StormMQ
Messaging as a Service

A Comparison of AMQP and MQTT

Introduction

AMQP and MQTT are both open protocols for asynchronous message queuing which have been developed and matured over several years. Recently, (4Q 2011) the organisations who developed them have made announcements that their latest protocol versions that are 'ready' for widespread adoption, and have submitted them for standardisation. AMQP has selected the OASIS industry standards group[1], with the intention of moving to becoming an ISO/IEC standard. MQTT has chosen to use the Eclipse foundation[2].

Overview

Both provide basic messaging needs; beyond that, AMQP provides a very much richer set of messaging scenarios. AMQP is almost a complete superset, lacking only explicit protocol support for Last-Value-Queues and will messages. However, its deliberate design for extensibility, using an IANA-like approach with a discursive approach, ensures that such features can be added in a forward-compatible, widely agreed upon way.

Both protocols are being promoted for 'widespread' use in the internet:-

- MQTT as a low-overhead, simple to implement way to send data, especially from embedded devices;
- AMQP as the asynchronous complement to HTTP

As such, both are being promoted as being ideal for cloud computing and the 'internet of things'. That essential thesis is correct; message queuing, with its asynchronous nature and minimal need for configuration when done right, is perfect for interoperating many different environments.

However, MQTT is constrained to providing basic messaging 'topics' in a single 'namespace', with no long-lived 'store-and-forward' queuing pragmatic. This makes it difficult, if not often impossible, to multi-tenant server resources, or to dynamically migrate them or provide simple 'development to production' switch-over. Even worse, a woefully naive security / user model makes proper resource sandboxing and analysis very limited. AMQP provides for sand-boxed, multi-tenanted or multi-hosted infrastructure, ideal for the modern cloud with multiple user security schemes appropriate to the modern internet.

Lastly, it's worth noting that MQTT, intended for telemetry transmission, is used in none of the world's biggest message queue based telemetry projects: Scripps Oceanography's monitoring of the Mid-Atlantic Ridge[3], and Smith Electric Vehicle's global fleet management[4], both use versions of AMQP.

---

[1] http://www.oasis-open.org/news/pr/amqp-tc

[2] http://mqtt.org/2011/11/eclipse-paho-open-source-and-other-news

[3] https://confluence.oceanobservatories.org/display/CIDev/Messaging+Service

[4] Green Car Congress: Smith Applies StormMQ Cloud-Based Message Queuing in EV Telematics System

Methodology

This white paper intends to primarily compare features at a high-level, rather than provide a blow-by-blow technical tick-list, with references to pages, sections and bullet points. Such exercises are a bit dry for the high-level view this paper takes,

Origins

AMQP comes from the finance community, and is primarily customer-driven: its originators wanted an open way to communicate the vastly increasing over-the-counter trace, risk and clearing market data they transfer, and do so without needing the pain of a bespoke protocol and its licensing headache. MQTT is vendor-driven; it comes from IBM and its partners as a reaction to the high cost of implementation MQSeries inflicts on its customers using small devices. These two approaches have strongly influenced the design and features of the protocols.

Intended Use of Protocol

The intended use of a protocol often influences its design; that is certainly the case for these. Both protocols 'sit' above TCP/IP, and are intended to be used to allow programs to send and receive messages asynchronously irrespective of their choice of hardware, operating system or programming language. However, from that basis, the protocols diverge; MQTT is designed to be of use for many small, relatively dumb devices sending small messages on low-bandwidth networks. AMQP, on the other hand, is designed to provide the full vibrancy of messaging scenarios that have been seen in the last 25 years. MQTT's design goals are a subset of its intended uses.

In particular, MQTT very much sees the network between the involved parties as a controlled, near private infrastructure; AMQP, on the other hand, is designed assuming it is in use between parties under different controls and who use network and infrastructure resources outside of those parties' control.

Optimisation of Framing

Both protocols provide for highly optimised 'on-the-wire' framing of data. MQTT uses a more stream-orientated approach, making it easy for low-memory clients to write frames. AMQP uses a buffer-orientated approach, making possible high-performance servers. MQTT does not permit fragmentation of messages, making it difficult to transmit large messages with constrained memory devices, however.

Messaging Scenarios

MQTT supports publish-subscribe messaging to topics. MQTT's messaging is effectively ephemeral: it is optimised for the use case of active routing of simultaneously connected publishers and subscribers. Consequently, it is very difficult to use it for classic long-lived message queuing. AMQP supports this use case, and more, with five different kinds of message publisher-consumer 'lifetime', from 'as long as connected' to 'nobody is using this queue'.

