MQTT

Message Queuing Telemetry Transport

Why not HTTP?

- Established
- Good for request and response
- Not so good for push and quality of service
- Text based requires more bandwidth
- Acting as a host requires web server
- Requires more battery

MQTT – Message Queuing Telemetry Transport

1. What is MQTT?

MQTT is a lightweight message queueing and transport protocol.
MQTT, as its name implies, is suited for the transport of telemetry data (sensor and actor data).

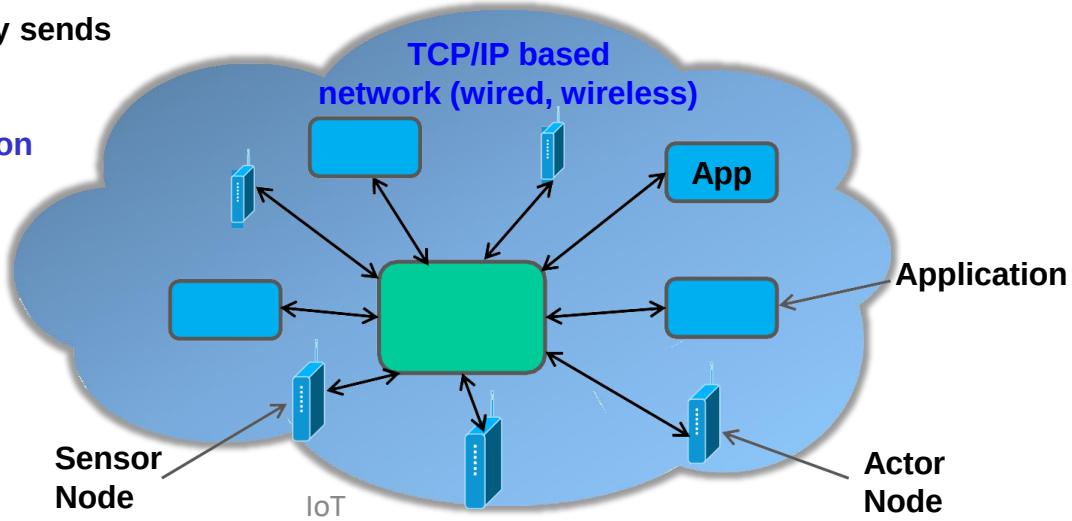
MQTT is very lightweight and thus suited for M2M (Mobile to Mobile), WSN (Wireless Sensor Networks) and ultimately IoT (Internet of Things) scenarios where sensor and actor nodes communicate with applications through the MQTT message broker.

Example:

Light sensor continuously sends sensor data to the broker.

Building control application receives sensor data from the broker and decides to activate the blinds.

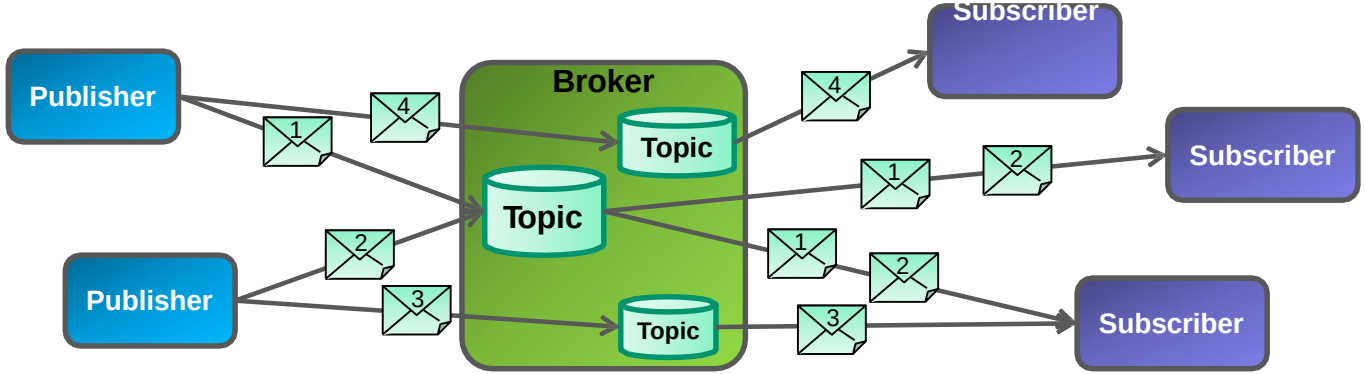
Application sends a blind activation message to the blind actor node through the broker.



MQTT Characteristics

MQTT Key features:

- Lightweight message queueing and transport protocol
- Asynchronous communication model with messages (events)
- Low overhead (2 bytes header) for low network bandwidth applications
- Publish / Subscribe (PubSub) model
- Decoupling of data producer (publisher) and data consumer (subscriber) through topics (message queues)
- Simple protocol, aimed at low complexity, low power and low footprint implementations (e.g. WSN - Wireless Sensor Networks)
- Runs on connection-oriented transport (TCP). To be used in conjunction with 6LoWPAN (TCP header compression)
- MQTT caters for (wireless) network disruptions



Origins and future of MQTT standard

The past, present and future of MQTT:

MQTT was initially developed by IBM and Eurotech.

