mediadriver

Media Driver Consulting
Blog Post

MQTT in the Enterprise

Author: Matt Pavlovich

September 21, 2016

About the author

Matt Pavlovich

Matt Pavlovich is a Founding Partner and Technical Practice Lead of Media Driver, a premier Solutions Provider and Systems Integrator with technologies for Integration, Business Process Management and DevOps. Media Driver provides consulting services, training courses and software products to assist organizations in their adoption of these technologies.

Matt directs the Media Driver Product Development Team. Media Driver products accelerate Enterprise application development while also increasing supportability. Media Driver tools provide a wide range of business users with greater access to Middleware solutions.

Matt has a lengthy resume of work within the Open Source Software community and is a Committer on the Apache ActiveMQ project. Matt is a hands-on technical leader who has led some of the largest and most well-known JBoss Fuse and JBoss A-MQ implementations in the world.

Summary

Recently, I have had a number of Enterprise customers ask me about using MQTT in Enterprise use cases. Technology trends often become rebranded as Enterprise "must haves" in order to expand an addressable market. What was classified a manufacturing automation, sensor tracking and SCADA now falls under the "IoT" umbrella. A quick Google of "IoT protocol" will return a number of pages suggesting MQTT and by the second page of search results, it's almost entirely MQTT-related links.

The questions IT Managers are asking boil down to these two: Do I need to adopt this protocol for my IoT-like use cases? Do I need to adopt an MQTT-specific messaging platform?

The answer for most Enterprises is yes for the first question and most definitely no for the second. Let's dig in further and balance the technical enthusiasm with some practical realities. MQTT's most unique feature can be critically beneficial—even life saving; however, the same feature has a shortcoming that must be considered in order to be operationally efficient in avoiding false alarms.

What is MQTT?

MQTT (or MQTT3) is a lightweight messaging protocol designed for lightweight communication between devices and computer systems. Originally designed for SCADA networks, manufacturing and low-bandwidth scenarios, MQTT has gained popularity recently due to the growth in the Internet-of-Things (IoT) space. The most recent revision of the MQTT OASIS standard is v3.1.1.

MQTT solutions are typically deployed very similarly to other Enterprise Messaging solutions, such as JMS or AMQP in a client-server-client model where devices publish messages to a centralized broker and another device or application connects to a centralized broker to receive messages.

MQTT is often mentioned as a peer-to-peer communication solution. This is a misnomer. In practice, "peer-to-peer" means that a broker is being embedded with each client node and effectively becomes a (client-server)-(server-client) architecture. The downside of this approach is significant. Peer-to-peer solutions mean that the full brunt of maintenance (security, patching, logging, and monitoring) of a Messaging System is required for each node, instead of just the centralized nodes in a standard architecture.

What makes MQTT unique?

MQTT has a feature that enables an application to configure a message to be sent in the event that connectivity from the device to the MQTT server is disconnected unexpectedly. This feature is called a Last Will and Testament message (aka LastWill). A separate application is then deployed to monitor for LastWill messages to immediately take action and/or produce alarm notifications. Practical scenarios for manufacturing and SCADA uses cases would be an event where a device fails, and other systems or devices can be instructed to take action to prevent equipment damage or injury to individuals.

The second most talked about feature is that MQTT has very low overhead in its protocol, making it ideal for computing scenarios where information needs to be transmitted over low-bandwidth networks or scenarios that involve a high volume of small messages—such as monitoring of manufacturing equipment, SCADA networks, sensor devices and scenarios involving low bandwidth, such as radio and cellular networks. However, this feature is largely overstated, as MQTT does not include message headers and other features common to messaging platforms. MQTT developers frequently have to implement these capabilities in the MQTT message payload to meet application requirements, so this benefit is largely negated.

What is the value to an Enterprise?

1. The LastWill message feature is unique and provides the ability to automate actions for planned and unplanned outages of devices and equipment.
2. Enterprises that need to operate on an embedded computing device with low memory footprint such as: sensor equipment, machine controls, SCADA networks, beacons, and manufacturing equipment can benefit from MQTT's lightweight footprint in languages such as C, C++ and others.
3. Enterprises that have low bandwidth connectivity scenarios such as: legacy cellular networks (EDGE or 2G), radio, or satellite networks may benefit from MQTT's minimal bandwidth overhead.

What are the sharp corners?

- MQTT does not include many features that are common in Enterprise Messaging Systems including message ID (1), expiration, timestamp, correlation ID, priority, transactions, or custom message headers.
- MQTT does not define a standard client API, so application developers have to select the best fit. The libraries are incomplete in their Enterprise feature, and developers are required to write complex test code in order to validate common test scenarios, such as connection timeout and simulating an abnormal disconnect in order to trigger the sending of the LastWill message (2).
- MQTT only supports binary message formats. Developers must apply best practices when converting text, XML and JSON data formats to binary in order to properly handle cross-platform differences.
- MQTT only supports the publish-subscribe (aka topics) messaging pattern and does not have a point-to-point (aka queues) messaging pattern.

