COMP 6331 Assignment3

MQTT Analysis Report

1. MQTT handshake of different QoS

1.1 QoS = 0

QoS (Quality of Service) 0 means “at most once” that is the message is sent only once and the client and broker don’t take additional steps to acknowledge delivery [1]. When the sender send a message to the receiver with QoS level 0, the receiver won’t send back an acknowledgement message to the receiver [2]. So, if receiver doesn’t receive the message, the sender won’t resend the message and this message will be lost forever. I wiresharked a message sent from a local broker to a local MQTT client with QoS level 0.

A picture containing table

Description automatically generated

Figure 1. Screenshot of a published message with QoS 0

Table

Description automatically generated with medium confidence

Figure 2. Screenshot of captured traffic between broker and client

From figure1, I can know that QoS level is a flag in the MQTT message’s header. From the figure2, I can see that the client didn’t send an acknowledge message to broker for the figure1’s message. So in QoS 0, there’s not a handshake process between the sender and the receiver.

Because the sender sends messages only once, the receiver doesn’t have message duplication problem. The sender sends messages in order because a server MUST by default treat each topic as an "Ordered Topic" [3]. But the receiver may receive messages in a wrong order. Some messages may arrive at the receiver through longer routes and take longer time than other packets. So that some packets sent earlier may reach the recipient later than those sent more recently.

When the receiver and sender have a stable connection or the receiver doesn’t mind if a few messages are lost, then QoS 0 is a good choice. A classic use case for QoS 0 is connecting a test sensor or a front end application to an MQTT broker over a wired connection [2].

1.2 QoS = 1

QoS (Quality of Service) 1 means “at least once” that is the message is re-tried by the sender multiple times until acknowledgement is received [1]. The sender stores the message until it gets a PUBACK packet from the receiver that acknowledges receipt of the message. It is possible for a message to be sent or delivered multiple times [2].

Table

Description automatically generated

Figure 3. Screenshot of published messages with QoS 1

In figure3, I captured a TCP packet which carried 7 MQTT messages together. Each MQTT message has a message identifier number. The expanded message in the figure has the id ‘614’. Its header also contains the QoS flag indicating the level 1.

A picture containing calendar

Description automatically generated

Figure 4. Publish Ack for the published message 614

Graphical user interface, text, application, table

Description automatically generated

Figure 5. Screenshot of captured traffic between broker and client

After the client received the MQTT message whose id is 614, it sent back an acknowledge message shown in Figure4. This Ack message only contains the message identifier to confirm. And its header has a flag, message type, which indicates that this is a publish acknowledge message.

If the broker doesn’t receive this Ack message, the broker may assume that the published message is lost, and the broker will send the message again with a DUP (duplicate) flag. In this circumstance, the client can receive a single message more than once which can cause the message duplication problem. The sender may send messages in wrong order because some messages may be resent. Like the QoS 0, the receiver may receive messages in a wrong order because of the similar reasons.

QoS 1 is used when the receiver needs to receive every message and it can handle duplication problem and the receiver want to receive messages as fast as possible. This QoS level is most used in remote monitoring scenarios where the status of critical devices or systems is reported on a regular basis.

1.3 QoS = 2

QoS (Quality of Service) 2 means “exactly once” that is the sender and receiver engage in a two-level handshake to ensure only one copy of the message is received [1]. At least two request/response flows (a four-part handshake) between the sender and the receiver offer the guarantee. To coordinate message delivery, the sender and receiver use the packet number from the original PUBLISH message. [2].

I captured many MQTT packets and screenshot four detailed packets as below to show the handshake process.



