MQ Telemetry Transport (MQTT) Programming

Tyler Lacroix & Roger Lacroix
IBM WebSphere MQ Telemetry

- WebSphere MQ Telemetry component is known as MQXR ('eXtended Reach')

- MQTT was added as an installable feature of IBM WebSphere MQ 7.0.1 before being fully integrated into WebSphere MQ version 7.1.

- MQTT is a feature of WebSphere MQ that extends the universal messaging backbone with the MQTT protocol to a wide range of remote sensors, actuators and telemetry devices.
IBM WebSphere MQ Telemetry

- Fully integrated / interoperable with WMQ
  - MQTT messages translated to standard WMQ messages
  - Administration included as part of WebSphere MQ Explorer

- Telemetry channels enable MQTT connections to the queue manager
  - Supports MQTTv3 protocol (most common in use)

- Scalability
  - 100,000+ clients

- Security
  - SSL channels
  - JAAS authentication

- Ships with reference Java (for MIDP upwards) and C clients
  - Small footprint clients
  - other APIs and implementations of MQTT available via 3rd parties
IBM WebSphere MQ Telemetry

- MQ applications use Publish/Subscribe to communicate with MQTT client applications.

- MQ applications can use Point-To-Point messaging to send a message directly to an MQTT client application (Note: This is one-way!!)
  - Connect to your queue manager
  - On the MQOPEN API call:
    - Set the QMgr Name to the MQTT Client Id
    - Set the Queue Name to the Topic
  - Use MQPUT to send messages directly to a particular MQTT client application
WebSphere MQ Telemetry Topology

Diagram showing the integration of sensors, mobile devices, and applications with WebSphere MQ through the MQXR daemon for devices.
WebSphere MQ Installation on Windows

- **Typical**: WebSphere MQ will be installed with the most common features. Recommended for new users.
- **Compact**: Installs the minimum features you need to run WebSphere MQ.
- **Custom**: Choose a uniquely identifiable installation name, the WebSphere MQ features you want installed and where they will be installed. Recommended for advanced users.
WebSphere MQ Installation on Windows
WebSphere MQ Installation on Linux

rpm -ivh MQSeriesXRService-7.5.0-0.x86_64.rpm
rpm -ivh MQSeriesXRClients-7.5.0-0.x86_64.rpm
MQ Explorer on Windows

Welcome to MQ Telemetry

The MQ Telemetry feature supports the connection of telemetry devices from the edge of a network, to WebSphereMQ. These telemetry devices range from sensors and actuators, to mobile phones, smart meters, medical devices, vehicles, and satellite locations. This connection is made possible by the WebSphere MQ Telemetry Transport (MQTT) protocol.

MQTT is an open message protocol that enables the transfer of messages from telemetry devices to a message server or vice versa. MQTT is designed to run on constrained devices and over constrained networks (for example, low bandwidth, high latency, or fragile networks). Examples of constraints on devices includes low memory and low processing power.

In order for a queue manager to accept connections from a telemetry device, one or more telemetry channels are needed. Running the Define sample configuration wizard creates a telemetry channel and starts the MQ telemetry service.

The sample configuration has been set up for this queue manager.

Run MQTT Client Utility...

MQ Telemetry documentation

MQ Telemetry Service: SYSTEM/MQXR SERVICE
Status: Started
MQ Explorer on Windows
MQ Explorer on Windows

![MQ Explorer on Windows](image_url)
MQTT Client Utility

Connection:
- Host: localhost
- Port: 1883
- Client identifier: mqtt_ACERTyler_1
- Status: Connected
- Options... Connect Disconnect

Client history:

<table>
<thead>
<tr>
<th>Event</th>
<th>Topic</th>
<th>Message</th>
<th>QoS</th>
<th>Retained</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected</td>
<td>testTopic</td>
<td></td>
<td>0</td>
<td>No</td>
<td>9/19/13 2:36 PM</td>
</tr>
<tr>
<td>Subscribed</td>
<td>testTopic</td>
<td></td>
<td>0</td>
<td>No</td>
<td>9/19/13 2:36 PM</td>
</tr>
<tr>
<td>Published</td>
<td>testTopic</td>
<td>Test M...</td>
<td>0</td>
<td>No</td>
<td>9/19/13 2:37 PM</td>
</tr>
<tr>
<td>Received</td>
<td>testTopic</td>
<td>Test M...</td>
<td>0</td>
<td>No</td>
<td>9/19/13 2:37 PM</td>
</tr>
</tbody>
</table>

Subscription:
- Topic: testTopic
- Request QoS: 0 - At most once

Publication:
- Topic: testTopic
- Message: Test Message
- QoS: 0 - At most once

View message... Clear history Scroll lock
What is MQ Telemetry Transport (MQTT)?

