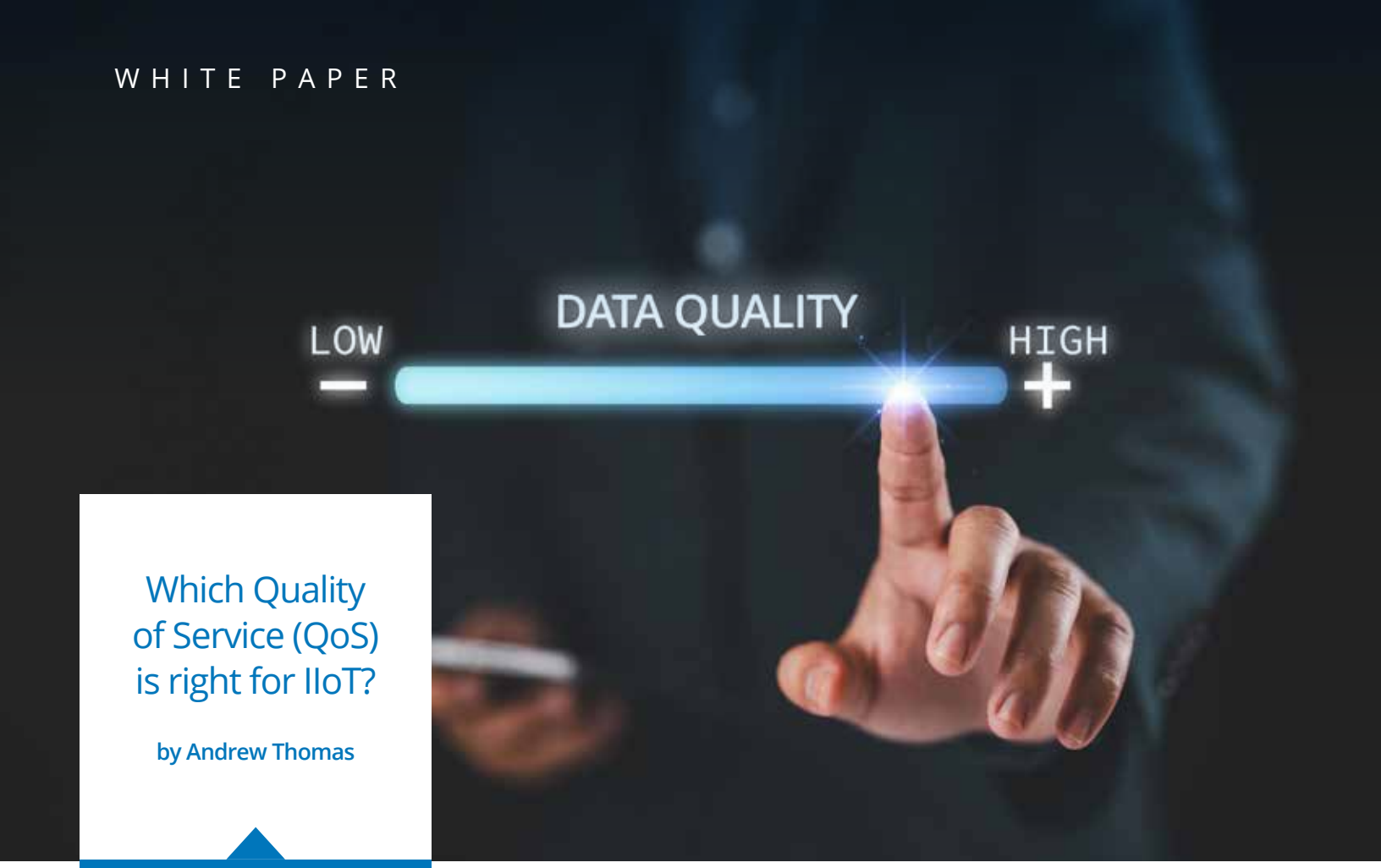
WHITE PAPER



Which Quality of Service (QoS) is right for IIoT?

by Andrew Thomas

Cogent DataHub™

The three Quality of Service levels (QoS) offered by MQTT may have been adequate for the original design goals, which was a one-hop connection for remote devices to a central location. But they do not adequately serve the needs of Industrial IoT. A higher standard is necessary for IIoT backbone and other applications: guaranteed consistency of data.

