Research Phase 4 – MQTT Protocol Guide

Smart-Agro

IoT Agricultural Solutions

Digital Labs - Sri Lanka Telecom PLC

Table of Contents

MQTT PROTCOL

MQ Telemetry Transport Protocol for IoT

Introduction

MQTT is a lightweight, publish/subscribe messaging protocol specially designed for resource constrained, limited bandwidth environments. With the increasing number of cheap, low power, sensors, and other devices it is highly possible that MQTT will serve multiple use cases in the IoT. It has also become an open OASIS standard and is gaining reputation as the most popular protocol in IoT as HTTP is to the web.

The protocol is designed to work over TCP/IP and can also run-on SSL/TLS, which is a security protocol built on top of TCP/IP.

Application Layer	MQTT
Transport Layer	TCP
Network Layer	IP
Physical Layer	IEEE 802.15.4 or other …

An IP network doesn't mean that an access to internet is necessary. Therefore, MQTT can work without an internet connection. Unlike HTTP, MQTT allows messages to travel in both directions between the client and server (bidirectional), since it's built on a TCP. HTTP servers can only respond to requests from clients.

***Origin**:* MQTT was originally designed by Andy Stanford-Clark (IBM) and Arlen Nipper in 1999 for low bandwidth oil pipeline telemetry systems over satellite. Although it was first named as Message Queuing Telemetry Transport, there is no longer any message queuing in MQTT.

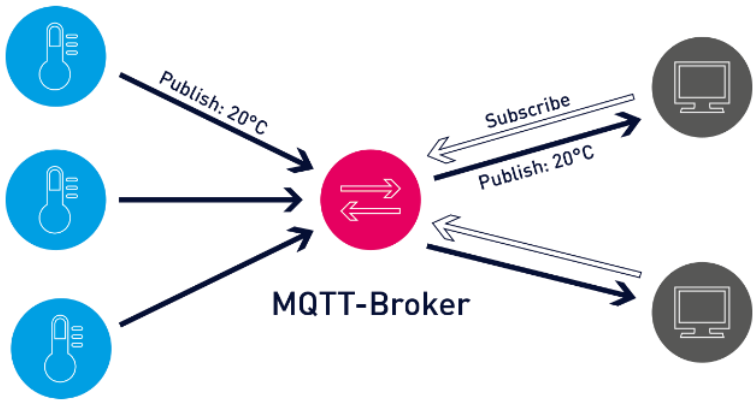
MQTT Versions

MQTT Versions

- MQTT v3.1.0 – original version.
- MQTT v3.1.1 – commonly used version currently
- MQTT v5 – a new version with currently limited use
- MQTT-SN – limited use. A version yet to be standardized

Out of these versions, the two main variants are MQTT v3.1.1 and MQTT v5

How MQTT works



The MQTT structure relies on a central broker/server to handle and distribute messages. A Publisher can publish messages to a broker on a topic, and subscribers can subscribe to those topics. In this case, both publishers and subscribers are system considered clients. In contrast to email systems, these clients do not have specific addresses. As a result, there is no simple way for a publisher and a subscriber to have a direct communication with one another.

Broker / server

- An MQTT broker functions as a go-between, allowing communication between a publishing client and a subscribing client.
- Brokers filter incoming messages based on topics and distribute them to the clients.
- They can restrict access to certain topics.
- Brokers do not store messages from any topic unless specified so.
- Commercial MQTT services provide three main options of integrating their brokers to a project
    1. Installing the broker locally
    2. Using a cloud-based server or virtual server
    3. Using a shared server application

MQTT Clients

- There are clients available for all main programming languages
- The Paho project has prepared client libraries for many of these languages.

- MQTT uses a client name (or client ID) to keep track of any subscriptions.
- These IDs are not of any significant importance. They are only used for recognition.
- But they should be *unique*. The reason is to avoid an infinite loop of disconnecting and reconnecting clients.