The previous protocol version 3.1 was made available under http://mqtt.org/.

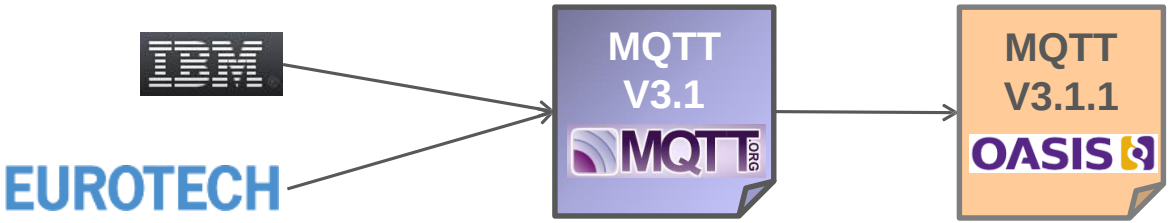
In 2014, MQTT was adopted and published as an official standard by OASIS (published V3.1.1).

As such, OASIS has become the new home for the development of MQTT.

The OASIS TC (Technical Committee) is tasked with the further development of MQTT.

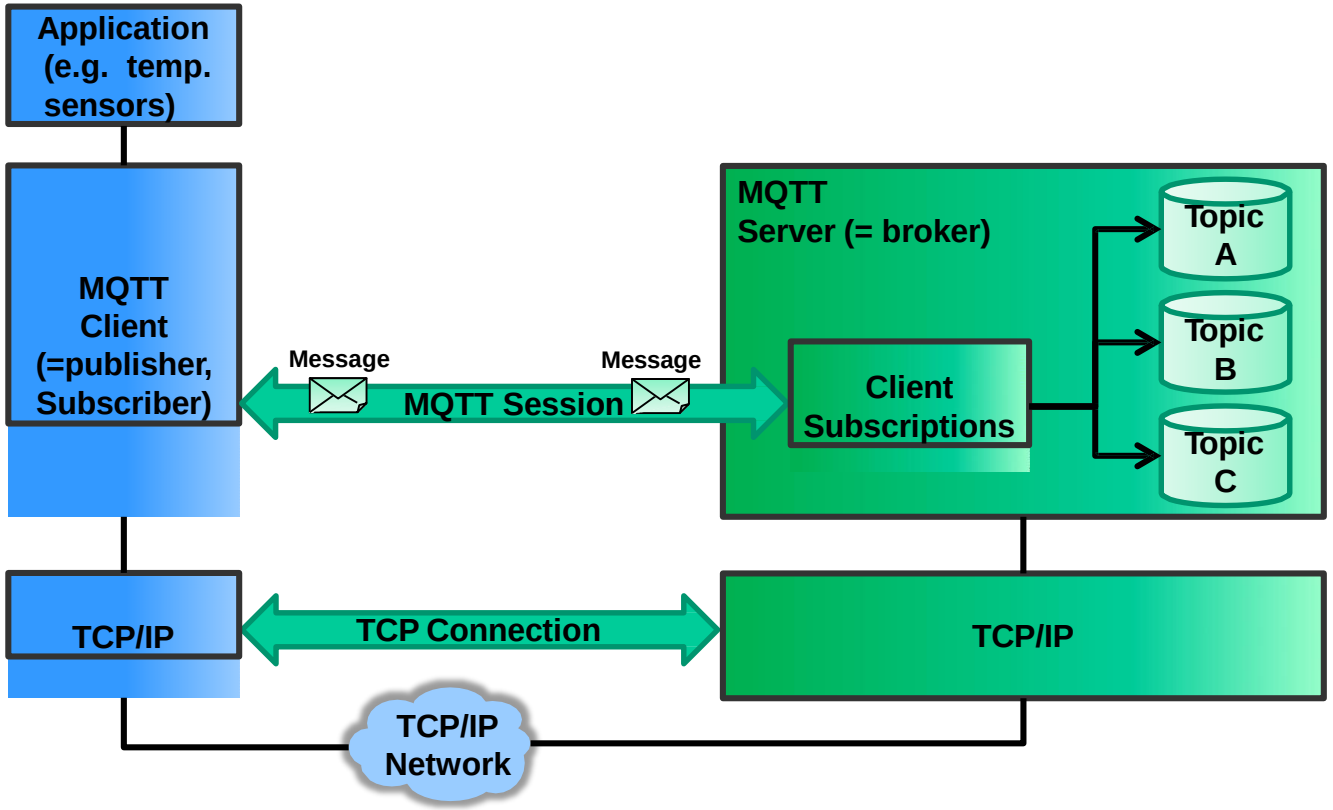
Version 3.1.1 of MQTT is backward compatible with 3.1 and brought only minor changes:

- **Changes restricted to the CONNECT message**
- **Clarification of version 3.1 (mostly editorial changes)**



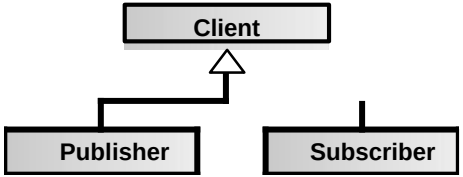
MQTT model

The core elements of MQTT are clients, servers (=brokers), sessions, subscriptions and topics.



