# MQTT For Sensor Networks (MQTT-SN)

# Protocol Specification

## Version 1.2

Andy Stanford-Clark and Hong Linh Truong
(andysc@uk.ibm.com, hlt@zurich.ibm.com)
November 14, 2013

# Contents

 

# 1 Change and Revision History

## 1.1 Version 1.0, November 29, 2007

- Initial version

## 1.2 Version 1.1, June 5, 2008

- New feature: support of sleeping devices

## 1.3 Version 1.2, May 20, 2011

- New feature: support of message lengths greater than 255 bytes
- Format for the forwarder encapsulation changed to the format proposed by Nicholas O'Leary (nick_oleary@uk.ibm.com)
- Return code value "0x03 Rejected, not supported" added
- Field "ReturnCode" added to the messages WILLTOPICRESP and WILLMSGRESP

# 2 Introduction

There is a recent increase of interest in Wireless Sensor Networks (WSNs), both from commercial and technical point of view, due to their simplicity, low cost and easy deployment. Those networks can serve different purposes, from measurement and detection, to automation and process control. A typical WSN consists of a large number of battery-operated sensors and actuators (SAs), which are usually equipped with a limited amount of storage and processing capabilities. It is important that those devices communicate wirelessly, since the number of SA-nodes is typically very large, and the cost of deployment of a wired infrastructure is prohibitively expensive. Such a network is by nature very dynamic: the wireless links may temporarily break at any time, and nodes may fail and be replaced very often. In such situations the conventional approach of using addresses for communicating with the individual nodes may become a nightmare. Applications residing on the fixed network and requiring interactions with the wireless SA devices would need to manage and maintain means to communicate with a large number of nodes. In most cases they do not need to know the address or identity of the devices which deliver the information, but are more interested in the content of the data. For example, an asset tracking application is more interested in the current location of a certain asset than in the network address of the GPS receivers that deliver that information. In addition, several applications may have interest in the same sensor data but for different purposes. In this case the SA nodes would need to manage and maintain communication means with multiple applications in parallel. This might exceed the limited capabilities of the simple and low-cost SA devices.

Another problem is the difference in the addressing schemes between the networks involved. For example, how does an application residing on a TCP/IP-based network address a SA device running on a ZigBee®[1]-based wireless network?

The problem described above may be overcome by using a data-centric communication approach, in which information is delivered to the receivers not based on their network addresses but rather as a function of their contents and interests. One well-known example of data-centric communication is the "Publish/Subscribe" (pub/sub) messaging system which is already being widely used in enterprise networks, mainly due to their scalability and support of dynamic network topology. Extending the enterprise pub/sub system into the WSNs also enables a seamless integration of the WSNs into the enterprise network, thus making the field data collected by the SAs available to all applications as any other enterprise information and enabling the control of the SAs from any

[1] ZigBee is a trademark of ZigBee Alliance in the US, other countries or both. Other company, product or service names may be trademarks or service marks of others.

enterprise application. This can be for example achieved by using the MQTT protocol, which is an open and lightweight publish/subscribe protocol designed specifically for machine-to-machine and mobile applications. It is optimized for communications over networks where bandwidth is at a premium or where the network connection could be intermittent. However MQTT requires an underlying network, such as TCP/IP, that provides an ordered lossless connection capability and this is too complex for very simple, small footprint, and low-cost devices such as wireless SAs.

The purpose of this document is to specify MQTT-SN, a pub/sub protocol for wireless sensor networks. MQTT-SN can be considered as a version of MQTT which is adapted to the peculiarities of a wireless communication environment. Wireless radio links have in general a higher failure rates than wired ones due to their susceptibility to fading and interference disturbances. They have also a lower transmission rate. For example, WSNs based on the IEEE 802.15.4 standard provide a maximum bandwidth of 250 kbit/s in the 2.4 GHz band. Moreover, to be resistant against transmission errors, their packets have a very short length. In the case of IEEE 802.15.4, the packet length at the physical layer is limited to 128 bytes. Half of these 128 bytes could be taken away by the overhead information required by supporting functions such as MAC layer, networking, security, etc.

MQTT-SN is also optimized for implementation on low-cost, battery-operated devices with limited processing and storage resources.

MQTT-SN was originally developed for running on top of the ZigBee®[1] APS layer. ZigBee®[1] is an open industrial consortium with the aim of defining an open and global communication standard for WSNs. To be global ZigBee®[1] has selected the IEEE standard 802.15.4 as the protocol for the PHY and MAC layers, and adds on top on this standard the required network, security and application layers, thus providing interoperability between products of different vendors.

MQTT-SN is designed in such a way that it is agnostic of the underlying networking services. Any network which provides a bi-directional data transfer service between any node and a particular one (a gateway) should be able to support MQTT-SN. For example a simple datagram service which allows a source endpoint to send a data message to a specific destination endpoint should be sufficient. A broadcast data transfer service is only required if the gateway discovery procedure is employed. To reduce the broadcast traffic created by the discovery procedure, it is desirable that MQTT-SN could indicate the required broadcast radius to the underlying layer.

# 3 MQTT-SN vs MQTT

MQTT-SN is designed to be as close as possible to MQTT, but is adapted to the peculiarities of a wireless communication environment such as low bandwidth, high link failures, short message length, etc. It is also optimized for the implementation on low-cost, battery-operated devices with limited processing and storage resources.

Compared to MQTT, MQTT-SN is characterized by the following differences:

1. The CONNECT message is split into three messages. The two additional ones are optional and used to transfer the Will topic and the Will message to the server.

2. To cope with the short message length and the limited transmission bandwidth in wireless networks, the topic name in the PUBLISH messages is replaced by a short, two-byte long “topic id”. A registration procedure is defined to allow clients to register their topic names with the server/gateway and obtain the corresponding topic ids. It is also used in the opposite direction to inform the client about the topic name and the corresponding topic id that will be included in a following PUBLISH message.

3. “Pre-defined” topic ids and “short” topic names are introduced, for which no registration is required. Pre-defined topic ids are also a two-byte long replacement of the topic name, their mapping to the topic names is however known in advance by both the client’s application and the gateway/server. Therefore both sides can start using pre-defined topic ids; there is no need for a registration as in the case of “normal” topic ids mentioned above.

 

Short topic names are topic names that have a fixed length of two octets. They are short enough for being carried together with the data within PUBLISH messages. As for pre-defined topic ids, there is also no need for a registration for short topic names.

4. A discovery procedure helps clients without a pre-configured server/gateway's address to discover the actual network address of an operating server/gateway. Multiple gateways may be present at the same time within a single wireless network and can co-operate in a load-sharing or stand-by mode.

5. The semantic of a "clean session" is extended to the Will feature, i.e. not only client's subscriptions are persistent, but also Will topic and Will message. A client can also modify its Will topic and Will message during a session.

6. A new offline keep-alive procedure is defined for the support of *sleeping* clients. With this procedure, battery-operated devices can go to a sleeping state during which all messages destined to them are buffered at the server/gateway and delivered later to them when they wake up.