- If a client attempts to connect to a broker with the same name as an existing client,
    1. The existing client's connection drops.
    2. Dropped client attempt to reconnect
    3. The now existing client again drops to make space for reconnecting client
    4. Cycle repeats

Client-Broker connections

- MQTT uses TCP/IP to connect to broker
- TCP (Transmission Control Protocol) is a connection-oriented protocol with error correction, and it guarantees that packets are received in order
- The TCP connection is similar to a telephone connection. Once a connection is established, even if the clients aren't communicating any data, the connection prevails until one end client hangs up.

- UDP (User Datagram Protocol) on the other hand is a connectionless protocol. This makes UDP relatively faster than TCP. However, this efficiency is acquired at the cost of reliability of data transmission. Here the packets give much care towards an arriving order either.
- The current MQTT protocol works only on TCP. But this might change with the development of IoT and its use cases.

Establishing a Connection

Before establishing a connection between a client and a broker, several parameters should be specified. Some of them are listed below (the parameters are written in the format specified by MQTT.js library using the JavaScript client. But these are common to all languages):

- **`host`** – mqtt hosting broker/ server/ cluster address

- **`protocol`** – the protocol provided by host for communication `mqtt`, `mqtts`, `tcp`, `tls`, `ws`, `wss` (mqtts – mqqt secure, ws – websockets and wss – webscokets secure)

- **`port`** – port open for communication

- **`protocolVersion`**: MQTT protocol version number. The default is 4 (v3.1.1). Can be modified to 3 (v3.1) and 5 (v5.0)

- **`keepalive`**: The TCP/IP connection is normally left open. If no data flows for some period of time, client generates a `PINGREG` and expect a `PINGRESP` from the broker, to confirm if the connection is still open. MQTT.js library - The unit is `seconds`, the type is integer, the default is `60` seconds, and it is disabled when it is set to `0`

- **`clean`**: Whether to clear the session. When **clean sessions / non-persistent connections** are enabled, broker remembers nothing when the client disconnects. By default, the clients establish a clean session with the broker (the default is `true`). When it is set to `true`, the session will be cleared after disconnection, and the subscribed topics will also be invalid. When it is set to `false`, i.e. if the session is a **non-clean session / persistent connection**, subscriptions are remembered, and when the client reconnects, they don't have to resubscribe. Messages with QoS of 1 and 2 are held and can be received even when offline

- **`reconnectPeriod`**: Interval time given to reconnect broker (MQTT.js - The unit is milliseconds, and the default is 1000 milliseconds. **Note:** When it is set to 0, the automatic reconnect will be disabled)

- **`connectTimeout`**: It is the waiting time before receiving a connection acknowledgement - CONNACK. (MQTT.js - The unit is milliseconds, and the default is 30000 milliseconds)

- **`clientId`**: the unique ID used to identify clients. As explained before, if two IDs are not unique, the connection process will run into an infinite loop of disconnection and reconnection. (MQTT.js - The default is `'mqttjs_' + Math.random().toString(16).substr(2, 8'`, and it can support custom modified strings)

- **`username`**: Authentication username. If Broker requires username authentication, a value has to be set. This is the most basic MQTT security measure.

- **`password`**: authentication password. If Broker requires username authentication, a value has to be set. This is the most basic MQTT security measure.

- **`will`**: Last will message is specified to react when the client disconnects abnormally, the Broker will publish a message to the will topic in the format below. It is a configurable object value.
  - **`topic`**: Topic sent by the will
  - **`payload`**: the message published by the will
  - **`QoS`**: QoS value sent by the will
  - **`retain`**: whether to retain will message or not

```
const host = 'broker.mqttdashboard.com';

const port = 8000;

const protocol = ws;

const path = 'mqtt';


const broker = `${protocol}://${host}:${port}/${path}`;


const options = {

    protocolId: 'MQTT',

    protocolVersion: 4,

    keepalive: 60,

    clean: true,

    reconnectPeriod: 5000,

    connectTimeout: 30000,

    // Authentication

    clientId: 'js_Client1',

    username: 'myUserName',

    password: 'myPassword',

    // last will message

    will: {

      topic: 'willMsg',

      payload: 'Connection Closed abnormally..!',

      qos: 1,

      retain: true

    }

  }


console.log('Connecting mqtt client...');
```

```
const client = mqtt.connect(broker, options);


client.on('connect', function (connack) {
  console.log('successfully Connected with CONNACK: ' + connack);
});
```

Failed Connections

If PINGREQ and PINRESP are not successful, broker will force disconnect the client.

