Java Magazine

Written by the Java community for Java and JVM developers

Coding, Libraries & Frameworks

Programming lightweight IoT messaging with MQTT in Java

June 3, 2022 | 18 minute read

Eric J. Bruno

MQTT is an open message protocol perfect for communicating telemetry-style data from distributed devices and sensors over unreliable or constrained networks.

MQTT is an open message protocol for Internet of Things (IoT) communication. It enables the transfer of telemetry-style data in the form of messages from distributed devices and sensors over unreliable or constrained networks to on-premises servers or the cloud. Andy Stanford-Clark of IBM and Arlen Nipper of Cirrus Link Solutions invented the protocol, which is now a standard maintained by OASIS.

In case you were wondering, *MQTT* originally meant "message queuing telemetry transport." However, that name has been deprecated, and there's no official meaning now to the acronym. (The same is true of *OASIS*, the name of the open standards consortium, which similarly doesn't spell out anything today, but which used to mean "Organization for the Advancement of Structured Information Standards.")

MQTT's strengths include simplicity and a compact binary packet payload (the compressed headers are

much less verbose than those of HTTP). The protocol is a good fit for simple push messaging scenarios such as temperature updates, stock price tickers, oil pressure feeds, or mobile notifications. It also works well connecting constrained or smaller devices and sensors to the enterprise, such as connecting an Arduino device to a web service.

The MQTT message types and headers

MQTT defines a small set of message types, such as those shown in **Table 1**.

Table 1. MQTT message types

Message Type	Value	Meaning
CONNECT	1	Client request to connect to server
CONNACK	2	Connect acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish acknowledgment
PUBREC	5	Publish received (assured delivery, part 1)
PUBREL	6	Publish release (assured delivery, part 2)
PUBCOMP	7	Publish complete (assured delivery, part 3)
SUBSCRIBE	8	Client subscribe request
SUBACK	9	Subscribe acknowledgment
UNSUBSCRIBE	10	Client unsubscribe request
UNSUBACK	11	Unsubscribe acknowledgment
PINGREQ	12	PING request
PINGRESP	13	PING response
DISCONNECT	14	Client is disconnecting
AUTH	15	Authentication exchange

Note that message type values 0 and 15 are reserved.

The message type is set as part of the MQTT message fixed header, shown in **Table 2**, which includes flags in the first byte indicating the message type (four bits), whether the message is being resent (one bit), a quality of service (QoS) flag (two bits), and a message retention flag (one bit). The remaining portion of the fixed header indicates the length of the rest of the message, which includes a header that varies by message type, and the message payload.

Table 2. MQTT message header

bit	**7 | 6 | 5 | 4**	**3**	**2**	**1**	**0**
byte 1	Message type	DUP	QoS level		RETAIN
byte 2	Message length (one to four bytes)				
byte 3	… if needed to encode message length				
byte 4	… if needed to encode message length				
byte 5	… if needed to encode message length				

As for the header fields, the message types were described in **Table 1**. The DUP flag is set to 1 when a message is resent. The QoS level will either be 0 (meaning no guarantee of delivery), 1 (meaning the message will never be lost but duplicate delivery may occur), or the strictest level of 2 (meaning exactly one delivery and the message is never lost and never duplicated).

The RETAIN flag, which applies only to published messages, indicates that the server should hold on to that message even after it has been delivered. When a new client subscribes to that message's topic, the message will be sent immediately. Subsequent messages for the same topic with the RETAIN flag set will replace the previously retained message. This is useful for cases where you don't want a client to wait for a published message (say, due to a change in value) before it has an existing value.

The length of the fixed header can be up to five bytes long depending on the total length of the message. An algorithm is used, whereby the last bit in each byte of the message length is used as a flag to indicate if another byte follows that should be used to calculate the total length.

For more details on the algorithm, see the MQTT specification, which is encoded in **Listing 1**. (Also see the sample application code I used in this article for the sender https://github.com/ericjbruno/mqtt_sender and the listener https://github.com/ericjbruno/mqtt_listener.)

