JMQT Version 1.0 Specifications

Table of Contents

1. What is JMQT (JSON Message Queueing and Transfer) ?

JMQT is a publish-subscribe based IoT and messaging protocol, which works on top of TCP/IP. It is a text based protocol, where the packets are build in JSON (JavaScript Object Notation) format and converted into text.

JMQT has been primarily developed for IoT and messaging applications, where multiple clients are connected to a server in order to exchange messages between them. JMQT also supports P2P (Point to Point) messaging in addition to publish-subscribe operations. JMQT protocol is open & free, and anyone can use this protocol without any license agreement.

2. Purpose of this document

This document covers the specifications of JMQT protocol version 1.0. The primary purpose of this document is to define the principles of the JMQT protocol 1.0 along with sample packets and operational behaviour.

3. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this specification are to be interpreted as described in IETF RFC 2119 [RFC2119].

Socket:
Standard socket connection over TCP/IP as defined in IETF RFC 147 [RCF147].
WebSocket:
Standard WebSocket connection over TCP/IP as defined in IETF RFC 6455 [RFC6455].
JSON:
Javascript Object Notation as defined in IETF RFC 7159 [RFC7159].
Client:
A program or device that uses JMQT. A Client always establishes the Network Connection to the Server. It can

- Publish Packets that other Clients might be interested in.
- Subscribe to request Packets that it is interested in receiving.
- Unsubscribe to remove a request for Packets.
- Disconnect from the Server.

Server:
A program or device that acts as an intermediary between Clients which publish packets and Clients which have made Subscriptions. A Server

- Accepts Network Connections from Clients.
- Accepts Packets published by Clients.
- Processes Subscribe and Unsubscribe requests from Clients.
- Forwards Packets that match Client Subscriptions.

Subscription:
A Subscription comprises a Channel Name and a maximum QOS (see section 9.a for QOS). A Subscription can associated with a single Session or can be persistent depending on the Persistence Flag (see section 9.c). A Session can contain more than one Subscription. Each Subscription within a session has a different Channel Name.
Channel Name:
The label attached to a packet which is matched against the Subscriptions known to the Server. The Server sends a copy of the published packet to each Client that has a matching Subscription.

Session:
A stateful interaction between a Client and a Server. Some Sessions last only as long as the Network Connection, others can span multiple consecutive Network Connections between a Client and a Server.
JMQT Packet:
A packet of information containing JSON data that is sent across the Network Connection. The JMQT specification defines fifteen different types of Packet, one of which (the Publish packet) is used to convey messages.

4. Version History

Date	Description	Prepared By	Remarks
3 Dec 2018	Initial Document Preparation	Shubhadeep Banerjee	Defining JMQT 1.0

5. Protocol Overview

JMQT consists of clients communicating to a server, primarily using Socket (or Web Socket) connection. A client may be either a publisher or a subscriber.

JMQT is built on the concept of Channels, which are used by the publishers to send messages, and are used by the subscribers to receive messages. When a publisher has a new message to distribute, it sends a 'publish' packet to the server with the channel name and the message. The server then distributes the message among the subscribers of that particular channel by sending the 'push' packets.

If the server receives a publish packet to a channel which has no current subscribers, it discards the message unless the publisher indicates that the message is to be retained (see section 9.b for retained packets). When a new subscriber subscribes to that channel (by sending a 'subscribe' packet), the retained message for that channel is sent to the subscriber allowing the new subscriber to receive the most current message of that channel rather than waiting for the next update from a publisher.

There are two special types of channels available in JMQT, Control channels and P2P channels. For details on P2P and control packets, see section 9.d.