Key takeaways

1. Message loss at MQTT QoS level 0 is unacceptable for IIoT, and levels 1 and 2 can produce long queues that can lead to catastrophic failures when data point values change quickly.

2. The choice of QoS level drives difficult compromises regarding performance and message order.

3. QoS levels 1 and 2 don't propagate well past the MQTT broker. The QoS promise cannot necessarily be kept among multiple clients.

4. Consistency of data can and must be guaranteed by managing message queues for each point, preserving event order, and notifying clients of data quality changes.

Introduction

Quality of Service (QoS) is a general term to indicate the delivery contract from a sender to a receiver. In some applications QoS talks about delivery time, reliability, latency or throughput. In IIoT, QoS generally refers to the reliability of delivery.

MQTT, a popular IIoT protocol offers three QoS levels. However, none of these is adequate for a robust IIoT backbone protocol. Instead, we suggest a more stringent yet more flexible standard, something altogether different—guaranteed consistency of data.

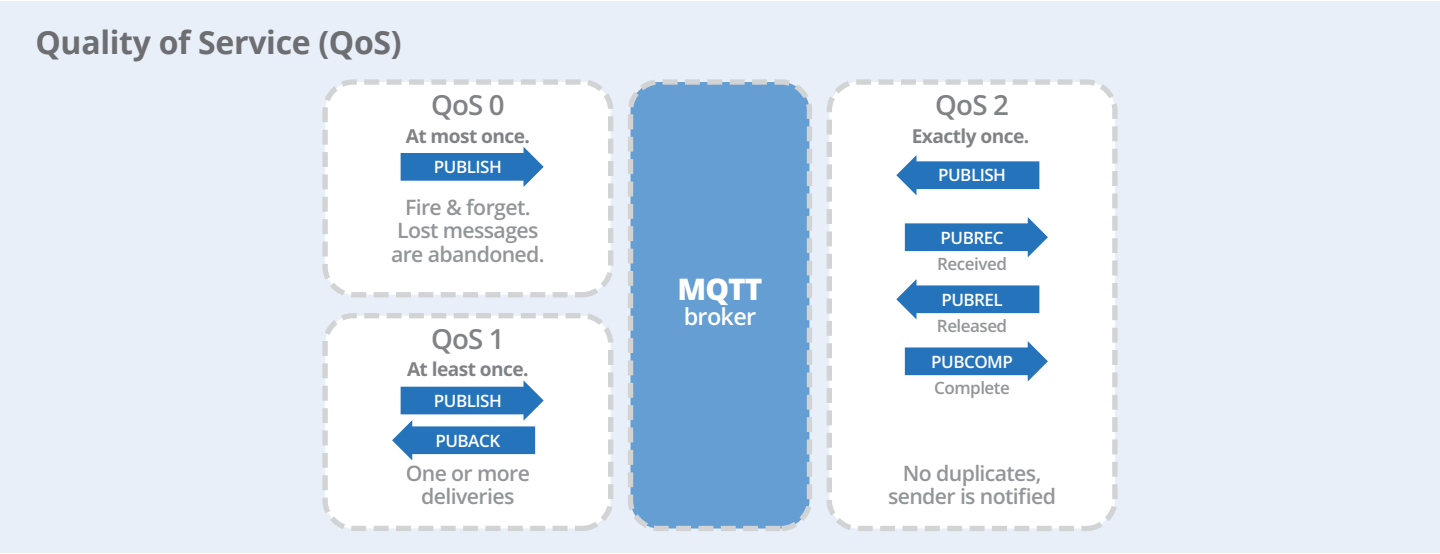
Let me explain. Here are the three Quality of Service levels that MQTT offers:

• **Level 0** – At most once. Every message will be sent out on a best-effort basis. If a message is lost in transit for whatever reason, it is abandoned—the receiver never receives it, and the sender does not know that it was lost.

• **Level 1** – At least once. Every message will be delivered to a receiver, though sometimes the same message will be delivered two or more times. The sender flags the message as a possible duplicate, placing the burden of distinguishing duplicates on the receiver. The sender is not certain whether the receiver received multiple copies of the message.