MQTT model

MQTT client (=publisher, subscriber):
Clients subscribe to topics to publish and receive messages.
Thus subscriber and publisher are special roles of a client.



MQTT server (=broker):
Servers run topics, i.e. receive subscriptions from clients on topics, receive messages from clients and forward these, based on client's subscriptions, to interested clients.

Topic:
Technically, topics are message queues. Topics support the publish/subscribe pattern for clients.
Logically, topics allow clients to exchange information with defined semantics.
Example topic: Temperature sensor data of a building.

MQTT model

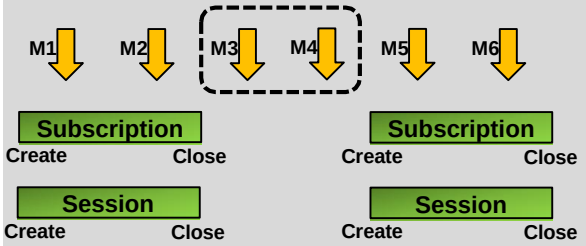
Session:

A session identifies a (possibly temporary) attachment of a client to a server. All communication between client and server takes place as part of a session.

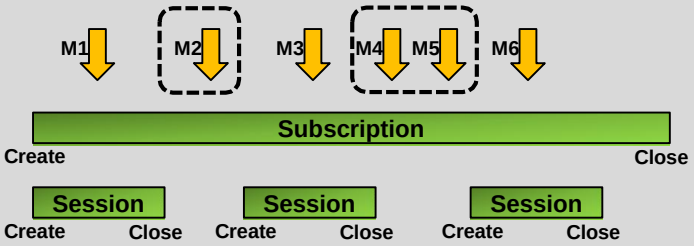
Subscription:

Unlike sessions, a subscription logically attaches a client to a topic. When subscribed to a topic, a client can exchange messages with a topic.
Subscriptions can be «transient» or «durable», depending on the clean session flag in the CONNECT message:

«Transient» subscription ends with session:
Messages M3 and M4 are not received by the client



«Durable» subscription:
Messages M2, M4 and M5 are not lost but will be received by the client as soon as it creates / opens a new session.



Message:

Messages are the units of data exchange between topic clients.
MQTT is agnostic to the internal structure of messages.

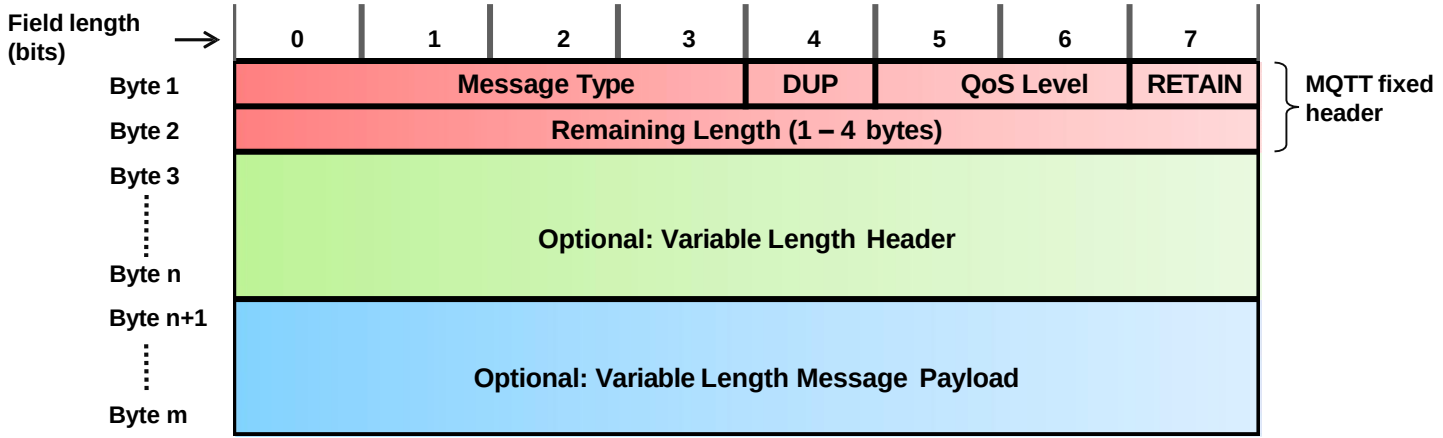
MQTT message format

Message format:

MQTT messages contain a mandatory fixed-length header (2 bytes) and an optional message-specific variable length header and message payload.

Optional fields usually complicate protocol processing.
However, MQTT is optimized for bandwidth constrained and unreliable networks (typically wireless networks), so optional fields are used to reduce data transmissions as much as possible.