# 4 MQTT-SN Architecture

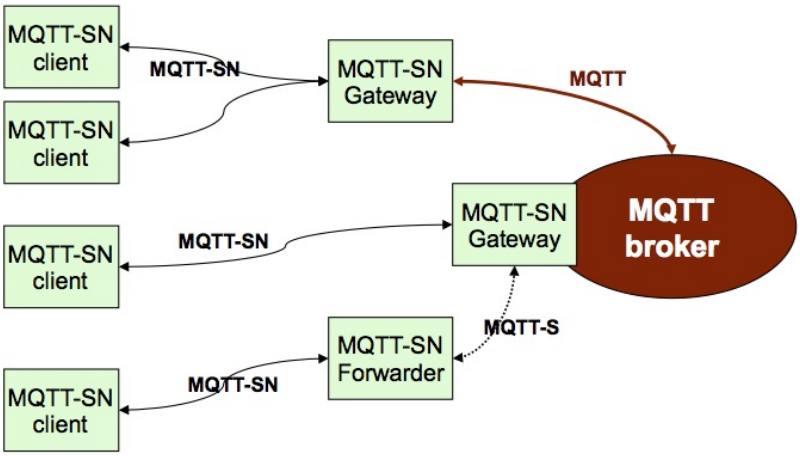


Figure 1: MQTT-SN Architecture

The architecture of MQTT-SN is shown in Fig. 1. There are three kinds of MQTT-SN components, MQTT-SN *clients*, MQTT-SN *gateways (GW)*, and MQTT-SN *forwarders*. MQTT-SN clients connect themselves to a MQTT server via a MQTT-SN GW using the MQTT-SN protocol. A MQTT-SN GW may or may not be integrated with a MQTT server. In case of a stand-alone GW the MQTT protocol is used between the MQTT server and the MQTT-SN GW. Its main function is the translation between MQTT and MQTT-SN.

MQTT-SN clients can also access a GW via a forwarder in case the GW is not directly attached to their network. The forwarder simply encapsulates the MQTT-SN frames it receives on the wireless side and forwards them **unchanged** to the GW; in the opposite direction, it decapsulates the frames it receives from the gateway and sends them to the clients, **unchanged** too.

Depending on how a GW performs the protocol translation between MQTT and MQTT-SN, we can differentiate between two types of GWs, namely *transparent* and *aggregating* GWs, see Fig. 2. They are explained in the following sections.

 

### 4.1 Transparent Gateway

For each connected MQTT-SN client a transparent GW will setup and maintain a MQTT connection to the MQTT server. This MQTT connection is reserved exclusively for the end-to-end and almost transparent message exchange between the client and the server. There will be as many MQTT connections between the GW and the server as MQTT-SN clients connected to the GW. The transparent GW will perform a "syntax" translation between the two protocols. Since all message exchanges are end-to-end between the MQTT-SN client and the MQTT server, all functions and features that are implemented by the server can be offered to the client.

Although the implementation of the transparent GW is simpler when compared to the one of an aggregating GW, it requires the MQTT server to support a separate connection for each active client. Some MQTT server implementations might impose a limitation on the number of concurrent connections that they support.

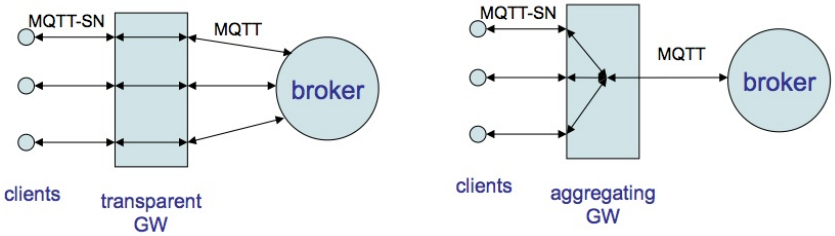


Figure 2: Transparent and Aggregating Gateways

### 4.2 Aggregating Gateway

Instead of having a MQTT connection for each connected client, an aggregating GW will have only one MQTT connection to the server. All message exchanges between a MQTT-SN client and an aggregating GW end at the GW. The GW then decides which information will be given further to the server. Although its implementation is more complex than the one of a transparent GW, an aggregating GW may be helpful in case of WSNs with very large number of SAs because it reduces the number of MQTT connections that the server has to support concurrently.

## 5 Message Formats

### 5.1 General Message Format

| Message Header (2 or 4 octets) | Message Variable Part (n octets) |
|---|---|

Table 1: General Message Format

The general format of a MQTT-SN message is shown in Table 1. A MQTT-SN message consists of two parts: a 2- or 4-octet long header and an optional variable part. While the header is always present and contains the same fields, the presence and content of the variable part depend on the type of the considered message.

### 5.2 Message Header

The format of the message header is illustrated in Table 2.

 

| Length (1 or 3 octets) | MsgType (1 octet) |
|---|---|

Table 2: Message Header

### 5.2.1 Length

The *Length* field is either 1- or 3-octet long and specifies the total number of octets contained in the message (including the *Length* field itself).

If the first octet of the *Length* field is coded "0x01" then the *Length* field is 3-octet long; in this case, the two following octets specify the total number of octets of the message (most-significant octet first). Otherwise, the *Length* field is only 1-octet long and specifies itself the total number of octets contained in the message.

The 3-octet format allows the encoding of message lengths up to 65535 octets. Messages with lengths smaller than 256 octets may use the shorter 1-octet format.

Note that because MQTT-SN does not support message fragmentation and reassembly, the maximum message length that could be used in a network is governed by the maximum packet size that is supported by that network, and not by the maximum length that could be encoded by MQTT-SN.

### 5.2.2 MsgType

The *MsgType* field is 1-octet long and specifies the message type. It shall be set to one of the values shown in Table 3 .

| **MsgType Field Value** | **MsgType** | **MsgType Field Value** | **MsgType** |
|---|---|---|---|
| 0x00 | ADVERTISE | 0x01 | SEARCHGW |
| 0x02 | GWINFO | 0x03 | reserved |
| 0x04 | CONNECT | 0x05 | CONNACK |
| 0x06 | WILLTOPICREQ | 0x07 | WILLTOPIC |
| 0x08 | WILLMSGREQ | 0x09 | WILLMSG |
| 0x0A | REGISTER | 0x0B | REGACK |
| 0x0C | PUBLISH | 0x0D | PUBACK |
| 0x0E | PUBCOMP | 0x0F | PUBREC |
| 0x10 | PUBREL | 0x11 | reserved |
| 0x12 | SUBSCRIBE | 0x13 | SUBACK |
| 0x14 | UNSUBSCRIBE | 0x15 | UNSUBACK |
| 0x16 | PINGREQ | 0x17 | PINGRESP |
| 0x18 | DISCONNECT | 0x19 | reserved |
| 0x1A | WILLTOPICUPD | 0x1B | WILLTOPICRESP |
| 0x1C | WILLMSGUPD | 0x1D | WILLMSGRESP |
| 0x1E-0xFD | reserved | 0xFE | Encapsulated message |
| 0xFF | reserved | | |

Table 3: Values of the MsgType field

## 5.3 Message Variable Part

The content of the message variable part depends on the type of the message. The following fields are defined for the message variable part.

### 5.3.1 ClientId

As with MQTT, the *ClientId* field has a variable length and contains a 1-23 character long string that uniquely identifies the client to the server.

### 5.3.2 Data

The *Data* field corresponds to payload of an MQTT PUBLISH message. It has a variable length and contains the application data that is being published

### 5.3.3 Duration

The *Duration* field is 2-octet long and specifies the duration of a time period in seconds. The maximum value that can be encoded is approximately 18 hours.

