COMP3310 Assignment 3: Testing MQTT

Hugo Kat (u7286091)

May 21, 2024

I discussed how to plot the experimental data with Jarrod Rose (u6899393). All other work is my own.

Question 1: Program Overview

We implemented an MQTT testing framework which will conduct all 180 experiments as a single program. This was accomplished by running the Analyser and Publishers on separate threads. We conducted the 'normal' assignment experiments using a local Mosquitto broker on an Ubuntu Linux Virtual Machine. Further details regarding program usage and requirements can be found in the `README.md`.

The Analyser collects all the required statistics for each `instancecount/qos/delay` combination. The Analyser also collects the following `$SYS/#` measurements:

- `$SYS/broker/clients/connected`
- `$SYS/broker/heap/current`
- `$SYS/broker/heap/maximum`
- `$SYS/broker/publish/messages/received`
- `$SYS/broker/publish/messages/sent`
- `$SYS/broker/publish/messages/dropped`

Question 2: Analysing Results

Question a.

There are three MQTT quality-of-service (QoS) levels: 0, 1, and 2. Each QoS level provides a different level of reliability with corresponding overhead [1].

- **Level 0**: Sends messages using a 'fire and forget' principal, meaning messages will be sent **at most once** and no acknowledgement is required. Level 0 provides the fastest delivery since it has minimal overhead, but does not guarantee message delivery. Level 0 should be used when message loss is acceptable.
- **Level 1**: Ensures that messages will be delivered **at least once**. To accomplish this, the sender stores a copy of the message until it receives acknowledgement from the receiver. The sender will resend the message if it does not receive acknowledgement in a reasonable time frame. Level 1 should be used when message loss is critical and message duplication is acceptable.
- **Level 2**: Ensures that messages are delivered **exactly once**. This is accomplished using a four-part handshake between the sender and receiver. Level 2 should be used when message loss is critical and message duplication is unacceptable.

We used the following Mosquitto command to subscribe to the `$SYS/broker/clients/connected` at the desired QoS level:
`mosquitto_sub -h test.mosquitto.org -p 1883 -t '$SYS/broker/clients/connected' -q <qos>`
Figures 1, 2, and 3 show the Wireshark log after subscribing to the topic, receiving **four** messages, and closing the connection for each QoS level respectively. Figure 1 demonstrates that no acknowledgement occurs when a message is published at QoS level 0. Figure 2 demonstrates that a two-way handshake occurs every time a message is published at QoS level 1 and Figure 3 demonstrates that a four-way handshake occurs every time a message is published at QoS level 2.

Figure 1: Wireshark log using QoS of 0

mqtt

No.	Time	Source	Destination	Protocol	Length	Info
14	1.231692598	10.0.2.15	91.121.93.94	MQTT	68	Connect Command
16	1.546707855	91.121.93.94	10.0.2.15	MQTT	60	Connect Ack
18	1.546824556	10.0.2.15	91.121.93.94	MQTT	90	Subscribe Request (id=1) [$SYS/broker/clients/connected]
20	1.862794703	91.121.93.94	10.0.2.15	MQTT	96	Subscribe Ack (id=1), Publish Message [$SYS/broker/clients/connected]
49	10.998976135	91.121.93.94	10.0.2.15	MQTT	91	Publish Message [$SYS/broker/clients/connected]
59	21.475175916	91.121.93.94	10.0.2.15	MQTT	91	Publish Message [$SYS/broker/clients/connected]
76	31.945019571	91.121.93.94	10.0.2.15	MQTT	91	Publish Message [$SYS/broker/clients/connected]
78	32.735452051	10.0.2.15	91.121.93.94	MQTT	56	Disconnect Req

