# MQTT, Das M2M

MQ Telemetry Transport ist ein schlankes Messaging Protokoll, welches auf den Bereich Mobile und Internet of Things zugeschnitten ist. MQTT erfreut sich zunehmender Beliebtheit auch für Home-Automation und bei Open Source Projekten.

Entwickelt wurde MQTT 1999 von Dr. Andy Stanford-Clark (IBM) und Arlen Nipper (Arcom) um eine Öl Pipeline in der Wüste zu überwachen. Inzwischen ist MQTT ein OASIS Standard und liegt aktuell in der Version 3.1.1 vor.

Neben Zuverlässigkeit war der sparsame Umgang mit Bandbreite und Ressourcen ein Ziel der Entwicklung. Das MQTT Protokoll ist einfach und leichtgewichtig. Für eine Umsetzung des Protokolls ist wenig Code notwendig, wie die Client Bibliothek für den Arduino Mikrokontroller zeigt.

MQTT eignet sich für die Kommunikation mit mobilen Geräten, wie z.B. Smartphones oder Internet of Things Geräten, bei denen Bandbreite und Energie knapp sind. Die Übertragung von Nachrichten erfolgt dank des kompakten Binärprotokolls annähernd in Echtzeit.

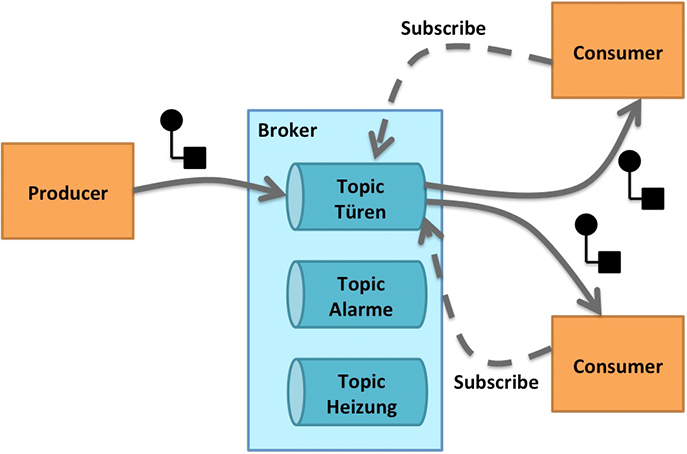
Über den Inhalt einer Nachricht macht MQTT keine Annahmen. Text-, Binär- oder Objektnachrichten sind möglich.

## Publish/Subscribe

Das von MQTT unterstützte Message Exchange Pattern ist das Publish/Subscribe Muster .

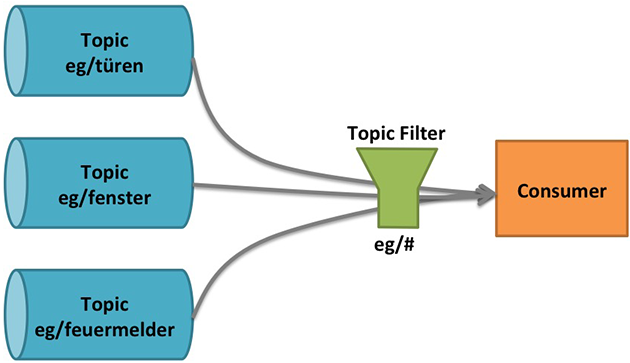
Bei Publish/Subscribe kommuniziert der Sender einer Nachricht nicht direkt mit dem Empfänger sondern wendet sich an einen Broker der die Zustellung der Nachricht übernimmt. Der Broker entkoppelt den Sender oder Producer vom Empfänger, der Consumer genannt wird. Ein Producer kann eine Nachricht selbst dann senden, wenn kein Consumer online ist. Tatsächlich weiß der Producer nicht, ob es keinen, einen oder mehrere Consumer gibt, die sich für seine Nachrichten interessieren. Der Broker ist für Consumer und Producer gleichermaßen der Server. Consumer und Producer sind Clients, die sich mit dem Broker verbinden.

Consumer sind oft nur an bestimmten Nachrichten interessiert, die ein Broker verteilt. Pro Thema kann ein Topic eingerichtet werden, an dem sich die Consumer anmelden (subscriben) können. In der Abbildung unten haben zwei Consumer eine Subscription auf das Topic Türen. Wird jetzt eine Nachricht z.B. über das Öffnen einer Türe an das Topic Türen gesendet, so übermittelt der Broker jeweils eine Kopie der Nachricht an die beiden Subscriber.



