SIC

Serviços e Infraestruturas de Comunicação

MQTT

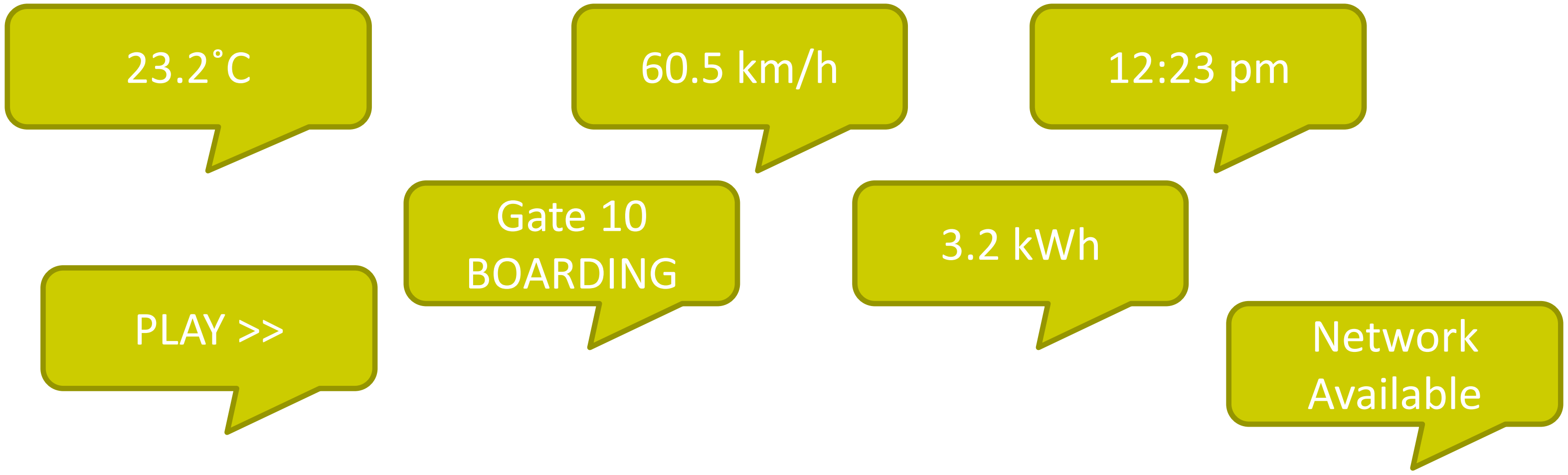
Message Queuing Telemetry Transport

Context

- Many heterogeneous IoT and M2M scenarios intrinsically require **loosely coupled communication mechanisms**, such as message queueing, for supporting telemetry and remote actuation

- However, not all message queuing solutions work well in these scenarios. Some requirements are especially relevant:
  - Low-bandwidth, high-latency, unreliable networks
  - Resource-constrained devices
  - Support for one-to-many communications

"Exposing State"



- Good at small, discrete data transfers
- Data may be triggered by local events
- Data may be read at any time by a client, in a decoupled fashion (e.g., using MQTT)
- Interface model is very simple

MQTT
Message Queuing Telemetry Transport

http://mqtt.org

Simple and lightweight publish/subscribe messaging protocol

Designed for resource-constrained devices
and low-bandwidth, high-latency or unreliable networks

Design principles: to minimize network bandwidth and device resource requirements whilst also attempting to ensure some reliability and some degree of assurance of delivery

This makes MQTT well tailored for machine-to-machine (M2M), IoT and mobile applications where bandwidth and battery power are at a premium

