incorporated directly into each message, enhancing the processing of these messages. For instance, a meta-tag that includes the unique identifier of the sending client or the

51

firmware version of the device platform can be added to the message header, facilitating analysis and processing by the receiver.

Additionally, MQTT 5 simplifies the process for the receiver by incorporating Payload Format Indicators. Now it's easier to differentiate between binary or text data, as MQTT 5 includes a MIME-style content type descriptor. This feature provides immense value across a plethora of use cases. Consider, for example, a toll road control system transmitting images of license plates for image recognition processing. In contrast, other messages might require a different processing style, like those including location coordinates. By specifying the format, MQTT 5 ensures efficient and appropriate processing of diverse data types.

4. How MQTT 5 is Better Than MQTT 3 as an IoT Standard

With its rich feature set, MQTT 5 has cemented its place as the go-to choice for diverse IoT use cases, successfully addressing the limitations of previous versions. We anticipate exponential MQTT adoption across industries in the coming years, hinting at MQTT's impending status as the universal IoT standard.

5. How MQTT 5 is Better Than MQTT 3 in Future-Proofing Your Applications

By addressing the limitations of MQTT 3, MQTT 5 paves the way for future enhancements in IoT applications. Its flexible and robust features make it adaptable to upcoming technology trends, ensuring your IoT systems stay current and ready for future advancements.

6. How MQTT 5 is Compatible With its Predecessors

One of the practical advantages of MQTT 5 is its compatibility with its predecessors. It supports MQTT 3.1 and MQTT 3.1.1 features, allowing a mix of MQTT 3 and MQTT 5 clients. This makes the migration process seamless, enabling organizations to gradually transition to MQTT 5 without disrupting existing IoT operations.

7. How is MQTT 5 Better Than MQTT 3 in Offering Improved Features

In this section, we delve into an array of MQTT5's enhanced features. From bolstering system robustness with Negative Acknowledgments to enabling customization with User Properties and boosting efficiency with Topic Aliases, MQTT 5 has significantly emphasized usability, flexibility, and performance. Moreover, Payload Format Indicators have been introduced to facilitate easy handling of diverse data types. Let's explore each of these advancements in detail:

Negative Acknowledgments

In a further bid to bolster system robustness, MQTT 5 incorporates the concept of **negative acknowledgments**. The broker can reject specific messages under predefined conditions or restrictions, such as exceeding the maximum message size, maximum quality of service (QoS), or using unsupported features. This proactive measure guards against MQTT clients that might start sending erroneous or malicious messages, significantly enhancing the overall system's security.

Topic Aliases for Efficiency

In large systems with complex topic structures, topic strings can become long, increasing network and processing load. MQTT 5 introduces topic aliases, which lets you replace these long topic strings with integer values. This can considerably reduce the demand on the network and the processing overhead, thereby boosting system efficiency and performance.

User Properties for Customization

MQTT 5 goes a step further in customization by introducing User Properties. These allow developers to add key-value properties to the message header of an MQTT message. This level of customization enables you to include application-specific information within each message. This data could be used to enrich message processing and integration, giving developers more flexibility and control over their applications.

Payload Format Indicators

With the diverse types of data in IoT applications, it's crucial to have a mechanism that simplifies data handling and processing. MQTT 5 meets this need with payload format indicators. These indicators identify whether the payload is binary or text and include a MIME-style content type. This helps improve data processing efficiency and makes the system more adaptable to various data use cases.

Chapter 15: MQTT Session Expiry and Message Expiry Intervals

In this chapter, we focus on two closely-related features: the **Session Expiry Interval and the Message Expiry Interval**. These functionalities represent key improvements in MQTT 5, and a **thorough understanding of them is crucial for optimal protocol utilization**.

How do Session Expiry Interval and Message Expiry Interval Work in MQTT 5?

Let's break down and analyze these two integral expiry interval features individually.

Session Expiry Interval in MQTT 5

The Session Expiry Interval is a parameter a client sets during the CONNECT packet stage, specified in seconds. This parameter indicates the duration the broker retains the client's session information. If this interval is set to zero, or if the CONNECT packet does not specify an expiry value, the session data is promptly deleted from the broker as soon as the client's network connection terminates.

Notably, the maximum session expiry interval is UINT_MAX (4,294,967,295), enabling an offline session to persist for an extended duration of just over 136 years following client disconnection.

The Session Expiry Interval can be defined in the connect packet

Message Expiry Interval in MQTT 5

Within MQTT 5, clients can set a unique Message Expiry Interval in seconds for each PUBLISH message. This interval establishes the duration the broker preserves the PUBLISH message for subscribers that match the topic but are currently offline. If the interval isn't defined, the broker must indefinitely hold the message for matching subscribers, yet disconnected subscribers. Furthermore, if the 'retained=true' option is selected during the PUBLISH message, the interval also dictates the length of time a message is retained on a particular topic.

Publish Packet with Message Expiry Interval set to 120 seconds. The retainFlag is set to "true", so the message will also be retained for 120 seconds.

Why Were the Expiry Intervals Introduced in MQTT 5?

Understanding the functionality of these expiry intervals merely scratches the surface. It's also essential to explore the reasons that prompted incorporating these features into the MQTT 5 specification, along with their practical applications.

In earlier chapters, we captured the spirit of the OASIS committee's deliberation process, which culminated in the establishment of MQTT 5 as a standard in March 2019. HiveMQ, a proud member of this committee, actively gathered user feedback, comprehended the needs of long-standing users of the MQTT 3.1 and 3.1.1 versions, and determined how best to evolve the protocol. The main driving force was to introduce features that expanded the realm of possibilities for users and enhanced the simplicity of their interactions with the protocol.

MQTT 5's Session Expiry Interval

The Session Expiry Interval in MQTT 5 masterfully fulfills two critical needs simultaneously. Earlier versions of MQTT offered a single avenue for removing **persistent sessions** as defined by the specification: a client bearing the same client ID as
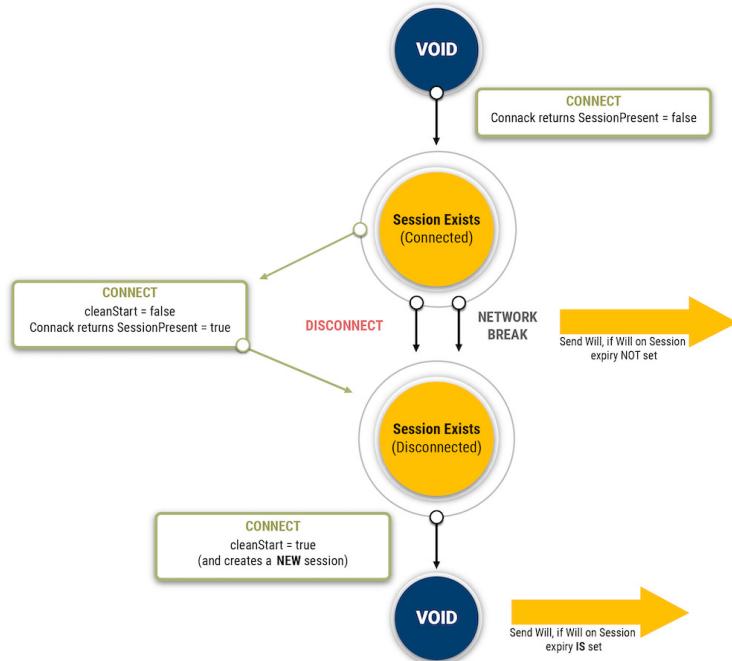
the session to be discarded would have to connect with the cleanSession=true flag set.

Consider scenarios where some IoT devices never reconnect due to decommissioning, destruction, or leftover sessions from inadequate cleanups post-load-testing. These residual sessions could impose an unnecessary burden on a broker's persistence. Enterprise-grade MQTT brokers like HiveMQ come equipped with sophisticated administrative tools such as the **HiveMQ Control Center** to facilitate the management of idle sessions. Nonetheless, it is still crucial to discern which sessions can be effectively removed.
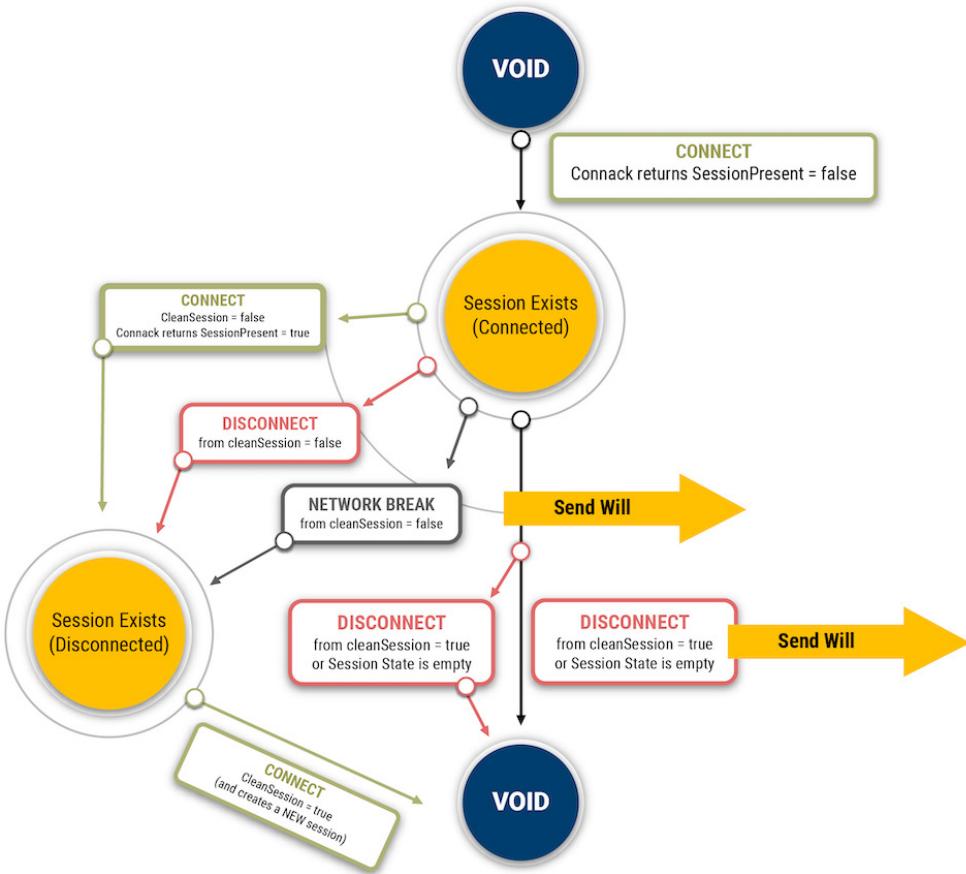
> Enter the session expiry interval feature of MQTT 5. This intuitive feature enables users to specify a sensible duration, after which the broker automatically purges an idle session, liberating valuable resources.

Beyond this automated cleanup capability, the session expiry interval has significantly simplified session state management. Ian Craggs graciously provided two diagrams illustrating the reduction in complexity in state transitions. This visual representation underscores how this new feature aids in streamlining state transitions and bolstering user efficiency.

MQTT V5 State Transition

MQTT V3.1.1 State Transtition

De-cluttering of state transition in MQTT 3.1.1 and MQTT 5 (courtesy of Ian Craggs)
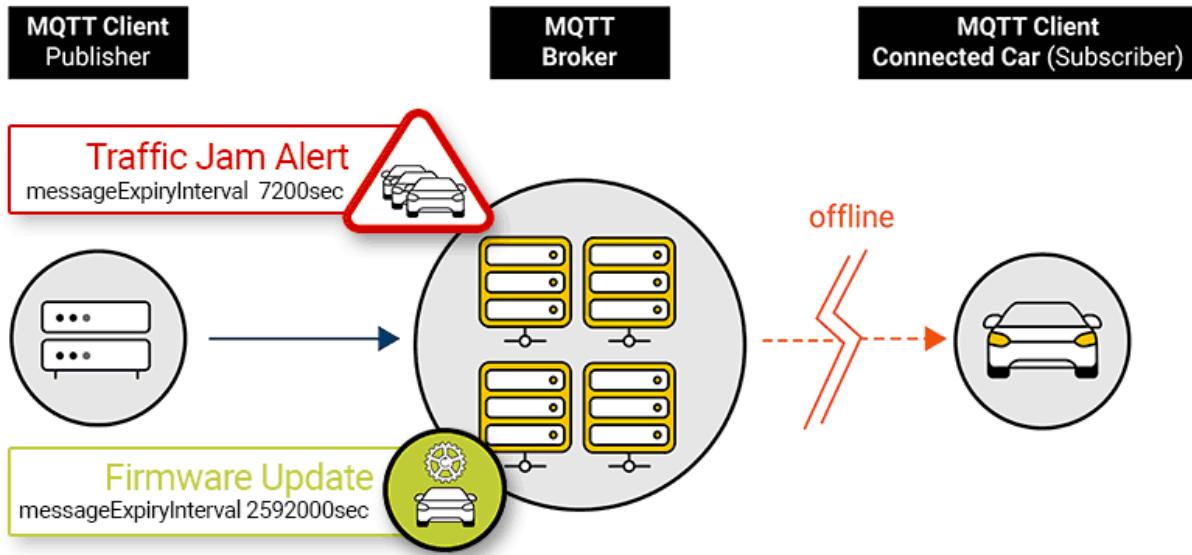
MQTT 5's Message Expiry Interval

Much like its counterpart, the session expiry's interval inception was fueled by a need for an automated maintenance mechanism. **Consider the myriad of IoT devices, such as connected cars designed to weather extended periods of internet disconnection. MQTT offers a lifeline for these scenarios with persistent sessions and message queueing**.