Listing 1. The fixed header length algorithm explained in Java

```
public class VariableLengthEncoding {
   public static void main(String[] args) {
       int value = 321;
       // Encode it
       Vector<Integer> digits = encodeValue(value);
```

```
        System.out.println(value + " encodes to "  + digits);
        // Derive original from encoded digits
        int value1 = decodeValue(digits);
        System.out.println("Original value was " + value1);
    }

    public static Vector encodeValue(int value) {
        Vector<Integer> digits = new Vector();
        do {
            int digit = value % 128;
            value = value / 128;

            // if there are more digits to encode
            // then set the top bit of this digit
            if ( value > 0 ) {
                digit = digit | 0x80;
            }
            digits.add(digit);
        }
        while ( value > 0 );
        return digits;
    }

    public static int decodeValue(Vector<Integer> digits) {
        int value = 0;
        int multiplier = 1;
        Iterator<Integer> iter = digits.iterator();
        while ( iter.hasNext() ) {
            int digit = iter.next();
            value += (digit & 127) * multiplier;
            multiplier *= 128;
        }
        return value;
    }
}
```

Copy code snippet

The length applies to the remaining portion of the message and does not include the length of the fixed header, which can be between two and five bytes.

Each message type contains its own header with applicable flags, called a variable header because it varies depending on message type.

For instance, the CONNECT message, used when a new client application connects to the MQTT server, contains the protocol name and version it will use for communication, a username and password, flags that indicate whether messages are retained if the client session disconnects, and a keep-alive interval.

By contrast, a PUBLISH message variable header contains the topic name and message ID only. The header components for the remaining message types are covered in the MQTT specification.

Publishing messages and subscriptions

The PUBLISH message has two use cases. First, the PUBLISH message type is sent by a client when that client wishes to publish a message to a topic. Second, the MQTT server sends a PUBLISH message to each subscriber when a message is available, such as when a message has been published by another client.

When a client sends a PUBLISH message (to send a message), the message must contain the applicable topic name and the message payload, as well as a QoS level for that particular message. The response to the sender will depend on the QoS level for the message sent, as follows:

- QoS 0: Make the message available to all interested parties; there is no response.
- QoS 1: Store the message in persistent storage, send to all interested parties, and then send a PUBACK back to the sender (in that order).
- QoS 2: Store the message in persistent storage (but don't send it to interested parties yet), and then send a PUBREC message to the sender (in that order).

When an MQTT client application wishes to subscribe to a topic, it first connects, and then it sends a SUBSCRIBE message. This message contains a variable length list of one or more topics (each specified by name) along with a QoS value for each topic that the client is subscribing to. The server will send a SUBACK message in response to the SUBSCRIBE request message.

In terms of the QoS level for a topic, a client receives messages at less than or equal to this level, depending on the QoS level of the original message from the publisher. For example, if a client has a QoS level 1 subscription to a particular topic, a QoS level 0 PUBLISH message to that topic stays at that QoS level when it's delivered to the client. However, a QoS level 2 PUBLISH message to the same topic will be downgraded to QoS level 1 for delivery to that client.

Message flow per MQTT QoS level

It's time to examine the message flow per QoS level in more detail.

QoS level 0 messages. For QoS level 0 messages, the server simply makes the published message available to all parties and never sends a message in response to the sender, as shown in **Figure 1**.