### 5.3.4 Flags

| DUP (bit 7) | QoS (6,5) | Retain (4) | Will (3) | CleanSession (2) | TopicIdType (1,0) |
|---|---|---|---|---|---|

Table 4: Flags field

The *Flags* field is 1-octet and contains the following flags (see Table 4):

- *DUP:* same meaning as with MQTT, i.e. set to "0" if message is sent for the first time; set to "1" if retransmitted (only relevant within PUBLISH messages);
- *QoS:* meaning as with MQTT for QoS level 0, 1, and 2; set to "0b00" for QoS level 0, "0b01" for QoS level 1, "0b10" for QoS level 2, and "0b11" for new QoS level -1 (only relevant within PUBLISH messages sent by a client);
- *Retain:* same meaning as with MQTT (only relevant within PUBLISH messages);
- *Will:* if set, indicates that client is asking for Will topic and Will message prompting (only relevant within CONNECT message);
- *CleanSession:* same meaning as with MQTT, however extended for Will topic and Will message (only relevant within CONNECT message);
- *TopicIdType:* indicates whether the field *TopicId* or *TopicName* included in this message contains a normal topic id (set to "0b00"), a pre-defined topic id (set to "0b01"), or a short topic name (set to "0b10"). The value "0b11" is reserved. Refer to sections 3 and 6.7 for the definition of the various types of topic ids.

### 5.3.5 GwAdd

The *GwAdd* field has a variable length and contains the address of a GW. Its depends on the network over which MQTT-SN operates and is indicated in the first octet of this field. For example, in a ZigBee network the network address is 2-octet long.

### 5.3.6 GwId

The *GwId* field is 1-octet long and uniquely identifies a gateway.

#### 5.3.7 MsgId

The *MsgId* field is 2-octet long and corresponds to the MQTT 'Message ID' parameter. It allows the sender to match a message with its corresponding acknowledgment.

#### 5.3.8 ProtocolId

The *ProtocolId* is 1-octet long. It is only present in a CONNECT message and corresponds to the MQTT 'protocol name' and 'protocol version'.

It is coded 0x01. All other values are reserved.

#### 5.3.9 Radius

The *Radius* field is 1-octet long and indicates the value of the broadcast radius. The value 0x00 means "broadcast to all nodes in the network".

#### 5.3.10 ReturnCode

The value and meaning of the 1-octet long *ReturnCode* field is shown in Table 5.

| ReturnCode Value | Meaning |
|---|---|
| 0x00 | Accepted |
| 0x01 | Rejected: congestion |
| 0x02 | Rejected: invalid topic ID |
| 0x03 | Rejected: not supported |
| 0x04 - 0xFF | reserved |

Table 5: Return Code Values

#### 5.3.11 TopicId

The *TopicId* field is 2-octet long and contains the value of the topic id. The values "0x0000" and "0xFFFF" are reserved and therefore should not be used.

#### 5.3.12 TopicName

The *TopicName* field has a variable length and contains an UTF8-encoded string that specifies the topic name.

#### 5.3.13 WillMsg

The *WillMsg* field has a variable length and contains the Will message.

#### 5.3.14 WillTopic

The *WillTopic* field has a variable length and contains the Will topic name.

### 5.4 Format of Individual Messages

This section specifies the format of the individual MQTT-SN messages. All messages are described with the 1-octet *Length* field. The message formats in case of the 3-octet *Length* field could be derived straightforwardly and are therefore not mentioned.

### 5.4.1 ADVERTISE

| Length (octet 0) | MsgType (1) | GwId (2) | Duration (3,4) |
|---|---|---|---|

Table 6: ADVERTISE Message

The ADVERTISE message is broadcasted periodically by a gateway to advertise its presence. The time interval until the next broadcast time is indicated in the *Duration* field of this message. Its format is illustrated in Table 6:

- Length and MsgType: see Section 5.2.
- GwId: the id of the gateway which sends this message.
- Duration: time interval until the next ADVERTISE is broadcasted by this gateway.

### 5.4.2 SEARCHGW

| Length (octet 0) | MsgType (1) | Radius (2) |
|---|---|---|

Table 7: SEARCHGW Message

The SEARCHGW message is broadcasted by a client when it searches for a GW. The broadcast radius of the SEARCHGW is limited and depends on the density of the clients deployment, e.g. only 1-hop broadcast in case of a very dense network in which every MQTT-SN client is reachable from each other within 1-hop transmission.
The format of a SEARCHGW message is illustrated in Table 7:

- Length and MsgType: see Section 5.2.
- Radius: the broadcast radius of this message.

The broadcast radius is also indicated to the underlying network layer when MQTT-SN gives this message for transmission.

### 5.4.3 GWINFO

| Length (octet 0) | MsgType (1) | GwId (2) | GwAdd* (3:n) |
|---|---|---|---|

(*) only present if message is sent by a client

Table 8: GWINFO Message

The GWINFO message is sent as response to a SEARCHGW message using the broadcast service of the underlying layer, with the radius as indicated in the SEARCHGW message. If sent by a GW, it contains only the id of the sending GW; otherwise, if sent by a client, it also includes the address of the GW, see Table 8:

- Length and MsgType: see Section 5.2.
- GwId: the id of a GW.

 

- GwAdd: address of the indicated GW; optional, only included if message is sent by a client.

Like the SEARCHGW message the broadcast radius for this message is also indicated to the underlying network layer when MQTT-SN gives this message for transmission.

### 5.4.4 CONNECT

| Length (octet 0) | MsgType (1) | Flags (2) | ProtocolId (3) | Duration (4,5) | ClientId (6:n) |
|---|---|---|---|---|---|

Table 9: CONNECT Message

The CONNECT message is sent by a client to setup a connection. Its format is shown in Table 9:

- Length and MsgType: see Section 5.2.
- Flags:
  - DUP, QoS, Retain, TopicIdType: not used.
  - Will: if set, indicates that client is requesting for Will topic and Will message prompting;
  - CleanSession: same meaning as with MQTT, however extended for Will topic and Will message (see Section 6.3).
- ProtocolId: corresponds to the "Protocol Name" and "Protocol Version" of the MQTT CONNECT message.
- Duration: same as with MQTT, contains the value of the Keep Alive timer.
- ClientId: same as with MQTT, contains the client id which is a 1-23 character long string which uniquely identifies the client to the server.

### 5.4.5 CONNACK

| Length (octet 0) | MsgType (1) | ReturnCode (2) |
|---|---|---|

Table 10: CONNACK Message

The CONNACK message is sent by the server in response to a connection request from a client. Its format is shown in Table 10:

- Length and MsgType: see Section 5.2.
- ReturnCode: encoded according to Table 5.

### 5.4.6 WILLTOPICREQ

| Length (octet 0) | MsgType (1) |
|---|---|

Table 11: WILLTOPICREQ and WILLMSGREQ Messages

The WILLTOPICREQ message is sent by the GW to request a client for sending the Will topic name. Its format is shown in Table 11: it has only a header and no variable part.

 

### 5.4.7 WILLTOPIC

| Length (octet 0) | MsgType (1) | Flags (2) | WillTopic (3:n) |
|---|---|---|---|

Table 12: WILLTOPIC Message

The WILLTOPIC message is sent by a client as response to the WILLTOPICREQ message for transferring its Will topic name to the GW. Its format is shown in Table 12:

- Length and MsgType: see Section 5.2.
- Flags:
  - DUP: not used.
  - QoS: same as MQTT, contains the Will QoS
  - Retain: same as MQTT, contains the Will Retain flag
  - Will: not used
  - CleanSession: not used
  - TopicIdType: not used.
- WillTopic: contains the Will topic name.

An empty WILLTOPIC message is a WILLTOPIC message without Flags and WillTopic field (i.e. it is exactly 2 octets long). It is used by a client to delete the Will topic and the Will message stored in the server, see Section 6.4.