Messages crafted for offline devices are stored on the broker, waiting for the device to regain connectivity for delivery. In grand-scale deployments, where the count of connected devices escalates into thousands or even millions, it's imperative to constrain the offline message queue for each client individually.

The longevity of message relevance to IoT devices can exhibit significant fluctuations. Take the example of the connected car: traffic updates maintain their relevance only briefly. Yet, when we consider over-the-air firmware upgrades, these need to be executed even if the car remains offline for an extended period, stretching into several weeks.

> The Message Expiry Interval in MQTT 5 emerges as the perfect tool to manage these differing time frames, adding to the protocol's versatility.

Example: traffic jam alerts usually become obsolete after 1-2 hours, but a firmware update should be available for many weeks. Message Expiry Interval is the perfect feature to define these different periods of time.

By assigning an optimal message expiry interval to messages with a limited relevance window and leaving perpetual relevance messages without an expiry, we ensure effective utilization of broker resources for clients that could be offline for a considerable duration. This strategy also spares clients from being inundated with irrelevant messages upon reconnection.

> For **retained messages**, the message expiry interval operates similarly, guaranteeing that these messages are only dispatched to new subscribers for a specified period.

One crucial aspect to note, however, is that when a client's session expires, all queued messages for that client also expire in sync with the session, irrespective of their individual message expiry status. This "GOTCHA" is an important reminder of the interconnectedness of sessions and their queued messages in the MQTT protocol.

Important Tips for MQTT 5 Session Expiry Interval and the Message Expiry Interval

Here're some important information:

- Both the Session Expiry Interval and the Message Expiry Interval are instrumental in refining resource management on the MQTT broker.
- Reflecting on the past, many MQTT 3 users expressed a need for expiry features. HiveMQ responded to this demand by introducing session and message expiry as supplementary features in our MQTT platform, predating their standardization in MQTT 5.
- The equivalent for cleanSession=true in the CONNECT packet of MQTT 3 is sessionExpiry=0 (or absent) and cleanStart=true in MQTT 5.
- Likewise, cleanSession=false from the MQTT 3 CONNECT packet finds its counterpart in MQTT 5 as a sessionExpire value exceeding zero, coupled with cleanStart=false.
- MQTT brokers, such as HiveMQ, offer the capability to configure a maximum value for these expiry intervals on the **server side**. This feature is handy in multi-vendor projects, particularly when the broker operator may lack authority over the MQTT client settings.

The advent of MQTT 5 has ushered in a host of new features designed to enhance the protocol's usability, flexibility, and efficiency. The Session Expiry Interval and the Message Expiry Interval are prime examples of this, acting as invaluable tools in resource management and ensuring the smooth functioning of your MQTT broker. Both these features truly embody the user-centric evolution of the MQTT standard, demonstrating its responsiveness to the demands and challenges of its users.

Chapter 16: MQTT 5's Improved Client Feedback & Negative ACKs

In this chapter, we focus on MQTT 5's Enhanced Client Feedback and Negative ACKs(NACKs). We will dissect this method that MQTT 5 implements for MQTT brokers to acknowledge clients, examining their potential to simplify the work of developers and operations teams, and making the protocol more robust and efficient.

Who Needs More Client Feedback While Using MQTT?

Over the years, MQTT has become a staple in numerous IoT projects. As noted in this eBook earlier, the OASIS committee thoroughly considered the feedback from actual protocol users while crafting the new MQTT 5 protocol standard. Among the predominant grievances was a glaring lack of transparency, primarily due to the insufficient return codes and limited means to communicate specific limitations or circumstances from the broker to the client. This deficiency resulted in increased debugging complexity, particularly in multi-vendor projects.

Whether it's delving into the causes of client disconnects, examining why messages fail to reach their designated targets, or guaranteeing consistency in MQTT client deployments across various teams, MQTT 3 users often find the need to augment the basic protocol features with technology, such as the **HiveMQ Extension SDK**. MQTT 5 has been deliberately designed to address these challenges more efficiently and in a standardized manner by introducing the following features.

MQTT 5's Specific Feature Set

Let's focus on several MQTT 5 features that provide more transparency and allow centralized system control by virtue of the broker.

Feedback on Connection Establishment in MQTT 5

With MQTT version 5, it is now possible for MQTT brokers to give additional feedback to MQTT clients on connection establishment. A number of different properties can be added to the connection acknowledgment packet that tells the client which features the broker supports or the client is allowed to use. This includes the following MQTT features:

• Retained messages
• Wildcard subscriptions
• Subscription identifiers
• Shared subscriptions
• Topic aliases
• Maximum quality of service level the client can use

In addition to notifying the client about enabled and disabled features, the new properties in the CONNACK packet also allow the server to give the client feedback on the limits granted to it by the broker. These limits include:

• Keep Alive
• Session expiry interval
• Maximum packet size
• Maximum number of topic aliases the client can send

Beyond supporting all of these limits, HiveMQ allows you to configure a maximum for the limits and to disable MQTT features that are not needed in your use case.

Better Reason Codes in MQTT 5

In previous iterations, namely MQTT version 3.1 and 3.1.1, the assortment of available reason codes was somewhat limited, with only five codes pertaining to unsuccessful operations. However, MQTT 5 substantially enhances this aspect, offering an expanded set of over 20 reason codes dedicated to unsuccessful scenarios.

Moreover, MQTT 5 introduces the possibility of integrating reason codes into packets that previously lacked this attribute in versions 3.1 and 3.1.1. These packets include UNSUBACK,

PUBACK, PUBREC, PUBREL, DISCONNECT, and PUBCOMP. This enhancement further strengthens the protocol's transparency and troubleshooting efficiency, illustrating the forward strides made in MQTT 5.

Enhancing Clarity with Contextual Reason Strings in MQTT 5

While adding additional reason codes certainly bolsters the feedback quality to clients, these codes often fall short in providing specific contexts. This is where MQTT 5 introduces the concept of 'reason strings' to bridge the gap, described in the specifications as, "… a human readable string designed for diagnostics …"

Reason strings offer a comprehensive context that developers and operation teams need to swiftly pinpoint the cause of an event. In essence, a reason string is a human-readable string meticulously designed for diagnostic purposes. This valuable tool aids in understanding the nuances of events in a way that reason codes alone can't provide.

However, it's worth noting that reason strings, despite their usefulness in development and diagnostics, can be deactivated within HiveMQ's configuration. This flexibility caters to scenarios where the exposure of such detailed information might not be preferred.

Server-Sent Disconnect Packets in MQTT 5

In MQTT 3.1 and 3.1.1, when a client surpasses a broker-defined limit, the broker responds by abruptly closing the client's connection. However, this action doesn't provide any direct insight to the client regarding the reason for the connection termination, which can lead to confusion and troubleshooting difficulties.

In contrast, MQTT 5 significantly improves this process by introducing server-sent disconnect packets. These allow the broker to deliver a DISCONNECT packet to the client before terminating the connection. Each server-sent disconnect packet includes a reason code and a corresponding reason string. These two elements work together to give the client

a comprehensive understanding of why the connection was severed.
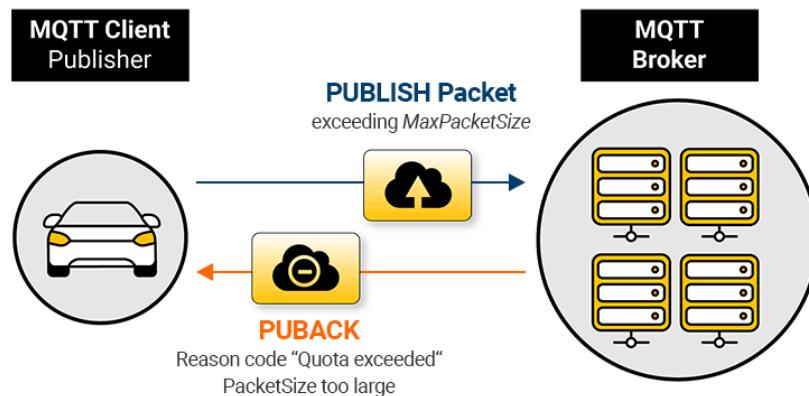
This streamlined method of connection termination not only adds clarity for the client but also substantially simplifies pinpointing the reason behind a broker-initiated connection closure. This significant enhancement in MQTT 5 dramatically improves the communication transparency between broker and client.

What are Negative Acknowledgments in MQTT 5

In MQTT 5, the communication mechanisms have been greatly improved with the addition of negative acknowledgments. Multiple packet types and message flows can now respond with a negative acknowledgment message, enhancing the messaging infrastructure's overall feedback loop.

Contrasting with MQTT 3, where the UNSUBACK packet sent to the client lacked a payload, leaving the client uninformed in case of an unsuccessful UNSUBSCRIBE attempt, MQTT 5 fundamentally changes this. The UNSUBACK packet in MQTT 5 includes a **reason code** informing the client about its UNSUBSCRIBE attempt's status, providing clear and actionable feedback. Several potential failure reasons are enumerated, such as the nonexistence of the initial subscription or a lack of client authorization to unsubscribe.

Acknowledgement packets from the publish flow, specifically PUBACK, PUBREL, PUBREC, and PUBCOMP, have also been enhanced with the ability to send a negative acknowledgment message. This ability is crucial when the server cannot process a message sent by the client, for example, due to the client not having the required authorization to publish on a certain topic. The enhanced ack packet now equips the client with all the necessary information to adapt and rectify the situation, reducing the need to contact operations or support teams to understand the issue. This marks a significant leap forward in maintaining efficient and smooth communication within MQTT 5 applications.

Exceeding the maxPacketSize is one of the new reason codes for negative acknowledgments.

MQTT 5: Refining MQTT Communication

- MQTT brokers such as HiveMQ let you set a **server side max value configuration for the mentioned limits**. This is extremely useful in multi-vendor projects in which the operator of the broker may not have control over the settings of the MQTT clients.
- The improved feedback for clients significantly simplifies diagnosis for development and operations.
- The added transparency also improves the interoperability between different MQTT clients and brokers.

Notably, the advancements in feedback for clients substantially streamline the diagnostic process for both development and operations teams. This enhanced feedback mechanism resolves issues faster and prevents potential bottlenecks in your MQTT systems.

Moreover, the transparency introduced with these improvements fosters better interoperability between different MQTT clients and brokers. This attribute is fundamental in creating robust and versatile IoT ecosystems, and it further emphasizes the strides MQTT 5 has made toward enhancing communication, debugging, and overall system control.

Chapter 17: MQTT User Properties

In this chapter, we guide you through another groundbreaking feature: User Properties. This addition is poised to revolutionize how you interact with MQTT, creating a more customized and insightful user experience.

In essence, User Properties are the user-defined properties that help you add metadata to MQTT messages and help transmit additional user-defined information.

Let's dive in.

MQTT 5 has been meticulously crafted to bridge the existing chasm between the offerings of MQTT 3.1.1 and the evolving expectations of users from this de facto standard protocol for the Internet of Things. With MQTT 5, we're ensuring that MQTT continues to lead the pack as the IoT protocol par excellence for many more decades.

How User Properties in MQTT 5 Work?

In MQTT 5, User Properties fundamentally operate as straightforward UTF-8 string key-value pairs. Their utility lies in their ability to be affixed to nearly every category of MQTT packet, with the sole exceptions of PINGREQ and PINGRESP. This broad application extends to various control packets like PUBREL and PUBCOMP.