MQTT
Message Queuing Telemetry Transport

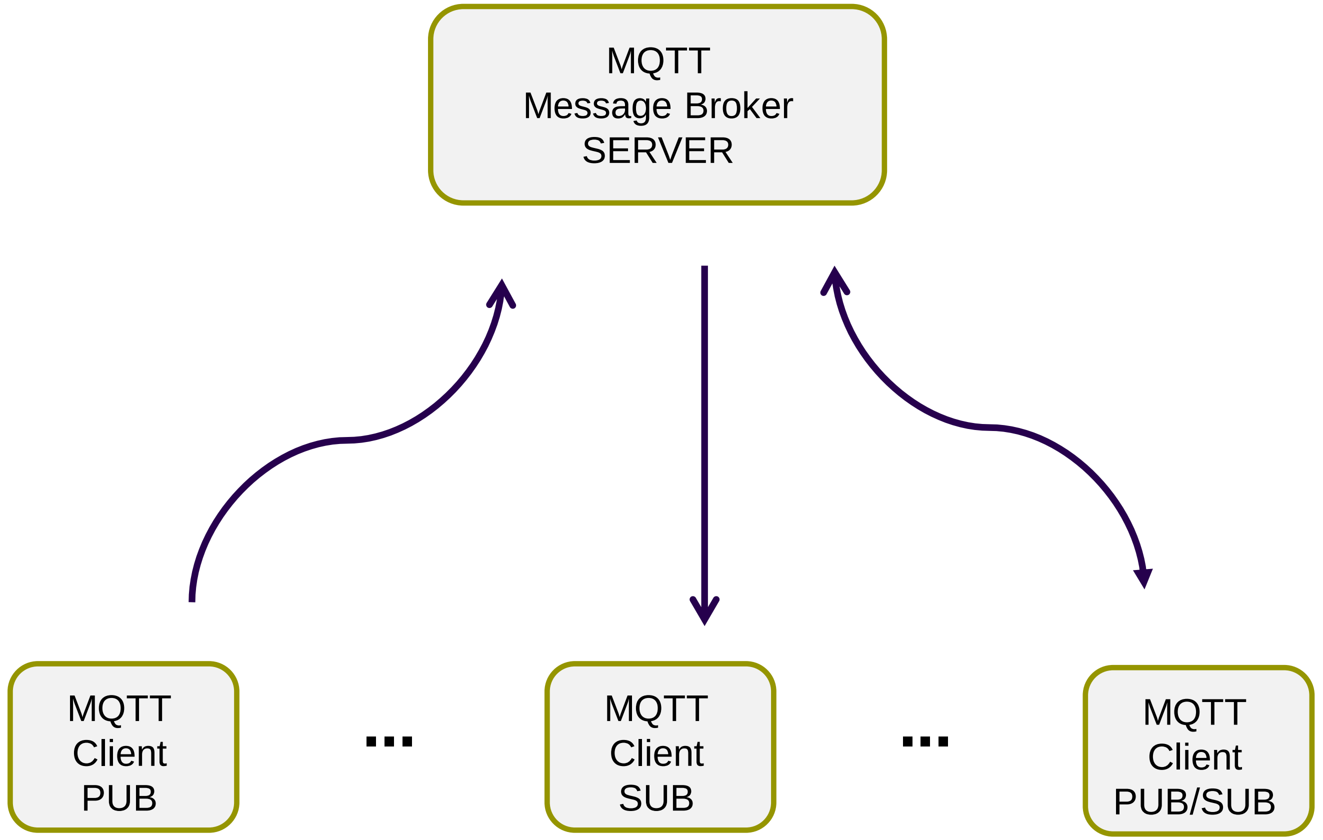
- Created in 1999 by A. Stanford-Clark (IBM) and A. Nipper (Arcom), as a connectivity protocol for Machine-to-machine (M2M and IoT)
- Has since become an OASIS (*Organization for the Advancement of Structured Information Standards*) standard (www.oasis-open.org)
- It is also an ISO standard (ISO/IEC PRF 20922)
- Public and royalty-free license
- Libraries available for Android, Arduino, Pi, C, C++, C#, Java...