• **Level 2** – Exactly once. Every message will be delivered exactly once to the receiver, and the sender will be aware that it was received.

All three of these QoS levels lack something critical for most industrial systems, which I will discuss below. But let's look at each one individually first.



QoS level 0 is unreliable. It is fine to lose a frame of a video once in a while, but not fine to lose a control signal that safely shuts down a stamping machine. If the sender is transmitting data more quickly than the receiver can handle it, there will come a point where in-flight messages will fill the available queue positions. At that point the broker must do one of three things: delete an old message to create queue space, delete the new message, or refuse to accept the new message. Since MQTT brokers do not interpret message payloads, none of these can ensure that an important value is not lost. If the broker refuses the message, the sender now has a responsibility for further queueing, in the expectation that queue space in the broker will be available later. The sender is now faced with the identical queuing problem – what happens when its queue is full? The client might have more information to intelligently discard data, but not always. For example, if the broker is the sender, it has no option but to delete a message. Systems that use QoS 0 have to hope that queues do not fill.

QoS level 1 seems pretty reasonable at first glance. Message duplication is not a problem in most cases, and where there is an issue the duplicates can be identified by the receiver and eliminated, assuming the receiver maintains enough history to be able to identify them.

However, problems arise when the sender is transmitting data more quickly than the receiver can process it. Since there is a delivery guarantee at QoS 1, the sender must be able to queue an infinite number of packets waiting for an opportunity to deliver them. Since memory is finite, this means that queues must overflow to disk. This hugely reduces performance, making queue exhaustion even more likely. You might say that queues do not need to be infinite, just large. But then what happens when that large queue fills? Delete the oldest message or newest message, or just refuse the message completely? That is effectively QoS 0.

Longer queues mean longer latencies. For example, if I turn a light on and off rapidly three times, and the delivery latency is 5 seconds simply due to the queue volume, then it will take 30 seconds for the receiver to see that the light has settled into its final state. In the meantime the client will be acting on false information. In the case of a light, this may not matter much (unless it is a visual alarm), but in industrial systems timeliness matters. The problem becomes even more severe if the client is aggregating data from multiple sources. If some sources are delayed by seconds or minutes relative to other, then the client will be performing logic on data values that are not only inconsistent with reality but also with each other.

Ultimately, QoS 1 cannot be used where any client could produce data faster than the slowest leg of the communication path can handle. Beyond a certain data rate, the system will effectively "fall off a cliff" and become unusable. I've personally seen this exact thing happen in a municipal waste treatment facility. It wasn't pretty. The solution was to completely replace the communication mechanism.

QoS level 2 is similar to QoS 1, but more severe. QoS 2 is designed for transactional systems, where every message

matters, and duplication is equivalent to failure. For example, a system that manages invoices and payments would not want to record a payment twice or emit multiple invoices for a single sale. In that case, latency matters far less than guaranteed unique delivery.

Since QoS level 2 requires more communication to provide its guarantee, it requires more time to deliver each message. It will exhibit the same problems under load as QoS level 1, but at a lower data rate. That is, the maximum sustained data rate for QoS 2 will be lower than for QoS 1. The "cliff" just happens sooner.

Maximum message rates, pipelining and message order

QoS 1 and 2 both require acknowledgements as part of their data transmission. That means that every message must wait for an acknowledgement before the next message can be transmitted. QoS 1 requires one network packet from client to broker, and one packet from broker to client. QoS 2 requires double that. This synchronization across the network makes the overall message rate highly dependent on network latency. For example, if the client is in Greece and the broker is in the US, the ping time is about 125ms. That means that QoS 1 would require 125ms per message. QoS 2 would require 250ms. In the worst-case scenario where all messages are on a single topic, QoS 1 could send at most 8 messages per second. QoS 2 would cap out at 4 messages per second.

By comparison, QoS 0 requires no acknowledgement beyond what is inherent in TCP, which is highly optimized. That makes it possible to pipeline messages using QoS 0, writing one after another without delay. The maximum messages rate depends on network bandwidth, not latency.