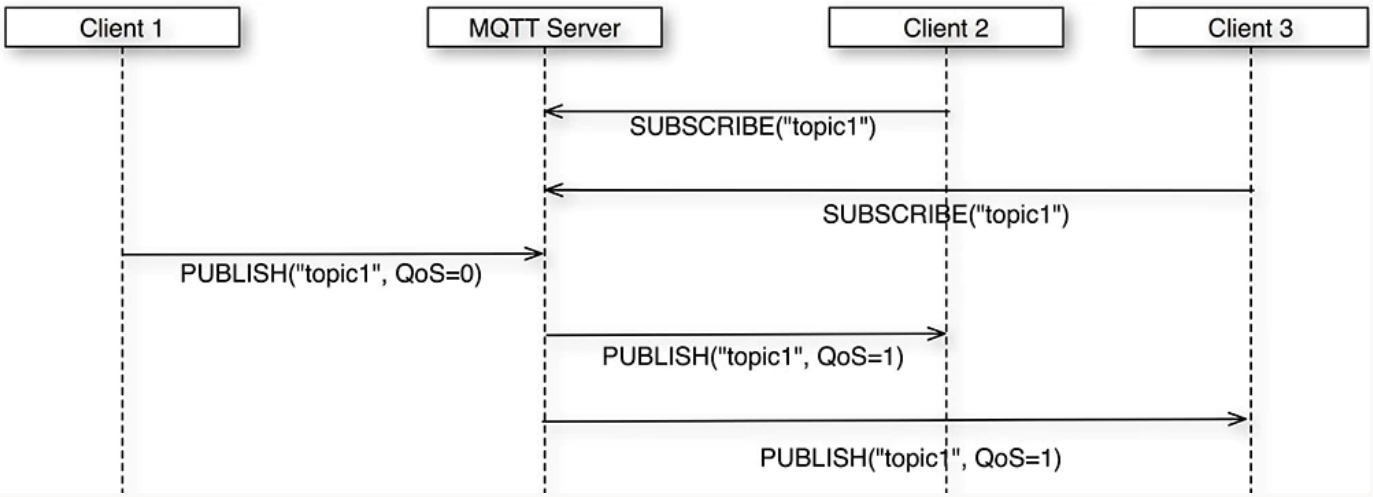


Figure 1. Message flow for a QoS level 0 published message

Since there are no guarantees with QoS level 0 messages, neither the sender nor the server persist the messages.

QoS level 1 messages. For QoS level 1 messages, the message flow is very different from QoS level 0 messages. See **Figure 2**.