6. Packet Types

JMQT protocol has the following packet types :

1. Auth (sent by the client to receive authentication token and client id)
2. Auth Acknowledgment (authentication status, token and client id sent back by the server)
3. Connect (sent by the client to establish a session, contains authentication token and client id)
4. Connect Acknowledgment (connection status sent back by the server)
5. Heartbeat (sent by the client in regular intervals to inform the server about its availability)
6. Heartbeat Acknowledgment (sent by the server to indicate that it is receiving the heartbeats)
7. Subscribe (sent by the client to subscribe to a channel)
8. Subscribe Acknowledgment (subscription status sent back by the server)
9. Unsubscribe (sent by the client to unsubscribe to a channel)
10. Unsubscribe Acknowledgment (unsubscription status sent back by the server)
11. Publish (sent by the client to publish a message to a channel)

12. Publish Acknowledgment (publish status sent back by the server, this message is option depending on the QOS)
13. Push (sent by the server to distribute a published message)
14. Push Acknowledgment (sent back by the client to indicate that it has received the push packet, this message is optional depending on the QOS)
15. Disconnect (sent by the client to indicate that the client is willingly disconnecting the server)

7. Packet Structure

JMQT packets are JSON packets with the root element of the packet indicating the packet type, i.e. the packets look like the following :

```
{ <packet type> : {<packet data in JSON>}}
```

Any kind of data transferred in a packet can be a JSON object or a plain string, depending on the business logic of the system. Any acknowledgement packet generated from the server contains a status code (see section 8 for the details on the JMQT status codes) which indicates the status of the action requested by the client.

Below is the detailed description of each of the packet types mentioned in section 6 with example packets indicating the real-life scenarios :

a. Auth Packet

Auth packets are sent by the client to the server to authenticate itself and get an authentication token for future communication. Auth packets contain the authentication information as a JSON object or a string. Auth packets can be used for login or registration in web based applications, registration of the client using IMEI or mac no., and so on.

Packet type code : auth

Parameters :

Parameter	Key	Type	Required
Auth Data	dt	JSON / String	Yes

Sample Packets :

1. Sample login :

```
{"auth":{"dt":{"login":{"email":"email@example.com","password":"my password"}}}}
```

2. Sample device registration using IMEI:

```
{"auth":{"dt":{"IMEI":"xxxxxxxxxxxxxx"}}}
```

b. Auth Acknowledgement

Server MUST validate an auth packet and acknowledge to the client with an Auth Ack packet. This auth ack packet contains a status code which denotes the status of the auth request (for details on status code see section 8), an optional authentication token & client id (if the authentication is

successful), and an optional message (mentioning the reason in case the authentication is unsuccessful). If the authentication is successful, the client will use the client id and auth token to send the connect packet to the server.

Packet type code : authAck

Parameters :

Parameter	Key	Type	Required
Status Code	st	Integer	Yes
Auth Token	at	String	Yes, if Status Code is 1 (OK)
Client ID	cl	String	Yes, if Status Code is 1 (OK)
Message	mg	String	Yes, if Status Code is not 1 (OK)

Sample Packets :

1. Status code 1 (OK) :

```
{"authAck":{"st":1,"at":"my token","cl":"client 1"}}
```

2. Status code 0 (FAILED) :

```
{"authAck":{"st":0,"mg":"Email and/or password not valid"}}
```

c. Connect Packet

Connect packets are sent by the client to the server to establish a JMQT session. Only the connected clients with established session can perform the JMQT operations like publish, subscribe, unsubscribe and receive push packets from the server. A connect packet contains the client id and auth token which the client may receive from the server in the auth acknowledgement or can be preset into the client program. A connect packet is followed by a connect ack generated by the server accepting or declining the session request.

Packet type code : conn

Parameters :

Parameter	Key	Type	Required
Auth Token	at	String	Yes
Client ID	cl	String	Yes

Sample Packets :

```
{"conn":{"at":"my token","cl":"client 1"}}
```

d. Connect Acknowledgement

Server responds to a connect packet by sending a connect ack. The connect ack notifies the client that the server has accepted the session request or not. The server MUST validate the auth token and client id sent by the client and return an appropriate status code along with the timeout seconds (no of seconds the server will wait before closing an idle connection) in the connect ack.

Packet type code : connAck

Parameters :

Parameter	Key	Type	Required
Status Code	st	Integer	Yes
Timeout Seconds	ts	Integer	Yes, if Status Code is 1 (OK)

Sample Packets :

1. Status code 1 (OK) :

```
{"connAck": {"st": 1, "ts": 15}}
```

2. Status code 0 (FAILED) :

```
{"connAck": {"st": 0}}
```

e. Heartbeat Packet

Once a session is established, the clients MUST send periodical heartbeat within the timeout seconds specified by the server in the connect ack. The heartbeat contains no data and just notifies the server that the client is online and active. The server shall close a session if the client does not send heartbeat packets for the timeout seconds. A client MUST send the heartbeats throughout an active session.