### 5.4.8 WILLMSGREQ

The WILLMSGREQ message is sent by the GW to request a client for sending the Will message. Its format is shown in Table 11: it has only a header and no variable part.

### 5.4.9 WILLMSG

| Length (octet 0) | MsgType (1) | WillMsg (2:n) |
|---|---|---|

Table 13: WILLMSG Message

The WILLMSG message is sent by a client as response to a WILLMSGREQ for transferring its Will message to the GW. Its format is shown in Table 13:

- Length and MsgType: see Section 5.2.
- WillMsg: contains the Will message.

| Length (octet 0) | MsgType (1) | TopicId (2,3) | MsgId (4:5) | TopicName (6:n) |
|---|---|---|---|---|

Table 14: REGISTER Message

#### 5.4.10 REGISTER

The REGISTER message is sent by a client to a GW for requesting a topic id value for the included topic name. It is also sent by a GW to inform a client about the topic id value it has assigned to the included topic name. Its format is illustrated in Table 14:

- Length and MsgType: see Section 5.2.
- TopicId: if sent by a client, it is coded 0x0000 and is not relevant; if sent by a GW, it contains the topic id value assigned to the topic name included in the TopicName field;
- MsgId: should be coded such that it can be used to identify the corresponding REGACK message.
- TopicName: contains the topic name.

#### 5.4.11 REGACK

| Length (octet 0) | MsgType (1) | TopicId (2,3) | MsgId (4,5) | ReturnCode (6) |
|---|---|---|---|---|

Table 15: REGACK Message

The REGACK message is sent by a client or by a GW as an acknowledgment to the receipt and processing of a REGISTER message. Its format is illustrated in Table 15:

- Length and MsgType: see Section 5.2.
- TopicId: the value that shall be used as topic id in the PUBLISH messages;
- MsgId: same value as the one contained in the corresponding REGISTER message.
- ReturnCode: “accepted”, or rejection reason.

#### 5.4.12 PUBLISH

| Length (octet 0) | MsgType (1) | Flags (2) | TopicId (3-4) | MsgId (5-6) | Data (7:n) |
|---|---|---|---|---|---|

Table 16: PUBLISH Message

This message is used by both clients and gateways to publish data for a certain topic. Its format is illustrated in Table 16:

- Length and MsgType: see Section 5.2.
- Flags:
  - DUP: same as MQTT, indicates whether message is sent for the first time or not.

 

  - QoS: same as MQTT, contains the QoS level for this PUBLISH message.
  - Retain: same as MQTT, contains the Retain flag.
  - Will: not used
  - CleanSession: not used
  - TopicIdType: indicates the type of the topic id contained in the *TopicId* field.
- TopicId: contains the topic id value or the short topic name for which the data is published.
- MsgId: same meaning as the MQTT "Message ID"; only relevant in case of QoS levels 1 and 2, otherwise coded 0x0000.
- Data: the published data.

#### 5.4.13 PUBACK

| Length (octet 0) | MsgType (1) | TopicId (2,3) | MsgId (4,5) | ReturnCode (6) |
|---|---|---|---|---|

Table 17: PUBACK message

The PUBACK message is sent by a gateway or a client as an acknowledgment to the receipt and processing of a PUBLISH message in case of QoS levels 1 or 2. It can also be sent as response to a PUBLISH message in case of an error; the error reason is then indicated in the *ReturnCode* field. Its format is illustrated in Table 17:

- Length and MsgType: see Section 5.2.
- TopicId: same value the one contained in the corresponding PUBLISH message.
- MsgId: same value as the one contained in the corresponding PUBLISH message.
- ReturnCode: "accepted", or rejection reason.

#### 5.4.14 PUBREC, PUBREL, and PUBCOMP

| Length (octet 0) | MsgType (1) | MsgId (2-3) |
|---|---|---|

Table 18: PUBREC, PUBREL, and PUBCOMP Messages

As with MQTT, the PUBREC, PUBREL, and PUBCOMP messages are used in conjunction with a PUBLISH message with QoS level 2. Their format is illustrated in Table 18:

- Length and MsgType: see Section 5.2.
- MsgId: same value as the one contained in the corresponding PUBLISH message.

| Length (octet 0) | MsgType (1) | Flags (2) | MsgId (3-4) | TopicName or TopicId (5:n) or (5-6) |
|---|---|---|---|---|

Table 19: SUBSCRIBE and UNSUBSCRIBE Messages

#### 5.4.15 SUBSCRIBE

The SUBSCRIBE message is used by a client to subscribe to a certain topic name. Its format is illustrated in Table 19:

- Length and MsgType: see Section 5.2.
- Flags:
  - DUP: same as MQTT, indicates whether message is sent for first time or not.
  - QoS: same as MQTT, contains the requested QoS level for this topic.
  - Retain: not used
  - Will: not used
  - CleanSession: not used
  - TopicIdType: indicates the type of information included at the end of the message, namely "0b00" topic name, "0b01" pre-defined topic id, "0b10" short topic name, and "0b11" reserved.
- MsgId: should be coded such that it can be used to identify the corresponding SUBACK message.
- TopicName or TopicId: contains topic name, topic id, or short topic name as indicated in the *TopicIdType* field.

#### 5.4.16 SUBACK

| Length (octet 0) | MsgType (1) | Flags (2) | TopicId (3,4) | MsgId (5,6) | ReturnCode (7) |
|---|---|---|---|---|---|

Table 20: SUBACK Message

The SUBACK message is sent by a gateway to a client as an acknowledgment to the receipt and processing of a SUBSCRIBE message. Its format is illustrated in Table 20:

- Length and MsgType: see Section 5.2.
- Flags:
  - DUP: not used.
  - QoS: same as MQTT, contains the granted QoS level.
  - Retain: not used.
  - Will: not used
  - CleanSession: not used
  - TopicIdType: not used

- TopicId: in case of “accepted” the value that will be used as topic id by the gateway when sending PUBLISH messages to the client (not relevant in case of subscriptions to a short topic name or to a topic name which contains wildcard characters)
- MsgId: same value as the one contained in the corresponding SUBSCRIBE message.
- ReturnCode: “accepted”, or rejection reason.

#### 5.4.17 UNSUBSCRIBE

An UNSUBSCRIBE message is sent by the client to the GW to unsubscribe from named topics. Its format is illustrated in Table 19:

- Length and MsgType: see Section 5.2.
- Flags:
  - DUP: not used.
  - QoS: not used.
  - Retain: not used.
  - Will: not used
  - CleanSession: not used
  - TopicIdType: indicates the type of information included at the end of the message, namely “0b00” topic name, “0b01” pre-defined topic id, “0b10” short topic name, and “0b11” reserved.
- MsgId: should be coded such that it can be used to identify the corresponding SUBACK message.
- TopicName or TopicId: contains topic name, pre-defined topic id, or short topic name as indicated in the *TopicIdType* field.

#### 5.4.18 UNSUBACK

| Length (octet 0) | MsgType (1) | MsgId (2-3) |
|---|---|---|

Table 21: UNSUBACK Message

An UNSUBACK message is sent by a GW to acknowledge the receipt and processing of an UNSUBSCRIBE message. Its format is illustrated in Table 21:

- Length and MsgType: see Section 5.2.
- MsgId: same value as the one contained in the corresponding UNSUBSCRIBE message.