The power of User Properties is in their uncapped potential - as long as the maximum message size isn't exceeded, you are free to employ an infinite number of User Properties. This opens up vast possibilities for enriching MQTT messages

with additional metadata, facilitating a fluid transmission of information between the publisher, broker, and subscriber.

> Conceptually, this feature closely mirrors the role of headers in HTTP. It's this resemblance that allows User Properties to inject a level of customizable complexity into MQTT 5, helping to create a protocol that is not only more robust but also more adaptable to user needs.

Why Were User Properties Introduced in MQTT 5?

MQTT 3 users identified two significant limitations: the protocol's constrained extensibility and the complexity of creating multi-vendor deployments. To rectify these concerns, MQTT 5 introduced the User Properties feature, effectively mitigating these challenges.

User Properties provide an avenue for enhancing flexibility by enabling users to transport virtually any piece of information across the entire MQTT system. This capability ensures that the MQTT protocol no longer restricts but promotes customized enhancements. This leap forward in functionality paves the way for users to augment standard protocol features, tailoring them to meet their specific requirements.

In doing so, MQTT 5 ensures that the protocol evolves in step with its users, facilitating greater adaptability and easing the integration of multi-vendor deployments.

Practical Use Case Examples of MQTT 5 User Properties

While the intricacies of the User Properties feature might
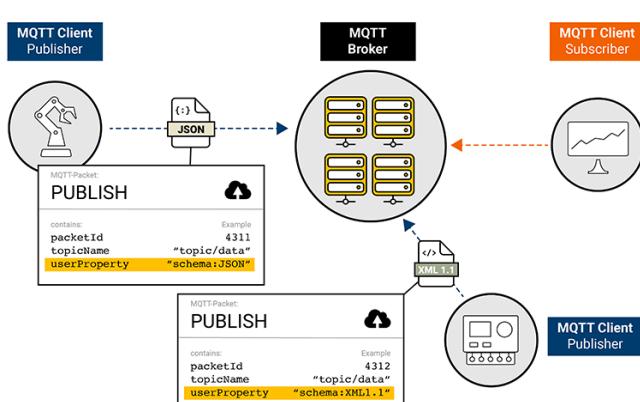
initially seem minor, the practical implications of possessing a mechanism to transfer metadata across the complete MQTT ecosystem are indeed substantial. To illustrate this transformative potential, let's delve into three common use cases that underscore the need for a feature like User Properties—a need articulated repeatedly by users eagerly awaiting the introduction of this component in the MQTT specification.

Saving Resources with Payload Metadata Using User Properties in MQTT 5

In environments where MQTT serves as a connector between diverse systems developed by different teams or vendors, variability in payload structures is quite commonplace. Clients may transmit data in many formats, including JSON, XML, or compressed formats such as Protobuf.

The advent of User Properties in MQTT 5 opens the door to appending metadata to messages, encapsulating specific details such as the markup language and version employed to encode the payload. This metadata provision obviates the need for the receiving client, or in certain instances, the broker, to unpack the payload and cycle through an array of possible parsers until the appropriate one is located.

Instead of this cumbersome process, each message arrives equipped with its parsing information, streamlining interpretation and significantly reducing the computational load across the entire system. This efficient use of resources amplifies the overall performance and speed of the MQTT network, showcasing the transformative power of User Properties.
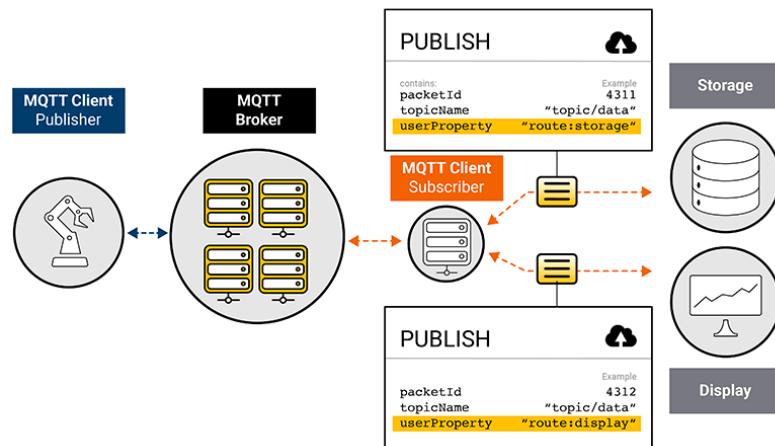
Including metadata about the used markup language in the payload can significantly relieve the system.

Increased Efficiency Through Application Level Routing Using User Properties in MQTT 5

With its proficiency in data transportation and routing, MQTT frequently serves as the backbone for large-scale data processing and streaming deployments. Such deployments typically involve a multitude of devices, systems, and applications. It's quite common for different systems to receive identical messages but for distinct purposes. For instance, one system may display live data while other archives the same data for long-term storage.

In such scenarios, User Properties can prove invaluable by serving as an additional application-level timestamp for the message. This attribute lets the broker quickly ascertain whether certain messages should not be passed to a specific subset of subscribers based on a predefined validity period. This feature introduces an added application-level layer that further refines message relevance according to the **Message Expiry Interval**.

Therefore, User Properties in MQTT 5 not only bolster the system's efficiency but also provide a finer level of control, thereby maximizing the utility and relevancy of each transmitted message.
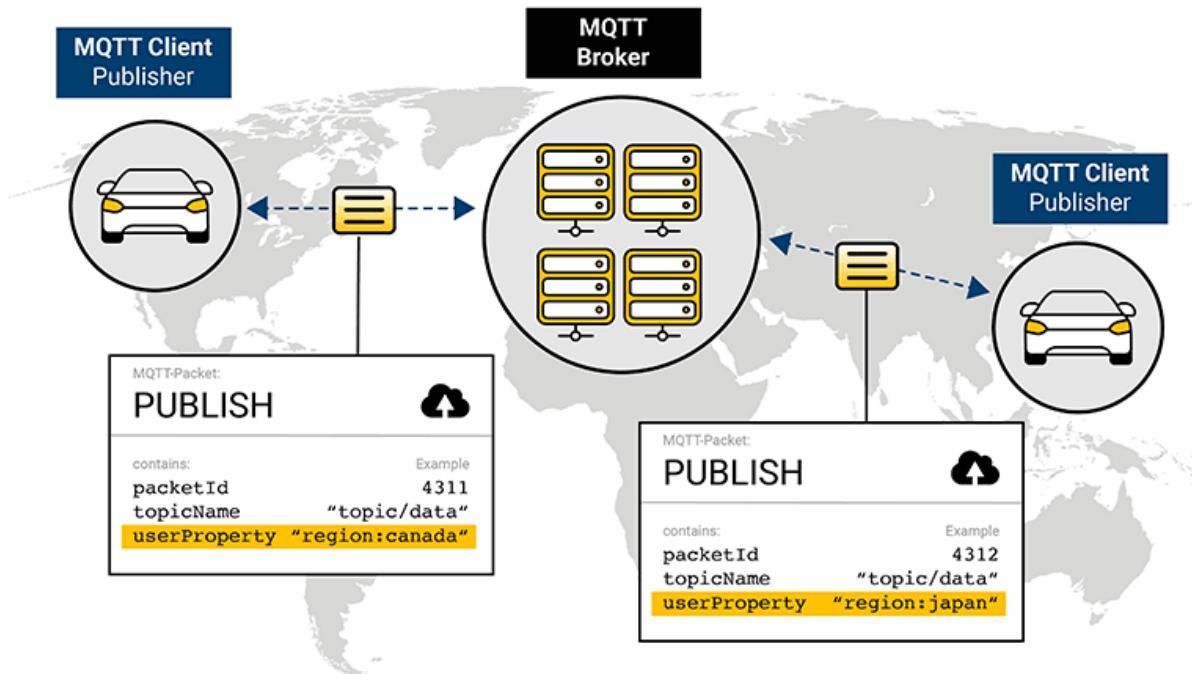
Example: user properties provide information for the broker, if messages should be routed to a storage or a display application.

Transparent Traceability in Complex Systems Using User Properties in MQTT 5

The landscape of IoT deployments often presents a maze of intricacy, with individual systems operating in parallel. Such complexity can cloud the origin of a specific message or the factors contributing to an unsuccessful multi-layer message flow. Under the MQTT 3.1.1 framework, there was no mechanism for a subscriber to discern the identity of a message's publisher. Although embedding a unique identifier in the topic is a viable strategy in 1-to-1 scenarios, this approach undermines several pivotal advantages of the publish-subscribe model.

In this regard, the advent of the User Property feature in MQTT 5 heralds a significant transformation. This innovative addition enables publishers to effortlessly include relevant self-identifying information, such as a client ID or the region where the publishing is conducted. Importantly, this information is relayed to all message recipients without necessitating any supplemental business logic.

Incorporating information about the publisher's region enhances the system's traceability while attaching a unique system identifier to MQTT messages enables comprehensive logging and tracking of the entire message flow from the sender to all subscribers. Implemented effectively, these identifiers can extend across multiple MQTT message flows, introducing unprecedented transparency and traceability.

Adding information about the publisher's region adds traceability to the system.

Such capabilities unlock a new horizon of possibilities, particularly for business-critical applications like premium paid services for end customers, where transparency and traceability become indispensable.

Additional Information on User Properties in MQTT 5

*   User properties serve as UTF-8 string key-value pairs that can seamlessly be incorporated into any MQTT message. This ability positions them as a versatile and invaluable addition to the MQTT protocol.
*   The implementation potential of User Properties in enhancing MQTT use cases is practically boundless. It offers a degree of customization that allows for many innovative applications, both in function and scope.
*   Deployments and projects that extend over multiple systems and vendors can leverage this feature to maintain consistency and ensure seamless communication across the entire infrastructure.
*   We are exhilarated to see the myriad of inventive ways MQTT users will harness the potential of this deceptively simple yet impactful feature.

User Properties in MQTT 5 represent a significant step forward in protocol extensibility and versatility, opening up exciting possibilities for future IoT applications.

Chapter 18: MQTT Shared Subscriptions

In this chapter, we'll delve into an especially interesting feature: Shared Subscriptions.

> Shared subscriptions, a core feature of MQTT 5, enable multiple MQTT clients to share a single subscription on the broker. In essence, this feature allows messages on a topic to be distributed among multiple clients, thereby improving load balancing and fault tolerance in an MQTT system.

Shared subscriptions: Bridging the MQTT Gap with V5

MQTT 5 was ingeniously designed to connect the gap between the functionality offered by MQTT 3.1.1 and users'

expectations for the Internet of Things' de facto standard protocol. With the integration of sought-after features such as Shared Subscriptions and **Session and Message Expiry Intervals**, MQTT 5 is poised to consolidate MQTT's standing as the go-to IoT protocol for the foreseeable future.

Due to the high demand for shared subscriptions, HiveMQ introduced this feature before the MQTT 5 specification was released. As a result, all MQTT brokers striving for 100% compatibility with the MQTT 5 specification must support this feature.

How MQTT Shared Subscriptions Work

In a standard MQTT subscription, each subscribing client is privy to a copy of each message broadcasted to that topic. With shared subscriptions, clients sharing a subscription in the same group receive messages in rotation, a process sometimes referred to as client load balancing. The message load of a single topic is distributed across all subscribers.

MQTT clients can subscribe to a shared subscription using standard MQTT mechanisms. Any **standard MQTT clients**, such as **Eclipse Paho**, can participate without requiring any modifications on the client side. It's important to note that shared subscriptions employ a unique topic syntax for subscribing.

