Introduction to MQTT

Rytis Paškauskas and Marco Rainone

PLAN

1. Introduction
2. Understanding MQTT Basics
3. The MQTT Architecture
4. MQTT Workflow
5. Hands-On Demonstration

CONSTRAINTS

2 hours

Introduction to MQTT

What is MQTT?

MQTT is is a lightweight, publish/subscribe messaging protocol designed for constrained devices and unreliable networks.

Pop quiz – “MQTT” stands for:

1. **M**essage **Q**ueuing **T**elemetry **T**ransport?
2. **MQ** Telemetry Transport?
3. MQTT?

What does “MQTT” stand for?

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.

MQTT is now simply the name of the protocol.

Although many sources label MQTT as a **M**essage **Q**ueue **T**elemetry **T**ransport protocol, this is not entirely accurate. While it is possible to queue messages in certain cases, MQTT is not a traditional message queuing solution. (from HiveMQ book).

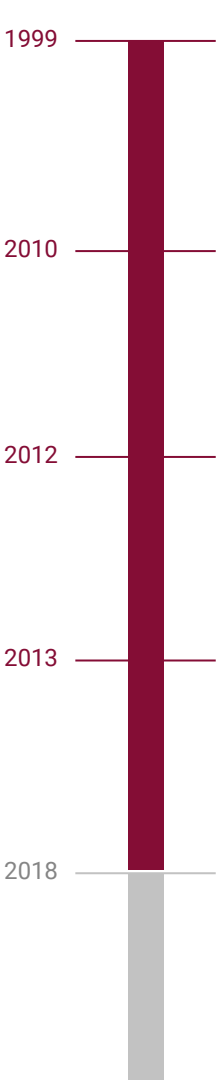
History and Evolution

Originally designed for SCADA systems. Supervisory Control and Data Acquisition (SCADA) systems are used for controlling, monitoring, and analyzing industrial devices and processes. The system consists of both software and hardware components and enables remote and on-site gathering of data from the industrial equipment.

Becoming an open standard helped to increase the visibility and adoption of MQTT among the developer community.

Popular (and open source) implementations are **Paho** (client in Python) and **Mosquitto** (broker and client in C). Both by Eclipse.

Mosquitto supports MQTT protocol versions 5.0, 3.1.1 and 3.1



1999

MQTT invented

Developed in 1999 by Andy Stanford-Clark and Arlen Nipper at IBM. First use cases in oil-pipeline monitoring. Over the next ten years, IBM used the protocol internally until they

2010

MQTT 3.1 opened as royalty-free protocol

In 2011, IBM contributed MQTT client implementations to the newly founded Paho project of the Eclipse Foundation.

2012

Mosquitto 1.0 released

Mosquitto is part of the Eclipse Foundation, and is an iot.eclipse.org project. The development is driven by Cedalo.

2013

OASIS TC formed

the OASIS MQTT Technical Committee (TC), which was formed in 2013 to work on the standardization of the MQTT protocol.

2018

MQTT 5.0 officially released

MQTT 5.0 improves error handling, adds session/message expiration, and supports topic aliases to reduce bandwidth. It introduces shared subscriptions for load balancing and user properties for metadata. These updates enhance flexibility, scalability, and security for IoT applications.

Why Use MQTT?

MQTT is used because it’s simple, efficient, and works well even when the internet connection is slow or unstable.

Asynchronous communication: Clients can publish data without worrying about whether subscribers are available, as the broker handles message routing. It allows devices to send and receive messages without waiting for an immediate response.

Low overhead and reduced power consumption: With a small packet size and a lightweight control mechanism, MQTT is efficient for transmitting data.

Support for Intermittent Connections: Devices in constrained environments often lose network connectivity. MQTT supports persistent sessions to handle this gracefully.