Text

Description automatically generated with medium confidence

Figure 6. Screenshot of published messages with QoS 2



Text, table

Description automatically generated

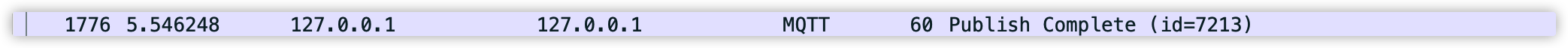
Figure 7. Screenshot of PUBREC for 7213



Text

Description automatically generated

Figure 8. Screenshot of PUBREL for 7213



Text

Description automatically generated

Figure 9. Screenshot of PUBCOMP for 7213

In figure 6, the broker sent a MQTT message with id 7213 to the client. In figure 7, the client received the message and replied a PUBREC (publish received) packet to the broker. In figure 8, the broker received the PUBREC and replied with a PUBREC (publish release) packet to the client. In figure 9, the client received the PUBREC and replied with a PUBCOMP (publish complete) packet to the broker.

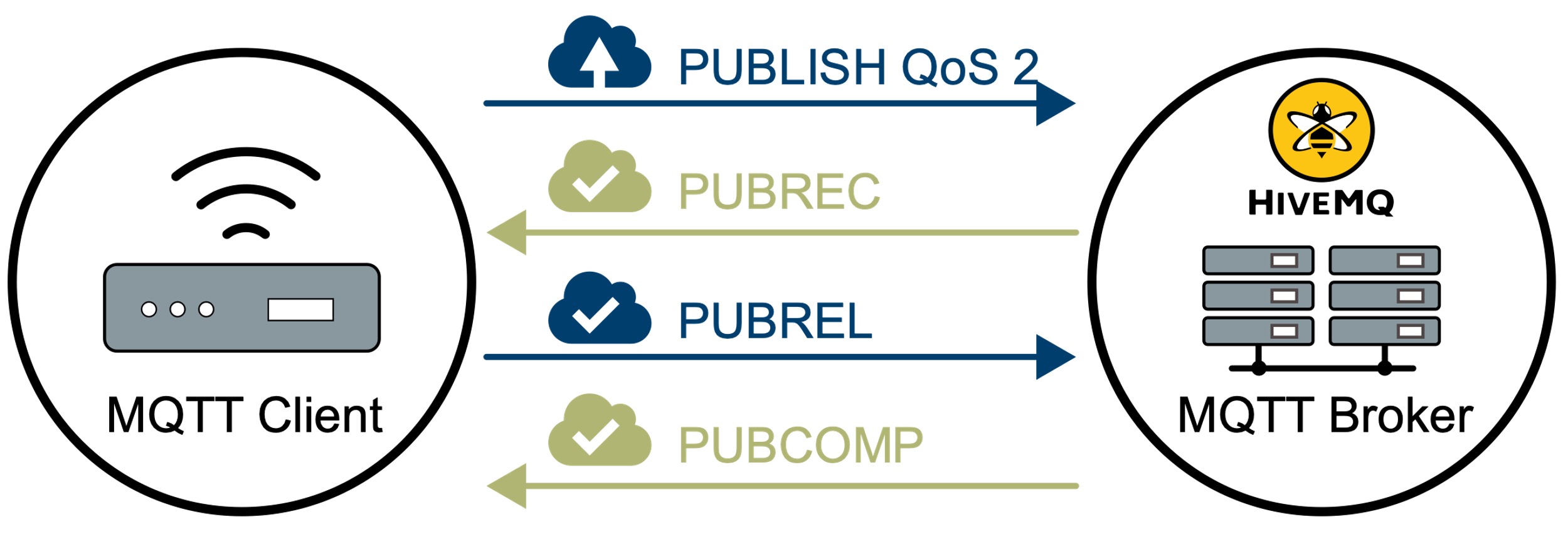


Figure 10. handshake process of QoS 2, source: https://www.hivemq.com/img/blog/qos-levels\_qos2.svg

The exactly once delivery is achieved using an addition PUBREL/PUBCOMP handshake so QoS 2 doesn’t have message duplication problem. Like QoS 0, the sender sends messages in order, but the receiver may not receive messages in order.