MQTT uses network byte and bit ordering.



MQTT message format

Overview of fixed header fields:

Message fixed header field	Description / Values	
Message Type	0: Reserved	8: SUBSCRIBE
	1: CONNECT	9: SUBACK
	2: CONNACK	10: UNSUBSCRIBE
	3: PUBLISH	11: UNSUBACK
	4: PUBACK	12: PINGREQ
	5: PUBREC	13: PINGRESP
	6: PUBREL	14: DISCONNECT
	7: PUBCOMP	15: Reserved
DUP	Duplicate message flag. Indicates to the receiver that this message may have already been received. 1: Client or server (broker) re-delivers a PUBLISH, PUBREL, SUBSCRIBE or UNSUBSCRIBE message (duplicate message).	
QoS Level	Indicates the level of delivery assurance of a PUBLISH message. 0: At-most-once delivery, no guarantees, «Fire and Forget». 1: At-least-once delivery, acknowledged delivery. 2: Exactly-once delivery. Further details see MQTT QoS.	
RETAIN	1: Instructs the server to retain the last received PUBLISH message and deliver it as a first message to new subscriptions. Further details see RETAIN (keep last message).	
Remaining Length	Indicates the number of remaining bytes in the message, i.e. the length of the (optional) variable length header and (optional) payload. Further details see Remaining length (RL).	

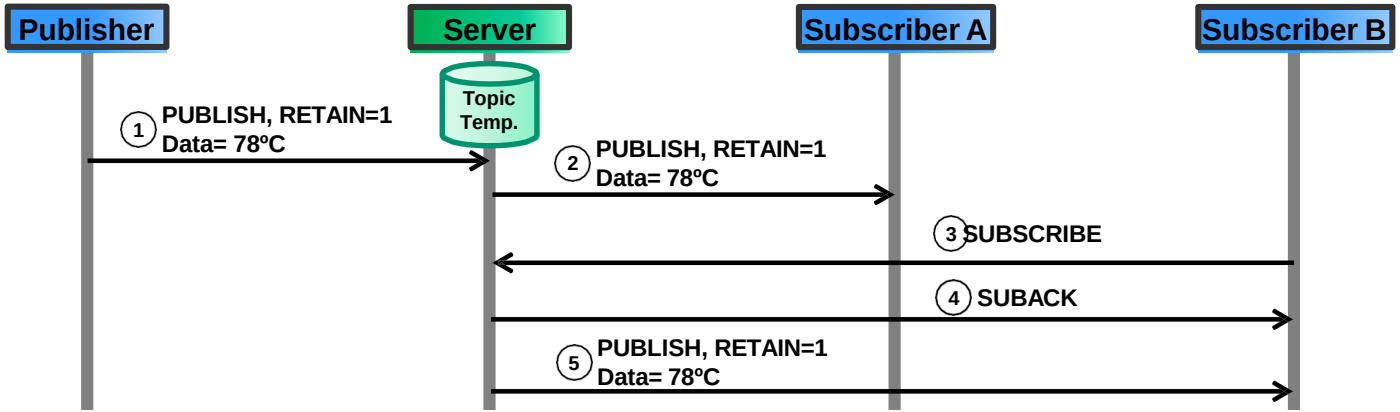
MQTT message format

RETAIN (keep last message):

RETAIN=1 in a PUBLISH message instructs the server to keep the message for this topic.
When a new client subscribes to the topic, the server sends the retained message.

Typical application scenarios:
Clients publish only changes in data, so subscribers receive the last known good value.

Example:
Subscribers receive last known temperature value from the temperature data topic.
RETAIN=1 indicates to subscriber B that the message may have been published some time ago.