Figure 2: Wireshark log using QoS of 1

mqtt

No.	Time	Source	Destination	Protocol	Length	Info
24	4.882335530	10.0.2.15	91.121.93.94	MQTT	68	Connect Command
26	5.197327620	91.121.93.94	10.0.2.15	MQTT	60	Connect Ack
28	5.197404228	10.0.2.15	91.121.93.94	MQTT	90	Subscribe Request (id=1) [$SYS/broker/clients/connected]
30	5.513060342	91.121.93.94	10.0.2.15	MQTT	98	Subscribe Ack (id=1), Publish Message (id=1) [$SYS/broker/clients/connected]
31	5.513134213	10.0.2.15	91.121.93.94	MQTT	58	Publish Ack (id=1)
56	13.830552954	91.121.93.94	10.0.2.15	MQTT	93	Publish Message (id=2) [$SYS/broker/clients/connected]
57	13.830624222	10.0.2.15	91.121.93.94	MQTT	58	Publish Ack (id=2)
73	24.305630682	91.121.93.94	10.0.2.15	MQTT	93	Publish Message (id=3) [$SYS/broker/clients/connected]
74	24.305713915	10.0.2.15	91.121.93.94	MQTT	58	Publish Ack (id=3)
85	34.776675277	91.121.93.94	10.0.2.15	MQTT	93	Publish Message (id=4) [$SYS/broker/clients/connected]
86	34.776827484	10.0.2.15	91.121.93.94	MQTT	58	Publish Ack (id=4)
92	35.602881230	10.0.2.15	91.121.93.94	MQTT	56	Disconnect Req

Figure 3: Wireshark log using QoS of 2

mqtt

No.	Time	Source	Destination	Protocol	Length	Info
58	0.583326453	10.0.2.15	91.121.93.94	MQTT	68	Connect Command
69	0.901116721	91.121.93.94	10.0.2.15	MQTT	60	Connect Ack
71	0.901205907	10.0.2.15	91.121.93.94	MQTT	90	Subscribe Request (id=1) [$SYS/broker/clients/connected]
93	1.215444957	91.121.93.94	10.0.2.15	MQTT	98	Subscribe Ack (id=1), Publish Message (id=1) [$SYS/broker/clients/connected]
94	1.215528196	10.0.2.15	91.121.93.94	MQTT	58	Publish Received (id=1)
96	1.529845886	91.121.93.94	10.0.2.15	MQTT	60	Publish Release (id=1)
97	1.529906346	10.0.2.15	91.121.93.94	MQTT	58	Publish Complete (id=1)
101	6.114802876	91.121.93.94	10.0.2.15	MQTT	93	Publish Message (id=2) [$SYS/broker/clients/connected]
102	6.114883884	10.0.2.15	91.121.93.94	MQTT	58	Publish Received (id=2)
104	6.433138328	91.121.93.94	10.0.2.15	MQTT	60	Publish Release (id=2)
105	6.433214969	10.0.2.15	91.121.93.94	MQTT	58	Publish Complete (id=2)
122	16.590311217	91.121.93.94	10.0.2.15	MQTT	93	Publish Message (id=3) [$SYS/broker/clients/connected]
123	16.590371667	10.0.2.15	91.121.93.94	MQTT	58	Publish Received (id=3)
125	16.908580291	91.121.93.94	10.0.2.15	MQTT	60	Publish Release (id=3)
126	16.908650574	10.0.2.15	91.121.93.94	MQTT	58	Publish Complete (id=3)
139	27.061760293	91.121.93.94	10.0.2.15	MQTT	93	Publish Message (id=4) [$SYS/broker/clients/connected]
140	27.061829383	10.0.2.15	91.121.93.94	MQTT	58	Publish Received (id=4)
142	27.376439026	91.121.93.94	10.0.2.15	MQTT	60	Publish Release (id=4)
143	27.376509525	10.0.2.15	91.121.93.94	MQTT	58	Publish Complete (id=4)
145	29.841101606	10.0.2.15	91.121.93.94	MQTT	56	Disconnect Req

Question b.

Figure 4 shows the message rate achieved using the different `analyser_qos`, `instancecount`, `publisher_qos`, and `delay` combinations. Unsurprisingly, the message rate decreased as the `delay` between messages increased, excluding the outlier point in the `instancecount=4` sub-figure. We expected the message rate to decrease as the `instancecount` increased since it induces a greater load on the broker; however, it did not have a drastic impact on the message rate. Conversly, the Analyser recorded some of the lowest messages rates with `instancecount=1`

Theoretically, the maximum message rate would be achieved using `analyser_qos = 0` and `publisher_qos = 0` since it has the lowest networking overhead. However, if enough messages are lost during transmission, the more reliable QoS levels could produce greater message rates. Figure 5 demonstrates that almost no message loss occurred during any of the experiments. This is expected since the experiments were conducted on a local Mosquitto broker.

It is important to note that the subscription-level QoS is the maximum level of QoS that can be received by the subscriber. If a publisher publishes at a lower level of QoS, the lower level will be used to send to the subscriber [2]. Therefore, `analyser_qos` is often determined by `publisher_qos`.

