# MQTT Primer

Version 1.00

For use with MQTT Client module versions 1.00 and above

Exported on 03/05/2019

# Table of Contents

# 1 System Overview

This introduces the main concepts.

## 1.1 Introduction

This primer introduces MQTT for those who want to implement MQTT as part of HCC Embedded's TCP/IP stack.

MQTT (originally termed Message Queueing Telemetry Transport) is a simple "publish and subscribe" lightweight messaging protocol for use over TCP/IP. It was designed to connect restricted devices in remote locations for sporadic messaging over low bandwidth, high-latency or unreliable networks, with minimal code size needed. Its original purpose was to collect data from multiple devices while using limited bandwidth and provide the information to several subscribers. It tries to ensure reliability and some degree of assurance of delivery.

MQTT is now mainly used as a Machine-to-Machine (M2M) Internet of Things (IoT) connectivity protocol. It is ideal for mobile applications because of its small size, low power usage, minimized data packets, and efficient distribution of information to one or many receivers. It involves many "clients" communicating with a centralized server ("broker") that distributes messages amongst the interested clients that have subscribed to the appropriate "topic". For example, it is used in sensors communicating to a broker via satellite link, over occasional dial-up connections with healthcare providers, and in a range of home automation and small device scenarios.

MQTT enables an embedded device to publish and receive messages from the cloud using just a few lines of code. It has minimal packet overhead compared to protocols like HTTP. Clients are easy to implement.

MQTT is an application protocol that operates over TCP, normally using one of two ports: 1883 for clear data and 8883 for connections over Transport Layer Security (TLS).

MQTT version 3.1.1 became an OASIS standard in 2014. The specification is open for anyone to implement.

## 1.2 Publish and Subscribe

As stated above, MQTT is a "publish and subscribe" protocol. In this setup:

- A client is both the producer and consumer of MQTT data.
- The publishing and subscribing elements never need to be directly connected.
- One published message can be sent to many subscribers interested in receiving information on the topic.
- Topic subscription supports a wildcard capability that can be used to describe the subscriber's interest in types of messages, or messages from a particular sender, depending on how the application's data dictionary has been defined.
- MQTT is agnostic about the content of a message payload. It does not specify the payload layout or how data is represented in a message.

A subscriber:

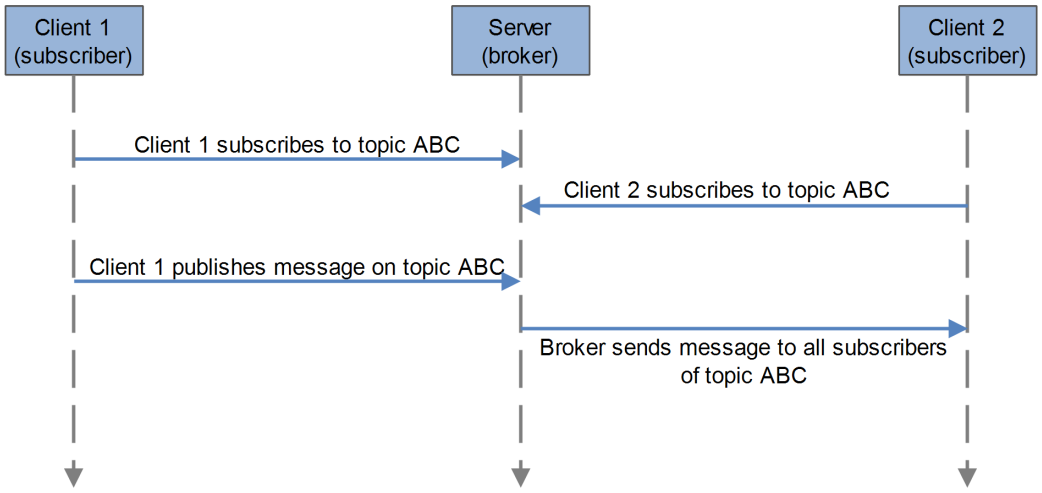
- Subscribes to one or more topics.
- Can unsubscribe from a topic at any time.
- Can receive messages from multiple publishers.

A message broker:

- Is an intermediary between subscribers. It is a server and the centralized system through which client data is communicated.
- Can be merely a 'Store and Forward' system or the main application that receives data and alerts from a sensor array.
- Does not just blindly pass data between the clients; it detects when a node drops off the network and can send memorized alerts to other entities interested in such notification.

With an application-defined "topic", a client publishes free-format data to a broker. This data is transmitted by the broker to other client(s) that have subscribed to that topic. Through use of wildcards, a single subscription can result in data from many clients being received. Similarly, data from a single publish action may be provided to many clients.

This diagram shows a simple publish and subscribe operation:




In simple terms, the process is as follows:

1. A client node comes online and connects to its broker, optionally providing id/password information and instructions stating what to do if fails to communicate within its Keep Alive period. At this time the node may also subscribe to topics.
2. Client nodes publish messages that are received by a broker.
3. As the broker receives messages containing topics matching a node's subscription filters, the packets are transmitted to that node.
4. The broker stores and/or forwards the messages to all appropriate entities that have subscribed to the message's topic with an appropriate filter.

In an MQTT system:

- There is no inherent limitation to the number of clients the architecture can support so long as the broker can keep up with all of the traffic.
- The publisher and subscriber devices can be identical, acting as both sensors and controllers.
- The client can also provide a "Last Will and Testament" packet for transmission by the broker should communication be lost.

# 1.3 MQTT Features

The features of the MQTT protocol are described below.

## Small Packet Overhead

MQTT control packet headers are kept as small as possible. A control packet has up to three parts:

- Two byte fixed header - this header is mandatory.
- Variable header - this may not be required, depending on the application. A variable header contains the packet identifier if the control packet uses these.
- Payload - if required, a payload up to 256 MB can be attached to a packet. Protocol defined values must be transmitted as UTF-8, but there are no requirements governing the format of published payloads.

This small header overhead makes MQTT appropriate for IoT applications as it reduces the amount of data transmitted over constrained networks.

It is easy to implement MQTT over a wide variety of IoT devices, platforms, and operating systems. Many MQTT applications can be developed by using just the CONNECT, DISCONNECT, PUBLISH and SUBSCRIBE control packets.