**Abbildung :** *Publish/Subscribe* Message Exchange Pattern MEP

Ein Topic Filter beschreibt für jede Subscription, an welchen Topics ein Consumer interessiert ist. In der Abbildung unten ist der Consumer an allen Topics interessiert, die Ereignisse vom Erdgeschoss (eg) liefern. Das Zeichen # ist eine Wildcard, die zu allen Topics im Erdgeschoss (eg) passt.



**Abbildung :** Topic Filter

Im Gegensatz zu Client/Server Protokollen wie HTTP arbeitet Publish/Subscribe ereignisorientiert. Ein Client muss nicht ständig beim Server anfragen, ob neue Daten vorliegen. Der Broker informiert die Consumer wenn neue Daten zu einem Topic vorliegen.

Trotz des "Q" im Namen gibt es bei MQTT keine 1 zu 1 Queues wie z.B. beim Java Messaging Service. Da sich jeder Consumer an einem Topic anmeldet werden bei MQTT die Consumer als Subscriber bezeichnet.

## Features

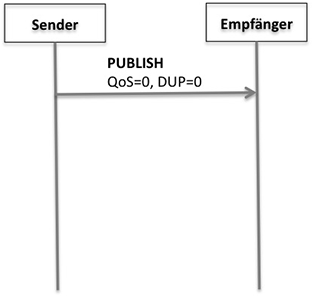
MQTT bietet wenige aber dafür nützliche und mächtige Features.

## Quality of Service

Quality of Service QoS steht für den Grad der Zuverlässigkeit, mit der Nachrichten zugestellt werden. MQTT kennt die drei QoS Stufen 0, 1 und 2. Jede Stufe steht für bestimmte Garantien, auf deren Einhaltung man sich verlassen kann. Jede Stufe stellt einen Kompromiss zwischen Zuverlässigkeit und Ressourcenverbrauch dar. Bei Homeautomation, Sensoren oder Internet of Things kann der Ressourcenverbrauch wichtiger sein als die Zuverlässigkeit. Der Sender kann bei jeder Nachricht individuell die QoS Stufe angeben.

### QoS 0: At most once

QoS der Stufe 0 garantiert lediglich, dass eine Nachricht höchstens einmal ankommt. Dass eine Nachricht überhaupt ankommt, garantiert QoS 0 nicht. Der Sender sendet eine PUBLISH Nachricht, die mit der QoS Stufe 0 gekennzeichnet ist und kümmert sich nicht weiter darum. Daher wird dieses Message Exchange Pattern auch Fire and Forget genannt. Fire and Forget ist schnell und Ressourcen schonend.



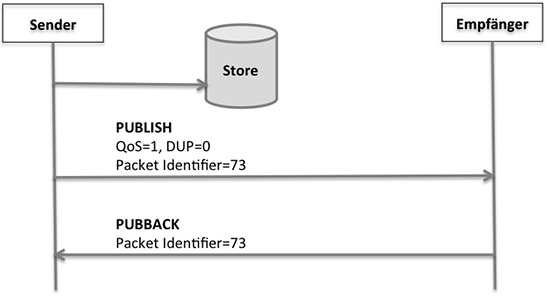
**Abbildung :** QoS 0: At most once: Fire and Forget mit MQTT

### QoS 1: At least once

QoS 1 garantiert, dass eine Nachricht mindestens einmal beim Empfänger eintrifft. Eine Nachricht kann aber auch mehrmals beim Empfänger eintreffen. Bei einer Bestellung könnten Duplikate einer Nachricht zu einer mehrfachen Auslieferung führen. Anders sieht es bei Messwerten aus, da schaden Duplikate meist nicht.

Vor dem Versand einer Nachricht kann der Sender eine Nachricht dauerhaft z.B. in einer Datei speichern. Die Nachricht steht dann auch nach einem Ausfall für eine erneute Übertragung zur Verfügung. Der Sender sendet mit dem PUBLISH Paket einen Packet Identifier zum Empfänger. Der Empfänger quittiert daraufhin den Empfang mit einem PUBACKnowlede Packet, das den selben Packet Identifier enthält. Vor dem Empfang des PUBACK Paketes darf der Sender eine Nachricht mit dem Packet Identifier und dem ursprünglichen Inhalt beliebig oft an den Empfänger senden. Nach Empfang des PUBACK Paketes wird eine erneute Nachricht mit dem selben Packet Identifier als neue Nachricht ausgelegt.