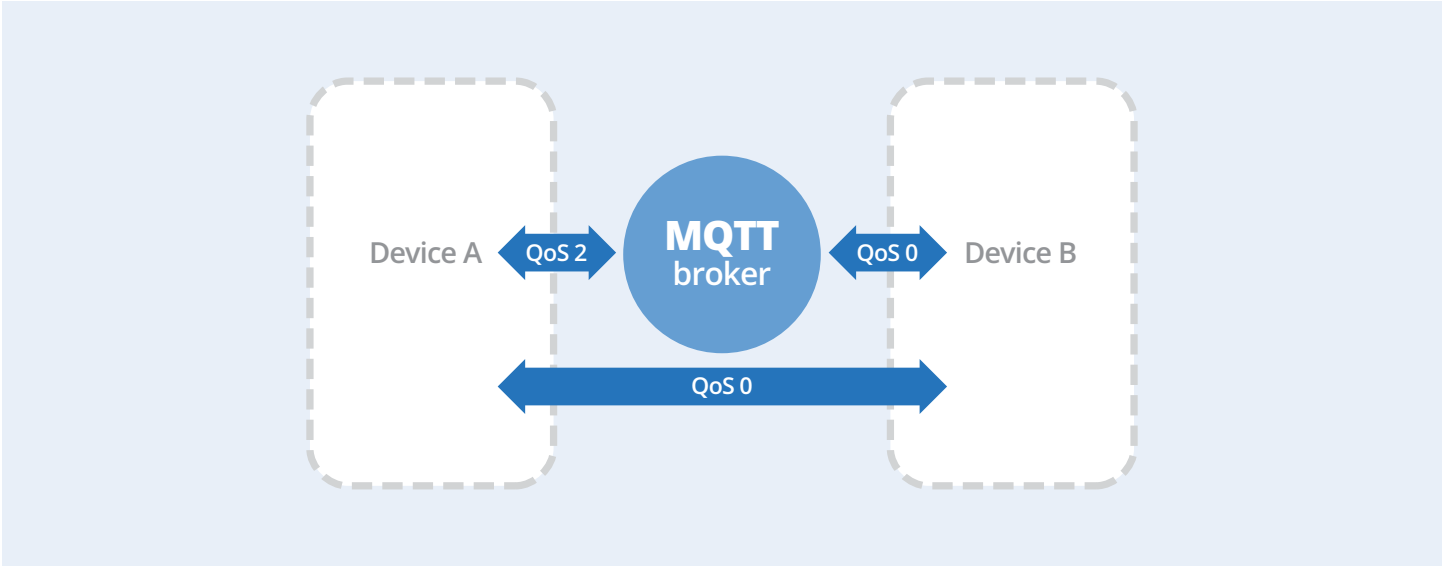
MQTT does not guarantee message delivery order for messages on different topics. QoS 1 and 2 will perform better if transmissions and acknowledgements for different topics are interleaved. In fact, without that they would be too slow to be useful outside a LAN. However, when transmissions are interleaved message order becomes unpredictable. MQTT does guarantee message order for multiple messages on the same topic. Consequently, there is a worst-case scenario for QoS 1 and 2 where all messages are sent to a small number of topics, making network latency the rate-limiting factor.

In effect, users must choose among performance, delivery promises and reliable message ordering. In most cases the user is unaware that the choice is even being made.

QoS Levels 1 and 2 Don't Propagate Well

All QoS levels, most importantly level 1 and level 2, suffer from another big flaw – they don't propagate reliably.