#### 5.4.19 PINGREQ

| Length (octet 0) | MsgType (1) | ClientId (optional) (2:n) |
|---|---|---|

Table 22: PINGREQ Message

As with MQTT, the PINGREQ message is an ”are you alive” message that is sent from or received by a connected client. Its format is illustrated in Table 22:

 

- Length and MsgType: see Section 5.2.
- ClientId: contains the client id; this field is optional and is included by a “sleeping” client when it goes to the “awake” state and is waiting for messages sent by the server/gateway, see Section 6.14 for further details.

#### 5.4.20 PINGRESP

| Length (octet 0) | MsgType (1) |
|---|---|

Table 23: PINGRESP Message

As with MQTT, a PINGRESP message is the response to a PINGREQ message and means ”yes I am alive”. Keep Alive messages flow in either direction, sent either by a connected client or the gateway. Its format is illustrated in Table 23: it has only a header and no variable part.

Moreover, a PINGRESP message is sent by a gateway to inform a sleeping client that it has no more buffered messages for that client, see Section 6.14 for further details.

#### 5.4.21 DISCONNECT

| Length (octet 0) | MsgType (1) | Duration (optional) (2-3) |
|---|---|---|

Table 24: DISCONNECT Message

The format of the DISCONNECT message is illustrated in Table 24:

- Length and MsgType: see Section 5.2.
- Duration: contains the value of the sleep timer; this field is optional and is included by a “sleeping” client that wants to go the “asleep” state, see Section 6.14 for further details.

As with MQTT, the DISCONNECT message is sent by a client to indicate that it wants to close the connection. The gateway will acknowledge the receipt of that message by returning a DISCONNECT to the client. A server or gateway may also sends a DISCONNECT to a client, e.g. in case a gateway, due to an error, cannot map a received message to a client. Upon receiving such a DISCONNECT message, a client should try to setup the connection again by sending a CONNECT message to the gateway or server. In all these cases the DISCONNECT message does not contain the *Duration* field.

A DISCONNECT message with a *Duration* field is sent by a client when it wants to go to the “asleep” state. The receipt of this message is also acknowledged by the gateway by means of a DISCONNECT message (without a duration field).

#### 5.4.22 WILLTOPICUPD

| Length (octet 0) | MsgType (1) | Flags (2) | WillTopic (3:n) |
|---|---|---|---|

Table 25: WILLTOPICUPD Message

 

The WILLTOPICUPD message is sent by a client to update its Will topic name stored in the GW/server. Its format is shown in Table 25:

- Length and MsgType: see Section 5.2.
- Flags:
  - DUP: not used.
  - QoS: same as MQTT, contains the Will QoS
  - Retain: same as MQTT, contains the Will Retain flag
  - Will: not used
  - CleanSession: not used
  - TopicIdType: not used.
- WillTopic: contains the Will topic name.

An empty WILLTOPICUPD message is a WILLTOPICUPD message without Flags and WillTopic field (i.e. it is exactly 2 octets long). It is used by a client to delete its Will topic and Will message stored in the GW/server.

#### 5.4.23 WILLMSGUPD

| Length (octet 0) | MsgType (1) | WillMsg (2:n) |
|---|---|---|

Table 26: WILLMSGUPD Message

The WILLMSGUPD message is sent by a client to update its Will message stored in the GW/server. Its format is shown in Table 26:

- Length and MsgType: see Section 5.2.
- WillMsg: contains the Will message.

#### 5.4.24 WILLTOPICRESP

| Length (octet 0) | MsgType (1) | ReturnCode (2) |
|---|---|---|

Table 27: WILLTOPICRESP and WILLMSGRESP Messages

The WILLTOPICRESP message is sent by a GW to acknowledge the receipt and processing of an WILLTOPICUPD message. Its format is illustrated in Table 27:

- Length and MsgType: see Section 5.2.
- ReturnCode: “accepted”, or rejection reason

 

#### 5.4.25 WILLMSGRESP

The WILLMSGRESP message is sent by a GW to acknowledge the receipt and processing of an WILLMSGUPD message. Its format is illustrated in Table 27:

- Length and MsgType: see Section 5.2.
- ReturnCode: “accepted”, or rejection reason

### 5.5 Forwarder Encapsulation

As mentioned in Section 4, MQTT-SN clients can also access a GW via a forwarder in case the GW is not directly attached to their WSNs. The forwarder simply encapsulates the MQTT-SN frames it receives on the wireless side and forwards them **unchanged** to the GW; in the opposite direction, it decapsulates the frames it receives from the gateway and sends them to the clients, **unchanged** too.

| Length (octet 0) | MsgType (1) | Ctrl (2) | Wireless Node Id (3:n) | MQTT-SN message (n+1,m) |
|---|---|---|---|---|

Table 28: Format of an encapsulated MQTT-SN frame

The format of an encapsulated MQTT-SN frame is shown in Table 28:

- Length: 1-octet long, specifies the number of octets up to the end of the “Wireless Node Id” field (incl. the Length octet itself)
- MsgType: coded “0xFE”, see Table 3
- Ctrl: The Ctrl octet contains control information exchanged between the GW and the forwarder. Its format is shown in Table 29:
  - Radius: broadcast radius (only relevant in direction GW to forwarder)
  - All remaing bits are reserved
- Wireless Node Id: identifies the wireless node which has sent or should receive the encapsulated MQTT-SN message. The mapping between this Id and the address of the wireless node is implemented by the forwarder, if needed.
- MQTT-SN message: the MQTT-SN message, encoded according to Table 1.

| reserved (bit 7:2) | radius (bit 1,0) |
|---|---|

Table 29: Format of the Ctrl octet

## 6 Functional Description

An important design point of MQTT-SN is to be as close as possible to MQTT. Therefore, all protocol semantics should remain, as far as possible, the same as those defined by MQTT. In the following we will focus on those points that either are new to or deviate from MQTT.

## 6.1 Gateway Advertisement and Discovery

This procedure is new and does not exist in MQTT.

A gateway may announce its presence by broadcasting periodically an ADVERTISE message to all devices that are currently parts of the network. A gateway should only advertise its presence if it is connected to a server (or is itself a server).

Multiple gateways may be active at the same time in the same network. In this case they will have different ids. It is up to the client to decide to which gateway it wants to connect. However at any point in time a client is allowed to be connected to only one gateway.

A client should maintain a list of active gateways together with their network addresses. This list is populated by means of the ADVERTISE and GWINFO messages received.

The time duration $T_{ADV}$ until the gateway sends the next ADVERTISE message is indicated in the *Duration* field of the ADVERTISE messages. A client may use this information to monitor the availability of a gateway. For example, if it does not receive ADVERTISE messages from a gateway for $N_{ADV}$ consecutive times, it may assume that the gateway is down and removes it from its list of active gateways. Similarly, gateways in stand-by mode will become active (i.e. start sending ADVERTISE messages) if they miss successively a couple of times advertisements from a certain gateway.

Since the ADVERTISE messages are broadcasted into the whole wireless network, the time interval $T_{ADV}$ between two ADVERTISE messages sent by a gateway should be large enough (e.g. greater than 15 min) to avoid bandwidth congestion in the network.

The large value of $T_{ADV}$ will lead to a long waiting time for new clients which are looking for a gateway. To shorten this waiting time a client may broadcast a SEARCHGW message. To prevent broadcast storms when multiple clients start searching for GW almost at the same time, the sending of the SEARCHGW message is delayed by a random time between 0 and $T_{SEARCHGW}$. A client will cancel its transmission of the SEARCHGW message if it receives during this delay time a SEARCHGW message sent by another client and identical to the one it wants to send, and behaves as if the SEARCHGW message was sent by itself.