MQ Telemetry Transport (MQTT) is a simple publish/subscribe lightweight messaging protocol.

It is open source and royalty-free, allowing easy adaptation for a wide variety of devices.

Ideal for constrained environments where network bandwidth is low and when remote devices may have limited processing capabilities. This design allows thousands of remote clients to be interconnected, resulting in “Internet of Things”.
What is MQ Telemetry Transport (MQTT)?

**Open**
- Open published spec designed for the world of “devices”
  - Invented by IBM and Eurotech
  - MQTT client code (C and Java) donated to the Eclipse "Paho" M2M project

**Lean**
- Minimized on-the-wire format
  - Smallest possible packet size is 2 bytes
  - No application message headers
- Reduced complexity/footprint
  - Clients: C=30Kb; Java=100Kb

**Reliable**
- Three qualities of service:
  - 0 – at most once delivery
  - 1 – assured delivery but may be duplicated
  - 2 – once and once only delivery
- In-built constructs to support loss of contact between client and server.
  - “Last will and testament” to publish a message if the client goes offline.
- Stateful “roll-forward” semantics and “durable” subscriptions.

**Simple**
- Simple / minimal pub/sub messaging semantics
  - Asynchronous (“push”) delivery
  - Simple set of verbs -- connect, publish, subscribe and disconnect.
MQTT Concept: Publish/Subscribe

- The MQTT protocol is based on the principle of publishing messages and subscribing to topics, which is typically referred to as a PUBLISH/SUBSCRIBE model. Clients can subscribe to topics and thereby receive whatever messages are published to those topics. Or clients can publish messages to topics, thus making them available to all subscribers to those topics.
MQTT Concept: Topics & Subscriptions

- Messages in MQTT are published to topics, which can be thought of as subject areas. Clients, in turn, sign up to receive particular messages by subscribing to a topic. Subscriptions can be explicit, which limits the messages received to the specific topic at hand, or use wildcard designators (+ and #) to receive messages across a variety of related topics.
MQTT Concept: Clean sessions & durable connections

When an MQTT client connects to the server, it sets the clean session flag. If the flag is set to true, then all of the client's subscriptions are removed when it disconnects from the server. If the flag is set to false, then the connection is treated as durable, and the client's subscriptions remain in effect after any disconnection. In this event, subsequent messages that arrive carrying a high QoS designation are stored for delivery once the connection is reestablished. Also note that this an optional behavior, and that messages may get lost. Even with QoS=2 messages may get lost because all of the server state is purged on reconnect.
MQTT Concept: Retained messages

With MQTT, the server keeps the message even after sending it to all current subscribers. If a new subscription is submitted for the same topic, any retained messages are then sent to the new subscribing client.
MQTT Concept: Wills

When a client connects to a server, it can inform the server that it has a will, or a message that should be published to a specific topic or topics in the event of an unexpected disconnection. This is particularly useful in alarm or security settings where system managers must know immediately when a remote sensor has lost contact with the network.
MQTT Concept: Qualities of Service

MQTT defines three Quality of Service (QoS) levels for message delivery:

- QoS = 0 "At most once", messages are delivered according to the best efforts of TCP/IP network. Message loss or duplication can occur. A response is not expected and no retry defined in the protocol.

- QoS = 1 "At least once", where messages are assured to arrive but duplicates may occur.

- QoS = 2 "Exactly once", where messages are assured to arrive exactly once.
MQTT Concept: Security

- You can pass a username and password with an MQTT connect packet in V3.1 of the protocol.

- Encryption across the network can be handled with SSL, independently of the MQTT protocol itself (it is worth noting that SSL is not the lightest of protocols, and does add significant network overhead).