This QoS level is used when it’s critical for receivers to receive all messages exactly once and receivers can afford overhead this level caused. For example, a server wants to push notification to users’ mobile phones, users don’t want to receive the same messages multiple times and they can afford the overhead of QoS 2. This situation can be a good for QoS 2. A door sensor which sends door’s status to monitor should also use this level. Because if two ‘open’ messages are sent to the monitor, the monitor will get confused by the duplication and may behave abnormally.

2. Results and analysis of analyzer against partner’s broker

My partner and I conducted a 40-minute long experiment at 3am May 21st. My partner set up a local Mosquito broker on his laptop and exported the service to me through the tool ngrok (<https://dashboard.ngrok.com/get-started/setup>).

The tool ngrok can provide a reverse proxy server to do NAT penetration for us. My partner and I are all located in Canberra and the proxy server is located in AWS Sydney.

The traceroute between the proxy server and me is shown below:

Text

Description automatically generated

The traceroute between the server and my partner is quite similar to mime. So the network latency between us can be estimated as 20ms \*2 = 40ms. I executed and measured all parameters in a single run of the analyzer program.

2.1 QoS. = 0

2.1.1 delay = 0ms

Text

Description automatically generated

Figure 11. screenshot of 2.1.1’s result

The overall average rate of messages I actually received across the period was 53699. The rate of message loss was 0. The mean of message gap was 0.018 and the median is 0.

This result shows that all messages have arrived in order and the gap between messages is stable. The average gap is 0.0186ms which means the CPU processing time for each message.

2.1.2 delay = 1ms

Text

Description automatically generated

Figure 12. screenshot of 2.1.2’s result

The expected message rate is 1000 and the message gap is 1ms.

The message rate, loss rate and message gap are very close to the expected result.

2.1.3 delay = 2ms

Graphical user interface, text

Description automatically generated

Figure 13. screenshot of 2.1.3’s result

The expected message rate is 500 and the message gap is 2ms.

The message rate, loss rate and message gap are close to the expected result. The message rate has some difference to the ideal one maybe because the previous workload on delay 0ms and 1ms made my partner’s computer a little overheated and the CPU’s performance dropped. The $SYS topic didn’t send any useful system information to help analyze this situation.

2.1.4 delay = 10ms

Text

Description automatically generated

Figure 14. screenshot of 2.1.4’s result

The expected message rate is 100 and the message gap is 10ms.

The real results have a relatively large gap with the ideal data. The gap between real data and ideal data is around 36%. I think the reason is similar to 2.1.3’s. The heavy workload before caused the computer’s performance drop.

2.1.5 delay = 20ms

Text

Description automatically generated

Figure 15. screenshot of 2.1.5’s result

The expected message rate is 50 and the message gap is 31ms. The result didn’t match the expectation and I think the reason is similar to 2.1.4’s.

2.1.6 delay = 100ms

Text

Description automatically generated

Figure 16. screenshot of 2.1.6’s result

The expected message rate is 10 and the message gap is 100ms. The gap between the real result and the expected one is closer because large delay values can make the workload smaller. And these measurement time can help the CPU recover from the overheated status.

2.1.7 delay = 200ms

Text

Description automatically generated

Figure 17. screenshot of 2.1.7’s result

The expected message rate is 5 and the message gap is 200ms. The gap between the real result and the expected one is closer. Because the workload for 200ms delay is pretty low and the process time for publisher and receiver is much less.