With no message loss, we would expect the lower QoS levels the produce the highest message rates. This occurred during experiments with `instancecount <= 3`, as low QoS combinations achieved the highest message rates. However, during experiments with `instancecount > 5`, the higher QoS combinations achieved the highest messages rates. The message rates of higher QoS combinations generally increased as `instancecount` increased while the message rate of lower QoS combinations decreased. We considered that the increased network traffic caused by the greater `instancecount` may have caused greater message loss in lower QoS combinations, reducing their message rate. However, since there was almost no message loss this explanation is not feasible.

The median inter-message gap followed a linear trend with the `delay`. There was minimal deviation between between experiments - the median inter-message gap was usually 1ms longer than the delay. We expected the higher QoS combinations to have a noticeably higher inter-message gap due to the additional overheads of their message transmission. However, these issues were likely minimised running on a local broker.

We did not observe any out-of-order messages. Once again, this is not surprising since the experiments were conducted on a local broker.

Question c.

The number of connected clients was six across all experiments. This is expected since we had one Analyser and five Publishers connected to the broker throughout every experiment.

The broker did not drop any published messages. This is likely a consequence of using a local broker.

Figure 7 demonstrates that the broker heap usage (size) increased with the `instancecount`. This is expected since more publishers are sending messages to the broker. Unfortunately, several of the experiments failed to receive any data regarding the heap size of the broker. From our limited samples, lower `qos` combinations used less heap memory than higher `qos` combinations. This is expected since the broker must store messages until they are delivered successfully with higher `qos` levels. Surprisingly, the heap size of the broker increased as the `delay` increased. This is unexpected since the broker receives less messages when `delay` increases as previously explained.

Figure 8 shows that the number of publisher messages received decreases as the `delay` increases and increases as the `instancecount` increases, mirroring the results in Figure 4. The number of publisher messages sent follows as similar trend as shown in Figure 9. These results also agree with our analysis of Figure 7.

Question 3: Broader Network Environment

Question a.

Designing an effective MQTT network to support high volumes of messages is challenging since a single weak link has the potential to bottleneck the entire network. There are three core components that should be considered.

- The capacity of the publishers and their surrounding network
- The capacity of the broker and its surrounding network
- The capacity of the subscribers and their surrounding network

The rate at which messages can be published is limited by the CPU and memory utilisation of the publishers. Delays will occur if the CPU cannot handle the required message rates. Message data will be stored in memory which could cause buffer-overflow or disk usage if they are not handled quickly enough. These issues might cause a publisher to fail publishing messages or crash entirely.

The subscribers are limited by their CPU and memory utilisation in a similar fashion to the publishers. If the subscribes cannot process the messages quickly enough it could led to buffer-overflow. It is also important to note that subscribers will usually do more than simply receiving the messages. Typically, they will process and analyse the incoming messages. These issues might cause a subscriber to drop incoming messages.

The broker is the most obvious bottleneck in the network, since everything else depends on it. The broker must have sufficient CPU and memory resources to process all incoming messages before directing them to the appropriate subscribers. If the broker persists messages, its disk space could be depleted when handling large volumes of messages. These issues might cause the broker to drop incoming messages.

The network infrastructure surrounding the publishers, subscribers, and broker also limit the message rate. The routers in the network must need to handle and redirect the high-volume of network traffic. If they cannot support the message rate, packets could be lost. Unreliable high-latency or ow-bandwidth LAN technologies could also bottleneck the message rate or cause packet loss. For example, a poor satellite connection likely could not support a high volume of messages.

Question b.

Using QoS 1 or 2 introduces more network traffic since additional packets must be sent to ensure reliable delivery. If these are used carelessly, the network traffic will be increased, which may reduce the performance of the MQTT network. Consequently, QoS 1 and 2 should only be used important messages when it is absolutely necessary. This is especially true for QoS 2. QoS 0 should achieve higher messages rates whilst putting less strain on the network.

Bonus Questions

Question 1.

Our interpretation of this question is based on Markus's answer to this Ed post.

The Analyser publishes to the request topics: `request/instancecount`, `request/qos`, and `request/delay` on a per-experiment basis. That is, at the start of each experiment, the Analyser will publish the parameters of the next experiment to the request topics. The Publishers must know the `instancecount`, `qos`, and `delay` before they can start the experiment, so the Publishers will block until they receive these values. Consequently, the Publishers will not immediately start after a single request topic is updated.