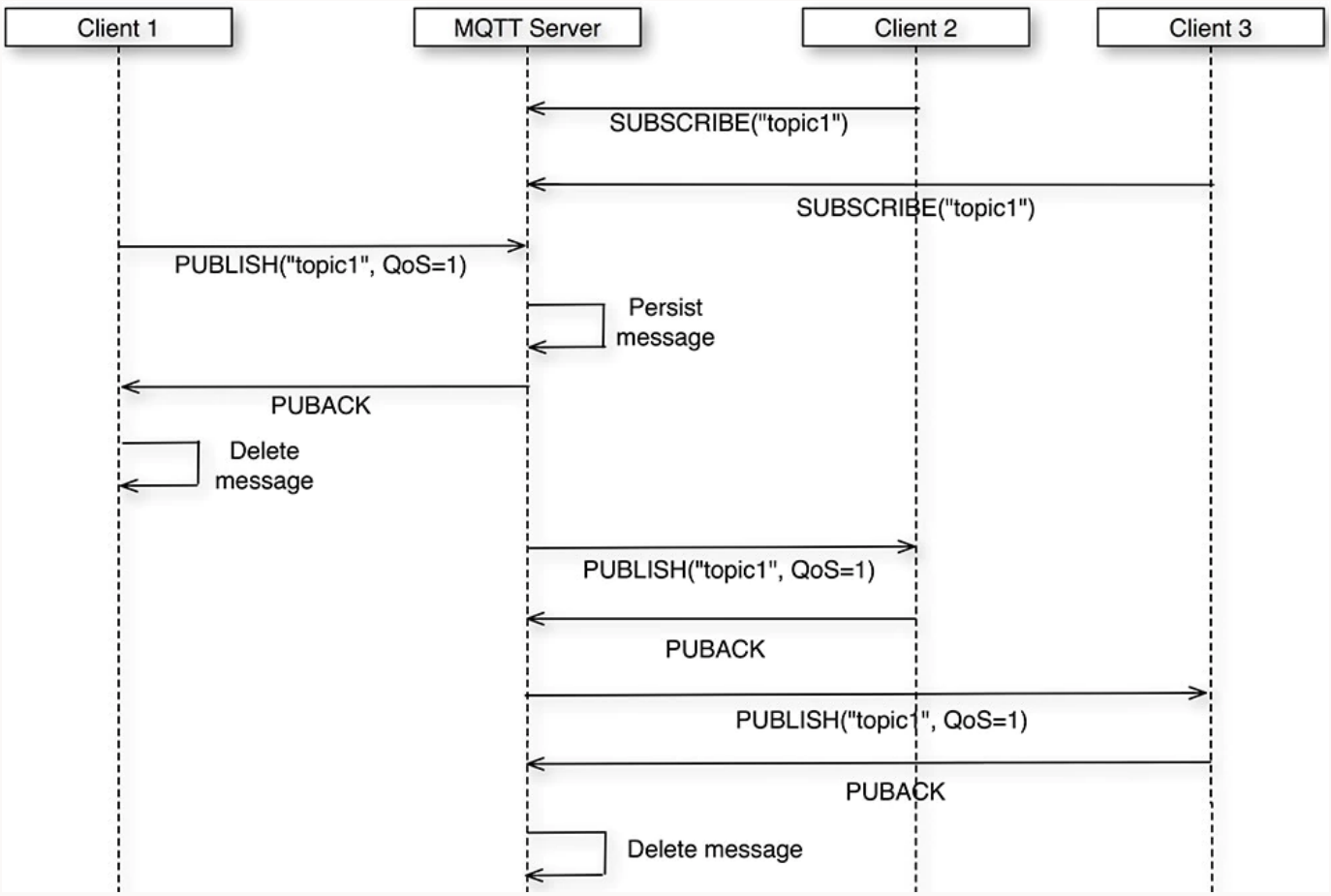
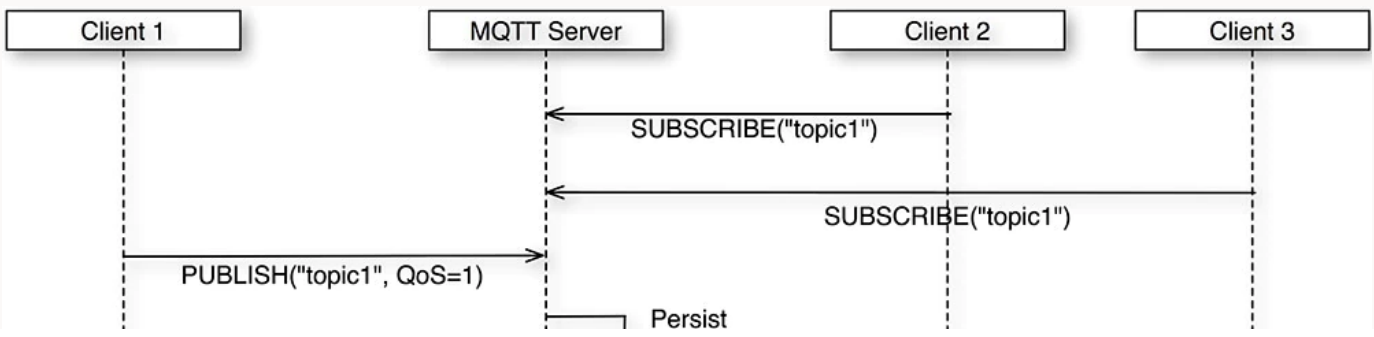


Figure 2. Message flow for a QoS level 1 published message

When a client publishes a message, the MQTT server first persists the message to ensure delivery. Next, it delivers the message to each subscriber by sending them PUBLISH messages with the message payload. After each subscriber acknowledges the message, the server deletes the message from the persisted store and sends the original sender an acknowledgment.

QoS level 2 messages. QoS level 2 messages are the strictest in terms of message delivery, guaranteeing once-and-only-once message delivery per client. The message flow is shown in **Figure 3**.