Feature/Protocol	MQTT	HTTP	CoAP	AMQP
Communication Model	Publish/Subscribe (asynchronous)	Request/Response (synchronous)	Request/Response (synchronous), Observe	Publish/Subscribe, Message Queue (asynchronous)
Transport Protocol	TCP	TCP (HTTP/3 uses UDP)	UDP (TCP is possible)	TCP
Overhead	Low	High	Very Low	Moderate to High
Message Delivery	QoS 0, 1, 2 (reliable with options)	TCP-level reliability, no QoS	Basic reliability with UDP	Highly reliable (transactional messaging)
Security	TLS	HTTPS (TLS)	DTLS	TLS, SASL

Feature/ Protocol	MQTT	HTTP	CoAP	AMQP
Statefulness	Session persistence for reconnecting clients	Stateless by design	Stateless, with optional Observe feature	Stateful
Scalability	High, designed for IoT environments	High, but overhead increases	Very High, designed for constrained devices	High, for large enterprise applications
Use Cases	IoT, remote sensors, constrained devices	Web applications, REST APIs, document exchange	Low-power IoT (smart grids, home automation)	Enterprise message brokering, financial systems

Feature/ Protocol	MQTT	HTTP	CoAP	AMQP
QoS/ Delivery Guarantees	QoS levels ensure message delivery guarantees	No built-in QoS beyond TCP reliability	Basic retransmissions (confirmable messages)	Guarantees through transactional messaging
Message Size	Small	Large	Small	Large
Latency	Low (ideal for low-latency environments)	High for IoT (due to overhead)	Low	Moderate

MQTT Publish/Subscribe Paradigm vs HTTP Request/Response

Feature	MQTT (Publish/Subscribe)	HTTP (Request/Response)
Communication Model	**Decoupled:** many-to-many communication with a broker	**Direct:** one-to-one communication
Message Flow	messages sent to topics, received by subscribers via broker	client requests data from a server, receives a response
Statefulness	Long-lived connections, maintains state	Stateless, each request is independent

When to Choose Each Protocol

MQTT: Ideal for low-bandwidth, high-latency environments where devices need to conserve power, such as IoT devices (sensors, actuators, smart home systems). Suitable when asynchronous, decoupled communication is needed, such as in telemetry or distributed control systems.

HTTP: Best for web applications and REST-based APIs where resource exchange and direct request-response interactions are required (e.g., file sharing, data retrieval from a server). Inefficient for IoT scenarios due to its high overhead and statelessness.

CoAP: Designed for constrained IoT devices that need efficient, low-overhead communication. Ideal for sensor networks, smart energy management, and low-power wireless communication.

AMQP: Preferred in enterprise-level applications where reliable, secure, and transactional messaging is required (e.g., financial institutions, message queuing systems, distributed systems). Suitable for complex routing and store-and-forward messaging scenarios where full delivery guarantees and reliability are crucial.

MQTT in the Modern IoT and Cloud Ecosystem

Use cases: Home automation, industrial IoT, autonomous vehicles, smart cities.

Supported by cloud services: like AWS IoT, Azure IoT Hub, Google Cloud IoT Core, IBM Watson, Oracle IoT

Commercial brokers: HiveMQ, …

MQTT-like protocols: Apache Kafka (LinkedIn), Pulsar, ActiveMQ, …

2. MQTT Protocol Overview

Core MQTT Concepts

Publish: The action of sending a message to a specific topic on the broker.

Subscribe: The action of a client registering to receive messages from a specific topic.

Topic: A string used to organize and categorize messages, acting as a routing key for message distribution.

Broker: Central server that manages communication between clients by routing messages from publishers to subscribers.

Client: Any device or application that connects to the broker to publish or subscribe to topics.

QoS (Quality of Service): Defines the level of guarantee for message delivery (0, 1, or 2).

Retain Flag: Option to store the last message sent to a topic for future subscribers.

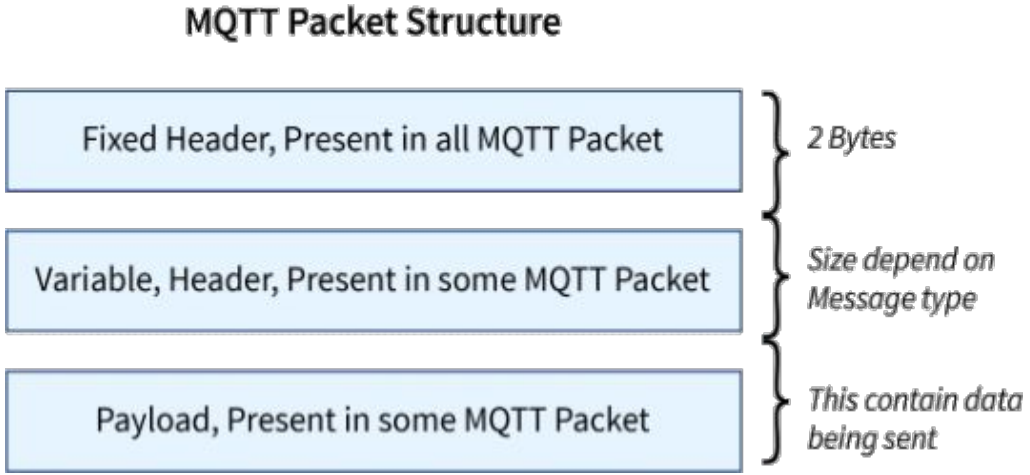
Will Message: A message defined by a client that the broker will send if the client unexpectedly disconnects.