The broadcast radius $R_b$ of the SEARCHGW message is limited, e.g. to a single hop in case of a dense deployment of MQTT-SN clients.

Upon receiving a SEARCHGW message a gateway replies with a GWINFO message containing its id. Similarly, a client answers with a GWINFO message if it has at least one active gateway in its list of active gateways. If the client has multiple GWs in its list, it selects one GW out of its list and includes that information into the GWINFO message.

Like the SEARCHGW message, the GWINFO message is broadcast with the same radius $R_b$, which is indicated in the SEARCHGW message. The radius $R_b$ is also given to the underlying layer when these two messages are passed down for transmission.

To give priority to the gateways a client will delay its sending of the GWINFO message for a random time $T_{GWINFO}$. If during this delay time the client receives a GWINFO message it will cancel the transmission of its GWINFO message.

In case of no response the SEARCHGW message may be retransmitted. In this case the time intervals between two consecutive SEARCHGW messages should be increased exponentially.

## 6.2 Client's Connection Setup

As with MQTT, a MQTT-SN client needs to setup a connection to a GW before it can exchange information with that GW. The procedure for setting up a connection with a GW is illustrated in Fig. 3, in which it is assumed that the client requests the gateway to prompt for the transfer of Will topic and Will message. This request is indicated by setting the Will flag of the CONNECT message. The client then sends these two pieces of information to the GW upon receiving the corresponding request messages WILLTOPICREQ and WILLMSGREQ. The procedure is terminated with the CONNACK message sent by the GW.

If Will flag is not set then the GW answers directly with a CONNACK message.

 

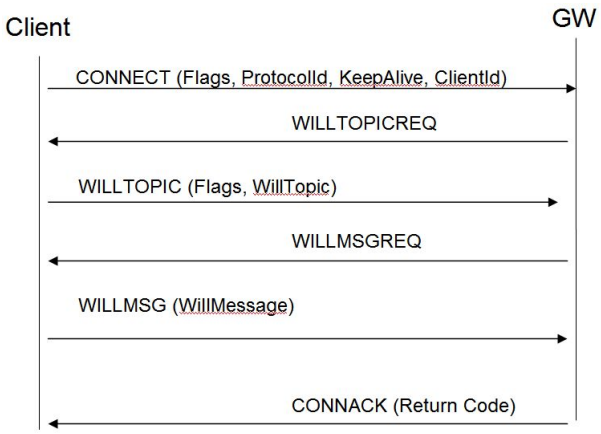


Figure 3: Connect procedure

In case the GW could not accept the connection request (e.g. because of congestion or it does not support a feature indicated in the CONNECT message), the GW returns a CONNACK message with the rejection reason.

## 6.3 Clean session

With MQTT, when a client disconnects, its subscriptions are not deleted. They are persistent and valid for new connections, until either they are explicitly un-subscribed by the client, or the client establishes a new connection with the "clean session" flag set.

In MQTT-SN the meaning of a "clean session" is extended to the Will feature, i.e. not only the subscriptions are persistent, but also the Will topic and the Will message. The two flags "CleanSession" and "Will" in the CONNECT have then the following meanings:

- **CleanSession=true, Will=true:** The GW will delete all subscriptions and Will data related to the client, and starts prompting for new Will topic and Will message.
- **CleanSession=true, Will=false:** The GW will delete all subscriptions and Will data related to the client, and returns CONNACK (no prompting for Will topic and Will message).
- **CleanSession=false, Will=true:** The GW keeps all stored client's data, but prompts for new Will topic and Will message. The newly received Will data will overwrite the stored Will data.
- **CleanSession=false, Will=false:** The GW keeps all stored client's data and returns CONNACK (no prompting for Will topic and Will message).

Note that if a client wants to delete only its Will data at connection setup, it could send a CONNECT message with "CleanSession=false" and "Will=true", and sends an empty WILLTOPIC message to the GW when prompted to do so. It could also send a CONNECT message with "CleanSession=false" and "Will=false", and use the procedure of Section 6.4 to delete or modify the Will data.

## 6.4 Procedure for updating the Will data

At any time during a connection a client could update its Will data stored in the gateway by sending a WILLTOPICUPD or a WILLMSGUPD message. The information contained in these two messages will overwrite the corresponding ones stored in the gateway. Both messages are acknowledged by the gateway. Both messages can be used independently from each other.

 

Note that an empty WILLTOPICUPD message will delete both the Will topic and Will message stored at the gateway.

## 6.5 Topic Name Registration Procedure

Because of the limited bandwidth and the small message payload in wireless sensor networks, data is not published together with its topic name as in MQTT. A registration procedure is introduced which allows both a client and a GW to inform its peer about the short topic id and its corresponding topic name before it can start sending PUBLISH messages using the short topic id.

To register a topic name a client sends a REGISTER message to the GW. If the registration could be accepted, the gateway assigns a *topicId* to the received topic name and returns it with a REGACK message to the client. If the registration could not be accepted, a REGACK is also returned to the client with the failure reason encoded in the *ReturnCode* field.

After having received the REGACK message with *ReturnCode*=*"accepted"*, the client shall use the assigned *topicId* to publish data of the corresponding topic name. If however the REGACK contains a rejection code, the client may try to register later again. If the return code was *"rejected: congestion"*, the client should wait for a time $T_{WAIT}$ before restarting the registration procedure.

At any point in time a client may have only one REGISTER message outstanding, i.e. it has to wait for a REGACK message before it can register another topic name.

A GW sends a REGISTER message to a client if it wants to inform that client about the topic name and the assigned topic id that it will use later on when sending PUBLISH messages of the corresponding topic name. This happens for example when the client re-connects without having set the "CleanSession" flag or the client has subscribed to topic names that contain wildcard characters such as # or +.

## 6.6 Client's Publish Procedure

After having registered successfully a topic name with the gateway, the client can start publishing data relating to the registered topic name by sending PUBLISH messages to the gateway. The PUBLISH messages contain the assigned topic id.

All three QoS levels and their corresponding message flows are supported as defined in MQTT. The only difference is the use of topic ids instead of topic names in the PUBLISH messages.

Regardless of the requested QoS level the client may receive in response to its PUBLISH a PUBACK message which contains either

- the *ReturnCode*= *"Rejection: invalid topic Id"*: in this case the client needs to register the topic name again before it can publish data related to that topic name; or
- the *ReturnCode*= *"Rejection: congestion"*: in this the client shall stop publishing toward the gateway for at least the time $T_{WAIT}$.

At any point in time a client may have only one QoS level 1 or 2 PUBLISH message outstanding, i.e. it has to wait for the termination of this PUBLISH message exchange before it could start a new level 1 or 2 transaction.

## 6.7 Pre-defined topic ids and short topic names

As described in Section 6.5, a topic id is a two-byte long replacement of the string-based topic name. A client needs to use the REGISTER procedure to inform the gateway about the topic name it wants to employ and gets from the gateway the corresponding topic id. It then will use this topic id in the PUBLISH messages it sends to the gateway. In the opposite direction, the PUBLISH messages also contain a 2-byte topic id (instead of the string-based topic name). The client is informed about the relation between topic id and topic name by means of either a former SUBSCRIBE procedure or a REGISTER procedure started by the gateway.