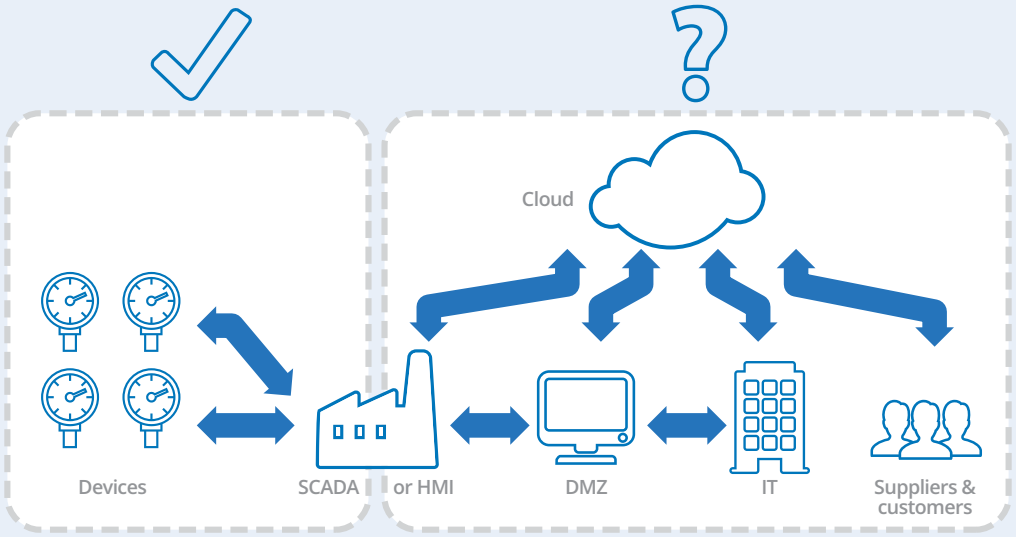
Consider a trivial system where two clients, A and B, are connected to a single broker. The goal is to ensure that B remains up to date with what A transmits. Suppose A sends its message with QoS 2. That ensures that the messages reach the broker. However, B needs data from many senders, so it subscribes using QoS 0 for speed. The net result is that B receives messages from the broker at QoS 0, even though A sent them at QoS 2. Any dropped message at QoS 0 would result in B being inconsistent with A. Obviously, B could subscribe to topics from A at QoS 2 to resolve this, but that implies that B knows more information about A than it should. A major goal of MQTT is to decouple senders and receivers. Obliging receivers to have intimate knowledge about senders violates that goal.



Since QoS 0 does not guarantee delivery, the message from A is not guaranteed to arrive at B. If that message would have updated a value in an HMI, for example, then that HMI will remain inconsistent with the source until a new value is transmitted. That is misleading in an HMI. It can be catastrophic in closed-loop control.

It gets even more complicated. As we have seen, QoS 1 and 2 can result in lengthy queues, which introduce delivery latency. One colloquial definition of "real time" is "a late answer is a wrong answer". The definition of "late" is application-dependent, but most control systems have one. In our trivial example, which QoS will reliably keep B up to date with A? The answer is: none of them.

In many networking scenarios, it is desirable to create a multi-server backbone, where brokers are connected to one another in a daisy-chain. Messages are passed between brokers, allowing clients in different networks to communicate with one another. This adds an extra opportunity for QoS mismatches, latency, ordering and performance issues to come into play.



Guaranteed Consistency

None of these QoS levels is really right for IIoT. We need something else, and that is **guaranteed consistency**. In a typical industrial system there are analog data points that move continuously, like flows, temperatures and levels. A client application would like to see as much detail as it can, but most critical is the **current** value of these points. If it misses a value that is already superseded by a new measurement, that is not generally a problem. However, the client cannot accept missing the most recent value for a point. For example, if I flick a light on and off 3 times, the client may not need to know how many times I did it, but it absolutely must know that the switch ended in the off position. The communication path needs to guarantee that the final "off" message gets through, even if some intermediate states are lost. This is the critical insight in IIoT. The client is mainly interested in the current state of the system, not in every transient state that led up to it.