Clean Session: An option to specify whether a client's subscription and message queue should be cleared when it disconnects.

Session Persistence: The broker can retain client subscriptions and undelivered messages between disconnections.

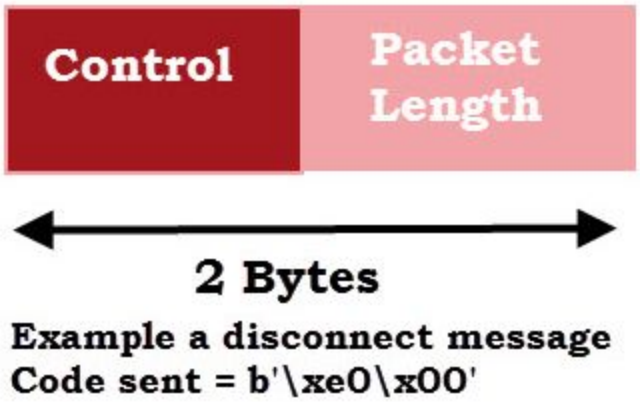
MQTT Packet Structure

- **Key components:** Fixed header, variable header, and payload.
- **Common packet types:** CONNECT, PUBLISH, SUBSCRIBE, PINGREQ, DISCONNECT.

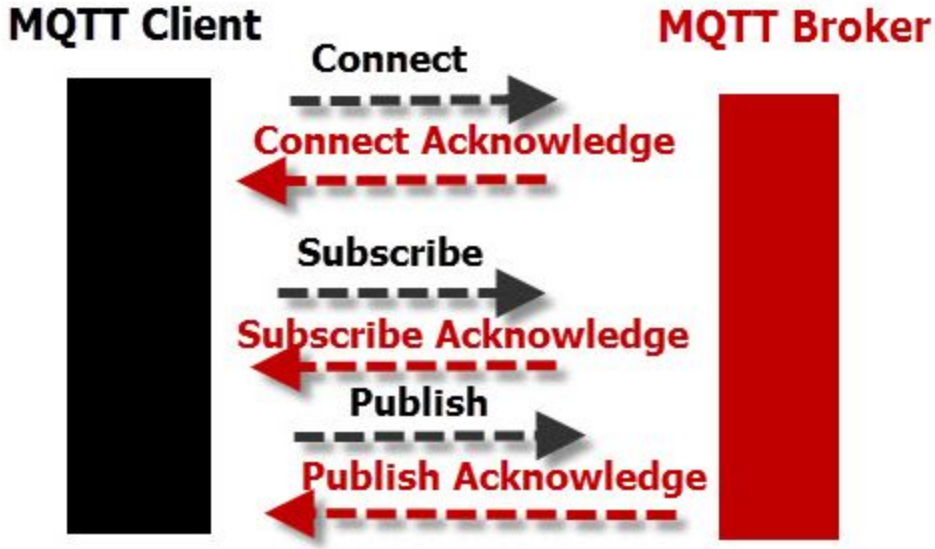


SUBSCRIBE	subscribe (client)	
PUBLISH	publish (client and broker)	
DISCONNECT	disconnect	
PINGREQ	ping request	
PINGRESP	ping response	
CONNACK	connection acknowledged	
SUBACK	subscribe acknowledged	
PUBACK	publish acknowledged	
PUBREC	publish request	
PUBREL	publish release	
PUBCOMP	publish completed	

MQTT minimum packet



MQTT Client and Broker “talking”