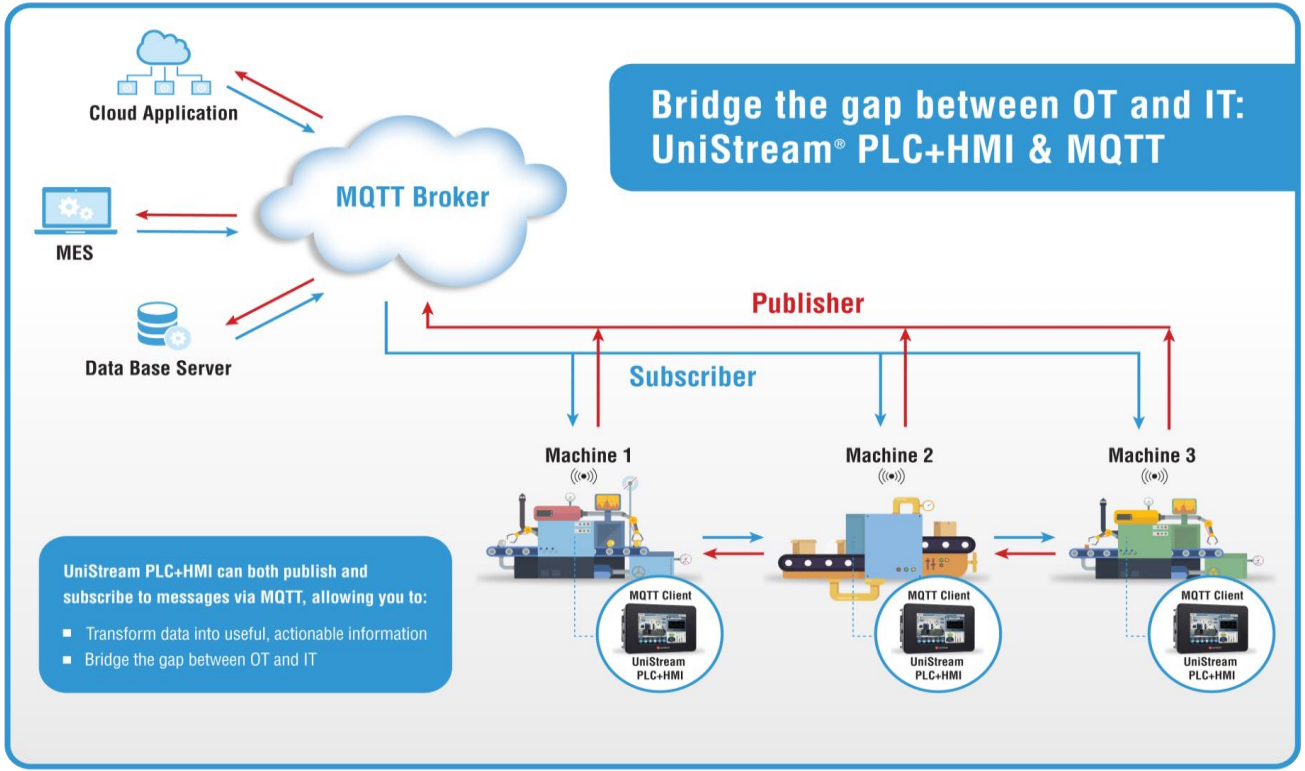
MQTT - SUMMARY

MQTT (Message Queuing Telemetry Transport) is a machine-to-machine connectivity protocol that runs over TCP/IP.

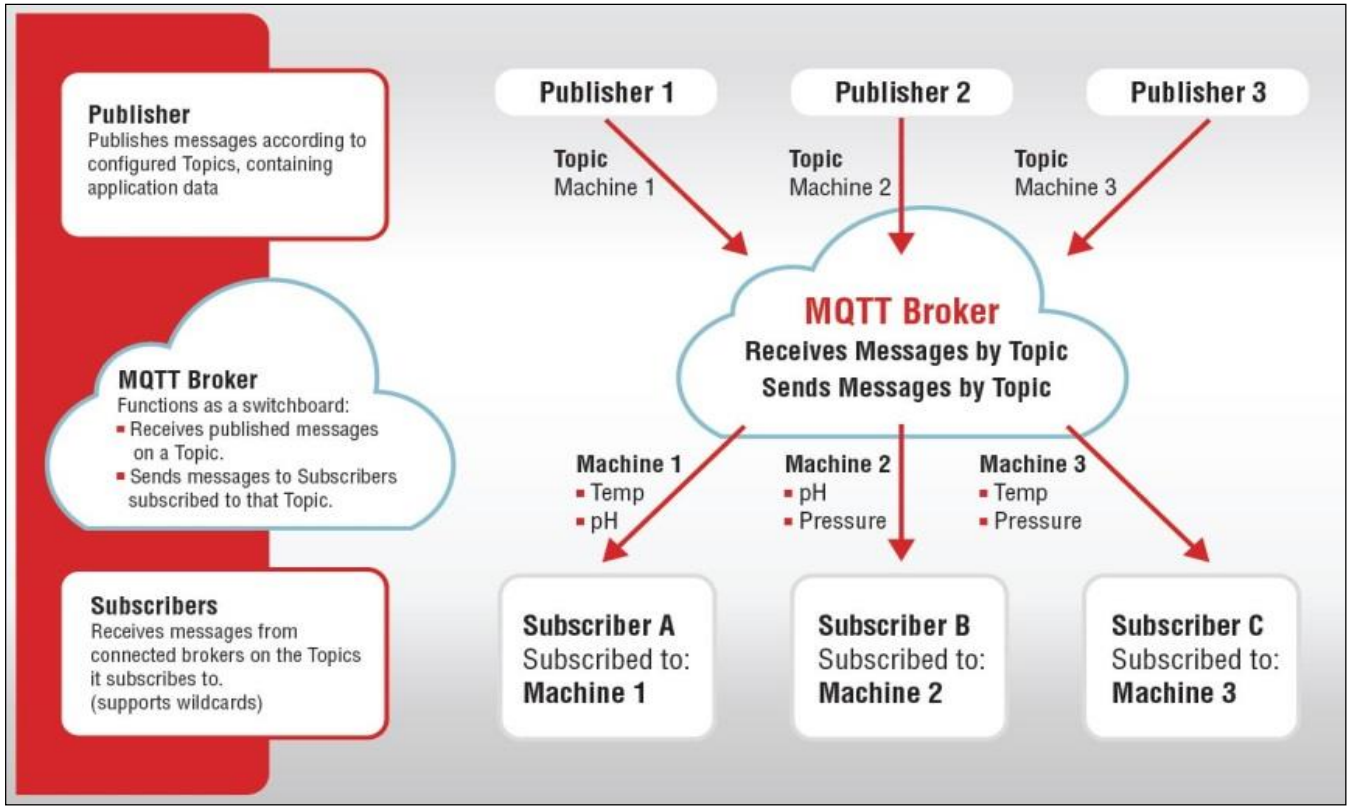
Lightweight, simple, MQTT is based on a publish- subscribe structure:

- A **Publisher** sends messages according to Topics, to specified Brokers.
- A **Broker** acts as a switchboard, accepting messages from publishers on specified topics, and sending them to subscribers to those Topics.
- A **Subscriber** receives messages from connected Brokers and specified Topics.

MQTT is based on a publish- subscribe structure



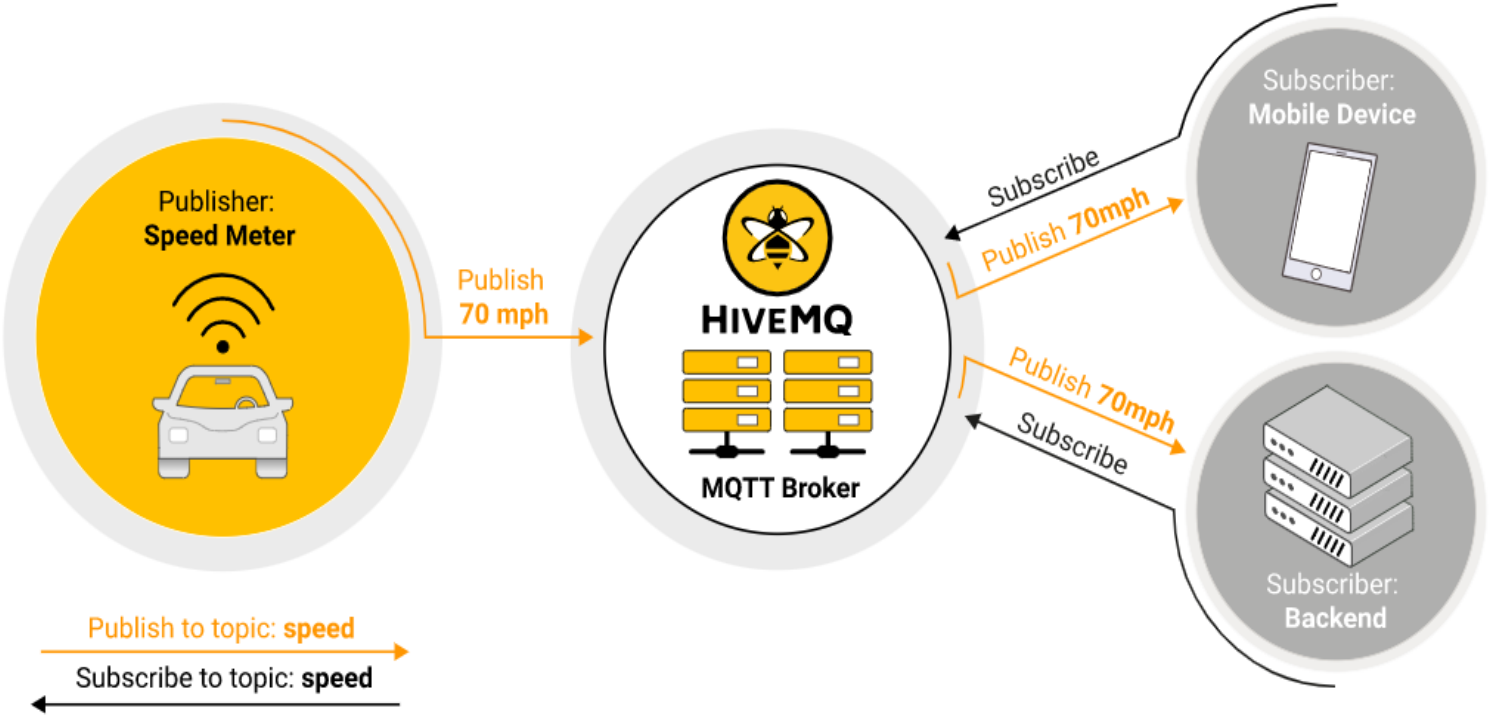
MQTT as a 'client' that can both publish and subscribe to messages



Publish-Subscribe

- The publish/subscribe pattern (also known as pub/sub) provides an alternative to traditional client-server architecture.
- In the client-sever model, a client communicates directly with an endpoint.
- The pub/sub model **decouples the client that sends a message (the publisher) from the client or clients that receive the messages (the subscribers)**.
- The publishers and subscribers never contact each other directly. In fact, they are not even aware that the other exists.
- **The connection between them is handled by a third component (the broker)**.
- The job of the broker is to filter all incoming messages and distribute them correctly to subscribers.