Packet type code : hb

Parameters : No parameters

Sample Packets : `{"hb":{}}`

f. Heartbeat Acknowledgement

Server MUST respond to the heartbeats received from the connected clients with the heartbeat ack. The heartbeat ack notifies the client that the server is receiving its heartbeats and is online. The server MUST not respond to any heartbeat packets if the client has not established the session yet by sending the Connection Packet or the session has been closed.

Packet type code : hbAck

Parameters : No parameters

Sample Packets : `{"hbAck":{}}`

g. Subscribe Packet

Clients generate subscribe packets and send to the server to subscribe to a particular channel. The subscribe requests MUST be validated by the server and it sends back a status code notifying that the subscription has been successful or not.

The subscribe packets also contains a flag notifying the persistence of the subscription. See section 9.c for the details of persistence subscription.

Subscription to control channels and P2P channels are not allowed (see section 9.d for details).

Packet type code : sub

Parameters :

Parameter	Key	Type	Required
Channel Name	cn	String	Yes
Persistence Flag	pr	Bit (0 or 1)	No, default is 0

Sample Packets :

1. No persistent flag (default) :

```
{"sub":{"cn":"my channel"}}
```

2. Persistent flag on :

```
{"sub":{"cn":"my channel","pr":1}}
```

h. Subscribe Acknowledgement

The subscribe requests are validated by the server and it acknowledges with a status code and the channel name notifying that the subscription has been successful or not.

Packet type code : subAck

Parameters :

Parameter	Key	Type	Required
Channel Name	cn	String	Yes
Status Code	st	Int	Yes

Sample Packets :

1. Status code 1 (OK) :

```
{"subAck": {"st": 1, "cn": "my channel"}}
```

2. Status code 0 (FAILED) :

```
{"subAck": {"st": 0, "cn": "my channel"}}
```

i. Unsubscribe Packet

Clients generate unsubscribe packets and send to the server to unsubscribe to a subscribed channel. The unsubscribe requests may be validated by the server, depending on the implementation, though it is not mandatory. Unsubscribing means that the client won't get any push messages published to that particular channel. If the client sends an unsubscription request to a channel which the client is not subscribed to, the server may or may not send an error code (status code other than OK) depending on the server implementation.

Packet type code : sub

Parameters :

Parameter	Key	Type	Required
Channel Name	cn	String	Yes

Sample Packets :

```
{"unsub":{"cn":"my channel"}}
```

j. Unsubscribe Acknowledgement

The unsubscribe requests are acknowledged by the server with a status code and the channel name notifying that the unsubscription has been successful or not. The unsubscribe requests may be validated by the server, depending on the implementation, though it is not mandatory.

Packet type code : unsubAck

Parameters :

Parameter	Key	Type	Required
Channel Name	cn	String	Yes
Status Code	st	Int	Yes

Sample Packets :

1. Status code 1 (OK) :

```
{"unsubAck": {"st": 1, "cn": "my channel"}}
```

2. Status code 0 (FAILED) :

```
{"unsubAck": {"st": 0, "cn": "my channel"}}
```

k. Publish Packet

Clients send publish packet to the server to publish a message (or data) to a channel. The data can be a JSON object or a String. A publish packet contains the channel and data, an optional QOS (for details see section 9.a), optional packet id (if QOS is set to 0) and an optional retained flag (see section 9.b for details).

Packet type code : pub

Parameters :

Parameter	Key	Type	Required
Channel Name	cn	String	Yes
Data	dt	JSON or String	Yes
QOS	q	Int	No, default is 0
Packet Id	id	String	Yes, if QOS is 1
Retained Flag	rt	Bit (0 or 1)	No, default is 0

Sample Packets :

1. Basic publish packet with string data (QOS 0, Retained flag off) :

```
{"pub":{"cn":"my channel","dt":"my message"}}
```

2. Basic publish packet with JSON data (QOS 0, Retained flag off) :

```
{"pub":{"cn":"my channel","dt":{"msg": "my message"}}}
```

3. Publish packet with QOS 1, Retained flag off :

```
{"pub":{"cn":"my channel","dt":"my message","q":1,"id":"4"}}
```

4. Publish packet with QOS 1, Retained flag on :

```
{"pub":{"cn":"my channel","dt":"my message","q":1,"id":"4","rt":1}}
```

l. Publish Acknowledgment

If the QOS of a publish packet is set to 1, the server MUST send a publish acknowledgement to the client. The server validates the publish packets and if the QOS is 1, it responds with a status code and the packet id sent by the client. The server may decline a publish request based on the business logic. If the client does not receive the acknowledgement of a QOS 1 packet, the client may retry sending the packet using the same packet id.