Shared subscriptions use the following topic structure:

$share/GROUPID/TOPIC

The shared subscription consists of 3 parts:
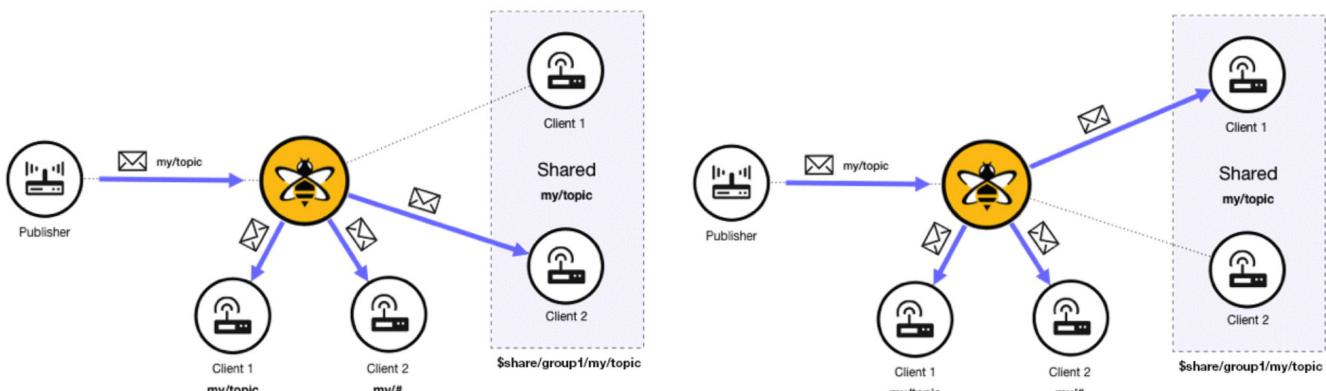
- A static shared subscription identifier ($share)
- A group identifier
- The actual topic subscriptions (may include wildcards)

A concrete example for such a subscriber would be:

$share/my-shared-subscriber-group/myhome/groundfloor/+/temperature
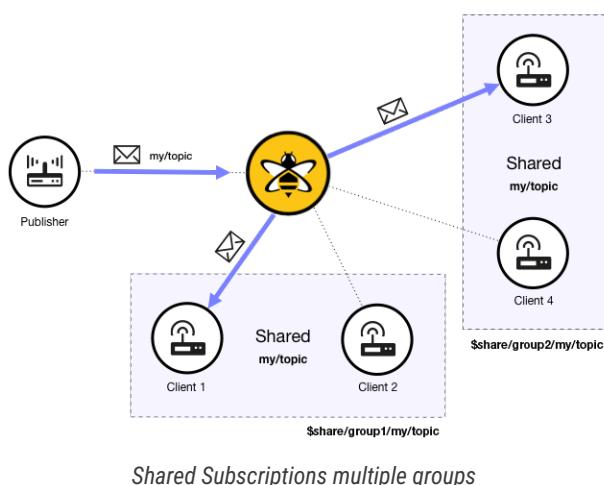
How Do MQTT Shared Subscriptions Work in HiveMQ MQTT Broker?

A shared subscription group can be conceptually envisaged as a virtual client acting as a proxy for numerous subscribers simultaneously. HiveMQ selects one subscriber from the group and delivers the message to that client. It typically uses a round-robin approach for distribution. The following picture demonstrates the principle:

Shared Subscriptions

For instance, a HiveMQ deployment might feature multiple shared subscription groups. These groups can have the same subscription but different group identifiers. Whenever a publisher sends a message with a matching topic, one unique client from each group receives the message. For example, the following scenario is possible:

Shared Subscriptions multiple groups

In the given scenario, we have two distinct groups, each encompassing two subscribing clients within their shared subscription group. Although these groups subscribe to the same topics, they are differentiated by unique group identifiers. When a publisher issues a message that aligns with a particular topic, a solitary client from each group — and only one client — is selected to receive the message.

MQTT Shared Subscription Use Cases

Shared subscriptions have a multitude of applications, particularly in high-scalability scenarios. These include:

- Client load balancing for MQTT clients that cannot handle the load on subscribed topics.
- Worker (backend) applications that ingest MQTT streams must scale horizontally.
- Optimizing subscriber node-locality for incoming publishes to reduce HiveMQ intra-cluster node traffic.
- The delivery semantics employ **QoS 1 and 2** though there's no necessity for guarantees on ordered-topic.
- Addressing hot topics causing scalability bottlenecks due to higher message rates.

How to Subscribe with Shared Subscriptions in MQTT?

Engaging your clients with shared subscriptions is a straightforward process. Below, we illustrate how to accomplish this using the MQTT CLI. The given command line execution code allows two MQTT clients to subscribe to the same subscription group and topic:

```
mqtt sub -h broker.hivemq.com -t
'$share/group1/my-share-topic' -i
client1 -q 1

mqtt sub -h broker.hivemq.com -t
'$share/group1/my-share-topic' -i
client2 -q 1
```

With these commands executed, both MQTT clients now have a shared subscription to the '`my-share-topic`' (part of the virtual group 'group1'). In this configuration, each client is allocated half of the MQTT messages dispatched over the MQTT broker on the topic '`my-share-topic`'.

Remember, the MQTT clients can join or depart from the subscription group whenever they prefer. For instance, should a third client decide to join the group, the distribution of relevant MQTT messages becomes equally divided among all three clients, each receiving one-third of the total.

For a deeper understanding of the semantics of shared subscriptions, the official **HiveMQ Documentation** is an excellent resource.

Scaling MQTT Subscribers with Shared Subscriptions

Shared subscriptions provide a simple way to integrate backend systems with MQTT, especially when it is not feasible to use **HiveMQ's extension system** or dynamic scaling is necessary. With the shared subscriptions, you can quickly add

subscribers as needed, distributing work in a push fashion. Shared subscriptions prove immensely valuable for scaling and load-balancing MQTT clients. Furthermore, HiveMQ clusters offer additional benefits in terms of latency and scalability through internal optimization of message routing.

For comprehensive details on shared subscriptions and HiveMQ clusters, reference the official **HiveMQ Documentation.**

In summary, shared subscriptions provide a compelling way to distribute messages across various MQTT subscribers using standard MQTT mechanisms. This feature simplifies implementing MQTT-client load balancing without the need for any proprietary modifications to your MQTT clients. This is especially beneficial for backend systems or "hot-topics" that might quickly overwhelm a single MQTT client.

Chapter 19: MQTT Payload Format Description and Content Type

In this chapter, we will focus on Payload Format Indicators, which specify the message content type, ensuring easier, more efficient parsing and interoperability between systems.

What is Payload Format Indicator in MQTT?

The Payload Format Indicator is a fundamental component of any MQTT packet that houses a payload. This includes a CONNECT packet encapsulating a WILL message or a PUBLISH packet. **This optional byte value has two possible settings: a 0**

indicates an "unspecified byte stream" while a 1 represents a "UTF-8 encoded payload." When the Payload Format Indicator isn't provided, it automatically defaults to 0.
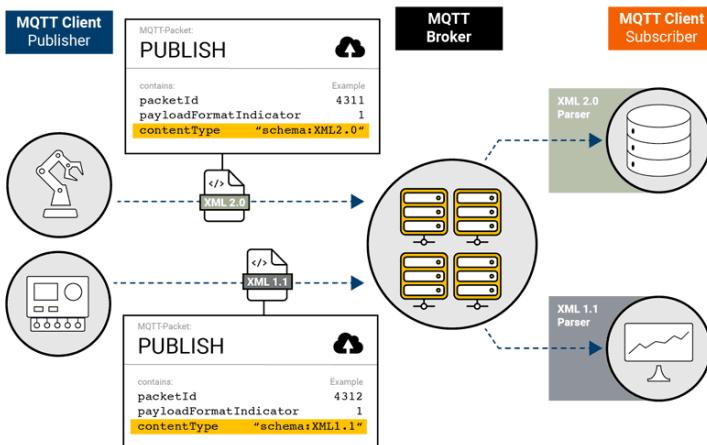
The Payload Format Description is optional. It can have the values "1" or "0".

MQTT Content Type

Similar to the Payload Format Indicator, the Content Type is also optional and can be incorporated in a CONNECT containing a WILL message or any PUBLISH packet. **The value for the Content Type must be a UTF-8 encoded string that identifies the payload's nature. When the Payload Format Indicator is set to 1, ideally, you should have a MIME content type descriptor (though it's not a hard requirement). A valid UTF-8 String is all you need.**

For UTF-8 encoded strings Content Types can be defined.

Payload Format Description allows pre-parsing without the need to open the payload

Why Describe the Payload Format?

The combined use of Payload Format Indicator and Content Type facilitates a transparent description of the payload content for any application message. This ability sets the stage for creating and defining industry-wide MQTT standards for varied payload formats. MQTT protocol experts view this standardization as the protocol's natural progression.

Having the payload content description in the headers can prove incredibly beneficial in individual deployments. It ensures every message is correctly processed without delving into the payload itself. Depending on the content type, different messages within a system may need various parsing methods. Moreover, in certain instances, message persistence could hinge on the payload's specific type. As the content-type definitions hinge on user design, the potential applications of this feature appear boundless.

In summary, the Payload Format Indicator discerns whether a payload is an undefined byte array or a UTF-8 encoded message. When dealing with UTF-8 encoded messages, the sender can use the content type to specify the payload's nature.

These features set the stage for transparent payload content definitions across large-scale systems and, potentially, entire industries. As the need for pre-parsing actual payloads diminishes, proper message processing can considerably enhance scalability.

Although it's anticipated that most users will rely on known MIME types to describe the content, they can also use arbitrary UTF-8 Strings.

Chapter 20: MQTT Request-Response Pattern

In this chapter, we spotlight two standout elements: Response Topic and Correlation Data.

Navigating the complexities of modern IoT projects demands teamwork across diverse vendors and teams. As MQTT has become the protocol par excellence for IoT, enhanced interoperability and system transparency emerged as prime requirements in the MQTT version 5 blueprint. The features we're delving into today answer these user needs by providing a standard solution for implementing a request-response pattern with MQTT.

What is Request-Response Pattern in MQTT?

MQTT is rooted in asynchronous messaging, adopting the **publish-subscribe paradigm**. This design allows senders and receivers to function independently of one another, facilitating one-to-many relationships. It's vital to grasp that MQTT's request-response pattern tackles challenges differently from synchronous, one-to-one-based protocols like HTTP.

> In MQTT, a response typically doesn't directly answer a request's "question". Instead, the request triggers a specific action in the receiver, and the response communicates the result of this action.

Sounds complicated? Don't fret; a concrete example will soon bring this into perspective!

What is Response Topic in MQTT 5?

A response topic is an optional UTF-8 string incorporated into any PUBLISH or CONNECT packet. If a value is present in the response topic, the sender immediately classifies the associated PUBLISH as a request. The response topic field indicates the topics where the message receivers' responses are expected. The initial PUBLISH (request) and the response topic can have multiple or no subscribers. Ideally, the original PUBLISH (request) sender should subscribe to the response topic before dispatching the request.

What is Correlation Data in MQTT 5?

Correlation data is optional binary data that trails the response topic. It helps the request's sender identify which specific request a later received response relates to. The correlation data allows the original request sender to manage asynchronous responses potentially sent from multiple receivers. It's important to note that this data is not relevant to the MQTT broker but serves to identify the relationship between sender and receiver.

What is Response Information in MQTT 5?
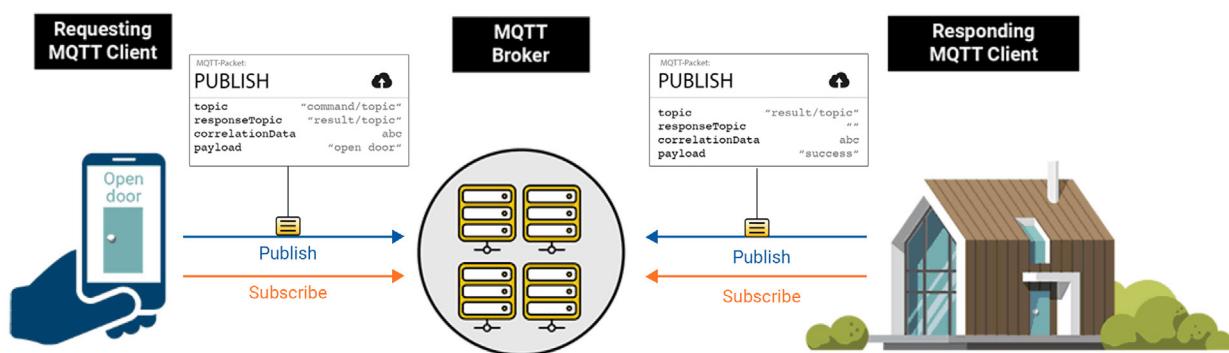
To foster transparent implementation and improved standardization, the MQTT 5 specification introduced the

Response Information property. Through a boolean field in the CONNECT, a client can request response information from the broker. When set to true, the broker can dispatch an optional UTF-8 String field (response information) in the CONNACK packet to communicate anticipated response topics.

This feature allows users to globally define a specific part of your topic tree on the broker, accessible by all clients indicating their intention to use the request-response pattern at the connection establishment.

End-to-End Acknowledgment in MQTT 5

MQTT ensures the complete decoupling of message senders and receivers. Use cases often call for an acknowledgment of message receipt from the intended recipient. For example, when opening the door of your smart home via a command, the sender (typically a mobile app) would want to know when and if the message was received and the outcome of the command.

Example: Smart door opening with a mobile device using the MQTT 5 Request Response feature.