- Latest version: MQTT V5.0 (http://mqtt.org)
- Internet Assigned Numbers Authority (IANA) reserved ports:
  - TCP/IP port 1883 (MQTT)
    TCP/IP port 8883 (MQTT over SSL)

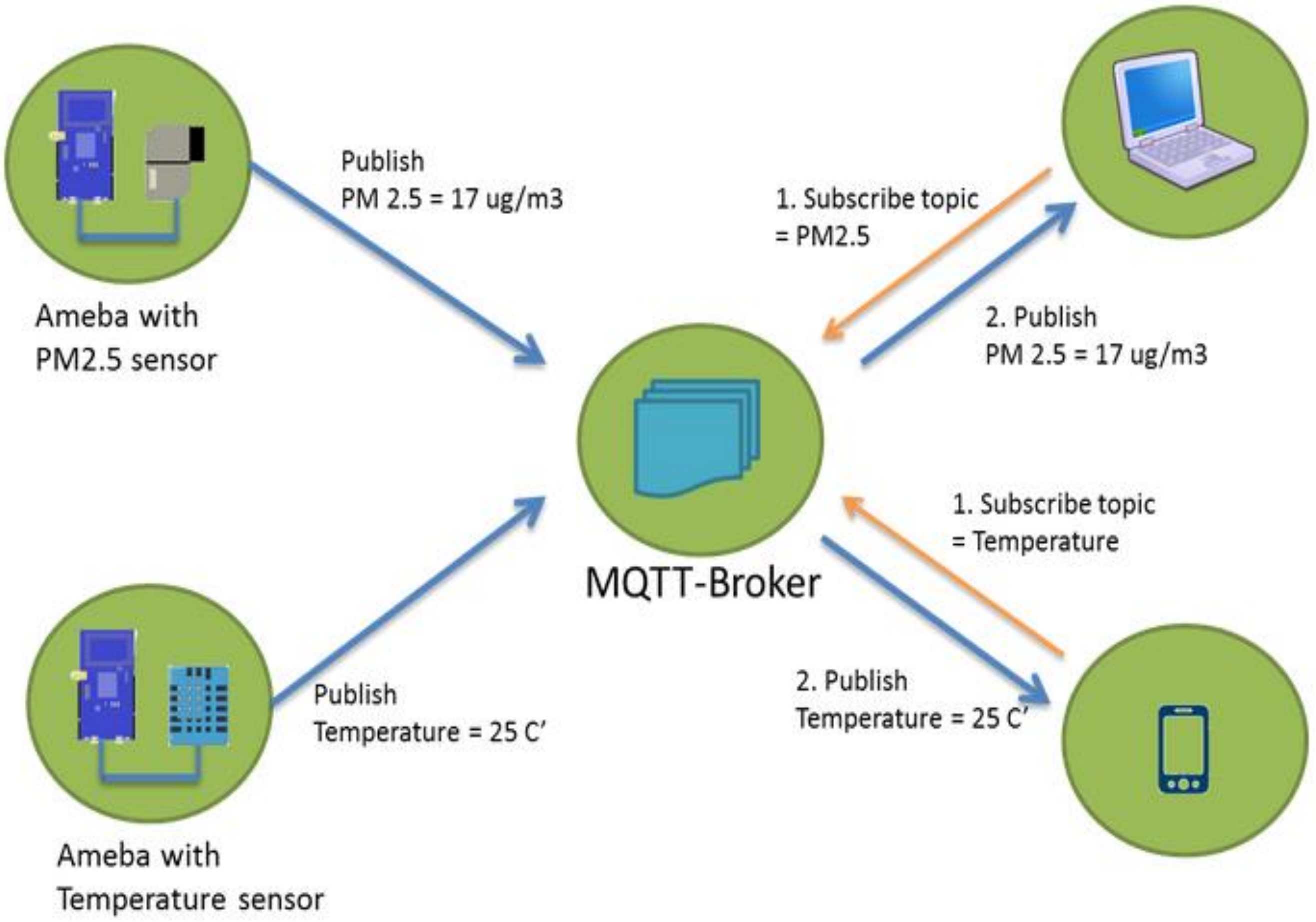
MQTT
Application Fields

- Home automation (e.g., lightning, smart metering)
- Healthcare
- Systems integration
- Mobile phone apps (e.g., messaging, monitoring)
- Industrial automation
- Automotive
- IoT applications in general