### Control Packets

This table lists all types of control packet.

| Packet | Function |
|---|---|
| CONNECT | Sent by a client requesting a connection to a server. |
| CONNACK | Sent by a server in response to a CONNECT received from a client. |
| PUBLISH | Sent by client to server or by server to client to transport an application message. |
| PUBACK | The response to a PUBLISH packet with QoS level 1. |
| PUBREC | The response to a PUBLISH packet with QoS 2. This is the second packet of the QoS 2 protocol exchange. |
| PUBREL | The response to a PUBREC. This is the third packet of the QoS 2 protocol exchange. |

| Packet | Function |
|---|---|
| PUBCOMP | The response to a PUBREL. This is the fourth and last packet of the QoS 2 protocol exchange. |
| SUBSCRIBE | Sent from the client to a server to create one or more subscriptions. Each subscription registers a client's interest in one or more topics.<br>A server sends PUBLISH Packets to the client in order to forward application messages that were published to topics that match these subscriptions. The SUBSCRIBE Packet also specifies (for each subscription) the maximum QoS with which the server can send application messages to the client. |
| SUBACK | Sent by a server to a client to confirm receipt and processing of a SUBSCRIBE Packet. |
| UNSUBSCRIBE | Sent by a client to a server, to unsubscribe from one or more topics. |
| UNSUBACK | Sent by a server to a client to confirm receipt and processing of an UNSUBSCRIBE Packet. |
| PINGREQ | Sent by a client to the server. This Packet is used in Keepalive processing. It can:<br>1. Indicate to the server that the client is alive (if the client has sent no other control packets to the server).<br>2. Request that the server responds, confirming that it is alive.<br>3. Indicate that the Network Connection is active. |
| PINGRESP | Sent by a server to a client in response to a PINGREQ. This indicates that the server is alive. |
| DISCONNECT | The last control packet sent by the client to a server. This indicates that the client is disconnecting cleanly. |

## Quality of Service - QoS

Every message sent must have a QoS level specified. The three types of QoS are as follows:

- QoS 0 (at most once delivery) - the best effort is made to deliver messages, given the underlying network. No response is sent by the receiver and no retry is performed by the sender. The message arrives at the receiver either once or never. Messages may be lost but this level has the lowest performance impact.
- QoS 1 (at least once delivery) - messages are guaranteed to arrive at least once once but duplicates may occur.
- QoS 2 (exactly once delivery) - message are guaranteed to arrive exactly once. This is the most reliable QoS but has the greatest overhead.

The delivery protocol is symmetric; the client and server can each take the role of either sender or receiver.

When a server delivers an application message to multiple clients, each client is treated independently.

IoT devices should choose the correct QoS for their requirements. This is important for maximizing performance.

## Topic-based Routing

MQTT uses a hierarchical topic-based routing scheme. This ensures fast data delivery. A topic is like a label added to every published message. It allows the broker to find all matching subscribers.

A "filter" is the string used to determine which topics are appropriate for delivery to a subscriber. Topic filters are provided by clients when they subscribe to topics and may contain topic wildcards, allowing access to multiple topics.

Topic subscription supports wildcards that can be used to describe the subscriber's interest in types of message, or messages from a specific sender, depending on how the application's data dictionary has been defined. Within topics:

- A forward slash (/) may be used to separate the levels in a topic tree and provide a hierarchical structure to topic names.
- A hash sign (#) acts as a multi-level wildcard character matching any number of levels in a topic. This wildcard represents the parent and any number of child levels.
- A plus sign (+) acts as a wildcard character that matches just one topic level.

**Note:** The following are the key points on wildcards:

- Where used, the '#' must be the final character in the topic filter.
- The single-level topic wildcard ('+') must occupy the entire level.

### Example

In this example meters in LA and SF measure temperature and power consumption:

```
meter/CA/LA/temperature/meter-id-202
meter/CA/LA/usage/meter-id-202
meter/CA/LA/temperature/meter-id-203
meter/CA/LA/usage/meter-id-203
meter/CA/SF/temperature/meter-id-678
meter/CA/SF/usage/meter-id-678
meter/CA/SF/temperature/meter-id-701
meter/CA/SF/usage/meter-id-701
```

The topic hierarchy for these meters is shown below:

```
meter
  CA
    LA
      temperature
        meter-id-202
        meter-id-203
      usage
        meter-id-202
        meter-id-203
    SF
      temperature
        meter-id-678
        meter-id-701
      usage
        meter-id-678
        meter-id-701
```

To receive messages from all the usage meters, a subscriber could subscribe using wildcards as follows:

```
meter/CA/+/usage/#
```

## "Client down" Notifications

A client can provide a "Last Will and Testament" (LWT) message to a broker when it first connects to it. If that client is disconnected uncleanly in the future, for example due to a power failure, the broker delivers its LWT message to other clients. This can be used, for example, to detect when an IoT device goes offline from a network. The LWT messages can be used to notify a monitoring application on a server.

## Retained Messages

Publishers can mark a message they send as "to be retained". Retained messages are stored permamently by a broker.

### Clean Session/Continuous Session Awareness

MQTT sessions can survive disconnection/reconnection events. If a client device goes offline for any reason, when it reconnects the session between it and the broker is resumed. The session state can be preserved over these events and when the client reconnects any outstanding messages are delivered to it.

## 1.4 MQTT Security

There are two ways to obtain secure MQTT connections.

### Using TLS

When a secure connection is required, the MQTT client can use TLS to create this connection, then use that connection to perform I/O. Encryption is handled by TLS, independently of MQTT itself but with added overheads. An application may encrypt the data that it sends and receives, but again this is independent of MQTT.

Port 8883 is used for connections over TLS. The use of TLS is largely transparent to the user of the MQTT API. For more information on HCC's TLS, see the TLS and DTLS User Guide.

### Authentication and Authorization

MQTT version 3.1 allows use of a user name and password within a packet to allow a client to authenticate itself with the broker.