 

A “pre-defined” topic id is a topic id whose mapping to a topic name is known in advance by both the client’s application and the gateway. This is indicated in the *Flags* field of the message. When using pre-defined topic ids, both sides can start immediately with the sending of PUBLISH messages; there is no need for the REGISTER procedure as in the case of ”normal” topic ids. When receiving a PUBLISH message with a pre-defined topic id, of which the mapping to a topic name is unknown, the receiver should return a PUBACK with the *ReturnCode*= *“Rejection: invalid topic Id”*. Note that this error situation cannot be resolved by means of re-registering as in the case of normal topic id.

A client is still required to subscribe to a pre-defined topic id, if it wants to receive PUBLISH messages relating to that topic id. To avoid confusion between a pre-defined topic id and a two-byte short topic name, the SUBSCRIBE message contains a flag indicating whether it is subscribing for a short topic name or a pre-defined topic id.

A “short” topic name is a topic name that has a fixed length of two octets. It could be carried together with the data within a PUBLISH message, thus no REGISTER procedure is needed for a short topic name. Otherwise, all rules that apply to normal topic names also apply to short topic names. Note however that it does not make sense to do wildcarding in subscriptions to short topic names, because it is not possible to define a meaningful name hierarchy with only two characters.

## 6.8 PUBLISH with QoS Level -1

This feature is defined for very simple client implementations which do not support any other features except this one. There is no connection setup nor tear down, no registration nor subscription. The client just sends its PUBLISH messages to a GW (whose address is known a-priori by the client) and forgets them. It does not care whether the GW address is correct, whether the GW is alive, or whether the messages arrive at the GW.

Only the following parameter values are allowed for a PUBLISH message with QoS level -1:

- QoS flag: set to “0b11”;
- TopicIdType flag: either “0b01” for pre-defined topic id or “0b10” for short topic name;
- TopicId field: value of the pre-defined topic id or of the short topic name;
- Data field: the data to be published.

## 6.9 Client’s Topic Subscribe/Un-subscribe Procedure

To subscribe to a topic name, a client sends a SUBSCRIBE message to the gateway with the topic name included in that message. If the gateway is able accept the subscription, it assigns a topic id to the received topic name and returns it within a SUBACK message to the client. If the subscription can not be accepted, then a SUBACK message is also returned to the client with the rejection cause encoded in the *ReturnCode* field. If the rejection cause is *“rejected: congestion”*, the client should wait for the time $T_{WAIT}$ before resending the SUBSCRIBE message to the gateway.

If the client subscribes to a topic name which contains a wildcard character, the returning SUBACK message will contain the topic id value 0x0000. The GW will the use the registration procedure to inform the client about the to-be-used topic id value when it has the first PUBLISH message with a matching topic name to be sent to the client, see also Section 6.10.

Similar to the client’s PUBLISH procedure, topic ids may also be pre-defined for certain topic names. Short topic names may be used as well. In those two cases the client still needs to subscribe to those pre-defined topic ids or short topic names.

To unsubscribe, a client sends an UNSUBSCRIBE message to the gateway, which will then be answered by means of an UNSUBACK message.

As for the REGISTER procedure, a client may have only one SUBSCRIBE or one UNSUBCRIBE transaction open at a time.

## 6.10 Gateway's Publish Procedure

Similar to the client's PUBLISH procedure described in Section 6.6, the gateway sends PUBLISH messages with the topic id value that was returned in the SUBACK message to the client.

Preceding the PUBLISH message the GW may send a REGISTER message to inform the client about the topic name and its assigned topic id value. This will happen for example when the client re-connects without clean session or has subscribed to topic names with wildcard characters. Upon receiving a REGISTER message the client replies with a REGACK message. The GW will wait for the REGACK message before it sends the PUBLISH message to the client.

The client could reject the REGISTER message with a REGACK message indicating the rejection reason; this corresponds to an unsubscribe to the topic name indicated in the REGISTER message. Note that unsubcribe to a topic name with wildcard characters can only be done with the unsubscribe procedure described in Section 6.9 and not with the rejection of a REGISTER message, since a REGISTER message never contains a topic name with wildcard characters.

If the client receives a PUBLISH message with an unknown topic id value, it shall respond with a PUBACK message with the *ReturnCode="Rejected: invalid Topic ID"*. This will trigger the gateway to delete or correct the wrong topic id assignment.

Note that in case either the topic name or the data is too long to fit into a REGISTER or a PUBLISH message, the gateway silently aborts the publish procedure, i.e. no warning is sent to the affected subscribers.

## 6.11 Keep Alive and PING Procedure

As with MQTT, the value of the Keep Alive timer is indicated in the CONNECT message. The client should send a PINGREQ message within each Keep Alive time period, which the GW acknowledges with a PINGRESP message.

Similarly, a client shall answer with a PINGRESP message when it receives a PINGREQ message from the GW to which it is connected. Otherwise the received PINGREQ message is ignored.

Clients should use this procedure to supervise the liveliness of the gateway to which they are connected. If a client does not receive a PINGRESP from the gateway even after multiple retransmissions of the PINGREQ message, it should first try to connect to another gateway before trying to re-connect to this gateway (see also section 6.13). Note that because the clients' keep alive timers are not synchronized with each other, in case of a gateway failure there is practically no danger for a storm of CONNECT messages sent almost at the same time by all affected clients towards a new gateway.

## 6.12 Client's Disconnect Procedure

A client sends a DISCONNECT message to the GW to indicate that it is about to close its connection. After this point, the client is then required to establish a new connection with the GW before it can exchange information with that GW again. Similar to MQTT, sending the DISCONNECT message does not affect existing subscriptions and Will data if the CleanSession flag is set. They are persistent until they are either explicitly un-subscribed, or deleted, or modified by the client, or if the client establishes a new connection with the CleanSession flag set. The gateway acknowledges the receipt of the DISCONNECT message by returning a DISCONNECT to the client.

A client may also receive an unsolicited DISCONNECT sent by the gateway. This may happen for example when the gateway, due to an error, cannot identify the client to which a received message belongs. Upon receiving such a DISCONNECT message a client should retry to setup the connection again by sending a CONNECT message to the gateway.

## 6.13 Client's Retransmission Procedure

All messages that are "unicasted" to the GW (i.e. sent using the GW's unicast address and not broadcasted) and for which a GW's reply is expected are supervised by a retry timer $T_{retry}$ and a retry counter $N_{retry}$. The retry timer $T_{retry}$ is started by the client when the message is sent and stopped when the expected GW's reply is received. If $T_{retry}$ times out and the expected GW's reply is not received, the client retransmits the message. After $N_{retry}$ retransmissions, the client aborts the procedure and assumes that its MQTT-SN connection to the gateway is disconnected. It should then try to connect to another gateway, and only if it fails to re-connect again to the former gateway.

## 6.14 Support of sleeping clients

*Sleeping* clients are clients residing on (battery-operated) devices that want to save as much energy as possible. These devices need to enter a sleep mode whenever they are not active, and will wake up whenever they have data to send or to receive. The server/gateway needs to be aware of the sleeping state of these clients and will buffer messages destined to them for later delivery when they wake up.