Packet type code : pubAck

Parameters :

Parameter	Key	Type	Required
Status Code	st	Int	Yes
Packet Id	id	String	Yes, if QOS is 1

Sample Packets :

1. Status code 1 (OK) :

```
{"pubAck": {"st": 1, "id": "4"}}
```

2. Status code 0 (FAILED) :

```
{"pubAck": {"st": 0, "id": "4"}}
```

m. Push Packet

When a client sends a publish packet to the server, the server distributes the data to the subscribed clients of the published channel using a push packet. The push packets look similar to the publish packets, but it is sent from the server to the clients. The QOS of the publish packet is replicated in the push packet. The packet id of the push packets are maintained by the server. Push packets contain the source client id of the message, and if the message is generated by the server, then the client id field is left blank (see section 9.e for details). The server may decide to store a push packet depending on the QOS and retained flag (see section 9.a and 9.b). If the message is a retained message, the retained flag will be sent 'on' or 1. Retained messages MUST not contain any packet id.

Packet type code : push

Parameters :

Parameter	Key	Type	Required
Channel Name	cn	String	Yes
Data	dt	JSON or String	Yes
Source Client Id	cl	String	Yes, can be empty
QOS	q	Int	No, default is 0
Packet Id	id	String	Yes, if QOS is 1
Retained Flag	rt	Bit (0 or 1)	No, default is 0

Sample Packets :

1. Basic push packet with String data (QOS 0, Retained flag off) :

```
{"push":{"cn":"my channel","dt":"my message","cl":"client 1"}}
```

2. Basic push packet with JSON data (QOS 0, Retained flag off) :

```
{"push":{"cn":"my channel","dt":{"msg": "my message"},"cl":"client 1"}}
```

3. Push packet with QOS 1, Retained flag off :

```
{"push":{"cn":"my channel","dt":"my message","cl":"client 1","q":1,"id":"400"}}
```

4. Push packet with Retained flag on :

```
{"push":{"cn":"my channel","dt":"my message","cl":"client 1","rt":1}}
```

n. Push Acknowledgment

If the QOS of a push packet is set to 1, the client MUST send a push acknowledgement to the server. The push ack looks similar to the publish ack. If the server does not receive a push ack, it MUST store the packet in some permanent storage and retry sending it when the client reconnects. Push ack for the retain packets are not required.

Packet type code : pushAck

Parameters :

Parameter	Key	Type	Required
Status Code	st	Int	Yes
Packet Id	id	String	Yes, if QOS is 1

Sample Packets :

```
{"pushAck": {"st": 1, "id": "400"}}
```

o. Disconnection Packet

If a client willingly closes a session, it MUST send a disconnect packet to the server before the disconnection. It notifies the server that the client connection is not being closed abruptly (due to network problem or some other technical problem). The server may clear the cache and buffer allocated for the client depending on the server implementation. The disconnect packet does not contain any data inside its body. Any non-persistent subscription (refer to section 9.c) will be removed from the server after the client sends a disconnection packet.

Packet type code : disconn

Parameters : No parameters
Sample Packets :

```
{"disconn":{}}
```

8. Status Codes

JMQT status code is a integer indicating the status of a request. The server sends status code with every acknowledgement against a request generated by the client. The JMQT client also sends a status code against the push packets sent by the server.