- MQTT does not define a server-to-server standard.
- MQTT's lack of a queuing semantic or server-to-server standard complicates solutions that need to span networks, such as a DMZ or to store-and-forward messages to back-office systems for collection and processing of data.
- MQTT support is not wide spread in the industry yet. Many commercial back-office products and ERP systems do not include native support for MQTT and would require a message bridge in order to complete back-office data collection and processing.
- MQTT-only products are limited in their Messaging Features and would also require a message bridge to move data reliably into the enterprise back office.
- MQTT's message retain feature support is tied to the producer-side. Generally, the broker or the consuming client sets consumer and subscription policies. This breaks the clean separation of producers, brokers and consumers that is provided by other messaging protocols. All consumers to a topic will have to handle the retained message behavior, even if only one requires it.
- MQTT's QoS feature is often implemented using broker-side durable subscriptions which overlaps with the retain feature. Administrators and developers are frequently confused to differentiate these concepts and the lack of a standard API means some MQTT clients do not return the retain and QoS information to consuming applications.
- MQTT's Last Will Message also includes retain and QoS features. Developers struggle with the complexity of these scenarios when combined. Additionally, the lack of a message timestamp makes it difficult to correctly determine the validity of a given Last Will Message that is configured to also use retain.

Note: All notes as of MQTT v3.1.1

What's the hype?

1. MQTT Quality of Service features are often touted as a differentiator. In practice, there is no practical improvement over what JMS, STOMP, AMQP or other enterprise messaging protocols provide.
2. MQTT protocol does minimalize bandwidth out-of-the-box. However, most JMS and AMQP providers offer a compression or tight encoding option to achieve similar results.
3. MQTT is not a replacement for general Enterprise Messaging use cases due to its lack of standard message headers, custom headers and binary-only message formats. Developers using MQTT frequently have to implement their own message body formats and server-side plugins to provide these capabilities.
4. MQTT's peer-to-peer messaging architecture is not a new solution or a recommended architecture in Enterprise use cases. Messaging solutions that require a broker on an edge device increase operation maintenance overhead and should only be used in specific use cases.

How do I ensure success?

1. Ensure application teams properly configure their MQTT client sessions. MQTT has some tricky configuration combinations that can lead to messages being retained on servers longer than desired. Grab an expert to help with this. We often perform short engagements (1-2 weeks MAX) with our clients to help resolve the MQTT client-side issues.
2. MQTT should be in utilized in applications running on devices, equipment or sensor monitors where the LastWill feature provides benefit.
3. Define requirements in order to properly handle LastWill messages that are sent during planned maintenance windows.

4. Define and enforce development policies and standards that includes documentation for the retain message feature and QoS handling to avoid troubleshooting headaches. For example, Last Will should rarely, if ever, be configured with retain message.
5. Enterprises should define a standard message payload format for use by all applications using MQTT.
6. Media Driver recommends Enterprises deploy MQTT applications using a messaging broker that supports multiple messaging protocols (such as IBM MQ or Apache ActiveMQ/Red Hat JBoss A-MQ) in order to account for the back-office side integration requirements and to better handle server-to-server use cases. This allows the device side of the application to leverage MQTT, but have the back-office side stick with standard JMS or AMQP protocols, which are more widely, adopted in commercial software and ERP systems.

What are the alternatives to MQTT?

1. There is no alternative to the LastWill feature.
2. Enterprises with solutions requiring more robust enterprise messaging features such as expiration, timestamps, message Id's, and transactions should look to using other industry standard messaging protocols such as JMS or AMQP.
3. An alternative for lightweight messaging for C, C++, Ruby/PHP/Python languages is STOMP. STOMP is similar to MQTT in that it has minimal bandwidth overhead, and has the advantage of also supporting transactions. Apache ActiveMQ supports STOMP, IBM MQ does not support STOMP.

How can I help make Enterprise Messaging better?

Glad you asked! Media Driver has submitted a request that the JMS v2 standard include a LastWill feature. This feature would provide the same LastWill capability to JMS v2, while still being able to retain all the other key Enterprise Messaging features that are available in JMSv2. Media Driver plans on submitting the same request to the STOMP and AMQP standards.

To help in having the feature adopted in JMS v2, vote on the feature request: https://java.net/jira/browse/JMS_SPEC-177

Notes

(1) Some MQTT client implementations, such as Eclipse's Paho provide a message Id in their specific implementation. However, since this is not defined in the MQTT v3.1.1 standard, the reliability of the message ID depends on which MQTT brokers and clients are used throughout the entire environment.

(2) Media Driver provided a patch to Eclipse's Paho MQTT Java client to allow for testing of unplanned disconnect. This is feature will be available in the v1.3.0 release. See: https://github.com/eclipse/paho.mqtt.java/issues/247

Terminology

Broker - A server-side application that collects messages from client applications that are producing messages and delivers messages to client applications that are registered to consume messages.

Publish-Subscribe - A messaging pattern where an application producing messages will have those messages replicated to one or more interested consuming applications. This is generally implemented using a Topic.

Point-to-Point - A messaging pattern where an application producing messages will be delivered to only one interested consuming application. This is generally implemented using a Queue.

References

[1] MQTT.org website: http://mqtt.org
[2] Eclipse IOT Project: http://iot.eclipse.org
[3] JMS v2 Last Will support: https://java.net/jira/browse/JMS_SPEC-177
[4] Eclipse Paho Java Abnormal Disconnect: https://github.com/eclipse/paho.mqtt.java/issues/247

Media Driver
3824 Cedar Springs Road
Suite 430
Dallas, TX 75219
Phone 855-4-SOA-TEAM
Email info@mediadriver.com
Web www.mediadriver.com

Media Driver Consulting
Blog Post