MQTT
Broker & Publish-Subscribe Model



MQTT
Broker & Publish-Subscribe Model



Source: mentor graphics

MQTT
Protocol Principles

- Based on the principle of <u>publishing</u> messages to <u>topics</u> and <u>subscribing</u> to <u>topics</u>
- Multiple clients connect to a broker (server) and subscribe to <u>topics</u> that they are interested in
- Clients also connect to the broker and publish messages on <u>topics</u>
- Several clients may subscribe to the same <u>topics</u>
- The broker and MQTT act as a simple switchboard, accepting messages from publishers of specified topics and sending them to subscribers of those topics.
- A client may act as a subscriber, as a publisher, or as both

MQTT
Protocol Features

- Use of *topics* to categorize messages
- Quality of Service
- *Retained* Messages
- Clean session / *Durable* connections
- Last Wills & Testament (LWT)
- Bridges

MQTT
Topics

- Messages in MQTT are published on topics
- There is no need to configure a topic, publishing on it is enough
- Topics are treated as a hierarchy, using a slash (/) as a separator
- This allows creation of sensible arrangements of common themes
- Wildcards can be used when subscribing:
  - **“+”** wildcard for a single level of hierarchy
  - **“#”** wildcard for all remaining levels of hierarchy

```
/Weather/sensors1/temp/temp1
/Weather/sensors2/temp/temp2
/Weather/sensors1/humidity
/Weather/sensors1/+
/Home/IOT/#
```

MQTT
Topics

Topics also provide a nice way to organize multiple message sources

For example:

/Sensors/MYHOUSE/temperature/ROOM_NAME

MQTT
Quality of Service (1/2)

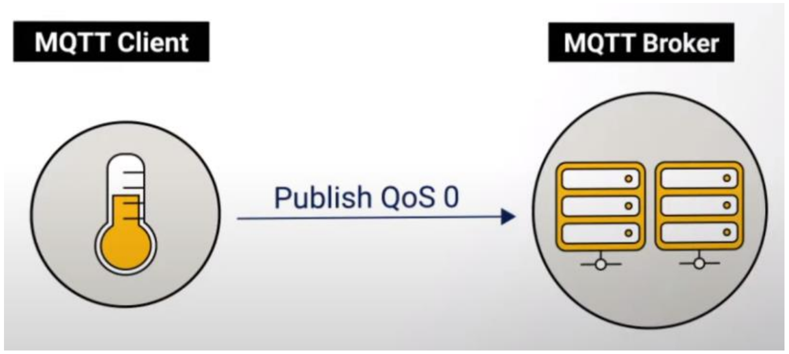
QoS settings define how hard the broker/client will try to ensure that a message is received

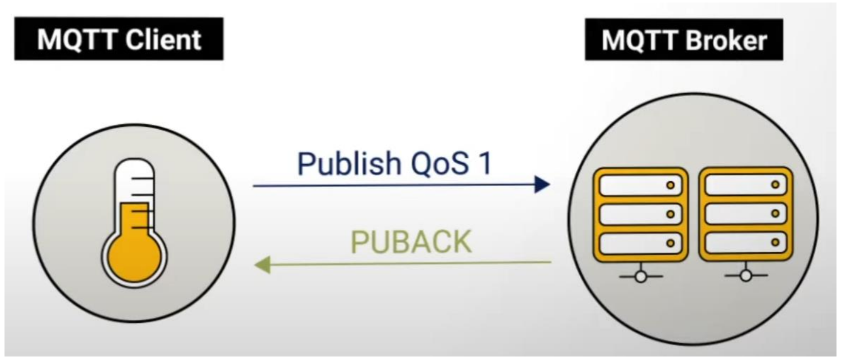
Higher levels of QoS are more reliable but involve higher latency and higher bandwidth requirements