The inclusion of the request-response pattern in the MQTT 5 specification was largely driven by the need for these "business ACKs." MQTT users sought the capability to provide end-to-end acknowledgments between an application message's sender and receiver. Adding response topics, correlation data, and response information as protocol fields significantly bolsters extensibility, dynamism, and transparency in application development using the request-response pattern.

MQTT 3 users previously embraced this pattern. By integrating response topics, correlation data, and response information directly into protocol fields, we've significantly enhanced application development's versatility, dynamism, and transparency under the request-response pattern.

Source Code Example of MQTT Request-Response Workflow

Below is a quick code snippet showcasing the HiveMQ MQTT Client's capabilities, providing a taste of the request-response pattern workflow in MQTT. Note that it's not a complete, functioning excerpt.

You can access a full example on **GitHub**.

```
//Requester subscribes to response topic
Mqtt5SubAck subAck = requester.subscribeWith()
        .topicFilter("job/client1234/result")
        .send();


//Requester publishes request
Mqtt5PublishResult result = requester.publishWith()
        .topic("job")
        .correlationData("1234".getBytes())
        .responseTopic("job/client1234/result")
        .payload(message.getBytes())
        .send();



//Responder subscribes to request topic
Mqtt5SubAck subAck = responder.subscribeWith()
        .topicFilter("job")
        .send();



//Responder sends response after receiving the request
 Mqtt5PublishResult result = responder.publishWith()
        .topic(publish.getResponseTopic().get())
        .payload(msg.getBytes())
        .correlationData(publish.getCorrelationData().get())
        .send();
```

Key Takeaways from Request-Response in MQTT

Here are a few key takeaways of request-response in MQTT:

- The request-response pattern in MQTT significantly differs from its counterpart in synchronous, client-server-based protocols like HTTP.
- MQTT allows multiple, single, or even no subscribers for requests and responses.
- Correlation data ensures the proper linking between request and response, enhancing message tracking.
- This pattern facilitates the implementation of "business acknowledgment" functionality, providing an extensible, dynamic, and transparent solution.

Best Practices to Consider While Using Request-Response in MQTT

Here are a few best practices to consider while using request-response in MQTT:

- Ensure the requester subscribes to the relevant response topic before sending a request.
- Employ unique identifiers within the response topic to avoid confusion.
- Ascertain that responders and requesters possess the necessary permissions to publish and subscribe to response topics.
- Dedicate a specific section of the topic tree for response purposes, and utilize the response information field to relay it to clients.

As we journey through the transformative features of MQTT v5, we discover the power of protocol evolution. These enhancements, like the request-response pattern, not only streamline existing practices but also unlock new realms of dynamic, transparent, and extensible application development.

Chapter 21: MQTT Topic Alias

In the previous chapter, we unraveled the **Request - Response Pattern**. In this chapter, we now direct our attention to another potentially impactful feature: Topic Alias.

What is MQTT Topic Alias?

Topic Aliases are integer values substituting topic names. It enables you to condense a lengthy and frequently utilized topic name into a 2-byte integer. This helps minimize the amount of bandwidth consumed during message publication.
Senders define the Topic Alias value in the PUBLISH message, followed by the topic name. Receivers then process this message like any other PUBLISH, establishing a mapping between the Topic Alias (integer) and Topic Name (string). Subsequent PUBLISH messages for the same topic can be sent with just the Topic Alias, omitting the topic name.

Why Use Topic Aliases in MQTT?

MQTT plays a pivotal role in your network with its efficiency in maintaining standing connections between your devices and the brokers. The **Keep Alive** mechanism guarantees the longevity of connections between clients and the broker, swiftly detecting any connection loss that could occur in unstable networks. PING packets - of just two bytes - need to be sent only every few minutes, enabling MQTT to sustain these connections with minimal power and bandwidth use.

This brings us to the Topic Alias feature, which is particularly beneficial in deployments involving a vast array of connected devices transmitting smaller, frequent messages. So, let's dive into this feature and see how it can optimize your MQTT 5 utilization.

How to Use MQTT Topic Names?

The MQTT client and the broker have the power to establish a Topic Alias for any PUBLISH message, provided they are the message's originator. Similarly, they can control the number of Topic Aliases permitted for each connection. The upper limit for Topic Aliases, known as the Topic Alias Maximum, is determined during the connection establishment phase.

The client indicates its Topic Alias Maximum in the CONNECT packet, while the broker does so in the CONNACK packet. Therefore, the client should only utilize Topic Alias values ranging from 1 to the broker-defined Topic Alias Maximum presented in the CONNACK packet. Similarly, the broker should respect the range of 1 to the client-defined maximum from the CONNECT packet.

In the absence of a specified Topic Alias Maximum, a default value of 0 is assumed, which effectively disables the use of Topic Aliases. This ensures clear communication boundaries and precise control of Topic Aliases within your MQTT deployment.

Use Case Examples of MQTT Topic Alias

MQTT stands out as a lightweight communication protocol that efficiently maintains TCP connections between clients

and brokers. Its **Keep Alive** mechanism minimizes energy and bandwidth usage, enabling users to establish cost-effective, always-connected device deployments. Moreover, it allows for real-time delivery of minimal data points, like measurements, negating the need for periodic bulk data transfers – a requirement of heavier data transfer technologies.

This inherent efficiency of MQTT finds utility in numerous applications, such as predictive maintenance, where the service's quality and responsiveness can be enhanced by transmitting small data points in real time. In these situations, an elaborate topic name might be more sizable than the actual data payload, which a single integer could represent. For instance, a topic name like '`data/europe/germany/ south/bavaria/munich/schwabing/box-32543y/ junction/consumption/current'` describes the current power consumption of a specific junction box, with the payload being a single integer value.

Topic Alias can substitute long and complex topic strings with single integers.

The Topic Alias feature of MQTT comes to the fore in these instances, replacing lengthy and complicated topic strings with single integers. This technique is especially useful when sending numerous small messages over extensive topic names in real time, offering two primary advantages: It amplifies performance while significantly reducing network traffic. Hence, Topic Alias becomes a powerful tool in your MQTT 5 arsenal, optimizing data transmission and network management.

Understanding MQTT Topic Alias

- Topic Aliases substitute UTF-8 String topic names with an integer.
- Topic Alias to Topic mapping is relevant only for a single connection.
- This feature's support is optional for brokers and clients.
- Broker and client negotiate to what degree this feature is supported during the connection establishment.
- Ensure your broker and client implementation supports Topic Aliases if you wish to use this feature.
- When used correctly, Topic Aliases can significantly impact the profit margins of your business case.

In summary, Topic Alias offers a versatile approach to utilizing the pub/sub model. When repeatedly publishing messages to a limited number of topics, particularly in high volumes, Topic Aliases can significantly conserve network and computing resources in an efficient manner.

Chapter 22: Enhanced Authentication in MQTT

Modern IoT projects have evolved into large, complex projects, especially when robust security measures are paramount. These expansive initiatives often involve collaboration between multiple vendors and teams. Adhering to internationally accepted standards becomes crucial to streamline the challenges encountered in such projects. Enhanced Authentication helps ensure compliance with these standards.

Implementing Challenge-Response Authentication
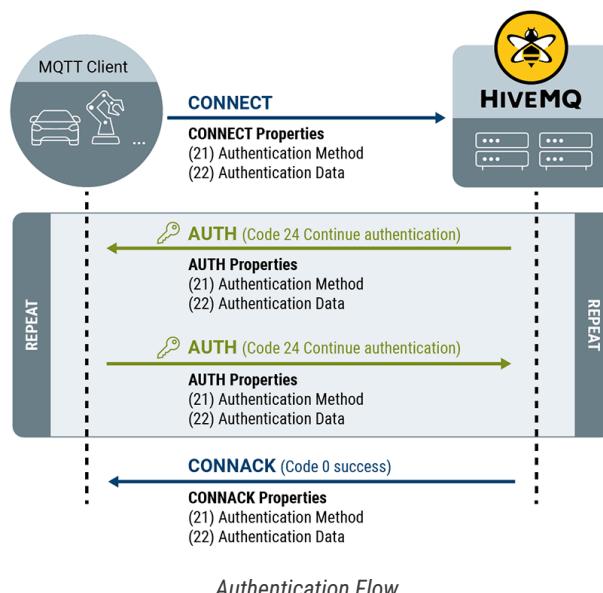
By incorporating challenge-response authentication into your MQTT 5 implementation, you can access industry-standard authentication mechanisms like the Salted Challenge

Response Authentication Mechanism (SCRAM) or the Kerberos protocol. These widely recognized protocols further bolster the security of your IoT infrastructure by adding a layer of verification.

What is Authentication Flow in MQTT?

The authentication flow in enhanced authentication relies on three MQTT message types: CONNECT, CONNACK (already present in MQTT v3), and the new MQTT v5 AUTH message. Clients send CONNECT messages, while the server sends CONNACK messages. Both message types are used once during each authentication process. On the other hand, AUTH messages can be used multiple times by both the server and the client.

The core of the authentication flow revolves around two message properties: the Authentication Method (identified by byte 21) and the Authentication Data (identified by byte 22). These properties are set on every message involved in the enhanced authentication flow.

Authentication Flow

Authentication Method in MQTT

With the Authentication Method the client and server can select and describe the agreed-upon authentication approach. It is represented by method strings commonly used to identify SASL (Simple Authentication and Security Layer) mechanisms. For instance, examples of method strings include SCRAM-SHA-1 for SCRAM with SHA-1 or GS2-KRB5 for Kerberos.

The Authentication Method assigns significance to the exchanged data during enhanced authentication and should remain constant throughout the process, ensuring consistency and integrity.

Authentication Data in MQTT

Authentication Data refers to binary information utilized during

the authentication process. It typically involves transferring encrypted secrets or protocol steps in multiple iterations. The specific content of the data heavily relies on the chosen mechanism employed in enhanced authentication and is specific to the application in use.

Source Code Example of Enhanced Authentication in MQTT

In this code snippet, we utilize the HiveMQ extension SDK to implement enhanced authentication. The purpose is to verify the support of the Authentication Method and determine the state of an MQTT client that is connecting after the exchange of two AUTH messages.

```java
public class MyEnhancedAuthenticator implements EnhancedAuthenticator {
    public void onConnect(EnhancedAuthConnectInput input, EnhancedAuthOutput output) {
        final ConnectPacket connectPacket = input.getConnectPacket();
        // Is the given authentication method supported?
        if (authenticationMethodIsSupported(connectPacket.getAuthenticationMethod())) {
            // Did the client provide valid authentication data?
            if (validateClientAuthenticationData(connectPacket.getAuthenticationData())) {
                // Send an AUTH message that contains a challenge!
                output.continueAuthentication(prepareServerAuthenticationData());
                return;
            }
        }
        // Fail the authentication and disconnect the client.
        output.failAuthentication();
    }
    public void onAuth(EnhancedAuthInput input, EnhancedAuthOutput output) {
        final AuthPacket authPacket = input.getAuthPacket();
        // Try to validate the response.
        if (validateClientAuthenticationData(authPacket.getAuthenticationData())) {
            // Allow the client to connect to the server.
            output.authenticateSuccessfully();
            return;
        }
        // Fail the authentication and disconnect the client.
        output.failAuthentication();
    }
}
```

The significance of Enhanced Authentication cannot be overstated. In a world where the proliferation of interconnected devices has amplified the importance of secure communication, MQTT 5 steps up to the challenge. This advanced authentication mechanism empowers organizations to safeguard their IoT infrastructure, sensitive data, and the privacy of their users.

Chapter 23: MQTT Flow Control
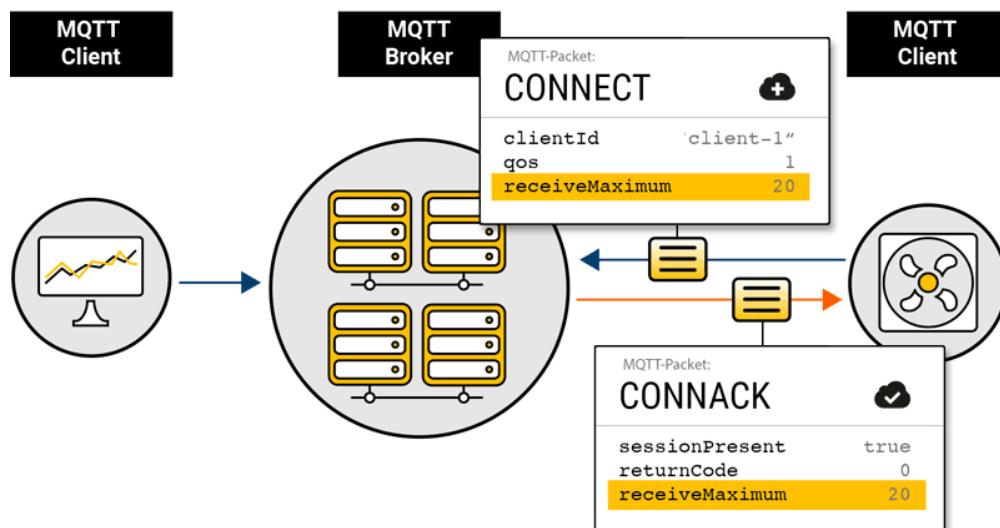
Flow Control is a dynamic feature introduced in MQTT 5 designed to regulate message traffic between IoT devices and brokers for efficient and stable communication.