The Analyser always publishes to the request topics with QoS 2 and the Publishers always subscribe to the request topics with QoS 2. This is done to guarantee that every Publisher receives the correct experiment parameters **exactly once**. The Analyser and Publishers still connect to the counter topics (i.e. `counter/<instance>/<qos>/<delay>`) using the Analyser QoS and Publisher QoS levels of the current experiment.

Question 2.

We ran the MQQT testing framework on the local Mosquitto broker with 10 publishers with `qos=0` and `delay=0ms`. These parameters were chosen to stress the broker as much as possible. However, the performance of the program remained stable as shown by Figure 10. This is unsurprisingly since the MQTT protocol was designed to potentially support thousands of subscribers and publishers. Interestingly, the average message rate of each instance increased slightly with `instancecount`. Similar to what was described in *Question 2.c*, the average heap size of the broker increased in the `instancecount` as shown by Figure 11.

Question 3.

We ran the program on the the `broker.hivemq.com` broker with `delay=0`. Unfortunately, the broker did not send any of the the `$SYS/#` measurements to the Analyser. Additionally, broker sent the counter value in an unexpected format, which our program was not prepared to handle. Due to this, the recorded median inter-message gap and out-of-order messages are invalid. However, the Analyser was able to record the message rate and loss rate as shown in Figures 12 and 13 respectively. Excluding the `analyser_qos = 0` and `instance_count = 1` outlier, the program achieved significantly slower messages than on the local broker. The connection was also significantly less stable than the local broker, as indicated by the high loss rates. These results are expected since the messages actually had to travel through the network to reach the broker.

Appendix

Figure 4: Message rate vs delay for different QoS and instancecounts on local Mosquitto broker. **Note**: Only y-values greater than zero are plotted