MQTT Publish and Subscribing

- All clients can publish (broadcast) and subscribe (receive).
- A client can only publish to one topic. It cannot publish to a group of topics.
- But message can be received by many subscribing clients

- When a client publishes a message, it needs to include:
  - Message topic
  - The message
  - QoS level
  - Retain flag (whether to retain message or not)

ex: publishing using the mqtt.js client library

```
client.publish(topic, message, [options], [callback])
```

Once a message is published:

- The broker distributes message to all subscribed clients.
- If the **Retain Flag** is false, the broker doesn't store any messages. if it's true it retains the last message with the flag.

- If no retain flags, persistent connections, QoS levels are specified:
    - Once message is sent to all clients, it is removed from the broker
    - If no clients have subscribed, the broker discards all messages
- **Last Will Message**:

  Last will message notifies a subscriber that the publisher is unavailable due to network outage. This is set by the publishing client on a per topic basis. Each topic has its own last will message. This message is stores on broker and if a connection failure occurs, the message is sent to all subscribing clients. If the publisher disconnects normally, the will message is not sent. A will message may include a connection request message also.

Subscribing to topics

- The broker cannot initiate a connection
- Only a client should establish a connection by subscribing
- A subscription has two components.
    - subscription topic
    - subscription QoS
- A client can subscribe to multiple topics
- All subscriptions are acknowledged using a subscription acknowledge message that includes a packet identifier. This is used to verify the success of subscriptions.

ex: subscribing using the mqtt.js client library

```
client.subscribe(topic/topic-array/topic-object, [options], [callback])
```

ex: unsubscribing using the mqtt.js client library

```
client.unsubscribe(topic/topic array, [options], [callback])
```

```
client.on('message', function (topic, payload, packet) {
    console.log('client received a published payload');
    console.log(`Topic: ${topic}, Message: ${payload.toString()}`);
});
```

MQTT Topics

- Topics are created dynamically by pub/sub clients when a client subscribes or publishes to a topic.
- These are not permanent unless retained message is set to true.
- They are structured in a hierarchy, using " / " symbol as the delimiter
  - ex: house/bedroom/light
- These topics are case sensitive and should be written in UTF-8 strings
  - ex: House/Bedroom/Light ≠ house/bedroom/light
- wildcard characters could be used to subscribe to multiple topics
  - **#** - multi level wildcard
  - **+** - single level wildcard
    - ex: house/# -> covers all hierarchies in the house. For example;
      - house/bedroom/light
      - house/bedroom/fan
      - house/bathroom/light
    - House/+/light -> covers all light components in the house. For example;
      - house/bedroom/light
      - house/bathroom/light
      - house/kitchen/light
    - house+ or house# has no topic level. Therefore, are invalid