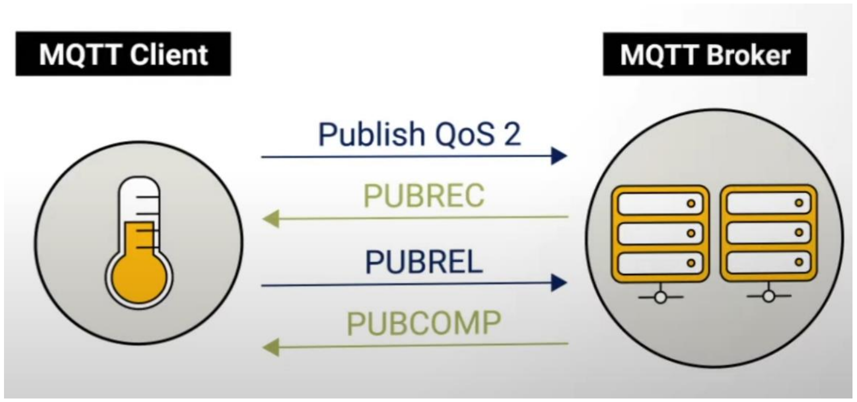
MQTT defines 3 levels of Quality of Service (QoS):

- **QoS 0** – broker/client will deliver the message at most once, no confirmation
- **QoS 1** – broker/client will deliver the message at least once, with confirmation required
- **QoS 2** – broker/client will deliver the message exactly once, by using a four-step handshake

MQTT
Quality of Service (2/2)







MQTT
Retained Messages

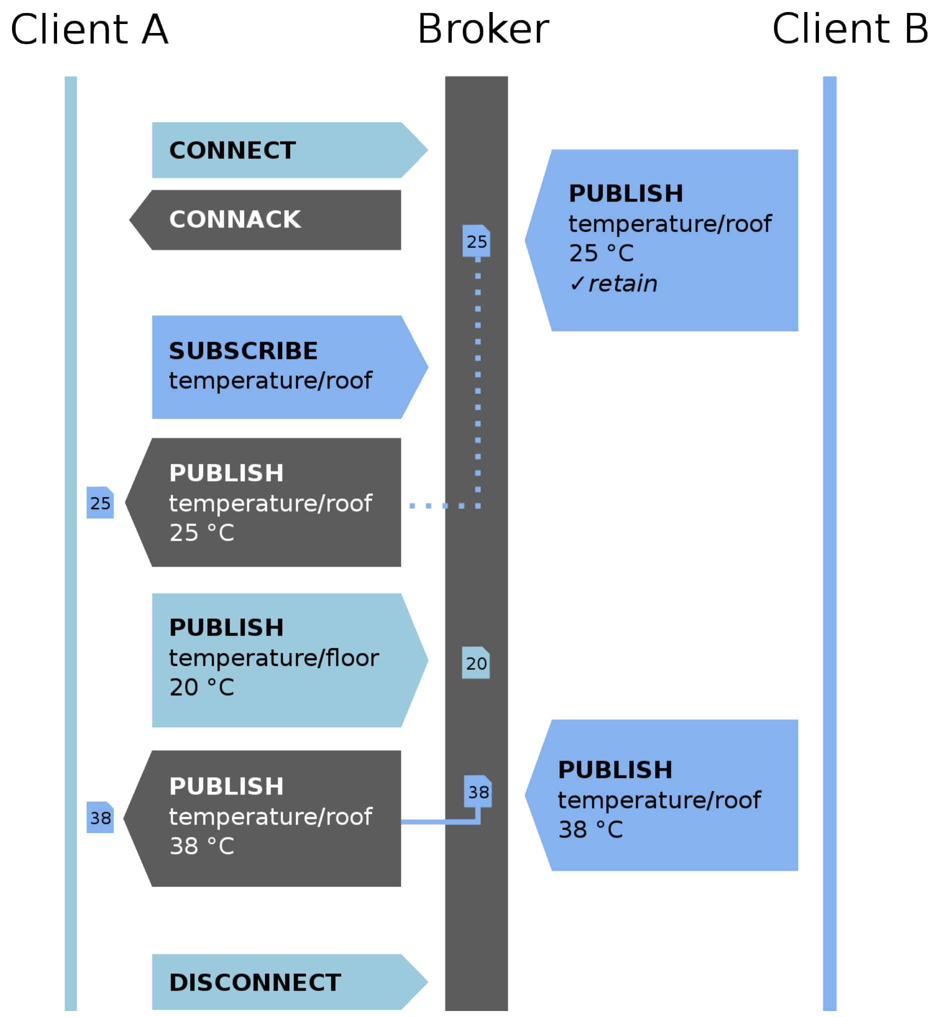
- All messages may be set to be retained
- This means the broker will keep the message even after sending it to all current subscribers
- If a new subscription is made, matching the topic of the retained message, the last (retained) topic message will be sent to the client
- This is useful as a "last known good" mechanism:
  - If a topic is only updated infrequently, without a retained message a newly subscribed client may have to wait a long time to receive an update
  - With retained messages the client will receive an instant update