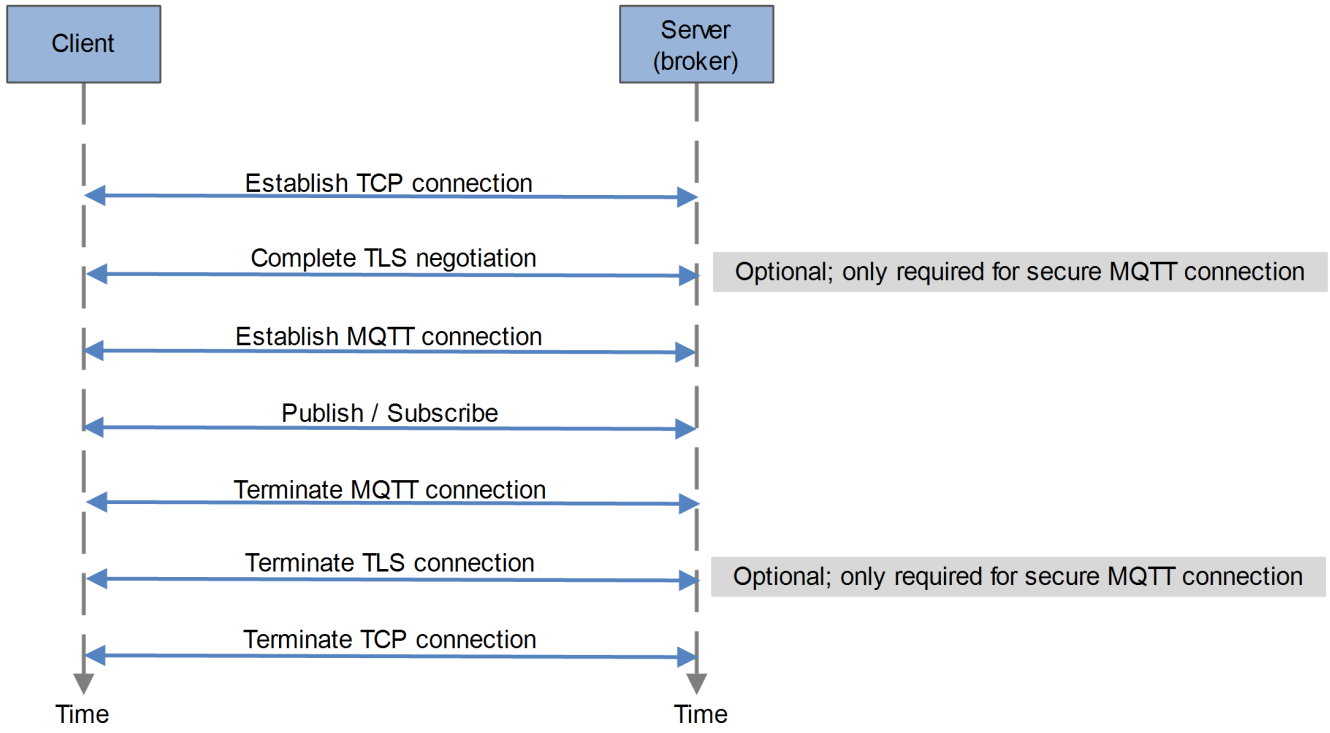
Additionally, when MQTT is run over a TLS connection, both the client and the server can authenticate each other by using X.509 certificates.

## 1.5 Connection Setup

This section describes MQTT's use with other protocols in the TCP/IP stack.

An MQTT client must first establish a TCP connection with the broker. After the TCP connection has been established, the MQTT client sends a CONNECT message to the broker, and waits for the receipt of a CONNACK, indicating a successful connection. A secure MQTT connection requires the successful completion of TLS negotiation between the client and the broker before the MQTT connection can be established.

The following diagram illustrates the order in which various protocols involved in a MQTT connection exchange messages with their peer entities:



# 2 Real World Use Cases

This section gives real world IoT use cases, showing how MQTT is being used. The first example shows use of MQTT for data collection and analysis, the others show its use in control systems.

To give an idea of how widely MQTT is used for IoT systems, all the following use MQTT for messaging:

- Amazon Web Services (AWS).
- AirVantage M2M Cloud
- IBM Bluemix
- Carriots
- CloudOne IoT Platform
- Everyware Device Cloud by Eurotech
- EVRYTHNG Engine
- Golgi
- Oracle Internet of Things Platform
- Pivotal Cloud Foundry
- ThingWorx
- Xively by LogMeIn

# 2.1 Example 1

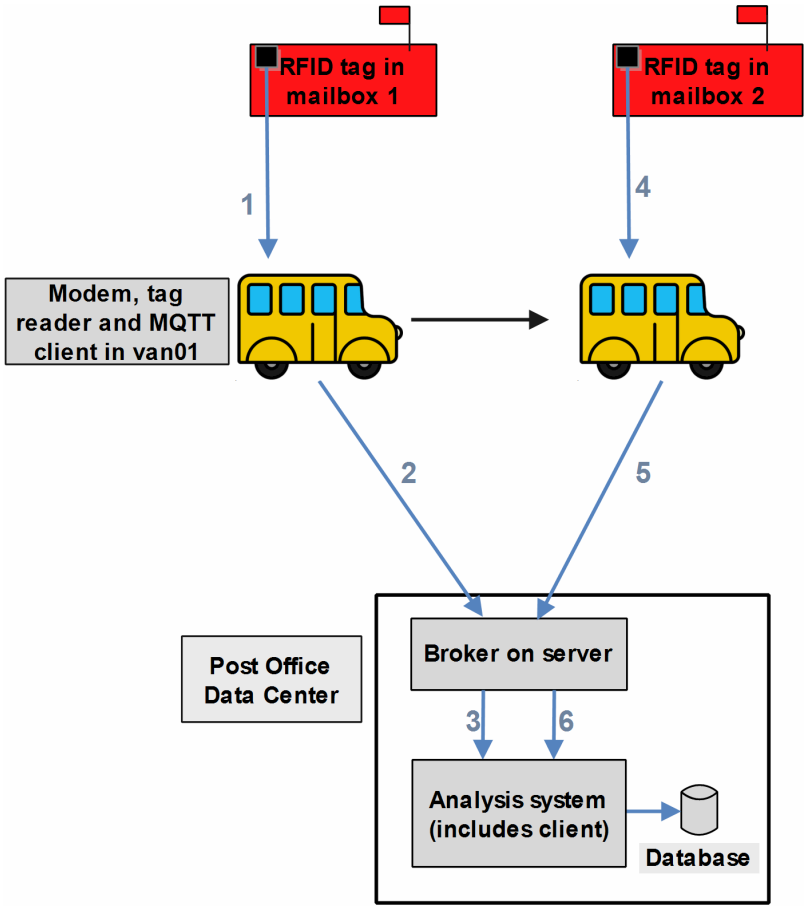
In this example MQTT is used to gather data easily for future study. Here the Post Office is interested in recalculating delivery routes by using Time and Motion analysis.

## Method

Their chosen method is the following:

1. With the homeowners' permission, place an RFID tag in every $N^{th}$ mailbox along each letter carrier's delivery route.
2. Attach a device containing a wireless modem, a tag reader, and an MQTT client in each delivery vehicle (or rotate these between them).
3. As the mobile device comes within range of a tag, it sends a simple message containing the tag information to the broker.
4. The broker forwards the message to a database-attached MQTT client that stores this information along with the current time stamp.

Over a period of six months millions of records are collected (250K addresses * 6 days * 30 weeks = 39M pieces of raw data).



## Sequence

The sequence of steps in the diagram is as follows:

1. As post van van01 approaches Mailbox 1, its tag reader detects the tag in the mailbox. Its MQTT client app publishes a message to topic "RFID_Van001". The modem sends this.
2. The broker running on the server at the PO Data Center receives the message.
3. The client in the Analysis System subscribes to all Van# topics so receives the message about Mailbox 1. The data is time-stamped and saved to the database.
4. As the van approaches the next tagged mailbox, steps 1 to 3 are repeated (labelled 4 to 6 in the diagram).

## Analysis

Graph Theory techniques can now be used to analyze the flow rate data of the measurement area, enabling:

- Identification of bottlenecks in delivery rate.
- Simulation of route changes.
- “Workarounds” that can be put into place for road repair, construction projects, traffic predictions based on scheduled sporting events, or protest marches.
- “Time vs Dollar” calculations for:
    - Vehicle repair
    - Truck vs on-foot delivery
    - Use of electricity vs gasoline
    - Use of alternate routes based on predicted weather (if it's snowing, deliver to Elm Street in the morning shortly after the snow plow clears the street).