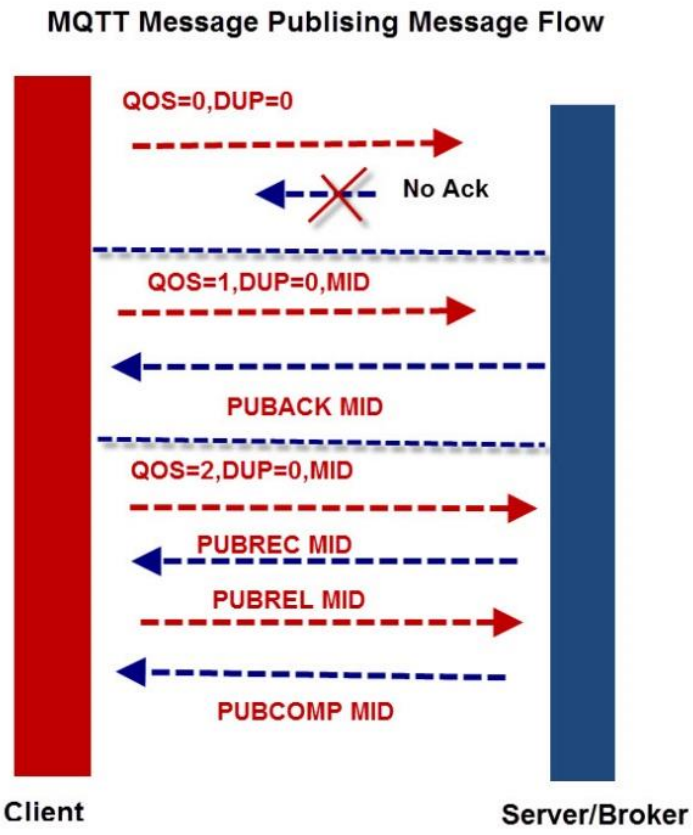
- $SYS topics are the only topics that has no standard structure specified. They are reserved topics created in broker by default. No other topic except for $SYS topic is created on a broker
- $SYS topic is used by most MQTT brokers to publish information about the broker.
- They are *read-only* topics

- A publisher can only publish to one topic. Therefore, using wildcards to publish topics are not allowed. If the same message has to be published to two topics, has to be published separately.
- Topics are removed from broker when the last subscribed client disconnects from broker while the clean session flag is true or when a new client connects with the clean session flag set to true.

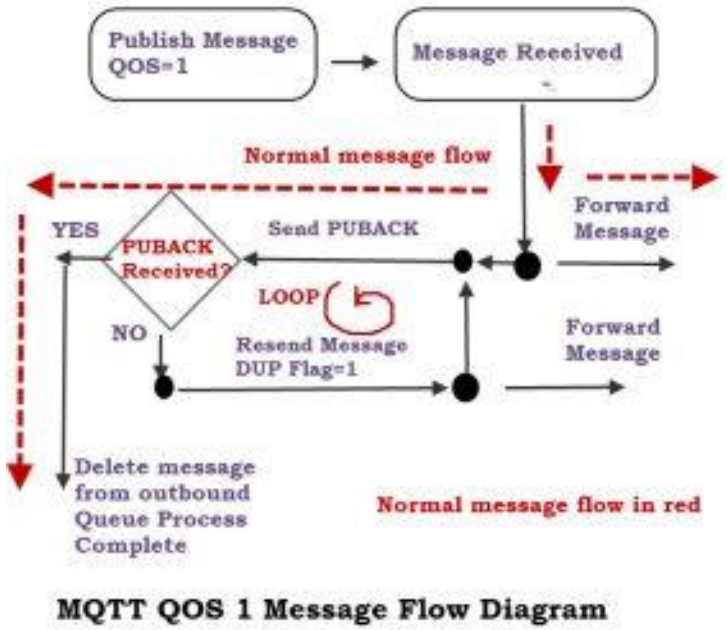
MQTT QoS

- Quality of Service (QoS) levels are designed to ensure delivery of messages. MQTT supports three QoS levels.
  - **QoS 0** – Default. No guarantee of message delivery.
  - **QoS 1** – Message is delivered at least once. Guarantees message delivery, but duplicates may be present
  - **QoS 2** – Delivers message at most once. Guarantees message delivery with no duplicates.

- QoS levels 1 and 2 could be used to ensure message delivery to a subscriber even though the subscriber is not online. Those level messages are acknowledged by the server and require a unique message id to do so. However, this may result message latency since the acknowledgement message is an added overhead.