IoT deployments encapsulate a wide range of device types. For instance, an MQTT client embedded in a compact sensor varies significantly from one incorporated in a high-performance backend server regarding processing speed and storage capabilities. Consequently, these MQTT clients demonstrate varying tolerance levels for managing in-flight messages. Here, an in-flight message refers to a PUBLISH command with a **Quality of Service level** of one or two awaiting acknowledgment.

Similarly, an IoT device might connect to multiple MQTT brokers, each with distinct limitations on managing in-flight messages from an MQTT client. To seamlessly manage such diversified conditions among MQTT clients and brokers, MQTT 5 introduces the Flow Control feature.

How Flow Control Works in MQTT 5?

The Flow Control feature functions through a negotiation between the client and broker to establish in-flight windows during the connection. This process involves setting an optional property known as "Receive Maximum" in the CONNECT packet, indicative of the maximum number of unacknowledged PUBLISH messages the client can accommodate. The broker reciprocates with a similar value for "Receive Maximum" in the CONNACK packet. If the "Receive Maximum" value is not specified, the default value of 65,535 is employed.

Client and broker negotiate their receive maximum.

What Are the Advantages of Flow Control in MQTT 5?

Flow Control enhances dynamic message flow adjustment for use cases involving diverse systems and devices, fostering transparency and adaptability when multiple teams or vendors collaborate on a project. No longer is it necessary for all parties to pre-establish in-flight windows. If an MQTT 5 client sends more unacknowledged messages than what the Server Receive Maximum permits, the broker sends DISCONNECT with Reason Code 0x93 (Receive Maximum exceeded). This flexibility allows the client and broker to send fewer in-flight messages than the corresponding Receive Maximum permits.

Chapter 24: MQTT Topic Tree & Topic Matching: Challenges and Best Practices Explained

As both the size and complexity of IoT projects continues to grow, we talk to many IT architects who are working to solve the technical challenges to design a data foundation built for scale. In a large MQTT deployment, there may be thousands or even millions of clients subscribing to different topics.

This chapter explains why finding the matching subscriptions among millions of subscribers is a challenge and how an MQTT broker can overcome this challenge.

What is Topic Matching in MQTT?
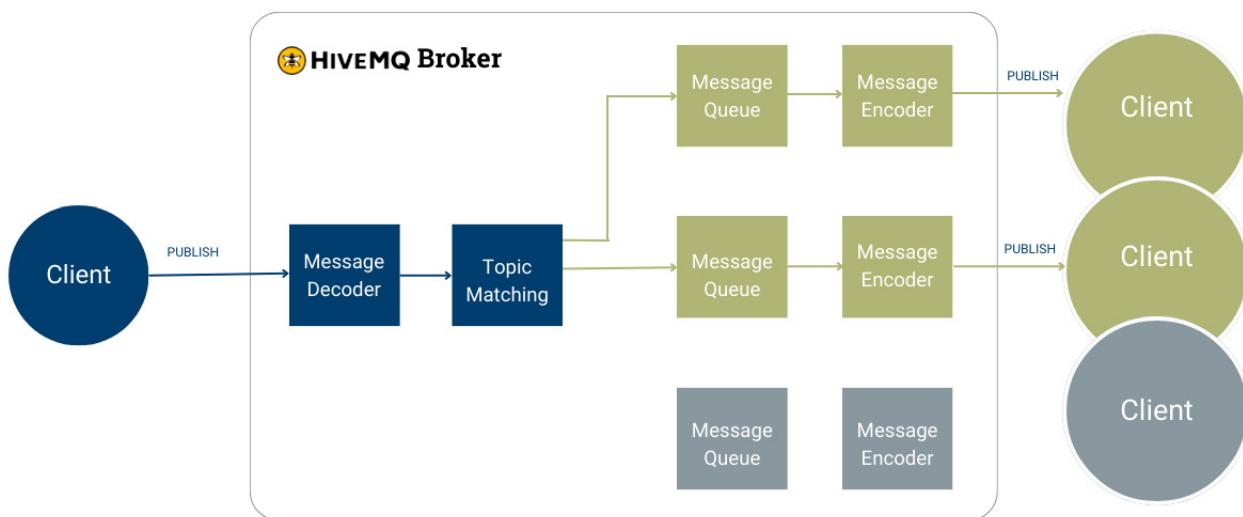
MQTT is a publish/subscribe protocol where devices act as MQTT Clients and exchange messages over an MQTT Broker. MQTT Clients send their data in the PUBLISH control packets to the specific topic. The topic is separate from the packet's payload, which allows the broker to avoid analyzing the packet's payload. The broker delivers the published message to every client subscribed with a matching topic filter.

For those unaware, the main distinction between the topic and topic filter is that the topic is used for publishing and cannot contain wildcard characters whereas the topic filter can. The wildcard characters are used to aggregate multiple streams of data into one and are thus used on the subscriber's side. It is possible to create a topic filter without wildcard characters, then it would only match at most one topic. That case is often referred to as an exact subscription.

In a nutshell, topic filter can be thought of as a selector for topics that the PUBLISH packets are sent to. The broker must be able to find the matching subscriptions for each published message.

Subscriptions can contain wildcard characters to match a broad range of topics. Subscriptions with wildcards are often used when there is uncertainty about the topics that publishing clients will use. For example, when the publishing clients include their ID as one topic level, it may be impossible to reliably receive messages for all such topics without the usage of wildcard characters. While this is useful for the clients, finding all the matching subscriptions presents a technical challenge. In some **real-life scenarios**, brokers check millions of subscriptions for every published message.

MQTT Wildcard Topic Matching Challenge Explained

Since there are many use cases for wildcards, let's examine the technical challenge of capitalizing on wildcard subscriptions. First, looking at every subscription for every published message is not scalable. The number of steps needed to find the matching subscriptions linearly increases with the number of subscriptions. Alternatively, the broker could map the subscriptions to their topic filters and check the map for all filters matching the topic of a published message. This method is also impractical because the number of potentially matching topic filters is rather large for topics with many levels. For instance, if a message is published to the topic "town/house/kitchen", all the subscriptions with the following topic filters would match:
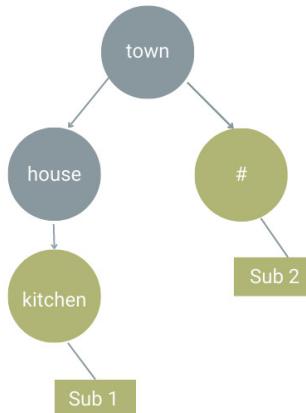
- #
- town/#
- town/house/#
- +/house/kitchen
- town/+/kitchen
- town/house/+
- +/+/kitchen
- town/+/+
- +/+/+
- town/house/kitchen
- …

The broker must also check the map for all these topic filters. In production workloads, the broker has to find the matching subscriptions for published messages thousands of times per second, so it needs a specialized data structure for a fast lookup.
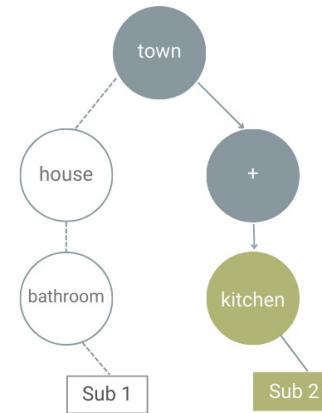
The Topic Tree

The Topic Tree is a data structure used to solve the challenges posed by the above wildcard topic matching problem. The topic of the published message is used to collect the matching subscriptions present in the topic tree. We start at the root of the topic tree and proceed through its levels using the topic segments to select the next node. If the current node has wildcard subscriptions (with # or +), they are added to the matching subscriptions. Once there are no more segments to match in the topic and there are non-wildcard subscriptions in the current node, there are exact subscriptions for this topic. An exact subscription filters topics of published messages for exact matches.

MQTT Topic Tree Structure

The broker can continue delivering the published message to the subscribed clients once it has found all matching subscriptions. This way of storing the subscriptions also reduces memory usage because topic levels shared across multiple subscriptions are only stored once.

Best Practices

For your application to filter messages to your specifications regardless of how the MQTT Broker defines topic matching, there are a few topic design considerations that you may leverage.

It is good practice to avoid topic levels that do not add additional information, like using the same topic level across all subscriptions. The most common example of such abuse is using the company name as the first level for every subscription. While some topic levels typically have less variety than others, you should omit topic levels that are the same for every topic. Similarly, leading with forward slashes must be matched, so should be avoided if you don't want them present on all topics.

Bad practice:

- home/livingarea/kitchen
- home/livingarea/bathroom
- home/garage

Bad practice:

- /livingarea/kitchen
- /livingarea/bathroom
- /garage

Good practice:

- livingarea/kitchen
- livingarea/bathroom
- Garage

In summary, MQTT's topic matching is crucial to its publish/subscribe protocol, enabling MQTT clients to exchange messages with the MQTT broker with minimal effort. Topic filters help select the topics to which PUBLISH packets are sent, and subscriptions with wildcard characters enable broad topic matching. However, finding all matching subscriptions presents a technical challenge, it can be solved using a specialized data structure called the Topic Tree. It is essential to use best practices when designing topics to make them agnostic of the implementation that the particular MQTT broker may have for topic matching.

Chapter 25: Additional Reading for Mastering MQTT

MQTT Over WebSockets

We've seen that MQTT is ideal for constrained devices and unreliable networks and that it is perfect for sending messages with a very low overhead. Naturally, it would be quite nice to send and receive MQTT messages directly in a browser. For example, on a mobile phone. MQTT over WebSockets is the answer. MQTT over WebSockets enables the browser to leverage all MQTT features. You can use these capabilities for many interesting use cases:
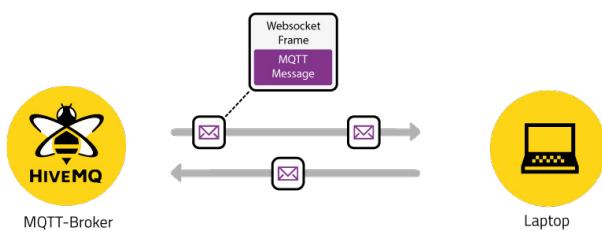
- Display live information from a device or sensor.
- Receive push notifications (for example, an alert or critical condition warning).
- See the current status of devices with LWT and retained messages.
- Communicate efficiently with mobile web applications.

What Does All This Mean from a Technical Point of View?

Every modern browser that supports WebSockets can be a full-fledged MQTT client and offer all the features described in the MQTT Essentials. The Keep Alive, Last Will and Testament, Quality of Service, and Retained Messages features work the same way in the browser as in a native MQTT client. All you need is a **JavaScript library** that enables MQTT over WebSockets and a broker that supports MQTT over WebSockets. Of course, HiveMQ Broker offers this capability straight out-of-the-box.

How Does It Work?

WebSocket is a network protocol that provides bi-directional communication between a browser and a web server. The protocol was standardized in 2011 and all modern browsers provide built-in support for it. Similar to MQTT, the WebSocket protocol is based on TCP.

In *MQTT over WebSockets*, the MQTT message (for example, a CONNECT or PUBLISH packet) is transferred over the network and encapsulated by one or more WebSocket frames. WebSockets are a good transport method for MQTT because they provide bi-directional, ordered, and lossless communication (WebSockets also leverage TCP). To communicate with an MQTT broker over WebSockets, the broker must be able to handle native WebSockets. Occasionally, people use a webserver and bridge WebSockets to the MQTT broker, but we don't recommend this method. When using HiveMQ, it is very easy to get started with WebSockets. Simply enable the native support in the configuration.

Why Not Use MQTT Directly?

Currently, it is not possible to speak pure MQTT in a browser because it is not possible to open a raw TCP connection. **Socket API** will change that situation; however, few browsers implement this API yet.