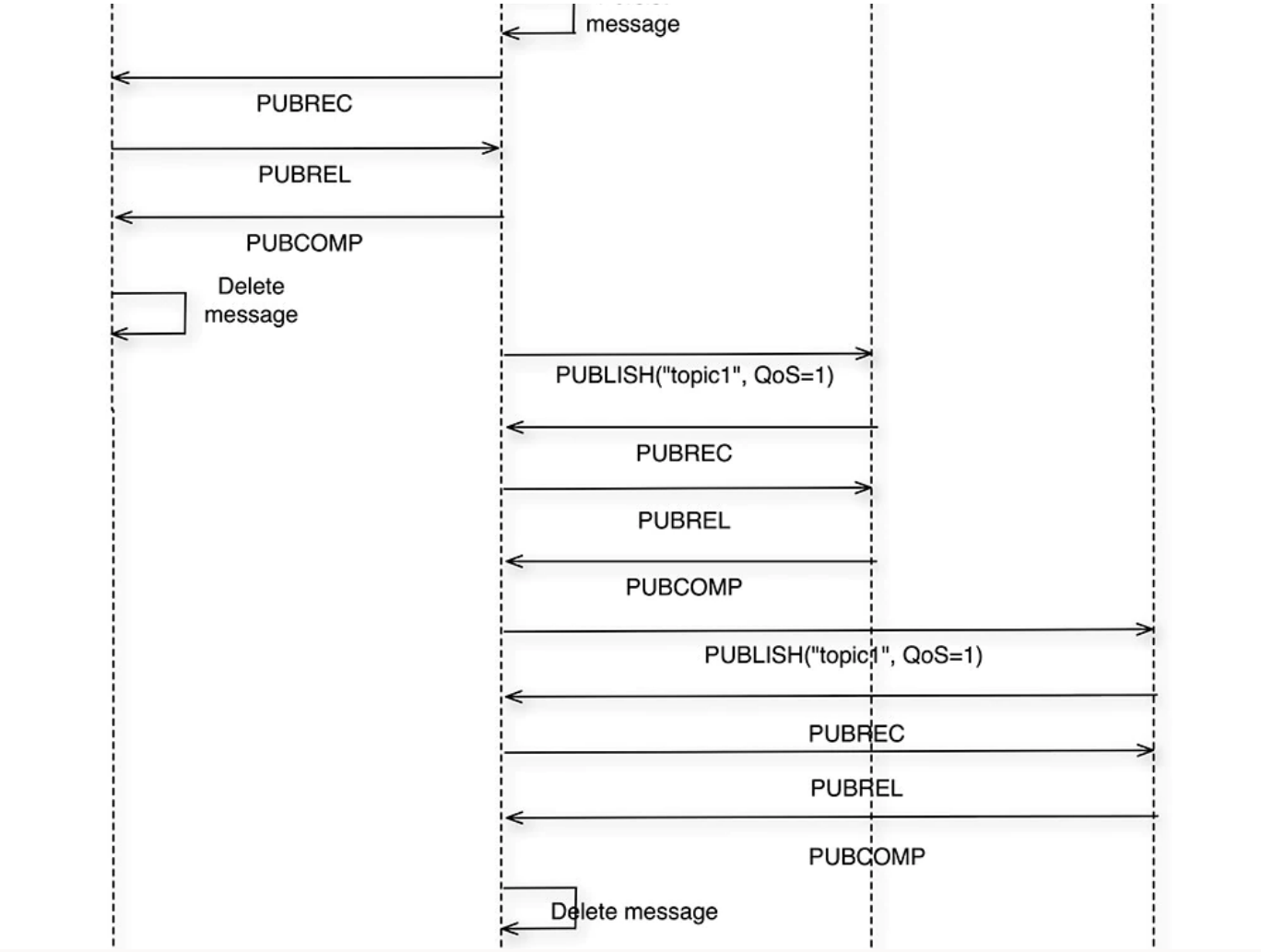


Figure 3. Message flow for a QoS level 2 published message

When a client publishes a QoS level 2 message, the MQTT server first persists the message to ensure delivery. Next, it sends a PUBREC (publish received) message to the sender, who then replies with a PUBREL (publish release) message.

When the server responds with a PUBCOMP (publish complete) message, the initial sender exchange is complete, and the sender can delete the message from its persistent store.

After that, the MQTT server delivers the message to each subscriber by sending them PUBLISH messages with the message payload. After message receipt, each subscriber sends a PUBREC message in acknowledgment, which the server responds to with a PUBREL message.