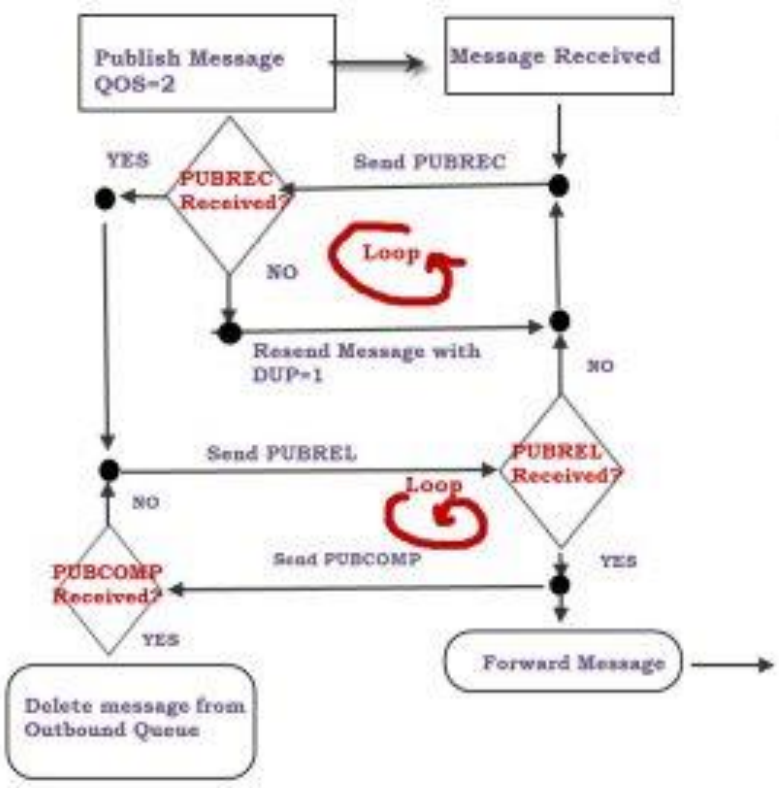

MQTT Message Publising Message Flow
QOS=0,DUP=0
No Ack
QOS=1,DUP=0,MID
PUBACK MID
QOS=2,DUP=0,MID
PUBREC MID
PUBREL MID
PUBCOMP MID
Client
Server/Broker

QoS 1


Publish Message
QOS=1
Message Received
Normal message flow
Forward
Message
YES
PUBACK
Received?
Send PUBACK
LOOP
NO
Resend Message
DUP Flag=1
Forward
Message
Delete message
from outbound
Queue Process
Complete
Normal message flow in red
MQTT QOS 1 Message Flow Diagram

1. Sends message and waits for ack. (PUBACK)
2. If ack is received, it notifies the client and deletes the message from the outbound queue
3. If ack not received, resend the message with the DUP flag set (Duplicate flag)
4. Message will be resent at regular intervals until ack is received
5. Broker forwards all messages to subs

QoS 2



MQTT QOS 2 Message Flow Diagram

1. Sender sends message and waits for ack (PUBREC) from receiver.
2. If sender doesn't receive PUBREC message, it resends the message with a DUP flag
3. If sender receives ack (PUBREC), it then sends a release message (PUBREL). Then message can be deleted from the queue
4. If receiver doesn't receive a PUBREL it will resend a PUBREC message
5. When receiver receives PUBREL, it forwards message to all subscribers and send a publish complete message (PUBCOMP) to sender

6. If sender doesn't receive PUBCOMP message, it will resend the PUBREL message
7. When sender receives PUBCOMP, it will delete the message from the outbound queue and also the message state.

The QoS level can be set by both publisher and subscriber. The overall QoS is always equal to the lowest of both. Since acknowledgement is an added overhead, it introduces latencies. QoS level 1 requires 2 messages to be exchanged while QoS level 2 requires 4 messages.

Retained Messages

The retained flag of the published topics is by default set to false. Therefore, the broker doesn't store any messages or topics. If the retain flag set to true, the last message with retain flag received by broker will be held and the published topic will be remembered.

For example, If the retain flag of a publishing sensor is set to true, when a new subscriber subscribes, they would know the current status of the sensor and receive messages. Otherwise, the subscriber has to wait until the publisher publishes to the topic again.