JMQT status code are listed below -

1	OK	The requested action has been successfully completed by the server
0	FAILED	The server has failed to perform the requested task
5	SERVER ERROR	Notifies that the server has encountered an error
6	INVALID TOKEN	The token passed in 'Conn' packet is invalid
7	NOT ALLOWED	The client is not allowed to perform the specific task
8	CLIENT OFFLINE	Special response code for P2P messages notifying that the destination client is offline.
10	INVALID PACKET	The request packet is malformed
11	INVALID CHANNEL	The channel is invalid. e.g. subscribing or unsubscribing to a control or P2P channel
9	NETWORK ERROR	Client program shall generate this error code if the client is unable to connect the server

9. Operational Behaviour

a. Quality of Service

JMQT has a concept of QOS (Quality of Service). Depending on the QOS, the server may or may not store and acknowledge to a 'publish' packet, and the client may or may not acknowledge to a 'push' packet. The QOS can either be 0 or 1. The 'q' key in a publish packet denotes the QOS. If no QOS is mentioned in a packet, i.e. the 'q' key is not set, the server MUST consider the QOS as 0. If QOS is set to 1, the client MUST include a packet id ('id' key) in the publish packet.

Example :

1. Default QOS (QOS 0) :

```
{"pub":{"cn":"my channel","dt":"my message"}}
```

2. QOS 1 :

```
{"pub":{"cn":"my channel","dt":"my message","q":1,"id":"1"}}
```

If the QOS is set to '0' in a publish packet, the server will distribute the message (by sending push packets which also have the QOS set to '0') among the connected clients which are subscribed to the channel and will forget the packet. This means, no offline subscriber will receive the message when they reconnects. The online subscribers who will receive the 'push' message from the server, do not need to acknowledge if the QOS is set to '0'.

Example of QOS 0 push packets:

```
{"push":{"cn":"my channel","dt":"my message","cl":"client id", "q":0}}
```

If the QOS is set to '1' in a publish packet, the server will first store the message, then it will distribute the message (by sending push packets with QOS set to '1') among the connected clients which are subscribed to the channel. The online subscribers who will receive the 'push' message from the server, MUST acknowledge if the QOS is set to '1'. If the server does not receive the push acknowledgement from a client or the client is offline, it will send the stored message again when the client reconnects to the server. The QOS 1 packets may be pushed to the client more than once to ensure the delivery. The packet id of the push packets are generated by the server and independent to the packet id sent by the client in the publish packet.

Example of QOS 1 push packets:

```
{"push": {"cn":"my channel","dt":"my message","cl":"client id", "id": "1", "q": 1}}
```

b. Retained Packets

The retained messages are used for allowing the new subscriber to receive the most current message of that channel rather than waiting for the next update from a publisher.

Retained messages are different from the QOS. If a channel does not have any subscribed clients, QOS 1 messages are discarded by the server, whereas the retained messages are still stored for the future subscribers. There can be only one (the latest) retained message stored in the server for a particular channel at any point of time. Server MUST send the retained messages to the client in two situations : 1. When the client connects to the server and it has persistent subscriptions to a channel (see section 9.c). 2. When the client subscribes to a channel. The push packets MUST have the retain flag on. Retained push packets do not have a packet id and retained flag is set to 1 to inform the client that the packet is a retained packet. The clients do not need to send push ack for any retained packets. Retained packets are not available for the Control and P2P channels.

A publish packet may have the retained flag ('rt' key) and QOS both set to 1, either of them set to 1, or both set to 0. If the retained flag is not mentioned in a packet, the server MUST consider the retained flag as 0 or off.

Example :

1. Publish packets with retain flag on :

```
{"pub":{"cn":"my channel","dt":"my message","rt":1}}
```

2. Push packets with retained messages :

```
{"push": {"cn":"my channel","dt":"my message","cl":"client id", "rt": "1"}}
```

c. Persistent Subscription

Persistent subscriptions are used to store the subscription in the server even after a client session is closed.

The subscribe packets contains a flag notifying the persistence of the subscription. If the persistent flag is on (i.e. 1), the server MUST store the subscription even if after the client session is closed. If the flag is off (i.e. 0), the server will remove the subscription mapping when the client disconnects.

The server MUST store all the QOS 1 packets of the persistent channels even if the client is offline. The server MUST resend the stored messages once the client reconnects and establishes a new session. Clients do not need to subscribe to channel again if the subscription is persistent until the

client unsubscribes to the channel. Non-persistent subscriptions are automatically unsubscribed when the client disconnects. To change the persistence of a subscription, the client MUST unsubscribe to the channel, and subscribe it again with the desired persistence flag.