Finally, after each subscriber acknowledges with a PUBCOMP message, the server deletes the message from the persisted store, and the QoS level 2 message send sequence is complete.

Setting up an MQTT broker

For the sample code below, I chose to use the Eclipse Mosquitto open source MQTT broker. Downloads are available for Windows, Linux (via `aptitude` or `apt`, if your distribution supports it), and macOS (via `Brew` if you choose to use it). Simply download binaries for Windows and macOS (via `Brew`), or install it for Linux using the following command:

```
>sudo apt install mosquitto
```

This works on a Raspberry Pi running Raspbian Linux as well. Using `apt`, Mosquitto will be installed in `/usr/bin`, and you can start it directly from there. However, it's best to start Mosquitto on Linux as a System V `init` service with the following command:

```
>sudo service mosquitto start
```

Subsequently you can stop Mosquitto with the following command:

```
>sudo service mosquitto stop
```

Alternatively, you can then start Mosquitto on a UNIX-based system with the following command:

```
>/usr/local/sbin/mosquitto
```

The default listen port is 1883, but you can specify a different port via the `–p` command-line parameter when you start Mosquitto. You can also supply an optional configuration file via the `–c` command-line parameter. Use the configuration file to specify alternative ports, certificate-based encryption, an access control list file, logging directives, and more, as described in the documentation.

The Eclipse Paho project contains MQTT libraries for Java and other languages. For Java, download the JAR files, or you can build from the source. For this article, I used the MQTTv5-compatible Paho library.

To build the Paho JAR files, you'll need Apache Maven. Be sure to define your `JAVA_HOME` environment variable or Maven will fail. The command would be the same on Linux; use the correct path to the JDK on your local system. Once you've done this, execute the following command to build Paho:

```
> mvn package –DskipTests
```

Sending and receiving MQTT messages

I'll begin by showing you how to create a sample application that simulates reading the current temperature from a temperature sensor, published as an MQTT message.

Create a Java project and add the Paho client library JAR file to the classpath, which is needed to connect to and communicate with an MQTT broker. The code is written for version 5 of the MQTT specification, so be sure to use version 5 of the Paho client library with the following Maven entry in the POM file:

```
<dependency>
      <groupId>org.eclipse.paho</groupId>
      <artifactId>org.eclipse.paho.mqttv5.client</artifactId>
      <version>1.2.5</version>
    </dependency>
```

Copy code snippet

Connect to the MQTT broker. To connect to the MQTT message broker, use the code in **Listing 2**.

Listing 2. The Java code to connect to an MQTT broker

```
MqttClient mqttClient= null;
// ...
try {
   MqttConnectOptions connOpts = new MqttConnectOptions();
   connOpts.setCleanStart(false);

   mqttClient =
     new MqttClient(
       "tcp://localhost:1883",    // broker URL
       "my_sender",               // a unique client id
       new MemoryPersistence() ); // or file-based, or omit

   mqttClient.connect(connOpts);
   System.out.println("Connected");
}
catch ( MqttException me ) {
   // ...
}
```

Copy code snippet

You can connect to any MQTT broker by changing the address in the URL parameter of the `MqttClient` class constructor. For instance, instead of `tcp://localhost:1883`, you can specify a different address and even the port, such as `tcp://192.168.1.10:1880`. The URL must begin with `tcp://`.

The second parameter is the client ID that must be unique for each client connecting to a specific broker. With the third parameter, you specify the way you want messages persisted *at the client* for guaranteed delivery. Keep in mind this does not affect message persistence at the broker; it affects only messages in flight to and from the client application.

This example uses `MemoryPersistence`, meaning messages will be stored in memory (which is fast) while awaiting notification from the broker that the message was received and/or delivered, depending on the QoS level specified. If the message needs to be resent to the broker for some reason, it's retrieved from the memory store.

In this case, message delivery is guaranteed only if the client application isn't shut down before the broker receives and stores the message. For additional reliability, choose `FilePersistence` instead (which can be slower). This guarantees message delivery even in cases where the client is shut down or crashes while messages are in flight and the client is subsequently restarted.