AMQP permits almost any form of messaging including classic message queues, round-robin, store-and-forward and combinations thereof. For example, some consumers can get copies of messages whilst others pull straight from the same queue, all using different filters. AMQP also uses message meta-data to support idempotent messages and message grouping.

Transactions

This is short but poignant. MQTT does not support transactions; it does support basic acknowledgments. AMQP supports different acknowledgment uses cases and transactions across message queues; it allows separation of the different transactional semantics, should that be needed, and for acknowledgments to be out-of-order or even delayed, and batched up as a performance optimisation. The protocol also defines, but makes optional, support for distributed transactions, such as X/Open XA transactions or MS DTC ones.

Connection Security

MQTT does not address connection security, although the community does provide advice. AMQP on the other hand, has specifically worked to integrate with TLS (eg TLS virtual server extensions, known as SNI) and SASL, the IETF set of RFCs that provide appropriate ways of securing the right to use a connection. Going further, it has ensured that modern SASL mechanisms, e.g. SCRAM-SHA and GS2, and security techniques (eg binding TLS channels to SASL mechanisms) works seamlessly. AMQP's core design allows separate negotiation of, and policies for, TLS and SASL mechanisms and upwards replacement with alternative techniques as they develop.

AMQP provides for different approaches to TLS negotiation, allowing its use over intermediate locked-down firewalls and SOCKS proxies, for example.

User Security

MQTT requires short user names and short passwords that do not provide enough entropy in the modern world. It has made these part of the protocol itself, so any change in policy, or security weakness, requires a new protocol version. AMQP uses SASL mechanisms, allowing organisations to choose the security that matters to them (e.g. Kerberos V5) without protocol change. It also supports a common security practice, the notion of proxy security servers, i.e. that the message queuing server (broker) is not the same as that providing the termination of the security layer(s). This allows organisations to use gatekeepers, nested firewalls, etc. AMQP's approach of authenticating the user before establishing a messaging connection allows for complete sand boxing of server resources.

Messages Matter, Too

To MQTT, a message is opaque. Notionally, this is a good thing; but it limits the infrastructure to nothing more than transmission from sender to receiver. In AMQP, they are not.

In practice, messages are not solely transmitted from a client to a receiver. They may be redirected or routed. They may pass through another pair of hands. There may be many consumers of those messages, interested in different subsets. MQTT can manage to support different 'topics', in a simple hierarchy, but that really isn't enough even in some simple cases.

For example, imagine a fleet of electric delivery trucks. Some days, you might want to find a problem with part of your fleet, but not affect ordinary operations by stealing their messages. So you attach the messaging equivalent of a multimeter, to listen for some subset. Perhaps by customer, by depot, by truck id, or may be later by truck model or date of manufacture. Those needs are orthogonal; only with perfect foresight could you have designed one hierarchy to capture all that.

AMQP separates the structure of a message, from its manner of delivery, with explicit and implicit meta-data which your infrastructure can use. Even better, its forwards-compatible so an older piece can still make use of newer messages. Some of this meta-data might need to mutate as a message passes through a network. Some of it must never change, as its used to calculate a signature (AMQP provides for cryptographically secure messaging needs).

Lastly, the MQTT topic is ‘global’ - it is a global namespace, equivalent to one queue or one node; in AMQP, their are as many queues as you wish to define.

Last-Value-Queues

MQTT has, with its ‘RETAIN’ command, the ability to support Last-Value-Queues (LVQs). These are useful when a consumer connects for the first time, and, rather than read a historic set of messages, just wants to get the latest state of play and then receive updates on it. AMQP does not support such a feature, although the protocol design easily allows for a vendor, or the entire standards body, to add one in a compatible way without breaking existing implementations. There is also an architectural argument that a LVQ should be implemented in the application infrastructure around the messaging, as it’s difficult for one message queue implementation to provide for the many scenarios in use, eg a Front-Office stock quote LVQ use case is quite different to a vehicle CANBUS telemetry one.

Reliable Messaging

Essentially, most users of messaging either care a message is sent and definitively received once, or they do not. Both protocols provide for ‘fire-and-forget, don’t try to hard’ messaging. AMQP provides fine-grained control over this, should it be required. This is useful when data delivery doesn’t have to be reliable, but order of delivery matters.

In practice, such use cases aren’t quite as common as them seem. A classic one is a stream of sensor data. On closer examination, though, the bandwidth for such a stream over a mobile network or serial line, say, is expensive, and that unreliability unpredictable. It’s rarely a simple ‘5%’ figure - often it’s ‘100% for 20 mins’ and ‘0% for 2 days’. Consequently, reliable messaging is far more useful.