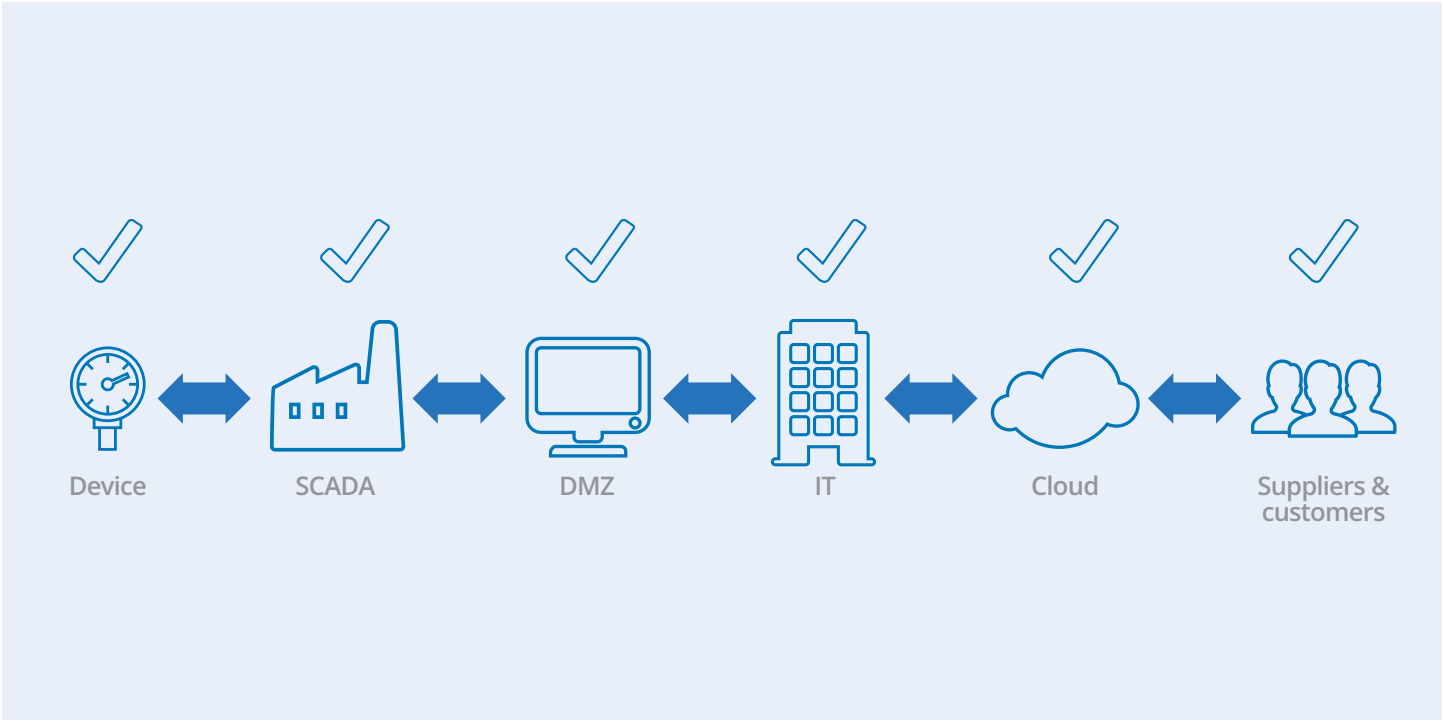
Guaranteed consistency for QoS is actually slightly more complex than that. There are really four critical aspects that are too often ignored:

1. **The server must know what it is managing.** MQTT brokers explicitly have no knowledge of the message contents. They do not know, for example, whether 3 sequential messages pertain to a single light switch, 3 different light switches, or something altogether different. They only know whether the messages are on the same topic. So long as the broker has no insight into the meaning of the messages, is cannot reliably know which messages constitute superseded values and can be safely discarded. Relying only on queue position or message age, typical heuristics for discarding stale messages, will fail.

2. Message queues must be managed for each data point and client. When communication is slow, old messages must be dropped from the queue in favor of new messages to avoid ever-lengthening latencies. This queuing must occur on a per-point, per-client basis. Only messages that are superseded for a specific point destined for a specific client can be dropped. If we drop messages blindly then we risk dropping the most recent message value for a point, as in the final switch status above. MQTT messages routinely contain values for multiple points. In that case, no message is safe to discard and we are back to the infinite queuing problem.

3. Event order must be preserved. When a new value for a point enters the queue, it goes to the back of the queue even if it supersedes a message near the front of the queue. If we don't do this, the client could see the light turn on before the switch is thrown. The relative order in which events occur in control systems is often critical for correct control and fault identification. Ultimately the client needs to maintain a consistent view of the data as that data changes.

4. The client must be notified when a value is no longer current. For the client to trust its data, it must know when data consistency is no longer being maintained. If a data source is disconnected for any reason, its data will no longer be updated in the client. The physical world will move on, and the client will not be informed. Although the data delivery mechanism cannot stop hardware from breaking, it can guarantee that the client knows that something is broken. The client must be informed, on a per-point basis, whether the point is currently active and valid or inaccessible and thus invalid. In the industrial world this is commonly done using data quality, a per-point indication of the trustworthiness of each data value.