Finally, call `MqttClient.connect` to connect to the specified broker. You can provide connection options via a parameter, such as minimum MQTT protocol version requirements (via `setMqttVersion`), whether or not to maintain state across restarts of the client (via `setCleanStart`), authentication details

whether or not to maintain state across restarts of the client (via `setCleanStart`), authentication details, and timeout intervals.

Publishing MQTT messages. Once your application has connected to a broker, it can begin to send and receive messages. The payloads of MQTT messages are sent as byte arrays, making it easier to avoid cross-platform or cross-language issues. **Listing 3** sends the current temperature, originally formatted as a `String`, to the topic `currentTemp`. You can see how the message is formatted as a `byte` array and then sent with the highest level of message QoS (once-and-only-once delivery).

Listing 3. Java code to publish an MQTT message

```
private void publishTemp(Long temp) {
   try {
       String content = "Current temp: " + temp.toString();
       MqttMessage message = new MqttMessage( content.getBytes() );
       message.setQos(2); // once and only once delivery
       mqttClient.publish("currentTemp", message);
   }
   catch ( MqttException me ) {
       // . . .
   }
}
```

Copy code snippet

Receiving MQTT messages. To receive messages, first connect to an MQTT broker, as shown in **Listing 2**. Next, subscribe to a topic (or multiple topics) by calling the `subscribe()` method on the `MqttClient` class, providing the topic and requested QoS level as parameters.

```
MqttSubscription subscription =
    new MqttSubscription(topic, qos);

IMqttToken token =
    client.subscribe(
        new MqttSubscription[] { subscription });
```

Copy code snippet

Messages published at a lower QoS will be received at the published QoS, while messages published at a higher QoS will be received using the QoS specified in the call to `subscribe`. Messages are delivered asynchronously to a callback, which in this case is a class that implements the `IMqttMessageListener` interface.

```
IMqttToken token =
    client.subscribe(
        new MqttSubscription[] { sub },
        new Mqtt MessageListener[] { this } );
```

Copy code snippet

The Java class supplied must implement the notification method, `messageArrived()`, as shown in **Listing 4**.

Listing 4. Implementing the IMqttMessageListener interface to receive message notifications

```
@Override
public void messageArrived(String topic, MqttMessage message)
 throws Exception {
   String messageTxt = new String( message.getPayload() );
   System.out.println(
       "Message on " + topic + ": '" + messageTxt + "'");
}
```

Copy code snippet

The `messageArrived()` method is called when a message is available from the broker for the supplied topic name. Providing the topic name is helpful in cases where you subscribe to multiple topics with a single callback. Optionally, you can set a separate callback for each topic, or you can specify some other arbitrary grouping.

Topic filter hierarchy and wildcards. MQTT supports hierarchical topic names, with subscriptions at any level within the hierarchy. For instance, say you define a topic hierarchy based on publishing temperature and other data for rooms across floors in a set of buildings, as shown below.

```
building/10/floor/1/room100/temperature
building/10/floor/1/room101/temperature
. . .
building/10/floor/20/room2001/temperature
. . .
```

To receive temperature readings for room 201 in building 2, subscribe to the topic `building/2/floor/2/room201/temperature`. Data can be published and subscribed to at any depth in the hierarchical tree (not just the leaves). For instance, you can publish/subscribe to occupancy data at `building/10/floor/1` as well as temperature data at `building/10/floor/1/room100/temperature`.

Leveraging this format, you can subscribe to sensor readings across entire floors and even entire buildings by using the *multilevel* wildcard string: #. For example, subscribing to `building/10/floor/1/room101/#` subscribes to all topics (assuming there are more devices than temperature sensors) for room 101. Further, subscribing to `building/10/floor/1/#` subscribes to all topics (and related sensors) for all rooms on the first floor of building 10. You can even subscribe to all topics in the system with just # as the topic name.