Case study: Basic usage example

```
> mosquitto_sub -h localhost -p 1883 -t '#'
```

```
1729511061: New connection from 127.0.0.1:44770 on port 1883.
1729511061: New client connected from 127.0.0.1:44770 as
auto-75A73584-46A3-85F6-BB86-F63D49823F2B (p2, c1, k60).
1729511061: No will message specified.
1729511061: Sending CONNACK to auto-75A73584-46A3-85F6-BB86-F63D49823F2B (0, 0)
1729511061: Received SUBSCRIBE from auto-75A73584-46A3-85F6-BB86-F63D49823F2B
1729511061:     # (QoS 0)
1729511061: auto-75A73584-46A3-85F6-BB86-F63D49823F2B 0 #
1729511061: Sending SUBACK to auto-75A73584-46A3-85F6-BB86-F63D49823F2B
1729511121: Received PINGREQ from auto-75A73584-46A3-85F6-BB86-F63D49823F2B
1729511121: Sending PINGRESP to auto-75A73584-46A3-85F6-BB86-F63D49823F2B
…
```

```
> mosquitto_pub  -h localhost -p 1883 -t 'hello' -m 'world'
```

```
1729511519: New client connected from 127.0.0.1:60848 as
auto-791F0260-5704-A25C-7820-3CAD54656FEB (p2, c1, k60).
1729511519: No will message specified.
1729511519: Sending CONNACK to auto-791F0260-5704-A25C-7820-3CAD54656FEB (0, 0)
1729511519: Received PUBLISH from auto-791F0260-5704-A25C-7820-3CAD54656FEB (d0, q0, r0,
m0, 'hello', ... (5 bytes))
1729511519: Sending PUBLISH to auto-75A73584-46A3-85F6-BB86-F63D49823F2B (d0, q0, r0,
m0, 'hello', ... (5 bytes))
1729511519: Received DISCONNECT from auto-791F0260-5704-A25C-7820-3CAD54656FEB
1729511519: Client auto-791F0260-5704-A25C-7820-3CAD54656FEB disconnected.
1729511541: Received PINGREQ from auto-75A73584-46A3-85F6-BB86-F63D49823F2B
1729511541: Sending PINGRESP to auto-75A73584-46A3-85F6-BB86-F63D49823F2B
…
```

```
$ mosquitto_sub -h localhost -p 1883 -t '#'
world
```

Quality of Service (QoS) Levels

QoS Level	It's like…	When to Use
QoS 0 (At most once)	Sending a postcard: no guarantee it will arrive, and no follow-up.	For non-critical data, such as sensor updates
QoS 1 (At least once)	Sending a registered letter: guaranteed delivery, but might arrive multiple times.	Guarantees message delivery but may result in duplicates
QoS 2 (Exactly once)	Sending a package with a signature: delivered once, and only once, no duplicates.	For critical messages, like financial transactions, where no loss or duplication is acceptable.

QoS 1 case study

```
> mosquitto_sub -q 1 -h localhost -p 1883 -t '#'

1729512609: New connection from 127.0.0.1:36906 on port 1883.
1729512609: New client connected from 127.0.0.1:36906 as
auto-ED3A5FED-675B-366D-EC69-0B6BD83796F4 (p2, c1, k60).
1729512609: No will message specified.
1729512609: Sending CONNACK to auto-ED3A5FED-675B-366D-EC69-0B6BD83796F4 (0, 0)
1729512609: Received SUBSCRIBE from auto-ED3A5FED-675B-366D-EC69-0B6BD83796F4
1729512609:     # (QoS 1)
1729512609: auto-ED3A5FED-675B-366D-EC69-0B6BD83796F4 1 #
1729512609: Sending SUBACK to auto-ED3A5FED-675B-366D-EC69-0B6BD83796F4
…
```

```
> mosquitto_pub -q 1 -h localhost -p 1883 -t 'hello' -m 'world'
```

```
1729512717: New connection from 127.0.0.1:46972 on port 1883.
1729512717: New client connected from 127.0.0.1:46972 as auto-35FE21EB-AE06-DF79-5138-A11159EAEA94 (p2, c1, k60).
1729512717: No will message specified.
1729512717: Sending CONNACK to auto-35FE21EB-AE06-DF79-5138-A11159EAEA94 (0, 0)
1729512717: Received PUBLISH from auto-35FE21EB-AE06-DF79-5138-A11159EAEA94 (d0, q1, r0, m1, 'hello', ... (5 bytes))
1729512717: Sending PUBLISH to auto-ED3A5FED-675B-366D-EC69-0B6BD83796F4 (d0, q1, r0, m1, 'hello', ... (5 bytes))
1729512717: Sending PUBACK to auto-35FE21EB-AE06-DF79-5138-A11159EAEA94 (m1, rc0)
1729512717: Received PUBACK from auto-ED3A5FED-675B-366D-EC69-0B6BD83796F4 (Mid: 1, RC:0)
1729512717: Received DISCONNECT from auto-35FE21EB-AE06-DF79-5138-A11159EAEA94
1729512717: Client auto-35FE21EB-AE06-DF79-5138-A11159EAEA94 disconnected.
```

```
$ mosquitto_sub -q 1 -h localhost -p 1883 -t '#'
world
```