Keywords here are the following:

- Time and Motion studies.
- Bottleneck Analysis.
- Graph theory.
- Energy consumption calculations.
- End-customer satisfaction.

## Summary

Note the following:

- By using MQTT in an embedded device, a tremendous amount of data can be sent to the broker, where it is stored in a database for delayed “Big Data” analysis.
- Simple devices using existing and well-known technologies can be deployed to a variety of markets, supporting "Big Data" analysis.
- MQTT can simplify and separate the data gathering activities from the rest of the project.

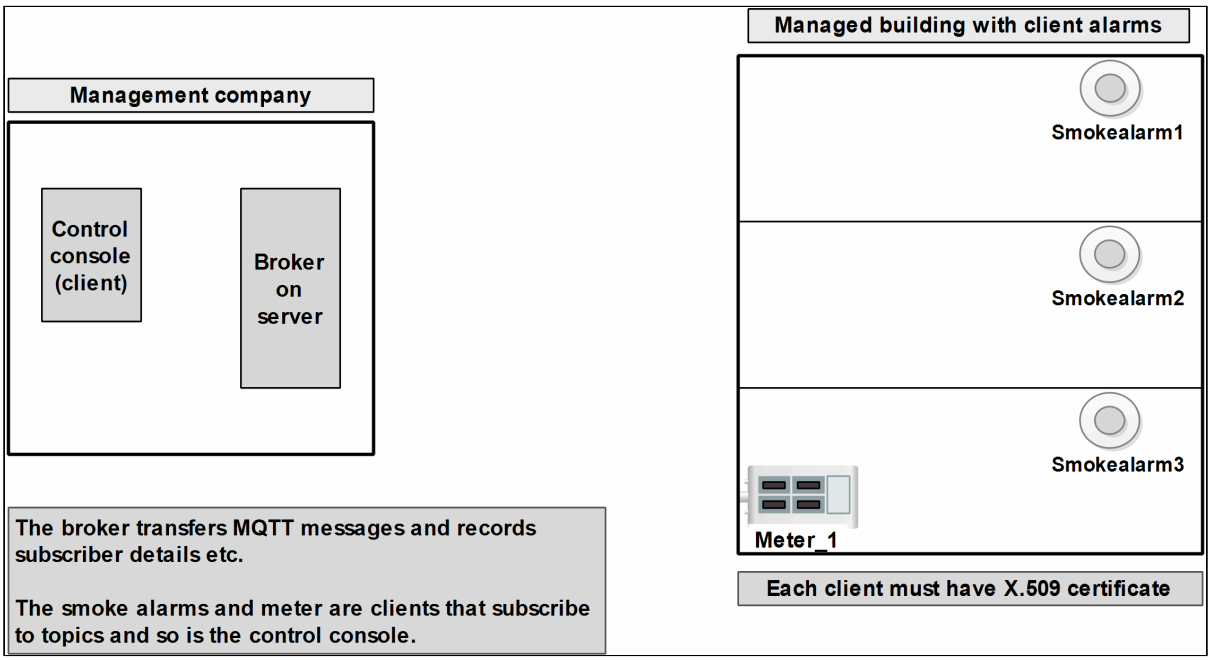
# 2.2 Example 2

The diagrams below show use of MQTT for messaging in a building control system.

## Initial situation

In this setup:

- The management company office is remote from the managed building itself. The building is one of a series of managed buildings and is named "building2".
- The building is monitored and managed from the console in the management company's office. The console is an MQTT client.
- The smoke alarms in the building all function as clients.
- There is a single power meter in the building.




**Note:** Although the broker and console client are shown as separate entities above, they could both be on the same logical device.

### Addresses

The addresses in this system are:

- Management console - Console1
- Alarms - Building2/Smokealarm1, Building2/Smokealarm2 and Building2/Smokealarm3
- Meter - Building2/Meter1

## Topics

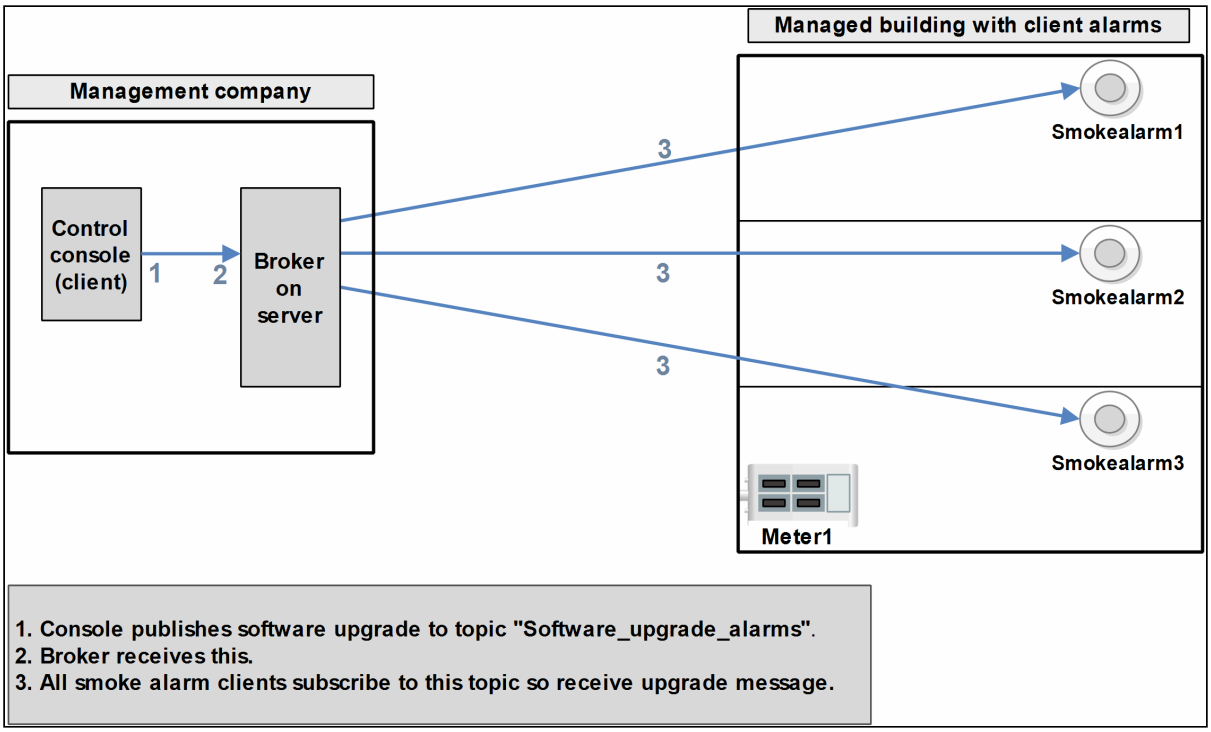
The clients must subscribe to topics as shown in this table:

| Topic | Subscribers | Description of Use |
|---|---|---|
| Software_upgrade_alarms | Console, Smoke alarms | The console publishes to this topic to upgrade the smoke alarms. |
| Fault_report_alarms | Console, Smoke alarms | A smoke alarm publishes to this topic to notify the manager of a fault. |
| Smoke_report_alarms | Console, Smoke alarms | A smoke alarm publishes to this topic to notify the manager of smoke detection. |
| Meter_reading | Console, Meter1 | The meter publishes to this to send the console a reading. |

The following sections show how messages sent to each topic may be used.