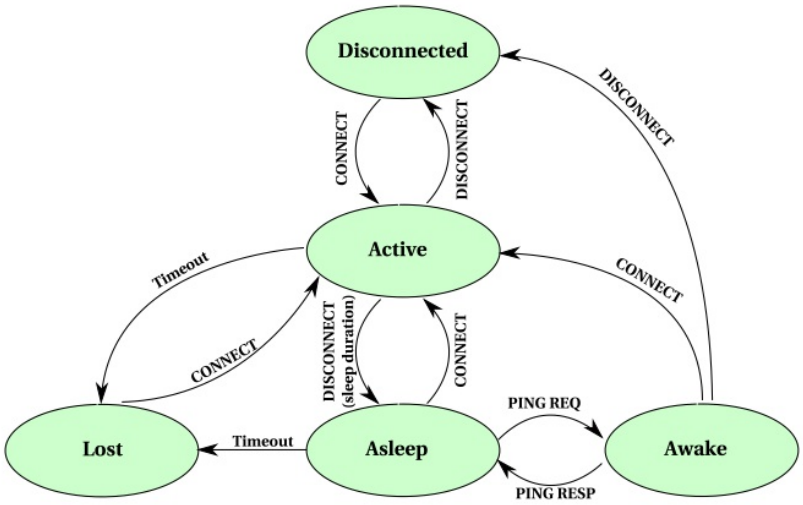


Figure 4: Client's state transition diagram

As illustrated in Fig 4, from the perspective of the server/gateway, a client may be in one of the following states: *active, asleep, awake, disconnected,* or *lost*. A client is in the *active* state when the server/gateway receives a CONNECT message from that client, as described in section 6.2. This state is supervised by the server/gateway with the "keep alive" timer as described in section 6.11. If the server/gateway does not receive any message from the client for a period longer than the keep alive duration (indicated in the CONNECT message), the gateway will consider that client as *lost* and activates for example the Will feature for that client.

A client goes to the *disconnected* state when the server/gateway receives a DISCONNECT without a duration field. This state is not time-supervised by the server/gateway.

If a client want to sleep, it sends a DISCONNECT message which contains a sleep duration. The server/gateway acknowledges that message with a DISCONNECT message and considers the client for being in *asleep* state, see also Fig. 5. The *asleep* state is supervised by the server/gateway with the indicated sleep duration. If the server/gateway does not receive any message from the client for a period longer than the sleep duration, the server/gateway will consider that client as *lost* and - as with the keep alive procedure - activates for example

 

the Will feature. During the *asleep* state, all messages that need to be sent to the client are buffered at the server/gateway.

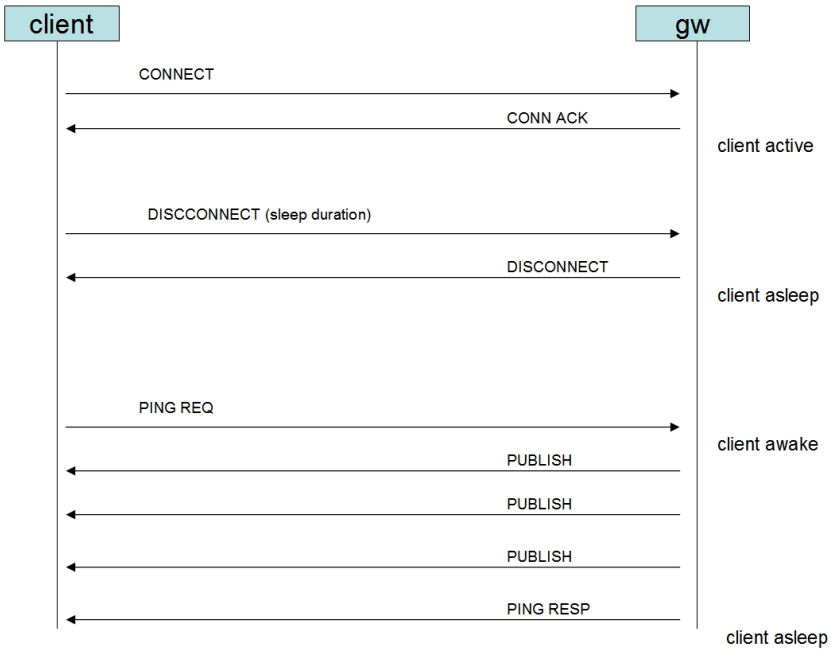


Figure 5: Sleep procedure

The sleep timer is stopped when the server/gateway receives a PINGREQ from the client. Like the CONNECT message, this PINGREQ message contains the *Client Id*. The identified client is then in the *awake* state. If the server/gateway has buffered messages for the client, it will sends these messages to the client. The transfer of messages to the client is closed by the server/gateway by means of a PINGRESP message, i.e. the server/gateway will consider the client as *asleep* and restart the sleep timer again after having sent the PINGRESP message.

If the server/gateway does not have any messages buffered for the client, it answers immediately with a PINGRESP message, returns the client back to the *asleep* state, and restarts the sleep timer for that client.

After having sent the PINGREQ to the server/gateway, the client uses the "retransmission procedure" of section 6.13 to supervise the arrival of messages sent by the server/gateway, i.e. it restarts timer $T_{retry}$ when it receives a message other than a PINGRESP, and stops it when it receives a PINGRESP. The PINGREQ message is retransmitted and timer $T_{retry}$ restarted when timer $T_{retry}$ times out. To avoid a flattening of its battery due to excessive retransmission of the PINGREQ message (e.g. if it looses the gateway), the client should limit the retransmission of the PINGREQ message (e.g. by a retry counter) and go back to sleep when the limit is reached and it still does not receive a PINGRESP message.

From the *asleep* or *awake* state a client can return either to the *active* state by sending a CONNECT message or to the *disconnected* state by sending a normal DISCONNECT message (i.e. without duration field). The client can also modify its sleep duration by sending a DISCONNECT message with a new value of the sleep duration.

Note that a sleeping client should go the *awake* state only if it just wants to check whether the server/gateway has any messages buffered for it and returns as soon as possible to the *asleep* state without sending any messages to the server/gateway. Otherwise, it should return to the *active* state by sending a CONNECT message to the server/gateway.

 

# 7 Implementation Notes

## 7.1 Support of QoS Level -1

Because PUBLISH messages with QoS level -1 could be sent at any time by clients (even with no connection setup) a transparent GW needs to maintain for those messages a dedicated MQTT connection with the server. An aggregating or hybrid GW may use any aggregating MQTT connection to forward those messages to the server.

## 7.2 "Best practice" values for timers and counters

Table 30 shows the "best practice" values for the timers and counters defined in this specification.

| Timer/Counter | Recommended value |
|---|---|
| $T_{ADV}$ | greater than 15 minutes |
| $N_{ADV}$ | 2 -3 |
| $T_{SEARCHGW}$ | 5 seconds |
| $T_{GWINFO}$ | 5 seconds |
| $T_{WAIT}$ | greater than 5 minutes |
| $T_{retry}$ | 10 - 15 seconds |
| $N_{retry}$ | 3 - 5 |

Table 30: "Best practice" values for timers and counters

The "tolerance" of the sleep and keep-alive timers at the server/gateway depends on the duration indicated by the clients. For example, the timer values should be 10% higher than the indicated values for durations larger than 1 minute, and 50% higher if less.

## 7.3 Mapping of Topic Ids to Topic Names

It is strongly recommended that in the gateway the mapping table between topic ids and topic names is implemented per client (and not by a single shared pool between all clients), to reduce the risk of an incorrect topic id from a client matching another client's valid topic, and thus causing a publication to the wrong topic, which could potentially have disastrous consequences.

## 7.4 ZigBee related issues

- In a ZigBee network a gateway need not be hosted by a coordinator node. It should however reside on an always-on router node to be able to receive client messages at any time.
- Due to short payload length of the ZigBee network/APS layer, the maximum length of a MQTT-SN message is restricted to 60 octets.

– END OF DOCUMENT –