Qos 2 case study

```
> mosquitto_sub -q 2 -h localhost -p 1883 -t '#'
```

```
1729513207: New connection from 127.0.0.1:49236 on port 1883.
1729513207: New client connected from 127.0.0.1:49236 as
auto-E41AFA10-1E5D-E733-F44E-5750B87AF619 (p2, c1, k60).
1729513207: No will message specified.
1729513207: Sending CONNACK to auto-E41AFA10-1E5D-E733-F44E-5750B87AF619 (0, 0)
1729513207: Received SUBSCRIBE from auto-E41AFA10-1E5D-E733-F44E-5750B87AF619
1729513207:     # (QoS 2)
1729513207: auto-E41AFA10-1E5D-E733-F44E-5750B87AF619 2 #
1729513207: Sending SUBACK to auto-E41AFA10-1E5D-E733-F44E-5750B87AF619
…
```

```
> mosquitto_pub -q 2 -h localhost -p 1883 -t 'hello' -m 'world'
```

```
1729513374: New connection from 127.0.0.1:50520 on port 1883.
1729513374: New client connected from 127.0.0.1:50520 as
auto-125DDDBC-DB9D-905F-78EF-87F602FFEE20 (p2, c1, k60).
1729513374: No will message specified.
1729513374: Sending CONNACK to auto-125DDDBC-DB9D-905F-78EF-87F602FFEE20 (0, 0)
1729513374: Received PUBLISH from auto-125DDDBC-DB9D-905F-78EF-87F602FFEE20 (d0, q2, r0,
m1, 'hello', ... (5 bytes))
1729513374: Sending PUBREC to auto-125DDDBC-DB9D-905F-78EF-87F602FFEE20 (m1, rc0)
1729513374: Received PUBREL from auto-125DDDBC-DB9D-905F-78EF-87F602FFEE20 (Mid: 1)
1729513374: Sending PUBLISH to auto-E41AFA10-1E5D-E733-F44E-5750B87AF619 (d0, q2, r0,
m1, 'hello', ... (5 bytes))
1729513374: Sending PUBCOMP to auto-125DDDBC-DB9D-905F-78EF-87F602FFEE20 (m1)
1729513374: Received DISCONNECT from auto-125DDDBC-DB9D-905F-78EF-87F602FFEE20
1729513374: Client auto-125DDDBC-DB9D-905F-78EF-87F602FFEE20 disconnected.
1729513374: Received PUBREC from auto-E41AFA10-1E5D-E733-F44E-5750B87AF619 (Mid: 1)
1729513374: Sending PUBREL to auto-E41AFA10-1E5D-E733-F44E-5750B87AF619 (m1)
1729513374: Received PUBCOMP from auto-E41AFA10-1E5D-E733-F44E-5750B87AF619 (Mid: 1,
RC:0)
```

Advanced MQTT Features

Last Will and Testament (LWT)

The Last Will and Testament (LWT) in MQTT is like a backup plan for when a device unexpectedly goes offline.

- Imagine a smart device that suddenly loses its connection without warning. The LWT is a special message the device prepares in advance. If the device disconnects without saying goodbye properly, the MQTT broker automatically sends this message to let other devices know something went wrong.
- For example, in a smart home, if a sensor stops working, the LWT could notify the system to alert the user or switch to a backup sensor. It helps keep everything running smoothly even when a device fails

Retained Messages

Retained Messages ensure that the latest message is always available to new subscribers, even if they weren't connected when it was first sent.