For those instances where it is critical to see every change in a process (that is, where QoS 1 or 2 is required), that critical information should be handled as close as possible to the data source, whether it's a PLC or an embedded device. That is, time-critical and event-critical information should be processed at its source, not transmitted via the network to a remote system for processing where that transmission could introduce latency or drop intermediate values. This topic of edge processing deserves its own white paper.

For the IIoT, the beauty of guaranteed consistency for QoS is that it can respond to changes in network conditions without slowing down, backing up, or invalidating the client's view of the system state. It has a bounded queue size and is thus suitable for resilient embedded systems. This quality of service can propagate through any number of intermediate brokers and still maintain its guarantee, as well as notify the client when any link in the chain is broken.

Can't MQTT guarantee consistency?

Seeing that guaranteed consistency has these advantages, is there any way to achieve it within the MQTT specification? At first glance, certain MQTT features suggest it could guarantee consistency of data across multiple connections, but a closer look at each of these shows it's not really possible.

• **A retained messages flag** delivers the last message on a topic to any new subscriber, which could in theory help ensure data consistency. But the retained message may not provide the latest values for all the underlying data points, particularly when a single payload can carry values for multiple points. Furthermore, the flag has no effect on messages that are dropped due to full queues or lost QoS 0 messages.

• **The Last Will and Testament (LWT) message** gets published automatically if a client drops. It seems this could be used to notify other clients that its data is now stale or invalid. Unfortunately, although this single message can inform a receiver that a source has failed, the receiver must know which data points are associated with that source, and also that each data point has only a single source. Encoding this kind of source knowledge into a receiving client is impractical to implement and violates the intent of decoupling in MQTT.

• **Persistent sessions** in QoS 1 and 2 ensure that a disconnected client will receive queued messages when it reconnects. This should be able to prevent data loss during temporary network outages. However, the MQTT specification allows for these sessions to time out after a while. Any client disconnection that exceeds the timeout will not recover data for that period. No client can rely on a persistent session surviving a disconnection and so must be prepared for any session to start from a "clean" state.

What about Sparkplug?

Sparkplug would probably be the most logical candidate for building guaranteed consistency into MQTT. This specification adds payload definition, topic hierarchy definition, source knowledge and lifetime information on top of MQTT. However, it is built on top of MQTT, and the limitations of MQTT still apply.

One issue is that in Sparkplug, missed messages can only be mitigated by a disconnect/reconnect cycle, where a client sends a BIRTH message to broadcast its presence and all current values. This approach breaks down on even a modest-sized implementation, where a system restart or network failure recovery can result in "birth storms" when all clients reconnect at once. These messages can saturate other clients, causing them to disconnect and re-emit BIRTH messages, triggering an endless cycle that can only be corrected by shutting down some of the clients and then restarting them manually over time.

Sparkplug inherits other problems from MQTT. All Sparkplug messages are sent at QoS 0, for good technical reasons. Full queues can only be handled by reconnections. Message order is important and not guaranteed, forcing clients to manage message re-ordering. That is something that TCP/IP solved 50 years ago.

These limitations of the protocol have become more evident as use of MQTT in IIoT expands far beyond its original design goals. To be realistic, along with the strengths of MQTT we need to understand its drawbacks. It can still play a key role in Industrial IoT edge scenarios, despite the limitations of QoS. But we cannot recommend MQTT as a backbone protocol for IIoT implementations of any size or complexity. It is good at event-driven telemetry transport, but was not designed for systems that require guaranteed state consistency.

So there's the answer. For IIoT, you don't want QoS 0. And once you understand the limitations and failure modes of QoS 1 or 2, you probably cannot accept either of them for more than single-hop connections. Beyond that you really need something more—guaranteed consistency. Although Skkynet software and services do support all QoS levels for MQTT, they also provide guaranteed consistency via other protocols to meet the highest possible standards for IIoT backbone applications.

About Skkynet

Skkynet is a global leader in real-time software and services that allow companies to securely acquire, monitor, control, visualize, network and consolidate live process data in-plant or in the cloud. DataHub™, and DataHub™ for Azure, enable secure, real-time data connectivity for industrial automation, Industrial IoT, and Industrie 4.0. Visit **skkynet.com** for more about the company and **cogentdatahub.com** for more about Cogent DataHub.