- Additional security can be added by an application encrypting data that it sends and receives, but this is not something built-in to the protocol, in order to keep it simple and lightweight.
Some C Code

Some code... Using Paho Asynchronous MQTT client library for C

#include "MQTTAsync.h"

#include "MQTTClientPersistence.h"
C Code: Connecting to MQTT Server

3 Steps:

1. Create a MQTTAsync

2. Create a MQTTAsync_connectOptions structure and set the options

3. Call MQTTAsync_connect and pass the MQTTAsync object and the MQTTAsync_connectOptions structure
C Code: Connecting to MQTT Server
Creating The Client

MQTTAsync client;

MQTTAsync_create(&client, "tcp://m2m.eclipse.org:1883", "clientId", MQTTCLIENT_PERSISTENCE_NONE, NULL);

MQTTAsync_setCallbacks(client, NULL, connectionLost, messageArrived, NULL);
C Code: Connecting to MQTT Server Setting Connection Options

```c
MQTTAsync_connectOptions conn_opts = MQTTAsync_connectOptions_initializer;
conn_opts.keepAliveInterval = 20;
conn_opts.cleansession = 1;
conn_opts.onSuccess = onConnect;
conn_opts.onFailure = onConnectFailure;
conn_opts.context = client;
```
C Code: Connecting to MQTT Server

More Connection Options

conn_opts.username = "yourUsername";
conn_opts.password =  "yourPassword";
conn_opts.ssl = ssl_structure;
conn_opts.will = will_structure;
conn_opts.context = client;
… and more
C Code: Connecting to MQTT Server

Connecting

MQTTAsync_connect(client, &conn_opts);
C Code: Callbacks

```c
void onConnect(void* context, MQTTAsync_successData* response) {}  
void onConnectFailure(void* context, MQTTAsync_failureData* response){}  
void connectionLost(void *context, char *cause) {}  
```
C Code: Subscribing to a Topic

2 Steps:

1. Create a `MQTTAsync_responseOptions` structure and set the options

2. Call `MQTTAsync_subscribe` and pass the `MQTTAsync` object and the `MQTTAsync_responseOptions` structure
C Code: Subscribing to a Topic Setting Subscription Options

```c
MQTTAsync_responseOptions opts = MQTTAsync_responseOptions_initializer;

opts.onSuccess = onSubscribe;

opts.onFailure = onSubscribeFailure;

opts.context = client;
```
C Code: Subscribing to a Topic

```c
int _qos = 0;
MQTTAsync_subscribe(client, "Topic", _qos, &opts);
```
C Code: Sending a Message

3 Steps:

1. Create a MQTTAsync_message

2. Create a MQTTAsync_responseOptions structure and set the options

3. Call MQTTAsync_sendMessage and pass the MQTTAsync_message and the MQTTAsync_responseOptions structure
C Code: Sending a Message
Creating The Message

MQTTAsync_message pubmsg = MQTTAsync_message_initializer;

char *message = "this is a test message";

pubmsg.payload = message;

pubmsg.payloadlen = strlen(message);

pubmsg.qos = 0;

pubmsg.retained = 0;
C Code: Sending a Message

Sending Options

```c
MQTTAsync_responseOptions opts =
    MQTTAsync_responseOptions_initializer;

opts.onSuccess = onSend;

opts.onFailure = onSendFailure;

opts.context = client;
```
C Code: Sending a Message

Sending

MQTTAsync_sendMessage(client, "Topic", &pubmsg, &opts);
int messageArrived(void *context, char *topicName, int topicLen, MQTTAsync_message *message)
{
    message->payload
    topicName
}

Some Java (Android) Code

Some code... Using Paho MQTT Client library for Java

import org.eclipse.paho.client.mqttv3
Java Code: MQTT

You must have an object that implements MqttCallback

```java
public void connectionLost(Throwable cause)
public void deliveryComplete(IMqttDeliveryToken token)
public void messageArrived(String topic, MqttMessage message) throws MqttException
```
Java Code: Connecting to MQTT Server

4 Steps:

1. Create a `MqttAsyncClient` object
2. Create a `MqttConnectOptions` object and set the options
3. Create a `IMqttActionListener` listener
4. Call `connect` method on `MqttAsyncClient` object and pass the `MqttConnectOptions` and the `IMqttActionListener` objects
Java Code: Connecting to MQTT Server
Creating The Client

MqttAsyncClient client; // Store Globally

try {
    client = new MqttAsyncClient("tcp://m2m.eclipse.org:1883", "clientId");
    client.setCallback(this); // Set Callback to object implementing MqttCallback

} catch (MqttException e) {
    // Catch Error
}
Java Code: Connecting to MQTT Server Setting Connection Options

MqttConnectOptions conOpt = new MqttConnectOptions();

conOpt.setCleanSession(true);
conOpt.setKeepAliveInterval(20);
conOpt.setPassword("password".toCharArray());
conOpt.setUserName("userName");