It's like leaving a sticky note with the most recent information for anyone who checks later.

Example:

Imagine you have a weather station that publishes temperature data.

If a new device (like a phone or dashboard) subscribes to the weather topic, without retained messages, it would only get new updates and miss the latest temperature.

With a retained message, the MQTT broker keeps the most recent temperature data and immediately sends it to any new subscriber.

MQTT Security: protecting data and devices during communication

1. **Username and Password**: Devices can use a username and password to connect, like logging into an account. This helps ensure that only authorized devices can communicate.
2. **Encryption (TLS/SSL):** MQTT can use encryption (like HTTPS on websites) to keep data safe while it’s being sent, so no one can read or change the messages during transmission.
3. **Access Control**: The broker can control who can publish or subscribe to topics, making sure only the right devices can send or receive specific messages.

Scalability Considerations

In technology, **scalability** ensures that a system can grow and manage increased demand efficiently.

It means how well a system can handle growth: as more users, devices, or data are added, a scalable system can continue to work smoothly without slowing down or breaking.

Here are some simple considerations for MQTT:

1. **Broker Capacity**: As more devices connect, the broker needs to handle more messages. You may need a more powerful broker or multiple brokers working together to keep things running smoothly.
2. **Network Traffic:** More devices mean more messages flying around. To avoid slowdowns, it's important to optimize how often devices send data and keep messages as small as possible.
3. **Shared Subscriptions:** To balance the load, MQTT can use shared subscriptions where multiple devices share the responsibility of processing incoming messages.
4. **Efficient Use of Resources:** As your system grows, it's important to manage how devices use resources like bandwidth, power, and processing, so everything runs efficiently even with more devices connected.

To improve scalability in MQTT

You can focus on several key areas:

1. **Increase server capacity:** Upgrade or add more powerful servers (brokers) to handle more traffic.
2. **Horizontal scaling:** Add more servers (brokers) and use clustering and load balancing to distribute the load evenly.
3. **Optimize network traffic:** Reduce message sizes and control how often devices send data to prevent bottlenecks.
4. **Efficient resource management:** Ensure that devices and systems use bandwidth, processing power, and memory efficiently.
5. **Multiple brokers for high availability:** Use multiple brokers for fault tolerance, ensuring that if one broker fails, others can take over.

Challenges and Considerations in Real-World MQTT Deployments

Scalability and Performance
Handling Unreliable Networks
Security Concerns

Scalability: As the number of connected devices increases, brokers need to handle higher traffic. This may involve horizontal scaling, clustering, and load balancing.

Fault Tolerance: Multiple brokers may be used for high availability, so if one fails, others can take over without downtime.

Security: Brokers must ensure secure communication (encryption, authentication) to protect data and control who can publish or subscribe to topics.

Message Management: The broker must efficiently store and forward messages to ensure they reach their intended devices, even after temporary disconnects (retained messages, Last Will, and Testament).

Network:

- **Network Reliability:** In real-world environments, network connections may be unreliable. The system must be designed to handle disconnects and reconnects without losing data.
- **Bandwidth Management:** With many devices communicating, it’s important to optimize how much data is sent to avoid overloading the network.
- **Latency:** The network should be optimized to reduce delays in communication, especially for real-time applications.
- **Data Traffic:** Managing large amounts of data from many devices requires network capacity planning to ensure smooth communication and avoid bottlenecks.

Devices:

- **Power Consumption:** Many IoT devices run on batteries, so communication must be efficient to conserve power.
- **Message Delivery Guarantees:** Devices need to use the right QoS (Quality of Service) to ensure important messages are delivered reliably.
- **Efficient Resource Usage:** Devices should use bandwidth, memory, and processing power efficiently to avoid performance issues.
- **Handling Disconnects:** Devices must be able to reconnect and continue communication after losing the network connection without data loss.

Demonstration

Implementing a Simple MQTT Communication

Setup Environment

Broker: Use Mosquitto MQTT broker.

Clients: Use Python paho-mqtt library for client implementation.

Code Walkthrough

Implement a simple Python script for

- Publishing messages to a topic.
- Subscribing to a topic and receiving real-time updates.

Exploring QoS Levels

Experiment with QoS 0, 1, and 2.

Observe how message delivery changes based on QoS settings.

Thanks!