QoS has no effect on retained messages. This is best for devices or sensors that don't change state often. (That publish status infrequently)

Clean Sessions and published and subscribed QoS

Clean session	Published QoS	Subscribed QoS	Message stored	Subscription remembered
True	Any	Any	No	No
False	0	0	No	Yes
False	0	1 or 2	No	Yes
False	1 or 2	0	No	Yes
False	1 or 2	1 or 2	Yes	Yes

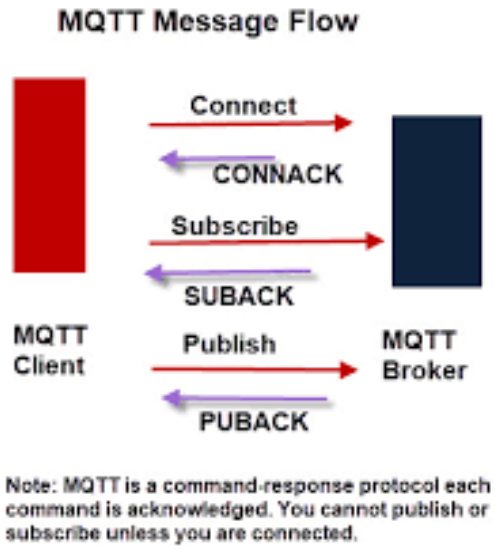
Retain Message, Clean Session and QOS Table

Clean Session Flag	Retain Flag	Subsribe QOS	Publish QOS	Published Message Always Reeeived
True	False	0	0	No
True	False	0	1	No
True	False	1	0	No
True	False	1	1	No
False	False	0	0	No
False	False	0	1	No
False	False	1	0	No
False	False	1	1	Yes – All messages
True	True	0	0	Yes –Last Message only
True	True	0	1	Yes –Last Message only
True	True	1	0	Yes –Last Message only
True	True	1	1	Yes –Last Message only
False	True	0	0	Yes –Last Message only
False	True	0	1	Yes –Last Message only
False	True	1	0	Yes –Last Message only
False	True	1	1	Yes – All messages

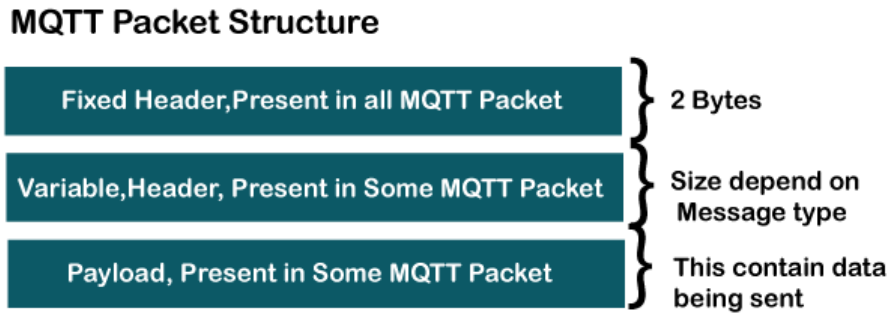
Note: QOS 1 and QOS 2 produce same result. Therefore for QOS 1 in the table read 1 or 2.

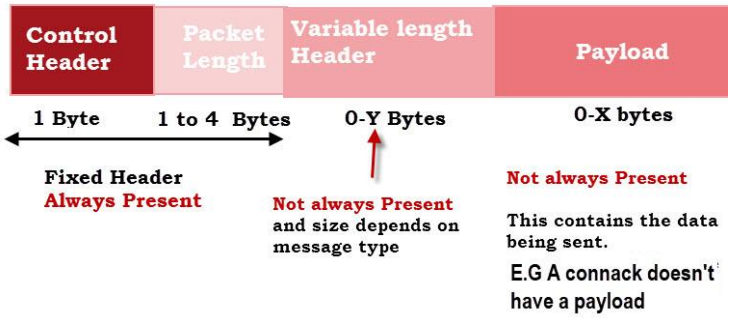
MQTT Message structure

- MQTT – binary based protocol. Control element are binary bytes.
- HTTP – Control element are text strings
- MQTT has a command-response message structure where each command is acknowledged



- Protocol information (topic names, client ID, usernames, passwords) are UTF-8 strings
- The Payload excluding protocol information is binary data