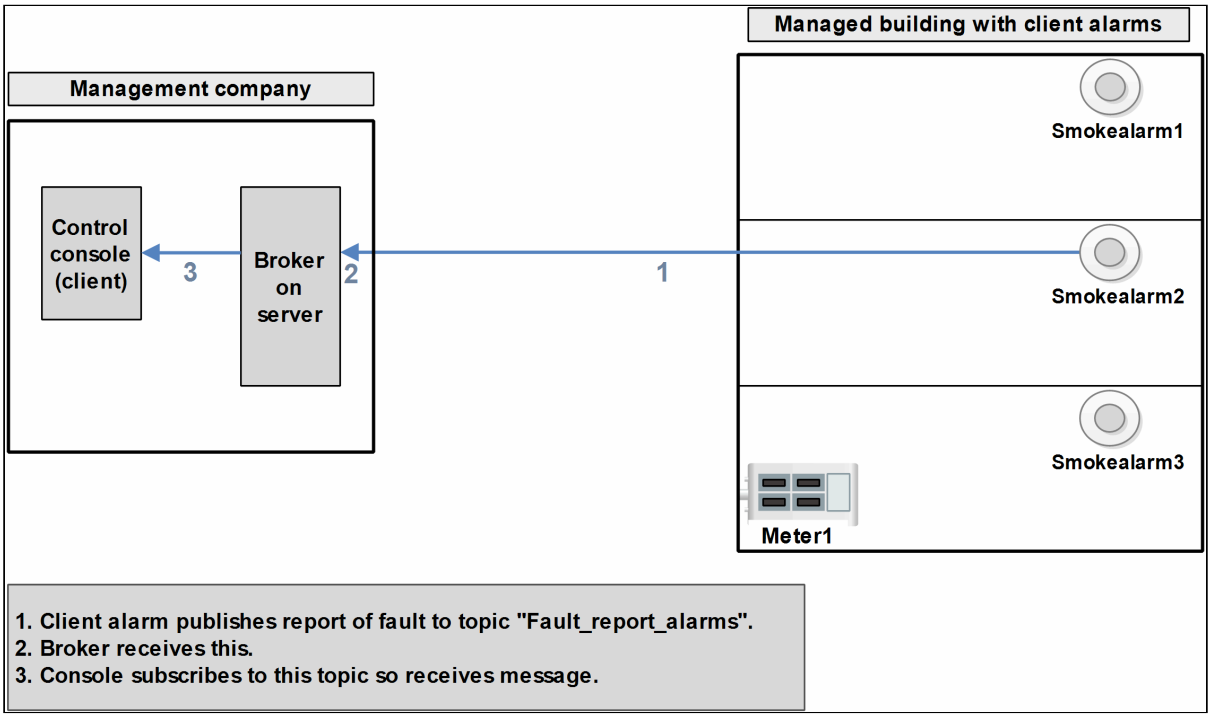
## Software upgrade

In this case the payload of the message contains the information needed for the upgrade. The QoS of this message should be 2 (message are guaranteed to arrive exactly once).



## Fault report

In this case the payload of the message contains information on the smoke alarm fault. The QoS of this message should be 1 or 2, guaranteeing the delivery of the message.



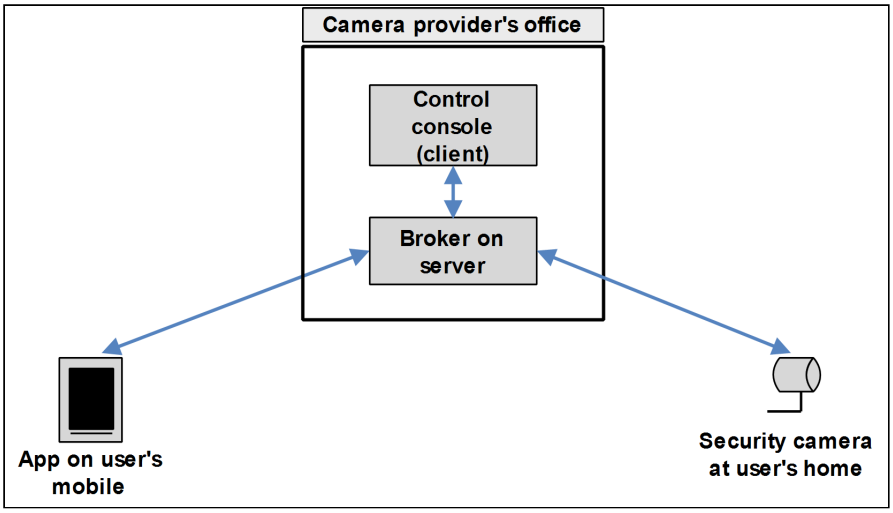
# 2.3 Example 3

The diagrams below show use of MQTT for messaging between a mobile phone app and the home security camera that it controls.

## Initial connection

This setup features:

- A home security camera managed by a camera provider company. This is an MQTT client named "Camera001".
- An app provided by the camera provider and installed on the home owner's mobile phone. The app appears as an MQTT client named "Viewcam001" on this phone.
- A broker application running on a server at the camera provider company.
- An application running on a PC at the camera provider company. This allows the camera provider to intervene to check the camera, upgrade its software, and so on in the same way as for the previous example. This is an MQTT client named "Console001".




### Addresses

The addresses in this system are:

- Home security camera - Camera001.
- Management console at camera provider company - Console001.
- Mobile phone app - Viewcam001.