2.2 QoS. = 1

2.2.1 delay = 0ms

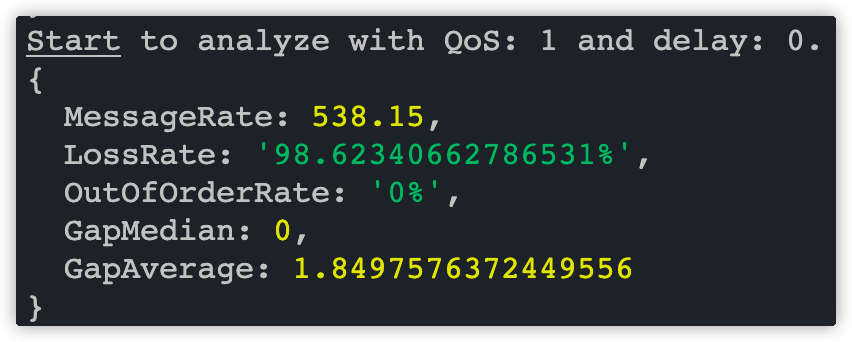


Figure 18. screenshot of 2.2.1’s result

The expected message rate and message gap are 2.1.1’s result. But the real message rate is much lower. The message gap is much higher than the expected. The loss rete is very high which means nearly half of messages were lost.

I looked up some $SYS topic messages and found some useful messages to explain this situation. QoS 1 should have been used to ensure that messages could be transferred and avoid message loss. But this mechanism can at least double the number of messages the publisher, broker, and receiver need to handle. There are many messages in $SYS topic like this line.

$SYS/broker/load/publish/dropped/1min, 96886.28

These messages indicate that the lost messages had been dropped because of inflight/queuing limits [4]. The default setting for queued messages is 1000 in mosquito. There were too many unsent messages that the queue in broker can’t handle and can only dropped them. This situation means QoS level 1 can’t always make sure the receiver can receive all messages. But this situation can be avoided or relieved by setting the **max\_inflight\_messages** and **max\_queued\_messages** options for brokers. The reason for a higher message gap is similar to 2.1.3’s mentioned before.

2.2.2 delay = 1ms

Graphical user interface, text

Description automatically generated

Figure 19. screenshot of 2.2.2’s result

The expected message rate is 1000. The expected gap is 1ms and the loss rate is much higher than the expected. The reason is similar to 2.2.1’s.

2.2.3 delay = 2ms

Graphical user interface, text

Description automatically generated

Figure 20. screenshot of 2.2.3’s result

The expected message rate is 500. The expected gap is 2ms. The reason is similar to 2.1.3’s.

2.2.4 delay = 10ms

Graphical user interface, text, application

Description automatically generated

Figure 21. screenshot of 2.2.4’s result

The expected message rate is 100. The expected gap is 10ms. The reason is similar to 2.1.4’s.

2.2.5 delay = 20ms

Graphical user interface, text

Description automatically generated

Figure 22. screenshot of 2.2.5’s result

The expected message rate is 50. The expected gap is 20ms. The reason is similar to 2.1.5’s.

2.2.6 delay = 100ms

Text

Description automatically generated

Figure 23. screenshot of 2.2.6’s result

The expected message rate is 10. The expected gap is 100ms. The reason is similar to 2.1.6’s.

2.2.7 delay = 200ms

Graphical user interface, text

Description automatically generated

Figure 24. screenshot of 2.2.7’s result

The expected message rate is 5. The expected gap is 200ms. The reason is similar to 2.1.7’s.

2.3 QoS. = 2

2.3.1 delay = 0ms

Graphical user interface, text, application

Description automatically generated

Figure 25. screenshot of 2.3.1’s result

This result is similar to 2.2.1’s but this is worse. Because every delivery of message need at least three more confirmation packets, this measurement period had a heavier workload than 2.2.1’s. More messages got dropped and the message delay was longer.

2.3.2 delay = 1ms

A screenshot of a computer

Description automatically generated with medium confidence

Figure 26. screenshot of 2.3.2’s result

The expected message rate is 1000. The expected gap is 1ms. This result is much worse than 2.2.2’s because of similar reason of 2.3.1.

2.3.3 delay = 2ms

Graphical user interface, text

Description automatically generated

Figure 27. screenshot of 2.3.3’s result

The expected message rate is 500. The expected gap is 2ms. This result is worse than 2.2.3’s because of similar reason of 2.3.1.

2.3.4 delay = 10ms

Graphical user interface, text

Description automatically generated

Figure 28. screenshot of 2.3.4’s result

The expected message rate is 100. The expected gap is 10ms. The reason is similar to 2.2.4’s.