Get Started

If you want to get started with MQTT over *WebSockets*, here are some useful resources:

- For testing and debugging, the HiveMQ **MQTT WebSocket client** is ideal. The public broker of the **MQTT Dashboard** is the default broker of this client. All features of the client are documented in detail and the source code is available on **GitHub**.
- If you want to integrate MQTT into your existing web application, check out this **step-by-step guide on how to build your own MQTT WebSockets client**.
- To learn more about how to set up your own broker with WebSockets support, read **MQTT over WebSockets**.

Secure WebSockets

You can leverage Transport Layer Security (TLS) to use secure WebSockets with encryption of the whole connection. This method works seamlessly with HiveMQ. However, there are a few points that you need to keep in mind. For more information, see the **Gotchas section of our user guide**.

Types of MQTT Brokers

There are different types of MQTT brokers available:

- Open source: These are license-free brokers that are suitable for small-scale projects.
- Commercial: These are available via licenses and offer several features and customization, making them suitable for production-level deployment.
- Cloud-managed: These are fully-managed MQTT brokers that need minimum maintenance and are easy to deploy.
- General purpose: These are available both license-free as well as for licenses and are suitable for small-scale projects.

The below image summarizes features offered by different types of brokers.

	Open Source MQTT Brokers	Commercial MQTT Brokers	Cloud Managed MQTT Brokers	General purpose Messaging brokers
Reliability	◔	◑	◕	◕
Scalability	◔	◕	●	◕
Security	◑	◕	●	◕
Observability	◕	◑	◑	◑
Extensibility	◑	◑	◑	◑
Simplicity	●	◕	●	◔

For in-depth information, download our 2023 Buyer's Guide: MQTT Platforms.

Open Source vs. Commercial MQTT Brokers

Open-source MQTT brokers have limited scalability, limited security options, cannot cluster for higher availability, are hard to manage when coded in difficult libraries, and have no overload protection from overactive publishers.

On the other hand, commercial MQTT brokers are customizable according to your needs, provide increased scalability, security, flexibility & reliability, and have overload protection.

HiveMQ offers a community edition MQTT broker as well as a commercial MQTT broker. Explore them now!

On-premises MQTT Brokers vs. Fully-Managed Cloud MQTT Brokers

On-premises MQTT brokers offer more control but require substantial setup and maintenance. Fully managed cloud-based MQTT brokers are easier to deploy, cost-effective, and have zero maintenance requirements.

HiveMQ offers an on-prem MQTT broker as well as a fully-managed cloud MQTT broker. Explore them now!

MQTT vs. Other IoT Protocols
MQTT vs. HTTP

	MQTT	HTTP
Full name	MQTT (the OASIS standardization group decided it would not stand for anything)	Hypertext Transfer Protocol
Architecture	Publish subscribe (MQTT does have a request/reply mode as well)	Request response
Command targets	Topics	URIs
Underlying Protocol	TCP/IP	TCP/IP
Secure connections	TLS + username/password (SASL support possible)	TLS + username/password (SASL support possible)
Client observability	Known connection status (will messages)	Unknown connection status
Messaging Mode	Asynchronous, event-based	Synchronous

	MQTT	HTTP
Message queuing	The broker can queue messages for disconnected subscribers	Application needs to implement
Message overhead	2 bytes minimum. Header data can be binary	8 bytes minimum (header data is text - compression possible)
Message Size	256MB maximum	No limit but 256MB is beyond normal use cases anyway.
Content type	Any (binary)	Text (Base64 encoding for binary)
Message distribution	One to many	One to one
Reliability	Three qualities of service: 0 - fire and forget, 1 - at least once, 2 - once and only once	Has to be implemented in the application

To learn more, read our blog, MQTT vs. HTTP

MQTT vs. AMQP

	MQTT	AMQP
Full name	MQTT (the OASIS standardization group decided it would not stand for anything)	Advanced Message Queuing Protocol
Architecture	Publish subscribe (MQTT does have a request/reply mode as well)	Queues, multicast (fanout), publish subscribe, request reply
Command targets	Topics	Exchanges, Queues
Underlying Protocol	TCP/IP	TCP/IP

	MQTT	AMQP
Secure connections	TLS + username/password (SASL support possible)	TLS + username/password (SASL support possible)
Client observability	Known connection status (will messages)	Unknown connection status
Messaging Mode	Asynchronous, event-based	Synchronous and asynchronous
Message queuing	The broker can queue messages for disconnected subscribers	Core capability, flexible configuration
Message overhead	2 bytes minimum	8 bytes (general frame format)
Message Size	256MB maximum	2GB theoretical, 128MB max recommended
Content type	Any (binary)	Any (binary)
Topic matching	Level separator: / Wildcards: + #	Level separator: . Wildcards: * #
Reliability	Three qualities of service: 0 - fire and forget 1 - at least once 2 - once and only once	Two qualities of service: -without acks (=0) - with acks (=1)
Connection "multiplexing"	No	Yes - channels
Message attributes	MQTT 5.0 only	Yes
Object persistence	Yes	Yes

To learn more, read our blog, MQTT vs. AMQP

HiveMQ logo top right

MQTT vs. ZeroMQ

	MQTT	ZeroMQ
Full name	MQTT (the OASIS standardization group decided it would not stand for anything)	ZeroMQ
Architecture	Client/server	Peer to peer
Command targets	Topics	ZeroMQ sockets
Underlying Protocol	TCP/IP	TCP/IP, UDP, shared memory
Secure connections	TLS + username/password (SASL support possible)	PLAIN (username/password), **CurveZMQ** and **ZAP**
Client observability	Known connection status (will messages)	None
Retained messages	Yes	No
Messaging Mode	Asynchronous, event-based	Application dependent
Message queuing	The broker can queue messages for disconnected subscribers	Some 0MQ socket types have it
Message overhead	2 bytes minimum	2 bytes minimum

	MQTT	ZeroMQ
Message Size	256MB maximum	2^63-1 bytes per frame
Message fragmentation	No	Yes
Content type	Any (binary)	Any
Pub/sub topic matching	Level separator: / Wildcards: + #	Prefix only
Message distribution	One to many, one to one	Various
Reliability	Three qualities of service: 0 - fire and forget 1 - at least once 2 - once and only once	Varies between socket types, but the equivalent of QoS 2 would have to be implemented in the application.

To learn more, read our blog, MQTT vs. ZeroMQ

MQTT vs. CoAP

	MQTT	CoAP
Full name	MQTT (the OASIS standardization group decided it would not stand for anything)	The Constrained Application Protocol
Architecture	Client/Server	Client/Server

	MQTT	CoAP
Command targets	Topics	URIs
Underlying transport	TCP	UDP
Default insecure TCP port	1883	5684
Default secure UDP port	8883	5684
Secure connections	TLS + username/password (SASL support possible)	**DTLS** (TLS for UDP)
Client observability	Known connection status (will messages)	None
Retained messages	Yes	No
Messaging Mode	Asynchronous, event-based	Synchronous (or asynchronous with observe extension)
Message queuing	The broker can queue messages for disconnected subscribers	None
Message overhead	2 bytes minimum	4 bytes minimum
Message Size	256MB maximum	Limit of underlying transport - RFC 7252 suggests 1152 bytes for UDP if nothing is known about the target.
Message fragmentation	Reliant on TCP	No (**RFC 7959** proposes an extension for this)
Content type	Any (binary)	Any

To learn more, read our blog, MQTT vs. CoAP

	MQTT	CoAP
Pub/sub topic matching	Level separator: /Wildcards: + #	No pub/sub (but **extensions** have been proposed)
Message distribution	One to many, one to one	One to many, one to one
Reliability	Three qualities of service: 0 - fire and forget 1 - at least once 2 - once and only once	Confirmable or not (roughly equivalent to QoS 1 and 0)

Implementing MQTT in C

To get started with the Paho MQTT C Client library, clone the repo and use the following code to install it.

Developers can use different client libraries to implement MQTT in different languages, such as Java, C, C#, etc.

```
git clone https://git.eclipse.org/r/paho/org.eclipse.paho.mqtt.c
make
sudo make install
```

For more instructions, read our blog Implementing MQTT in C.

Implementing MQTT in Java

Developers can use an MQTT Java Client Library, such as HiveMQ MQTT Client for Java or Eclipse Paho Java MQTT Client Library. To use Paho Java, install it by downloading it from the Eclipse project page. If you're using Maven, add the following code to your pom.xml file.

```
<repositories>
    <repository>
        <id>Eclipse Paho Repo</id>
        <url>https://repo.eclipse.org/content/repositories/paho-releases/</url>
    </repository>
</repositories>
```

Then add the following code to your dependencies section.

```
<dependency>
    <groupId>org.eclipse.paho</groupId>
    <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
    <version>1.0.2</version>
</dependency>
```

For more instructions, read our blog Implementing MQTT in Java.

Implementing MQTT in C#

You can install the HiveMQ C# MQTT client by using the code:

dotnet add package HiveMQtt

For more instructions, read our blog Implementing MQTT in C#.

Implementing MQTT in Python

You can install Paho MQTT for Python 3.12 in the command line with the code:
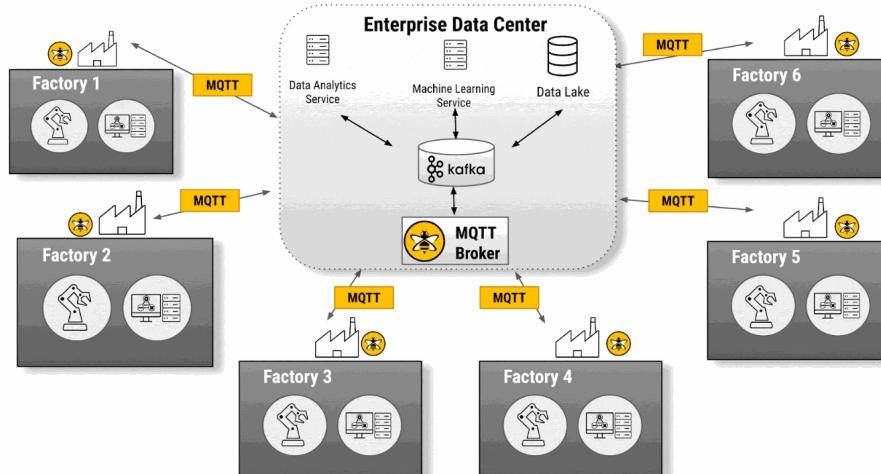
pip3 install paho-mqtt<2.0.0

For more instructions, read our blog Implementing MQTT in Python.

MQTT Broker Architecture in Smart Manufacturing

MQTT brokers can be used for several smart manufacturing use cases, such as intra-factory connections, inter-factory connections, etc.
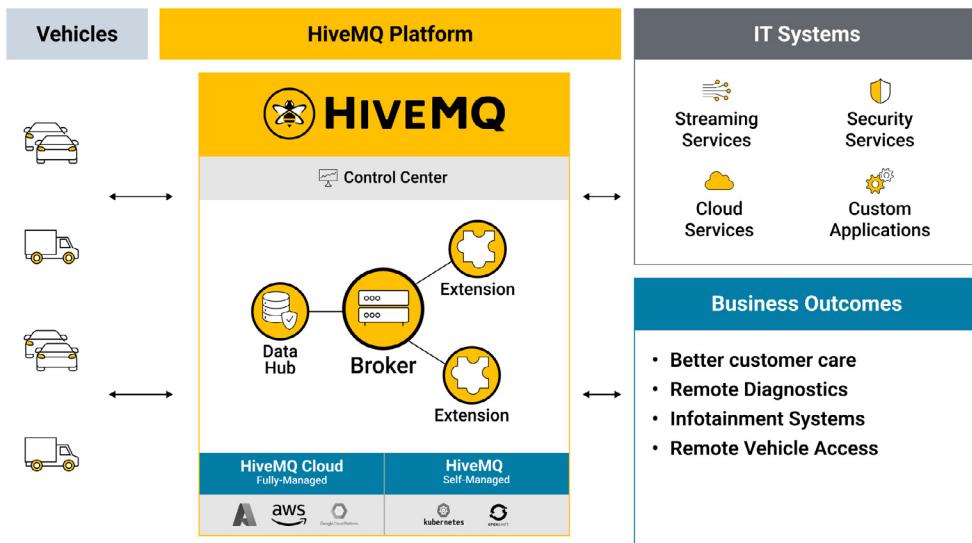
Here's an example of inter-factory communication using the HiveMQ MQTT Broker.

To learn more, read our whitepaper Modernizing the Smart Manufacturing Industry with MQTT.