MQTT Standard Packet Structure

MQTT Security

client authentication

- Security mechanisms are configured on MQTT broker
- Most brokers provide three methods for client authentication
  - Client IDs
  - Username and Password
  - Client Certificates
- These are the most basic security measures that can be taken in MQTT

Client IDs

- The Client ID links to the topic and the TCP connection when a client subscribes to a topic.
- It should be unique as mentioned before.
- With persistent connections, broker remembers client id and subscribed topics

Username and Password

- These restricts access to broker and topics
- Username/password are transported in clear test. Therefore, they are not secure without some form of transport encryption

X509 Client certificates

- Most secure method but most difficult to implement
- Need to deploy and manage certificates of many clients
- Only suited to a small number of clients with high level of security

Securing Data

- In order to secure content of an MQTT message:
  - TLS or SSL security
  - Payload encryption

  methods can be used

TLS / SSL security

- The technology used on the web to secure TCP/IP.
- Provides an encrypted pipe down which MQTT messages can flow.
- Parts of the message can be protected.
- The main issue with this method is that it requires client support, and it is unlikely to be available on simple clients.

Payload Encryption

- This is done at application level and not by broker.
- The method encrypts data end to end
- However, it does not encrypt usernames and passwords but only the message, since it is not involved in broker configuration

MQTT over WebSockets

- The protocol of the web is HTTP. Modern browsers can also support WebSocket.

- Unlike HTTP WebSocket is an event-driven protocol, which means communication (data exchange) can be done real time. In HTTP, request updates have to be constantly made, but with WebSockets, updates are sent immediately when they are available.

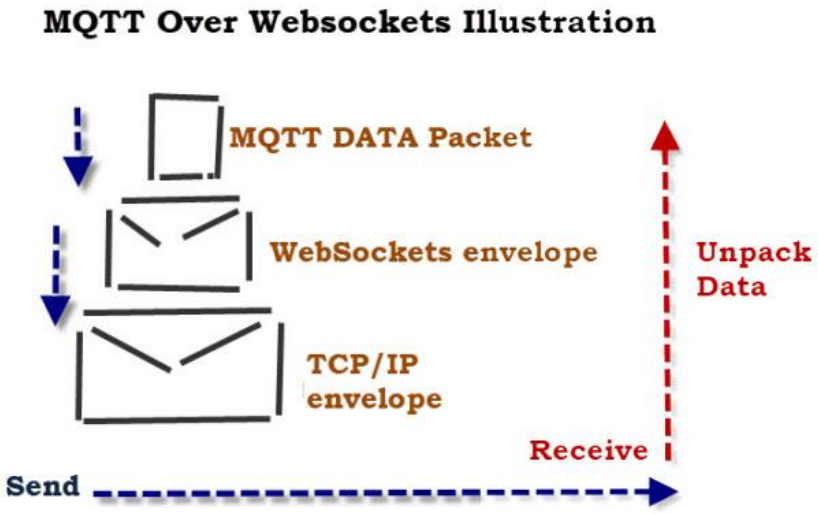
- Either way, browsers still do not have MQTT built in. Using MQTT over WebSockets allow web browsers to directly receive MQTT data.

- MQTT WebSocket support for web browsers is provided by the JavaScript client
- The two options available at the moment are:
    - MQTT.js client library (written in JavaScript for node.js and the browser.)
    - Paho JavaScript client library

- This client enables the creation of web Apps that use the MQTT protocol for displaying and sending data.

- WebSocket can set up a full duplex channel over TCP/IP. It establishes the connection initially using HTTP and then switch to WebSockets to allow clients and server to exchange binary data over the connection. Likewise, WebSockets connection forms an outer pipe for MQTT protocol to communicate.

- MQTT broker places MQTT packet in the WebSocket packet and send it to client. Client unpacks it on client side once received. In simple MQTT communication this data packet is directly placed in the TCP/IP packet.



- However, not all cloud based brokers support WebSocket. Therefore, a suitable broker has to be selected for use.

- To test connections, MQTTBox / MQTTLens chrome extensions can be used.