Both protocols claim to provide reliable messaging, essentially using a series of acknowledgments to give ‘exactly-once’ receipt of a message. However, under analysis, this is not always the case with MQTT. MQTT assumes ‘general reliability’ of the parties involved. This is simply not the case in the real world. AMQP addresses these scenarios with ‘link recovery’, which allows fine-grained control, and will ensure eventual delivery under hostile conditions. MQTT also naively assumes that messages are always accepted by the server. In practice, this is not the case, and AMQP provides control to allow both a server and a client to reject and ‘return-to-sender / forward-to’ in the same way the postal service does. MQTT’s only option here to specify a ‘Will Message’, to be sent on a client’s behalf, if a connection dies. In essence, this acts a bit like the Royal Navy’s Letters of Last Resort, and, in the same way, is a nuclear option that provides no finesse.

Message Namespaces

MQTT's only 'namespace' is a hierarchal topic space, into which all messages go. The implementors' of it have developed some naming conventions for it. This is quite limiting. In AMQP, there are multiple such spaces ('nodes' or 'queues'), each of which can have many different ways of finding messages. The method of finding a message is at a consumer's choice, not necessarily by the server. As such, many consumers can share a queue, in which some always pull messages off, and others receive copies, each using the same or different expression to find messages. Such an expression could be a topic; or it can be anything else, and AMQP's use of message meta-data allows implementors to choose some original schemes for finding messages.

More than Just a Connection

In practice, there's more to life that just a connection. Some clients are ephemeral, connecting once and then disappearing into the light. Both MQTT and AMQP support those sort. Others are long-lived, and have state, such as which messages they think they've sent, which have bits missing and which they think they didn't (but did). AMQP provides for this using 'Containers'; 'MQTT' does in a small way, using a client-id, but this is marginally useful. And lastly, some clients are themselves as capable as a server, sending and receiving all at once to many queues. In AMQP, a client is the same as a server; all concepts are bidirectional, so it doesn't matter which behaviors one uses. In MQTT, the relationship is asymmetric, and a client can never be so powerful.

Some clients might be quite powerful, and capable of sending and receiving on multiple threads. AMQP supports this multiplexing, using a concept called 'sessions'. Such a set up might very easily require flow control quite different to that in the underlying TCP. MQTT does not support this, but AMQP does, ensuring that a memory-constrained device that needs reliable messaging is never swamped by more messages than it can hold onto before acknowledging them. Some clients can go further, and have different needs at the same time. AMQP provides for this using links.

It's Alive

Message queuing implementations can live a very long time, especially 'in the field'. Different parts of the infrastructure can come and go, be upgraded or replaced. MQTT can only provide for very basic needs here using DNS redirects. AMQP goes much further, for example allowing a server to redirect to a peer, at either the level of an entire connection or just for a particular queue. In the later case, that allows for fine-grained load-balancing of popular queues, or to provide for when some of a fleet of vehicles is sold-on.

Implementation

It is certainly easier to implement MQTT; it is a much smaller protocol. However, that is arguably mute in today's world. Open protocols result in open source libraries. The vast majority of users will simply choose the open source client library for their operating system or language. However, a simple protocol does not necessarily mean less operational size. Both AMQP and MQTT have been implemented in devices with less than 64Kb of RAM, so it would seem that any comparison here is moot.

Extensibility

AMQP has explicitly defined points of extensibility allowing vendor-specific and standards-agreed future extensions in a way compatible with, and usable by, existing implementations. MQTT requires a completely new protocol draft. AMQP's protocol is layered, allowing change in one part of the specification to be isolated from another.

Availability of Server Implementations

Today, both protocols have several implementations. MQTT, however, has several that are IBM backed and only one that seems open source and separate, Mosquito. It is not obvious if commercial support is available for this. AMQP has implementations available from StormMQ, VMWare and RedHat, with further product likely from other contributors such as Microsoft.

Conclusion

MQTT and AMQP are both message queuing protocols, suitable for use in hardware and software and on all major operating systems and platforms. MQTT is suited to its use case of simple clients talking to a server, but any infrastructure using it is exposed to serious security weaknesses and an inability to make best use of resources or to support additional use cases. AMQP is suited to these uses cases and many others, supports far better use of resources, far more pragmatic security and message reliability and has a future place as an ISO standard. Its origins as customer-orientated protocol, and its backing by big, competing names in IT bodes well for customers traditionally worried about support of open protocols and vendor lock-in.

Author

If you have any questions, comments, or would like to know more about StormMQ's Messaging-as-a-Service offering, please visit www.stormmq.com or contact:-

Raphael Cohn

Chief Architect

StormMQ

raphael.cohn@stormmq.com

+44 7590 675 756