MQTT-Publish-Subscribe Architecture



Publish-Subscribe Messaging- One to Many

MQTT Publish / Subscribe Architecture

MQTT-Publish-Subscribe Architecture

- **MQTT embodies all the aspects of pub/sub that we have mentioned in the last slide.**
- MQTT decouples the publisher and subscriber spatially. To publish or receive messages, publishers and subscribers only need to know the hostname/IP and port of the broker.

- MQTT decouples by time. Although most MQTT use cases deliver messages in near-real time, if desired, the broker can store messages for clients that are not online.

- A Publish Subscribe messaging protocol allowing a message to be published once and multiple consumers (applications / devices) to receive the message providing decoupling between the producer and consumer(s)

- A producer sends (publishes) a message (publication) on a topic (subject)
- A consumer subscribes (makes a subscription) for messages on a topic (subject)

- A message server / broker matches publications to subscriptions
- If no matches the message is discarded
- If one or more matches the message is delivered to each matching subscriber/consumer

Publish Subscribe Messaging

- A topic forms the namespace
  - Is hierarchical with each “sub topic” separated by a /
  - An example topic space
    - A house publishes information about itself on:
      - *<country>/<region>/<town>/<postcode>/<house>/energyConsumption*
      - *<country>/<region>/<town>/<postcode>/<house>/solarEnergy*
      - *<country>/<region>/<town>/<postcode>/<house>/alarmState*
      - *<country>/<region>/<town>/<postcode>/<house>/alarmState*
    - *And subscribes for control commands:*
      - *<country>/<region>/<town>/<postcode>/<house>/thermostat/setTemp*

Publish Subscribe Messaging

- A subscriber can subscribe to an absolute topic or can use wildcards:
  - Single-level wildcards “+” can appear anywhere in the topic string
  - Multi-level wildcards “#” must appear at the end of the string
  - Wildcards must be next to a separator
  - Cannot be used wildcards when publishing

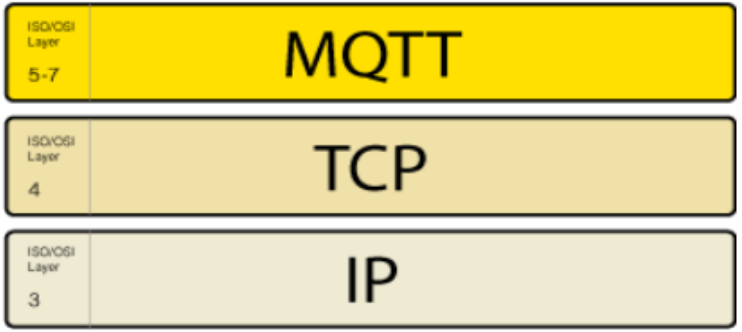
  - For example
    - *UK/Hants/Hursley/SO212JN/1/energyConsumption*
      - Energy consumption for 1 house in Hursley
    - *UK/Hants/Hursley/+/+/energyConsumption*
      - Energy consumption for all houses in Hursley
    - *UK/Hants/Hursley/SO212JN/#*
      - Details of energy consumption, solar and alarm for all houses in SO212JN

Publish Subscribe Messaging

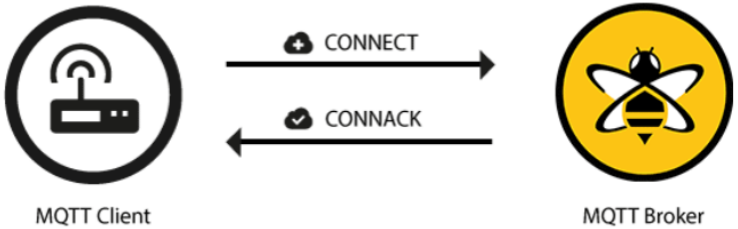
- A subscription can be durable or non durable
  - Durable:
    - Once a subscription is in place a broker will forward matching messages to the subscriber:
      - Immediately if the subscriber is connected
      - If the subscriber is not connected messages are stored on the server/broker until the next time the subscriber connects
  - Non-durable: The subscription lifetime is the same as the time the subscriber is connected to the server / broker
- A publication may be retained
  - A publisher can mark a publication as retained
  - The broker / server remembers the last known good message of a retained topic
  - The broker / server gives the last known good message to new subscribers
    - i.e. the new subscriber does not have to wait for a publisher to publish a message in order to receive its first message

MQTT Connection

The MQTT protocol is based on TCP/IP. Both the client and the broker need to have a TCP/IP stack.



- The MQTT connection is always between one client and the broker.
- Clients never connect to each other directly.
- To initiate a connection, **the client sends a CONNECT message to the broker. The broker 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.