Ein zeitliches Überscheiden eines erneuten Sendeversuches ( Retry ) mit dem PUBACK Packet kann zu einem Duplikat der Nachricht führen. Bei QoS 0 sollte der Empfänger mit Duplikaten umgehen können.



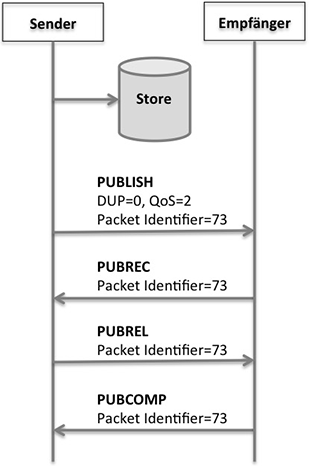
**Abbildung :** QoS 1: At least once

Mit dem Empfang von PUBACK beim Sender geht die Verantwortung für die Nachricht auf den Empfänger über.

### QoS 2: Exactly once

Die höchste Stufe QoS 2 garantiert, dass keine Nachrichten verloren gehen und dass keine Duplikate entstehen. Eine Nachricht kommt exakt einmal beim Empfänger an. Ein zweistufiger Bestätigungsablauf sorgt dafür, dass die Garantien eingehalten werden. Durch die zusätzlichen Nachrichten verzögert sich die Zustellung und wertvolle Bandbreite wird beansprucht.

Der Empfänger antwortet auf den Empfang einer Nachricht mit einem PUBRECeived Paket. Darauf sendet der Sender ein PUBRELease Paket auf welches der Empfänger mit einem PUBCOMPlete antwortet. Den genauen Ablauf zeigt das Diagramm.



**Abbildung :** QoS 2: Exactly once

Die Tabelle unten zeigt alle QoS Stufen in der Übersicht.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **QoS** | **Name** | **Beschreibung** | **Nachrichten- Verlust ist möglich** | **Duplikate sind möglich** |
| 0 | At most once | Nachricht wird genau einmal verschickt. Es gibt keine Antwort und keine Bestätigung vom Server. Ein Retry wird nicht durchgeführt. Schnellste Art des Nachrichtenversandes | ja | nein |
| 1 | At least once | Nachricht kann mehrfach verschickt werden. Server antwortet mit einer Bestätigung. | nein | ja |
| 2 | Exactly once | Erhöhter Overhead | nein | nein |

## Last Will Testament LWT

Nachdem die Verbindung zum Broker unterbrochen wurde, kann der Broker stellvertretend für einen Publisher eine Nachricht als dessen letzten Willen versenden. Ein Publisher kann so bestimmen, welchen Wert oder welches Kommando seine Subscriber, nach dessen Verbindungstrennung bekommen. Nützlich ist das Testament, um Subscriber zu informieren, dass die Verbindung getrennt wurde.

Im folgenden Beispiel legt der Client fest, dass die Nachricht „Ich bin gestorben“ an das Topic friedhof gesendet wird, wenn der Broker die Verbindung zum Client verliert.

var client = mqtt.connect('mqtt://localhost', {

clientId : "me",

will : { topic : "friedhof", payload : "Ich bin gestorben!" }

});

**Listing 1:** Last Will Testament mit Javascript und MQTT.js

## Message Persistence

Wird die Verbindung zu einem Client unterbrochen, so kann der Server neue Nachrichten für diesen Client für eine spätere Zustellung zwischenspeichern.

## Retained Messages

Verbindet sich ein Consumer zum ersten Mal mit einem Topic, so bekommt er normalerweise erst eine Nachricht, wenn ein Producer danach eine Nachricht an das Topic sendet. Eine Anzeige für die Temperatur würde nach der Verbindung mit einem Topic also so lange keinen Wert anzeigen, bis ein Temperaturfühler einen neuen Wert an das Topic sendet. Mit Retained Messages bekommt ein Consumer den letzten Wert, der vor der Subscription an das Topic gesendet wurde, sofort zugestellt. Die Temperaturanzeige könnte also anstatt keinem Wert zumindest einen älteren Wert anzeigen. Dieses Konzept wird auch Last Value Queue genannt.