…. And More
Java Code: Connecting to MQTT Server Connection Listener

IMqttActionListener conListener = new IMqttActionListener() {

    public void onSuccess(IMqttToken asyncActionToken) {
        //Connected
    }

    public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
        //Failed to Connect
    }

};
Java Code: Connecting to MQTT Server

Connecting

try {
    client.connect(conOpt, "Connect sample context", conListener);
} catch (MqttException e) {
    // Catch Error
}
Java Code: Subscribing to a Topic

2 Steps:

1. Create a IMqttActionListener listener

2. Call subscribe method on the MqttAsyncClient object and pass the IMqttActionListener object
Java Code: Subscribing to a Topic
Subscription Listener

IMqttActionListener subListener = new IMqttActionListener() {

    public void onSuccess(IMqttToken asyncActionToken) {

        //Subscription Successful

    }

    public void onFailure(IMqttToken asyncActionToken, Throwable exception) {

        //Subscription Failed

    }
};
Java Code: Subscribing to a Topic

Subscribing

```java
try {
    int qos = 0;
    client.subscribe("testTopic", qos, "Subscribe sample context", subListener);
}
catch (MqttException e) {
    // Error
}
```
Java Code: Publishing a Message

2 Steps:

1. Create a IMqttActionListener listener

2. Create MqttMessage and call publish method on the MqttAsyncClient object and pass the IMqttActionListener object
Java Code: Publishing a Message
Publishing Listener

IMqttActionListener pubListener = new IMqttActionListener() {

    public void onSuccess(IMqttToken asyncActionToken) {
        //Publish Successful
    }

    public void onFailure(IMqttToken asyncActionToken, Throwable exception) {
        //Publish Failed
    }
};
Java Code: Publishing a Message
Publishing the Message

```java
try {
    MqttMessage message = new MqttMessage("test message".getBytes());
    message.setQos(0);
    client.publish(topicName, message, "Pub sample context", pubListener);
} catch (MqttException e) {
    //Error when trying to send message
}
```
public void messageArrived(String topic, MqttMessage message) throws MqttException {

    System.out.println("Message Arrived:" + new String(message.getPayload()) +
                     "At Topic:\t" + topic +
                     " QoS:\t" + message.getQos());
}
Demo 1

Message Arrived: testing
From iOS!
From Topic: testTopic
onSend
onSubscribe
onConnect
Demo 2

iOS

MQTT Server

Android

<table>
<thead>
<tr>
<th>MQTT Banking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account #</td>
</tr>
<tr>
<td>Total Balance</td>
</tr>
</tbody>
</table>

Transfer Money

| To: #00001 | Amount: $5.00 |

History

| To #00004 | Amount: -$90.00 |
| From #00004 | Amount: $65.86 |

<table>
<thead>
<tr>
<th>MQTT Banking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account #</td>
</tr>
<tr>
<td>Total Balance</td>
</tr>
</tbody>
</table>

Transfer Money

| To: #00001 | Amount: $5.00 |

History

| To #00004 | Amount: -$90.00 |
| From #00004 | Amount: $65.86 |
Demo 2

iOS

MQTT Banking

<table>
<thead>
<tr>
<th>Account #</th>
<th>00002</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Balance:</td>
<td>$726.14</td>
</tr>
</tbody>
</table>

Transfer Money

To: #00001  Amount: $5.00

Send

History

To #00004  -$90.00
From #00004  +$65.86

Android

MQTT Banking

<table>
<thead>
<tr>
<th>Account #</th>
<th>00002</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Balance:</td>
<td>$726.14</td>
</tr>
</tbody>
</table>

Transfer Money

To: #00001  Amount: $5.00

Send

History

To #00004  -$90.00
From #00004  +$65.86
Demo 2

iOS

MQTT Banking

Account #: 00002
Total Balance: $726.14

Transfer Money
To: #00001 Amount: $5.00

Send

History
To #00004 - $90.00
From #00004 + $65.86

Android

MQTT Banking

Account #: 00002
Total Balance: $726.14

Transfer Money
To: #00001 Amount: $5.00

Send

History
To #00004 - $90.00
From #00004 + $65.86
Demo 2

iOS

MQTT Banking

Account #: 00002
Total Balance: $726.14

Transfer Money
To: #00001 Amount: $5.00

Send

History
To #00004 - $90.00
From #00004 + $65.86

Android

MQTT Banking

Account #: 00002
Total Balance: $726.14

Transfer Money
To: #00001 Amount: $5.00

Send

History
To #00004 - $90.00
From #00004 + $65.86
Questions & Answers