## Topics

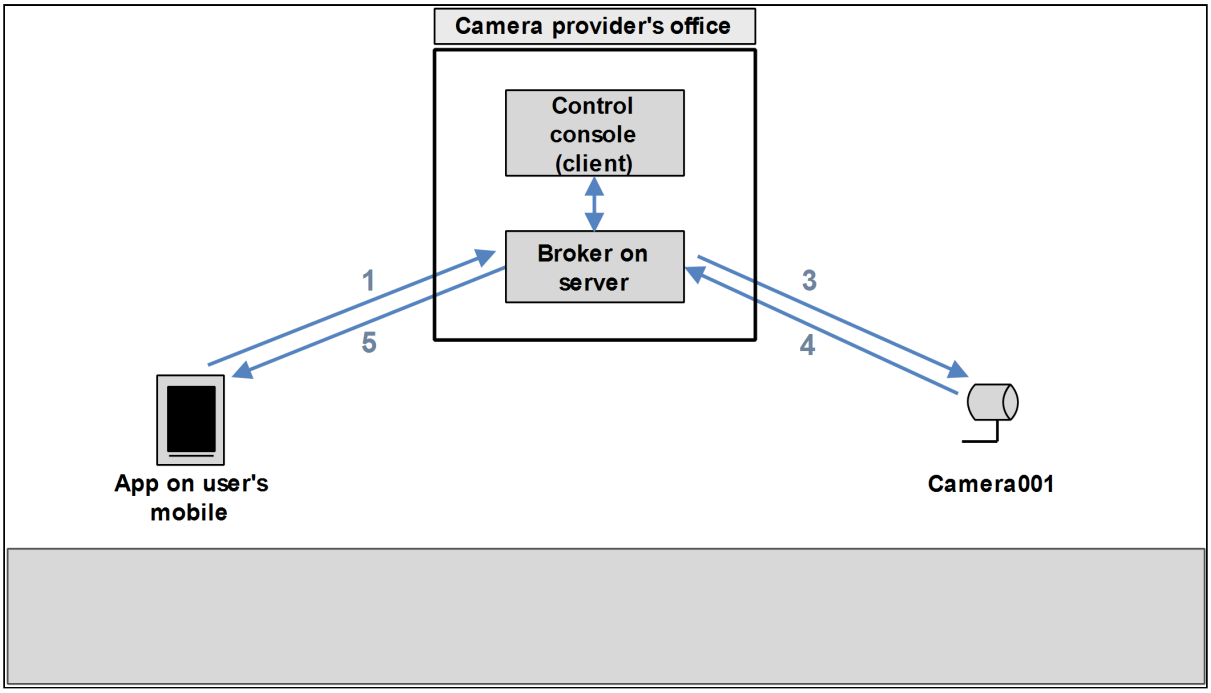
The clients must subscribe to topics as shown in this table:

| Topic | Subscribers | Description of Use |
|---|---|---|
| Check_camera001 | Console, App on mobile, Camera | The mobile app publishes to this topic to check the security camera. |
| Alarm_camera001 | Console, App on mobile, Camera | The camera publishes to this topic to notify of suspicious activity. |
| Video_camera001 | Console, App on mobile, Camera | The camera publishes to this topic. The payload is a minute's video from the camera. |

The following sections show how messages sent to each topic may be used.

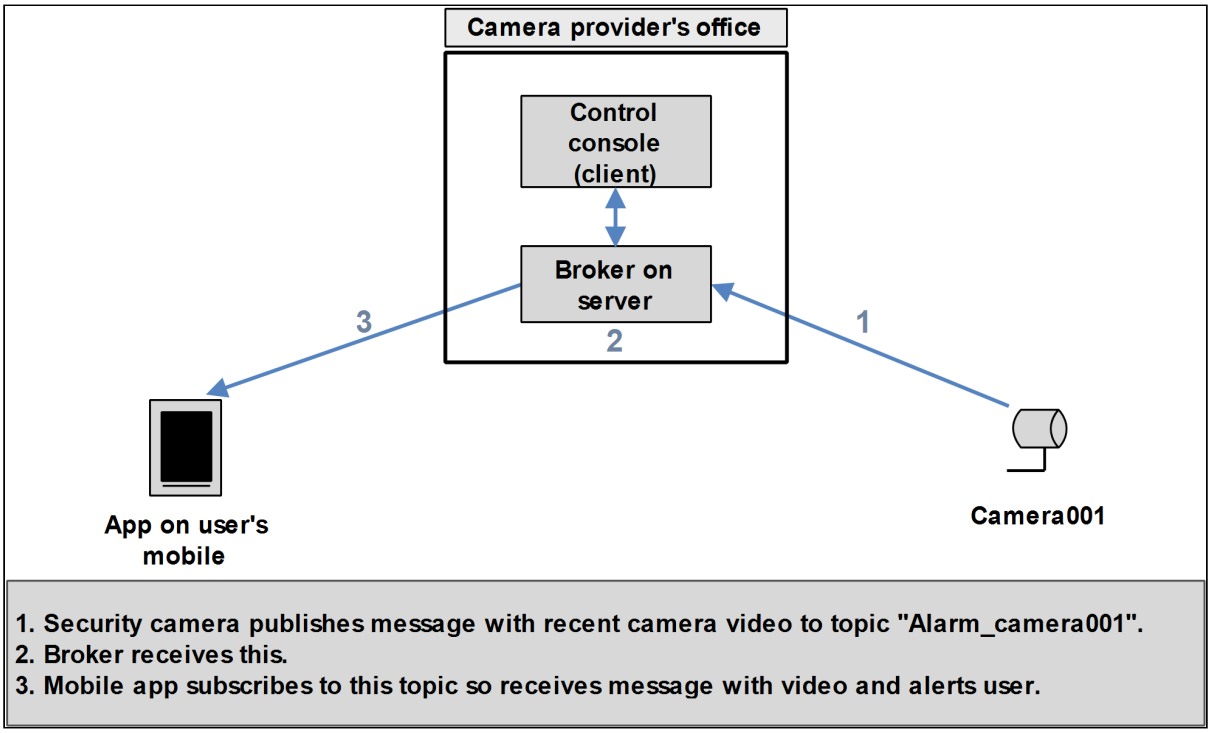
## Home owner at work checks camera at home

In this case the payload of the message published by the camera contains camera video. The QoS of this message should be 1 or 2 (messages are guaranteed to arrive exactly once).



## Camera reports suspicious activity

In this case the camera raises an alarm and streams live footage:



# 2.4 Example 4

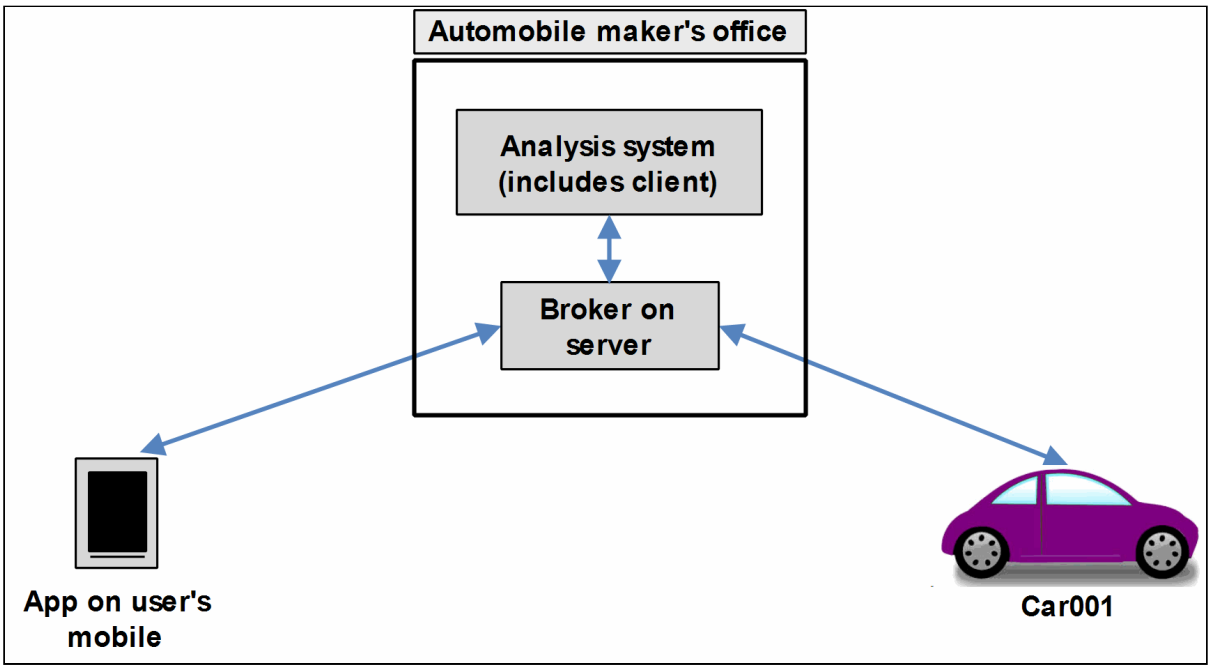
An MQTT-based platform enables automobile makers to offer real time diagnostics, safety, security, information and entertainment services, also apps and Wi-Fi hotspot services.

The diagrams below show use of MQTT for messaging between sensors in the car, a computer at the automobile manufacturer that analyzes the data, and an app. on the owner's mobile phone.

## Initial connection

This setup features:

- A control module in an automobile. This collects data from sensors throughout the car. It is an MQTT client named "Control/Car001".
- A broker application running on a server at the automobile maker.
- An analysis system running on a PC at the automobile maker. The company uses this to analyze the data from sensors in its cars and notify the owner when necessary. This system includes an MQTT client named "Analyzer001".
- An app provided by the automobile maker and installed on the car owner's mobile phone. The app appears as an MQTT client on this phone.




### Addresses

The addresses in this system are:

- Control module in car - Control/Car001.
- Analysis system at automobile company - Analyzer001.
- Mobile phone app - Owner/Car001.

## Topics

The clients must subscribe to topics as shown in this table:

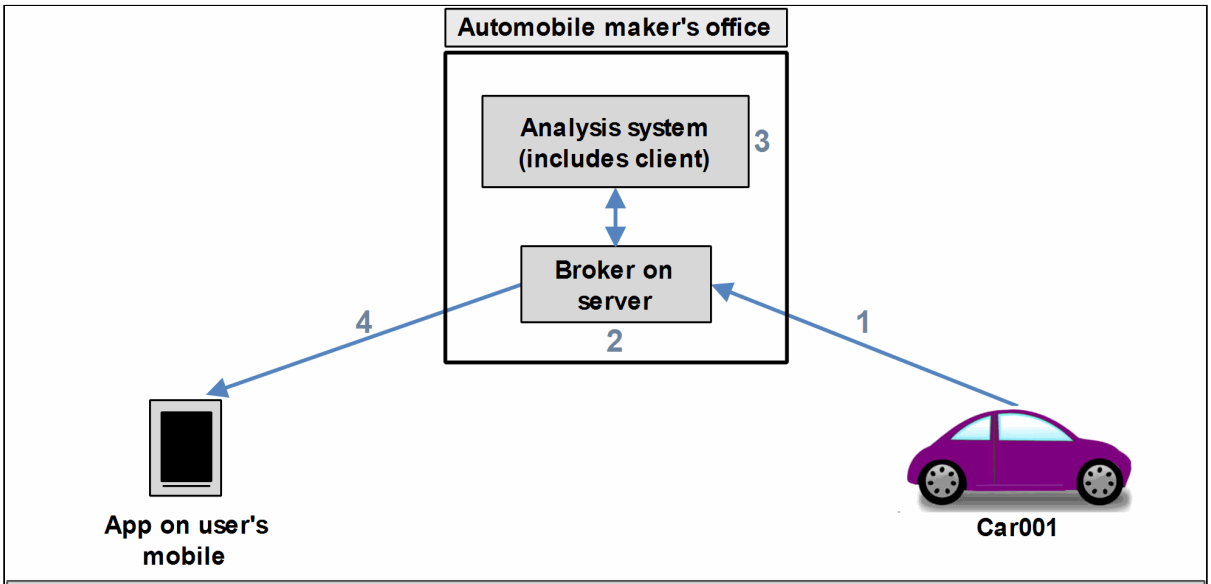
| Topic | Subscribers | Description of Use |
|---|---|---|
| Message_Car001 | Analyzer, Car, App on mobile | The mobile app publishes to this topic. |
| Warning_Car001 | Analyzer, Car | The car publishes to this topic to supply operational data from its sensors. |
| Owner_Car001 | Analyzer, Car, App on mobile | The car and analyzer publish to this topic to notify the car owner. |

The QoS of all messages should be 1 or 2 (messages are guaranteed to arrive at least or exactly once).

The following sections show how messages sent to each topic may be used.

## A sensor in the car reports that a service is needed

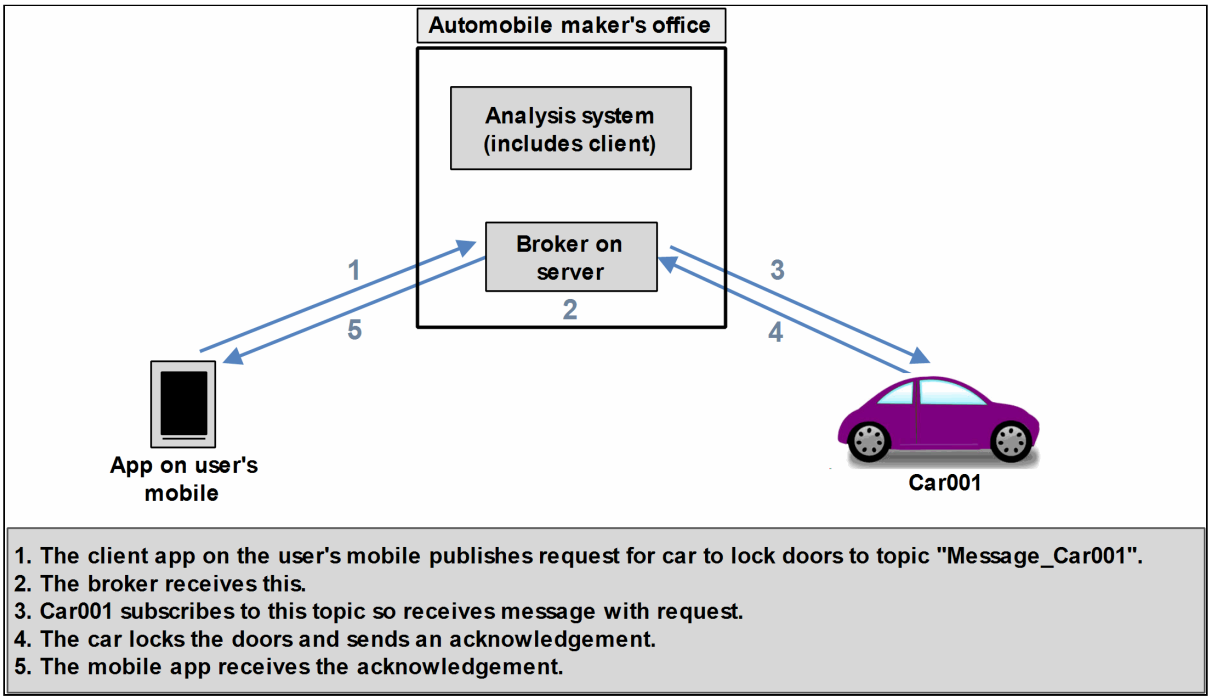
In this case one of the car's sensors reports an issue. The car's control module then starts the report operation.