MQTT

Retained Messages – example

Example of an MQTT connection (QoS 0) with connect, publish/subscribe, and disconnect.

The first message from client B is stored due to the retain flag.



Client A
Broker
Client B
CONNECT
CONNACK
PUBLISH temperature/roof 25 °C ✓retain
25
SUBSCRIBE temperature/roof
PUBLISH temperature/roof 25 °C
25
PUBLISH temperature/floor 20 °C
20
PUBLISH temperature/roof 38 °C
38
38
PUBLISH temperature/roof 38 °C
DISCONNECT

Source: Simon A. Eugster

MQTT
Clean session / Durable connections

- On connection, a client sets the "clean session" flag, which is sometimes also known as the "clean start" flag
- If a clean session is set to false, the connection is treated as *durable*
- What data is stored in persistent sessions
  - Session data (e.g., clientID), subscriptions, unACK QoS messages, queued messages
- With *durable connections* when the client disconnects, any subscriptions it has will remain and any subsequent QoS #1 or QoS #2 messages will be stored until it connects again in the future
- If a clean session is true, then all subscriptions will be removed for the client when it disconnects

MQTT
Last Wills & Testament (LWT)

- When a client connects to a broker, it may inform the broker that it has a *will* (topic + message)
- This is a message that it wishes the broker to send to a specified topic if/when the client disconnects unexpectedly
- The *will* message has a topic, a QoS policy, and a retain status just as any other message

MQTT
Bridges

- Multiple brokers (message servers) may be connected together, using the bridging functionality
- This is useful where it is desirable to share information between locations, but where not all the information needs to be shared
- By defining topic patterns and direction parameters you can control the data flow between the bridged servers
- For example:
  - Bridge messages with Topic X from Server A to B
  - Bridge messages with Topic Y from both Servers
  - Do not bridge messages with other Topics

MQTT
Other Features

- Security: authentication using username and password, encryption using SSL/TLS

- Persistence: MQTT has support for persistent messages stored on the broker (cf., durable connections)

- MQTT-SN (protocol for sensor network)
works on non-TCP/IP networks (e.g., Zigbee)

- MQTT over web sockets possible
(browser as MQTT client)

MQTT
OpenSource Implementations

Erlang MQTT Broker (EMQ)

- EMQ project, created by Feng Lee in 2012
- Fully open-source MQTT Broker written in Erlang/OTP and licensed under the Apache Version 2.0.
- Scalable open-source MQTT broker
- http://emqtt.io/downloads

Mosquitto

- https://mosquitto.org
- Use cases
  - https://mqtt.org/use-cases

MQTT
Simple Exercises…

- Think about how to use MQTT to:
  - Organize a topics' structure to receive multiple environmental data (temperature, humidity, luminosity, gas detection) from multiple rooms, in multiple buildings
  - Organize a topics' structure to receive messages related to airport flights. You should be able to subscribe:
    - all departure/arrival messages
    - all messages from a specific airline
    - all messages from a specific destination

- Sketch your solutions to pave the way for future implementations