However, the following shows another scenario where each room supports temperature and humidity data readings with topics:

```
building/10/floor/1/room101/temperature
building/10/floor/1/room101/humidity
```

If your application needs to get all temperature data (but not humidity) for all rooms on the first floor, you can insert the *single-level* wildcard character + (not the multilevel #) within the topic hierarchy to subscribe to, as follows:

```
building/10/floor/1/+/temperature
```

Note that the single-level wildcard subscribes to all topic names in that one level of the hierarchy only. For example, if you subscribe to `temperature/average/state/+`, you would receive temperature data for every state in the country. However, if you subscribe to `temperature/average/state/#`, you *also* receive temperature data for each major city within each state, as follows:

```
temperature/average/state/NY/Albany
temperature/average/state/NY/Kingston
. . .
temperature/average/state/CA/Sacramento
temperature/average/state/CA/Pasadena
. . .
```

You can use the single-level wildcard more than once within the hierarchy. For example, to receive temperature data for all rooms across all floors in building 10, subscribe to the following topic:

```
building/10/floor/+/+/temperature
```

Request/response messaging. When a message is published, the publisher can request a reply by including a response topic name in the message properties, as shown below.

```
MqttProperties props = new MqttProperties();
props.setResponseTopic("RespondTopic");
props.setCorrelationData( ("" + (correlationId++)).getBytes() );
message.setProperties(props);
client.publish("MyTopic", message);
```

Copy code snippet

When a subscriber receives this message, it can check for the existence of the response topic and then send a reply message, as shown in **Listing 5**.

Listing 5. Implementing request/response messaging with MQTT

```
@Override
public void messageArrived(String topic, MqttMessage mqttMessage)
    throws Exception
{
    String messageTxt = new String( mqttMessage.getPayload() );
    // ...

    MqttProperties props = mqttMessage.getProperties();
    String responseTopic = props.getResponseTopic();
    if ( responseTopic != null ) {
        String corrData = new String(props.getCorrelationData());

        MqttMessage response = new MqttMessage();
        props = new MqttProperties();
        props.setCorrelationData(corrData.getBytes());
        response.setProperties(props);

        String content = "Received. Correlation data=" + corrData;
        response.setPayload(content.getBytes());

        client.publish(responseTopic, response);
    }
}
```

Copy code snippet

Of course, this means that the publisher needs to have already subscribed to the response topic prior to sending the request. The correlation ID can be used to match responses to requests, if the receiver sets it in the response message.

```
MqttSubscription sub = new MqttSubscription("response", 0);
    IMqttToken token =
        client.subscribe(
            new MqttSubscription[] { sub },
            new IMqttMessageListener[] { this });

    //...

    MqttProperties props = new MqttProperties();
```

```
        correlationId++;
        props.setResponseTopic("response");
        props.setCorrelationData( (""+correlationId).getBytes() );
        message.setProperties(props);
```

Copy code snippet

This way you can have one subscriber for all request/response messages, yet you will still be able to match them with the correlation data or the topic name.

Selecting MQTT broker implementations

There are quite a few MQTT brokers available under varying license models. Additionally, there are cloud services that support MQTT integration, such as Oracle Internet of Things Production Monitoring Cloud Service and Oracle Internet of Things Cloud Service.

The following are a few open source MQTT implementations:

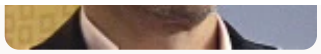
- Eclipse Mosquitto
- ActiveMQ MQTT
- Paho
- RabbitMQ

Conclusion

MQTT messaging is straightforward, multiplatform, and intended to scale larger servers to constrained devices and sensors. It supports fast and reliable message delivery, and it's easy to understand and code for.

Dig deeper

- HTML5 server-sent events with Micronaut.io and Java
- Fast data access in Java with the Helidon microservices platform
- How to test Java microservices with Pact
- Curly Braces #2: The design phase in Agile Java development



Eric J. Bruno

Eric J. Bruno is in the advanced research group at Dell focused on Edge and 5G. He has almost 30 years experience in the information technology community as an enterprise architect, developer, and analyst with expertise in large-scale distributed software design, real-time systems, and edge/IoT. Follow him on Twitter at @ericjbruno.

‹ Previous Post