Use of Constrained Networks

- Designed for constrained networks:
  - Protocol compressed into bit-wise headers and variable length fields.
  - Smallest possible packet size is 2 bytes
  - Asynchronous bidirectional "**push**" delivery of messages to applications **(no polling)**
    - Client to server and server to client
  - Supports always-connected and sometimes-connected models
  - Provides Session awareness
    - Configurable keep alive providing granular session awareness
    - "Last will and testament" enable applications to know when a client goes offline abnormally
  - Typically utilises TCP based networks e.g. Websockets
  - Tested on many networks – vsat, gprs, 2G….
- Provides multiple deterministic message delivery qualities of service (QoS)
  - 0 – message delivered at most once.
  - 1 – message will be delivered but may be duplicated
  - 2 – once and once only delivery
  - *QOS maintained over fragile network even if connection breaks*

Use of Constrained Devices

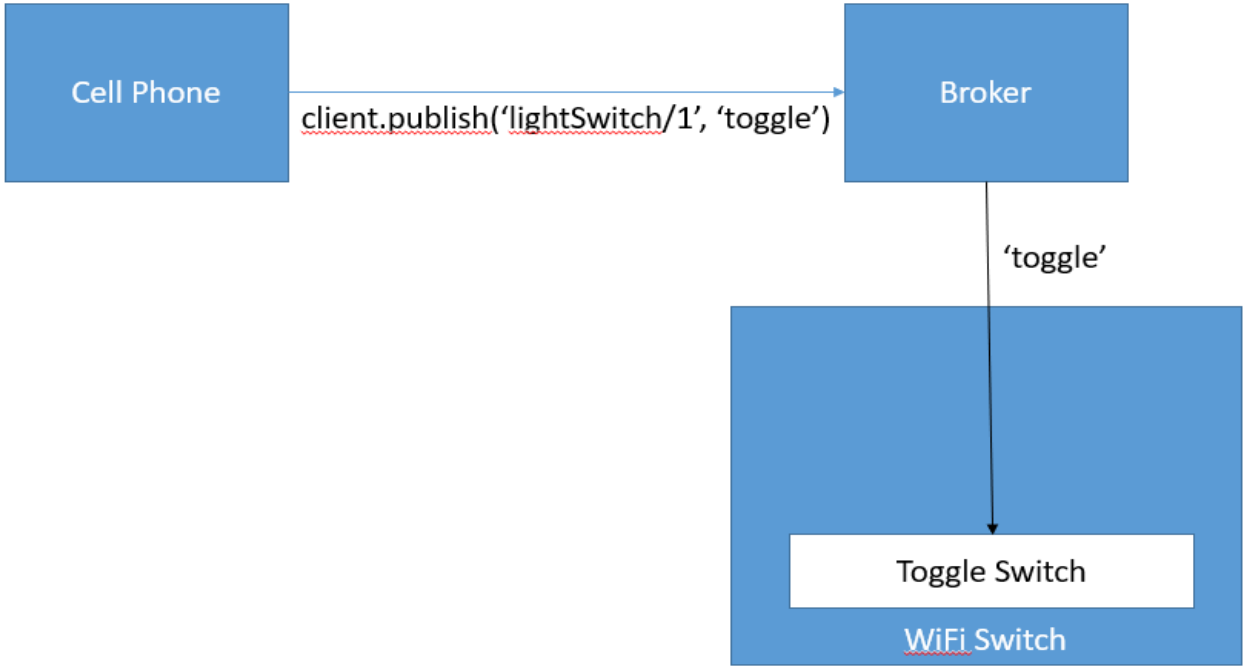
- Designed for constrained devices:
  - Suited to applications / devices that may have limited resources available
    - 8 Bit controllers upwards
    - Battery
  - Multiple MQTT client implementations available in many form factors / languages
    - Tiny footprint MQTT client (and server) libraries e.g. a c client lib in 30Kb and a Java lib in 64Kb

MQTT Clients and APIs

- You can develop an MQTT client application by programming directly to the MQTT protocol specification …… however it is more convenient to use a prebuilt client
- Open Source clients available in Eclipse Paho project
- C, C++, Java, JavaScript, Lua, Python and Go
- Clients for other languages are available, see mqtt.org/software
- E.g. Delphi, Erlang, .Net, Objective-C, PERL, PHP, Ruby
- Not all of the client libraries listed on mqtt.org are current. Some are at an early or experimental stage of development, whilst others are stable and mature.

Application

MQTT Hypothetical Light Switch



Applications

Automotive:
HiveMQ: BMW Car-Sharing application relies on HiveMQ for reliable connectivity.
https://www.hivemq.com/case-studies/bmw-mobility-services/

Logistics:
Reliable IoT communication enables real-time monitoring of Matternet's autonomous drones.
https://www.hivemq.com/case-studies/matternet

Smart Home:
IBM Telemetry use case: Home energy monitoring and control
https://www.ibm.com/docs/en/ibm-mq/8.0?topic=cases-telemetry-use-case-home-energy-monitoring-control

Transportation:
Deploying IoT on Germany's DB Railway System.
https://iot.eclipse.org/community/resources/case-studies/iot-on-railway-systems-db/