2.3.5 delay = 20ms

Graphical user interface, text

Description automatically generated

Figure 29. screenshot of 2.3.5’s result

The expected message rate is 50. The expected gap is 20ms. The reason is similar to 2.2.5’s.

2.3.6 delay = 100ms

Graphical user interface, text

Description automatically generated

Figure 30. screenshot of 2.3.6’s result

The expected message rate is 10. The expected gap is 100ms. The reason is similar to 2.2.6’s.

2.3.7 delay = 200ms

Graphical user interface, text

Description automatically generated

Figure 31. screenshot of 2.3.7’s result

The expected message rate is 5. The expected gap is 200ms. The reason is similar to 2.2.7’s.

The constrained user ID and password can be configured for Mosquitto broker. First, I created a password file to store the password 33102021. Then I set the password file path in the mosquitto.conf and finally started the broker with the configure file. The configured authentication method was successfully applied. When my partner tried to connect with my broker firstly without specified username and password, the connection was rejected. For better security concern, the broker can also be configured with some plugins or even MQTTS (MQTT’s TSL version).

3. broader end-to-end network environment

In real world IoT scenario, the network structure is usually much more complicated than it in this assignment. In this chapter, I’m going to discuss in a situation with millions of sensors publishing to thousands of subscribers.

In the worst situation, these millions of sensors publish messages at the same time, there will be millions of messages per second sent to the broker. This situation’s workload is much heavier than 0ms delay in this assignment, where throughput was around 50k messages per second. The broker is the performance bottleneck because it’s responsible for receiving all these messages and then sent them to thousands of subscribers. During this process, the broker must use CPU resource to parse and construct network packets. But this process’s complexity is O(1) and each message’s CPU processing time is around 0.02ms according to 2.1.1’s analysis. So the CPU time for processing all millions of messages is around 20 seconds. The machine may need multiple CPUs to finish all tasks in parallel within one second. In terms of memory consuming, millions of messages received for the broker have to be stored in memory (in a queue).

In 2.1.1, the topic $SYS reported that the broker stored some messages as below:

$SYS/broker/store/messages/count, 39

$SYS/broker/store/messages/bytes, 331

Each message’s length is around 331 / 39 ≈ 10 bytes. The total length for one million messages is 1M \* 10 = 10M bytes. With the above preliminary calculations, I think that memory resources will not be the bottleneck of the broker.

For the network, if one message’s length is 100 bytes, then 1M messages’ total length is 100M. So the least network throughput for the broker is around 800Mbps. If every message received need to be sent to thousands of subscribers, then the network throughput need to times 1000 more. If publishers and subscribers use QoS 1 or 2, the throughput needs to be doubled or even more.

In summary, this application scenario is the most challenging for the network and more challenging for the CPU, but not much for memory resources. Applying distributed brokers can be a solution for this situation.

QoS 0 has the least challenge for brokers and QoS 2 has the largest challenge for brokers. According to the measurement in chapter 2, if sensors need to publish messages very frequently (hundreds of times per second), QoS 0 is the best choice both for sensors and brokers. Because this scenario can cause heavy workload for brokers and QoS 1 or QoS 2 can’t relieve the loss rate. For example, when the message delay is from 0ms to 20ms, QoS 1 and QoS 2 had much higher loss and latency than QoS 0 in chapter 2. QoS1 and 2 are the recommended choice only if the sensor does not transmit at a high frequency and reliability is required.

In real world situation, it is rare for all sensors to send messages at the same time and the workload for brokers can vary from time to time. The broker's peak workload does not last long. But in the assignment’s experiment, the workload is consistent for the same QoS level and delay value. The challenge for CPU, memory and network is similar for both situations.

References:

1. <https://en.wikipedia.org/wiki/MQTT>
2. <https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/>
3. <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718105>
4. <https://mosquitto.org/man/mosquitto-8.html>