## Persistent Sessions

Zwischen Client und Server kann eine Sitzung aufgebaut werden, die länger lebt als eine physikalische Netzwerkverbindung. Ein Consumer kann angehalten und neu gestartet werden, anschließend kann der Consumer eine bestehende Session wieder aufnehmen. Der Server kann daraufhin Nachrichten, die der Consumer in der Zwischenzeit verpasst hat, an ihn ausliefern. Eine Session wird über einen Client Identifier eindeutig identifiziert.

## Heartbeats & Keep Alive

Clients können dem Server über die Keep Alive Zeit mitteilen, wie lange eine inaktive Verbindung gehalten werden soll. Wird die Zeit überschritten, kann der Server die Verbindung trennen. Sendet der Client ein PINGREQ Paket, beginnt die Zeitmessung für den Keep Alive von vorne. PINGREQ dient auch dazu zu prüfen, ob der Server und die Netzverbindung noch aktiv sind.

## Web Sockets Transport

Web Browser können über MQTT over Web Sockets direkt mit einem Broker eine Verbindung aufbauen. Über diese Verbindung können Push Nachrichten an den Browser geschickt werden.

## Performance

MQTT wurde für Netzwerke mit großer Verzögerungszeit und kleiner Bandbreite entwickelt. Besonders kleinste Nachrichten können in kürzester Zeit und großer Zahl ausgetauscht werden. Die notwendigen Header Felder wie QoS oder Flags werden kompakt als Binärwerte gespeichert. Weitere selbst definierte Header werden nicht unterstützt. Damit eignet sich MQTT besonders für kleinste Nachrichten wie z.B. einzelne Messwerte eines Sensors. Eine Komprimierung auf Protokollebene gibt es nicht. Bei größeren Nachrichten ab wenigen Kilobytes kann ein Transport über HTTP mit Komprimierung effizienter sein.

# MQTT für Embedded Devices

Das einfache Protokoll, der geringe Footprint und der QoS Handshake machen den Transport von MQTT unabhängig von TCP. Geräte ohne TCP/IP Stack wie z.B. der Arduino können MQTT über einen Bus oder über die serielle Schnittstelle übertragen.

# Skalierbarkeit

MQTT wurde dafür entwickelt, dass sich tausende von Sensoren über das Internet gleichzeitig mit einem Server verbinden können. Für den Transport von vielen kleinen Nachrichten zwischen vielen kleinen Geräten ist MQTT ideal.

## Sicherheit

MQTT selbst unterstützt nur eine Absicherung über Benutzername und Passwort. Die Kommunikation kann mit SSL bzw. TLS auf Transportebene verschlüsselt werden. Broker können zusätzlich Client Zertifikate für die Authentifizierung verwenden oder den Zugriff über Access Control Lists z.B. durch die Überprüfung der IP einschränken.

## Einsatz von MQTT

Facebook verwendet MQTT als Protokoll für seinen Messanger.

## MQTT Broker

Ein Broker nimmt Nachrichten von Producern entgegen und verteilt diese an die Consumer. MQTT Broker können in zwei Gruppen aufgeteilt werden: Broker, die speziell für MQTT entwickelt wurden und Broker, die MQTT über einen Konnektor unterstützen.

Native Broker sind wesentlich kleiner als Broker, bei denen MQTT nur ein Protokoll von vielen ist. Durch den geringen Footprint eignen sich einige diese Broker wie z.B. [mosquitto](http://mosquitto.org/) für den Betrieb auf einem Mikrokontroller oder dem Raspberry Pi.

Wer bereits ActiveMQ oder RabbitMQ einsetzt, kann MQTT über einen Konnektor oder Adapter nutzen. Sensordaten, die über MQTT verteilt werden, können dann von Enterprise Anwendungen über JMS oder AMQP empfangen werden.

Wer selbst keinen Broker aufsetzen möchte um mit MQTT zu experimentieren, kann einen der öffentlichen Broker zu Testzwecken verwenden.

## MQTT Client Bibliotheken

Client Bibliotheken gibt es für die unterschiedlichsten Plattformen und Programmiersprachen. Unterstützt werden u.a. die Plattformen Java, .Net, Arduino und Raspberry Pi. Zu den unterstützten Sprachen gehören u.a.:

* C/C++
* C#
* Go
* Java
* Javascript
* Lua
* PHP
* Objective-C
* Ruby