**1. A sensor in the car reports an issue. The car's control module publishes a message with engine data to topic "Warning_Car001".**
**2. The broker receives this.**
**3. The car manufacturer's system analyzes the data.**
**4. It concludes that a service is required and publishes a message to topic "Owner_Car001".**
**5. The mobile app subscribes to this topic so receives the service message and alerts the owner.**

## Car owner locks car doors remotely

In this case the car owner suspects they have left the car unlocked. The car is in a car park a mile away. They use the phone app to lock the doors.




The payload of the message sent contains the instruction to lock the door. A range of other instructions could be sent in the same way.

The manufacturer's analysis system does not need to be involved in this operation.

# 3 Sources of MQTT Brokers

This section briefly describes three possible sources of MQTT brokers, one Open Source and two commercial.

## 3.1 Eclipse Mosquitto

Eclipse Mosquitto™ is an open source message broker that implements MQTT. It can be used for testing and maybe for building real systems using MQTT.

For full details, see the mosquitto.org website. This provides "man pages" on the broker and how to configure it, on password files and use with TLS, on the two commands referenced below, and on **libmosquitto** client library programming. There is also information on the **libmosquitto** API and on the provided Python module.

Mosquitto uses command line commands, for example:

- mosquitto - sets up the broker.
- mosquitto_pub - a client command that publishes a single message on a topic then exits.
- mosquitto_sub - a  client command for subscribing to topics.

### mosquitto broker

**mosquitto** is a broker for the MQTT protocol version 3.1. For full details see mosquitto broker.

The command line is:

```
mosquitto [-c config file] [ -d | --daemon ] [-p port number] [-v]
```

The broker is configured using a configuration file. Other options allow you to run **mosquitto** in the background as a daemon, to change the port from the default 1883, and to use verbose logging.

### mosquitto_pub

**mosquitto_pub** is a client command that publishes a single message on a topic then exits.

General options include the topic, a keepalive time, the port to connect to, QoS, an LWT message, a retain message and the payload. File contents can be sent.

Security options include a username and password to be used for authentication by the broker, a list of TLS ciphers to support in the client, pre-shared-key, and a client identity for use with TLS.

**Examples**

This example shows the command publishing light switch status. Message is set to retained because there may be a long period of time between light switch events:

```
mosquitto_pub -r -t switches/kitchen_lights/status -m "on"
```

This example shows the command sending parsed electricity usage data from a Current Cost meter, reading from *stdin* with one line/reading as one message:

```
read_cc128.pl | mosquitto_pub -t sensors/cc128 -l
```

## mosquitto_sub

**mosquitto_sub** is a client command for subscribing to topics. Most of the options are similar to those listed above.

**Examples**

Subscribe to temperature information on localhost with QoS 1:

```
mosquitto_sub -t sensors/temperature -q 1
```

Subscribe to hard drive temperature updates on multiple machines/hard drives. This expects each machine to be publishing its hard drive temperature to the topic sensors/machines/HOSTNAME/temperature/HD_NAME.

```
mosquitto_sub -t sensors/machines/+/temperature/+
```

# 3.2 Amazon Web Services IoT

Amazon Web Services (AWS) IoT provides secure, bi-directional communication between the AWS cloud and Internet-connected "things" (Amazon's term for devices such as sensors, actuators, embedded devices and smart appliances). This enables you to collect telemetry data from multiple devices and store and analyze the data. You can also create applications that enable your users to control these devices from their phones or tablets.

To publish and subscribe, clients can either use the raw MQTT protocol or MQTT over WebSocket. Clients can use the HTTP REST interface to publish to topics.

## Summary

Internet-connected devices can use AWS IoT to connect to the AWS cloud and applications in the cloud can interact with Internet-connected things. IoT applications may collect and process telemetry from devices. or enable users to control a remote device.

Devices report their state by publishing messages to MQTT topics. MQTT topics have hierarchical names, identifying the thing whose state is being updated. After a message is published to an MQTT topic, the message is sent to the AWS IoT MQTT message broker that is responsible for ensuring that all messages published on an MQTT topic are sent to all client subscribers to the topic.

Communication between AWS IoT and devices is protected by using X.509 certificates. Either AWS IoT can generate these certificates or users can employ their own. Either way, the certificate must be registered and activated with AWS IoT then copied onto the device. When a device communicates with AWS IoT, it presents the certificate to confirm its identity. There is an entry in the server's "thing registry" for every AWS IoT device, holding information on the device and its certificate.

The AWS IoT server stores state information on each device: its last reported state and the desired state requested by an application. Applications can request a device's current state information and can control a device by requesting a change in its state. The server sends a state change request to the device. When it gets this, the device receives the message, changes its state, then reports the new state.

## 3.3 Google Cloud IoT Core

In this system all devices and gateways connect to Google Cloud over MQTT or HTTP. It allows all of an organization's devices to be managed as a single global system.

In Google Cloud IoT core, devices can use an "MQTT bridge" to communicate with Cloud IoT Core. For details, see the Google documentation: Using the MQTT bridge. The MQTT or HTTP bridge is a central component of Cloud IoT Core. When MQTT is used, the main features are as as follows:

- Device connection is maintained.
- Full-duplex TCP connection is used.
- JSON Web Tokens (JWT) are sent in the password field of the CONNECT message.
- Telemetry events are pushed to Cloud Publication/Subscription.
- Device connection status is reported.
- Device configurations are propagated via subscriptions.
- Most recent configuration (whether newer or not) is always received by devices on subscription.
- Device configurations are acknowledged (ACKed) when using QoS 1.