Example :

1. Subscription with persistent flag on :

```
{"sub":{"cn":"my channel","pr":1}}
```

2. Subscription with persistent flag off / default :

```
{"sub":{"cn":"my channel","pr":0}}
```

d. P2P and Control Packets

There are two special types of channels available in JMQT, control channels and P2P channels.

Control channels are similar to the HTTP API calls. Control channels (starts with $) are used to retrieve some information from the server itself. Implementation of control channels is optional and server dependent. The client may publish a packet with or without data to a control channel which the server MUST process and respond to the client. Any packet published to a control channel MUST contain a packet id.

Example of control packets:

Step 1. Client publishes a message to control channel (to get a list of the subscribed channels) -

```
{"pub":{"cn":"$mySubscriptions","dt":{},"id":"2"}}
```

Step 2. Server responds back with publish ack with required data (list of the subscribed channels) -

```
{"pubAck": {"st": 1, "id": "2", "dt": {"channels": ["my channel"]}}}
```

P2P channels (starts with #) are used to send messages directly to one client from another client. In order to send a P2P message, the sender MUST know the P2P address of the receiver. P2P messaging is also optional, but it's recommended in the applications where the clients always need to talk to each other directly (e.g. chat applications) without the need of broadcasting a message to multiple clients. The behaviour of P2P channels are just like the normal channels (i.e. it supports QOS) except the P2P channels does not support retained packets.

Example of P2P packets:

1. P2P publish packet with QOS 1 -

```
{"pub":{"cn": "#my client 1", "dt": "my p2p message", "id": "5", "q": 1}}
```

2. P2P push packets -

```
{"push":{"cn":"#my client 1","dt":"my p2p message","cl":"my client 2","id": "401","q": 1}}
```

Clients are never allowed to subscribe to a control channel or a P2P channel. Subscriptions to the P2P channels are controlled by the server (see section 9.e for details).

e. Server side Sub, Unsub and Push

JMQT protocol includes the possibility to perform subscription and unsubscription initiated by the server.

When a client establishes a new session, the server may make the client subscribe and/or unsubscribe to certain channels without the subscription and/or unsubscription request generated by the client. In case of P2P channels, the subscription request should not be generated by the client, instead the server MUST perform the subscription to the P2P channels on behalf of the client when the client establishes a session. In certain applications, it is also possible that the server will completely decline any subscription or unsubscription request from the clients, and subscription or unsubscription operations will be performed by the server itself.

Server can also push messages to all the available channels without having a publish packet received from a client. This ensures that the server may push required information (such as software update notification, billing notification etc.) directly to the clients anytime the server wants. The push packets generated by the server does not have any value assigned to the source client id (the 'cl' key in push packets), instead the value is left empty (blank string). The QOS principle is similar to that of the normal push packets. Follow is an example of a server side QOS 1 push packet :

```
{"push": {"cn": "server channel", "dt": "You have an update", "cl": "", "id": "402",
"q": 1}}
```

f. Disconnection behaviour

The client can disconnect the server (or end a session) in three ways - 1. The client can just close the socket or websocket connection, 2. The client may send a disconnection packet to the server, 3. Both 1 and 2. The recommended way of disconnection is way no. 3, as it ensures the server that the client is initiating the disconnection willingly and the server can clear the cache (if any) assigned to that particular session. Disconnecting the socket or websocket is also important because an unused socket or websocket connection may lead to network error in the client system.

When a session is closed, the server MUST remove all non-persistent subscriptions from the disconnected client along with the QOS 1 push packets mapped for that particular client. The client SHOULD stop sending the heartbeat packets once the session is closed.

10. Authentication and security

In order to connect to the server, a client MUST send a combination of client id and authentication token to the server. This authentication token and client id can be pre-configured in the client, or the client can retrieve those credentials from the server by sending the Auth packet, which will contain the authentication information like user id and password.

JMQT does not define any built-in encryption protocol. However, as it uses TCP connection (Socket and WebSocket) for communications, TLS (Transport Layer Security) can easily be implemented on top of socket and WebSocket to secure the incoming and outgoing messages.