MQTT Broker Architecture in Connected Cars

MQTT brokers can be used for several connected car use cases such as remote monitoring, remote diagnostics, Over-the-Air (OTA) updates, theft recovery, energy management, etc.
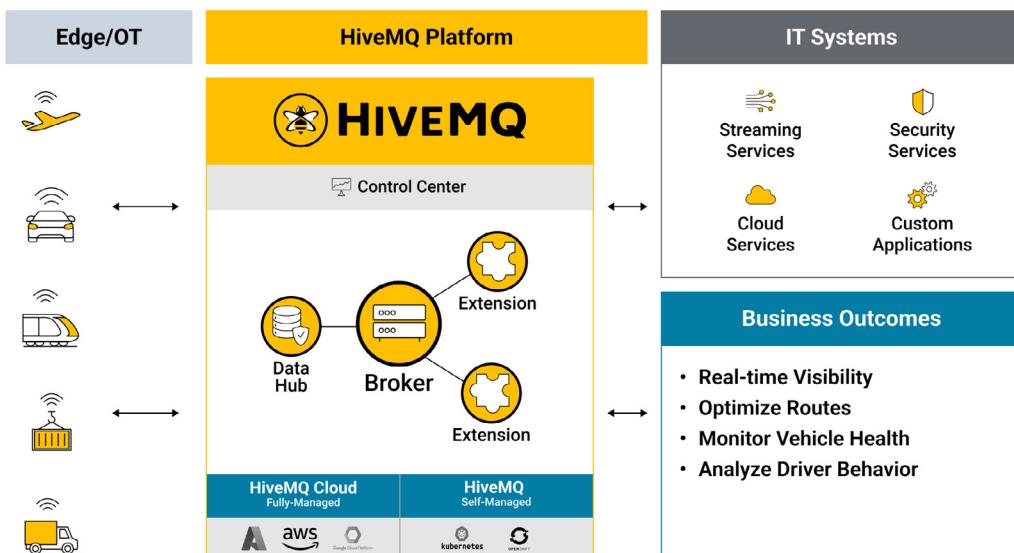

To learn more, read our blog Building New Connected Car Features on the Back of a Robust Connectivity Platform.

MQTT Broker Architecture in Transportation and Logistics

MQTT brokers are well-suited for use in the transportation and logistics industry, where real-time data communication is crucial for efficiency and safety. Some of the use cases include fleet management, supply chain visibility, warehouse management, driver and cargo safety, etc.
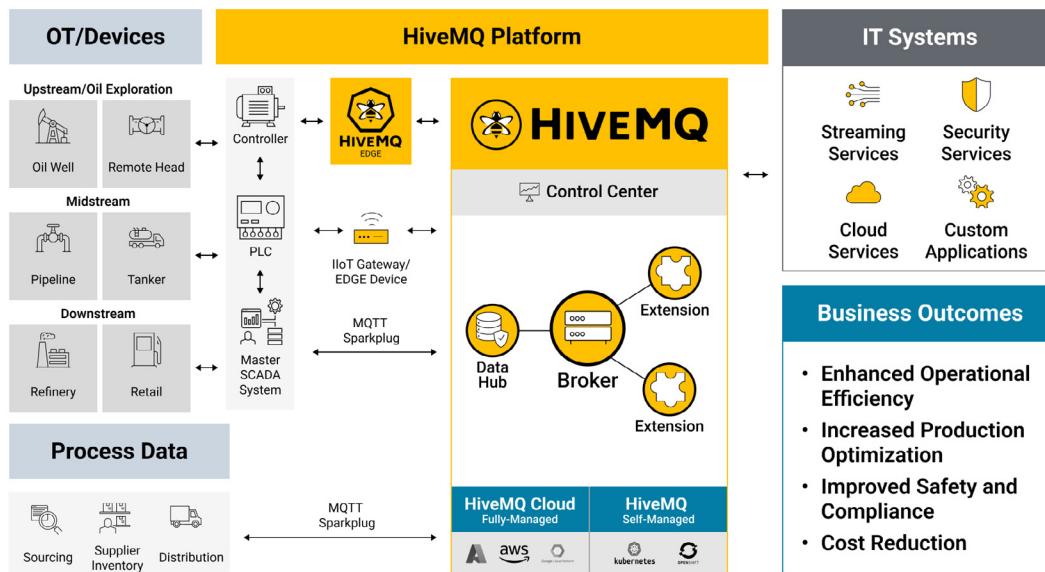
Here's an example of how an MQTT broker, like HiveMQ, can help real-time communication between several vehicles and backend systems.

MQTT Broker Architecture in the Energy Industry

MQTT brokers can play a crucial role in optimizing the operations and OT/IT interoperability in the energy industry. MQTT brokers can be used in energy use cases such as Asset Performance Management, Predictive Maintenance, Asset Tracking, etc.

Here's an example of how an MQTT broker, like HiveMQ, can be used for Remote Asset Management in Oil & Gas.

Chapter 26: Next Steps – Choosing the Right MQTT Broker

Choosing the right MQTT broker depends on your architectural requirements (a.k.a. Non-Functional Requirements). An MQTT broker comparison based on these architectural requirements should give you insight into how to find the best MQTT broker for your needs.

Here are some of the functionalities you should look for in an MQTT broker if you want to use it in large-scale production:

- **Handling large numbers of concurrent connections reliably**: Depending on its implementation, a broker has the capability to manage millions of MQTT client connections reliably and securely, with a robust load balancing feature. This facilitates communication between diverse devices, networks, and software systems in near real-time.

- **Filtering and routing messages**: MQTT brokers can filter messages based on the subscription topic and determine which client(s) should receive the message.

- Session management: MQTT brokers can maintain session data for all connected clients, including subscriptions and missed messages, for clients with persistent sessions.

- **Authentication and authorization**: The broker is responsible for authenticating and authorizing clients based on credentials provided by the client. The broker is extensible, facilitating custom authentication, authorization, and integration into backend systems. In addition to authentication and authorization, brokers may provide other security features, such as encryption of messages in transit and access control lists.

- Scalability and monitoring: An MQTT broker must be scalable to handle large volumes of messages and clients, integrate into backend systems, be easy to monitor, and be failure-resistant. To meet these
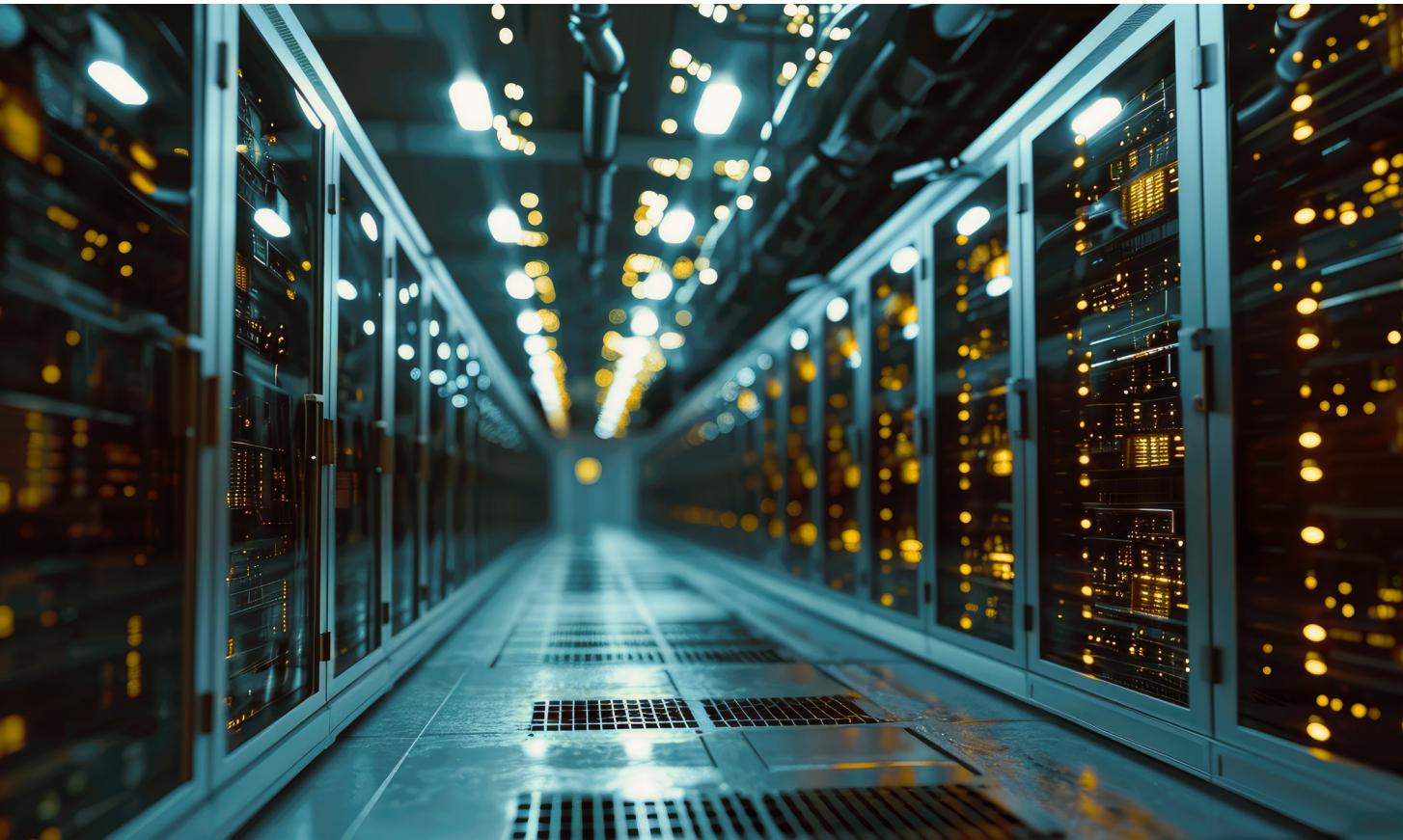
requirements, the MQTT broker must use state-of-the-art event-driven network processing, an open extension system, and standard monitoring providers. Brokers may also provide advanced features for managing and monitoring the MQTT system, such as message filtering, message persistence, and real-time analytics.

- **Clustering**: MQTT brokers can support clustering, allowing multiple instances of the broker to work together to handle large numbers of clients and messages.
- **Provides control over how IoT data flows within your data pipeline**: Prominent MQTT brokers can provide you with the ability to have control over your data with visualization and management tools, powerful security features, and an integrated policy engine that can validate, enforce, and transform data in motion.

HiveMQ Self-Managed Enterprise Broker offers all the above features. Hundreds of active customers, across the globe trust HiveMQ MQTT Broker for reliable, flexible, observable, secure, and scalable data communication.
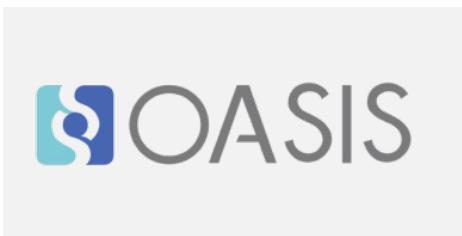
Here are some of the functionalities you should look for in an MQTT Broker if you want to use it in small-scale projects:

- **Ease-of-use**: For small-scale projects, especially DIY projects and PoCs, an MQTT broker should be easy to deploy. There are several open-source and fully-managed MQTT brokers that help you connect devices with minimal effort.
- **Scalability**: MQTT brokers should be able to help you scale as you go so your PoCs turn into production-level projects soon.
- **Cost-efficiency**: There is pay-as-you-go pricing available for MQTT brokers so you don't have to constrain yourself while working on a PoC.
- **Security**: It goes without saying how important it is to secure your device connectivity from end to end, irrespective of your scale of the project.

HiveMQ Cloud is a fully managed MQTT broker that offers all the features above. Do check it out.

HiveMQ is active in the open source community, working with several organizations to advance the adoption of the MQTT protocol for IoT.
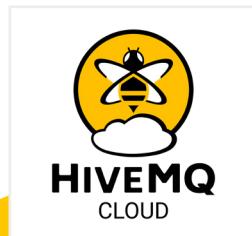
OASIS

We are a member of the OASIS MQTT and MQTT-SN Technical Committees.

iot eclipse.org

We are a member of the OASIS MQTT and MQTT-SN Technical Committees.

Sparkplug

We are a member of the OASIS MQTT and MQTT-SN Technical Committees.

HiveMQ Cloud:

Develop, test, deploy, and scale production IoT use cases without a large investment.

HiveMQ Trial:

Trial the HiveMQ platform with all of the enterprise features needed for large-scale deployment.

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**

HiveMQ

www.hivemq.com