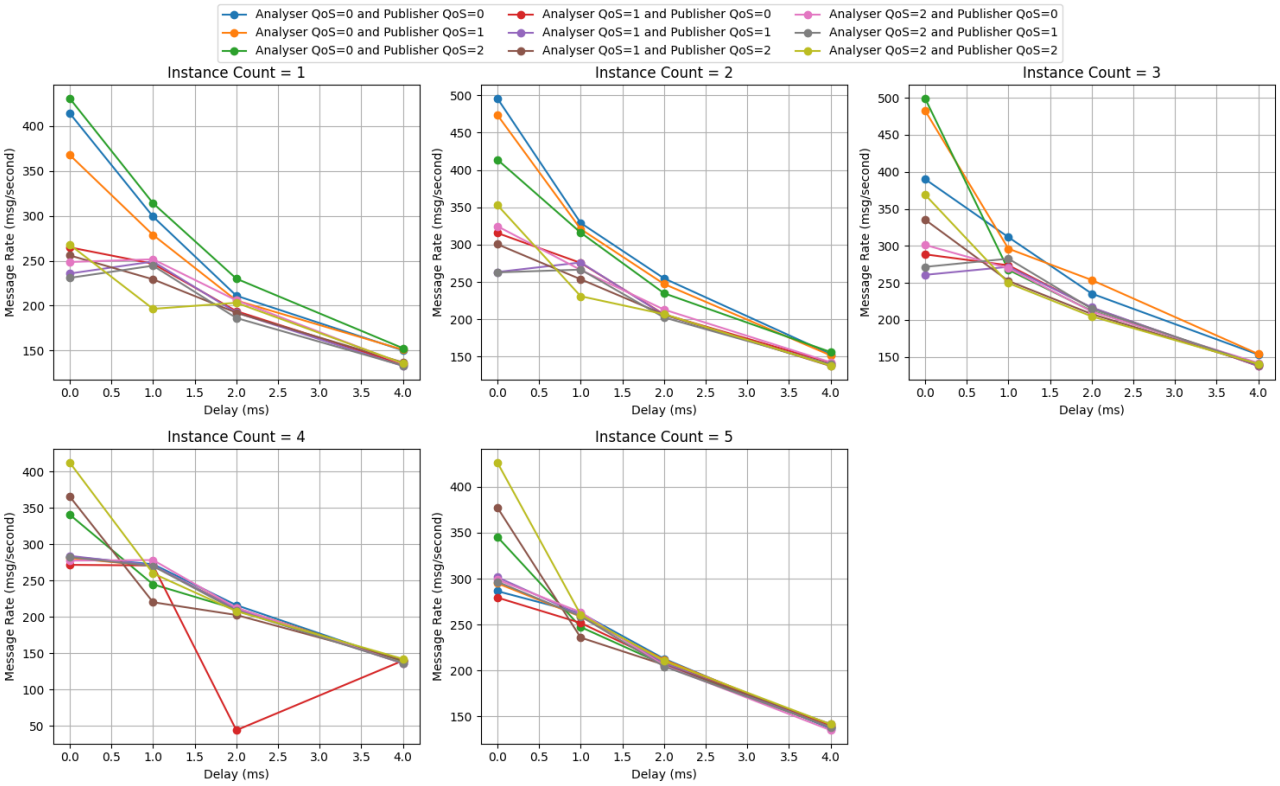


Figure 5: Loss rate vs delay for different QoS and instancecounts on local Mosquitto broker.

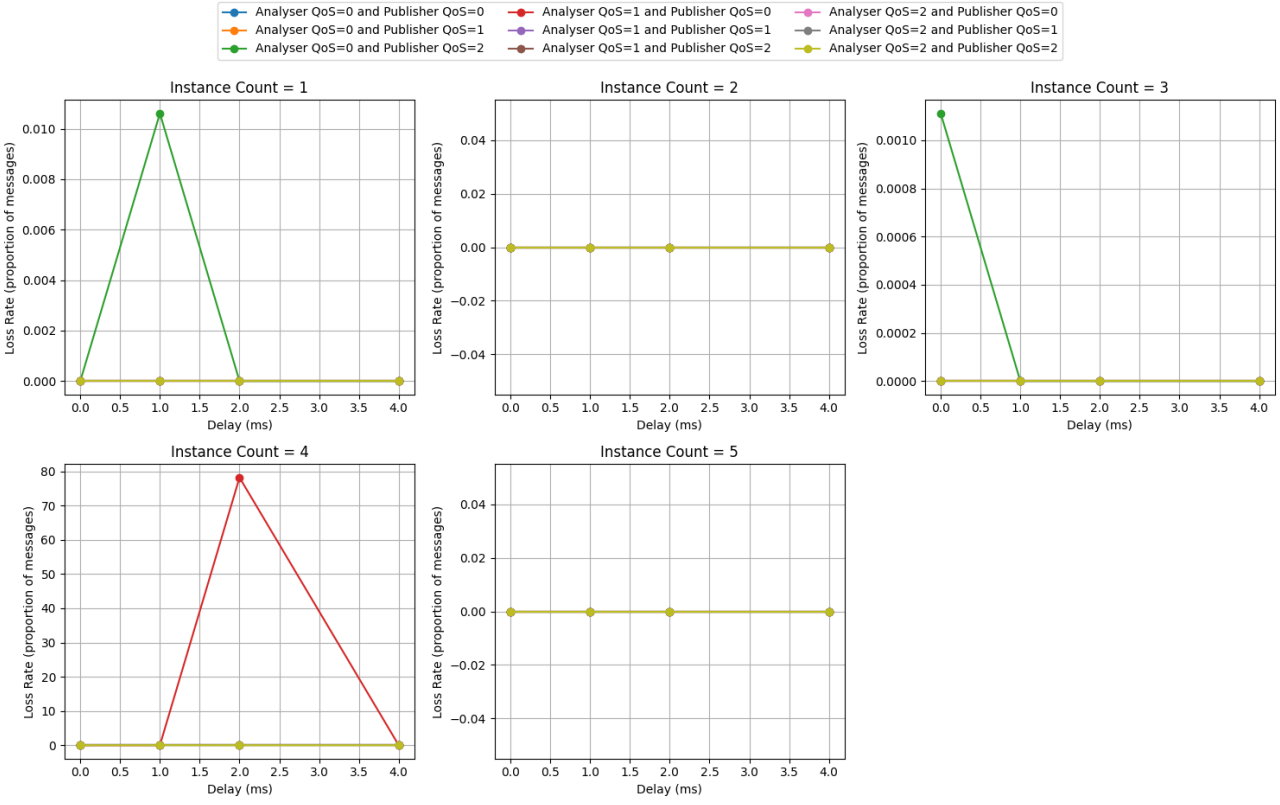


Figure 6: Median inter-message gap vs delay for different QoS and instancecounts on local Mosquitto broker. **Note**: Only y-values greater than zero are plotted

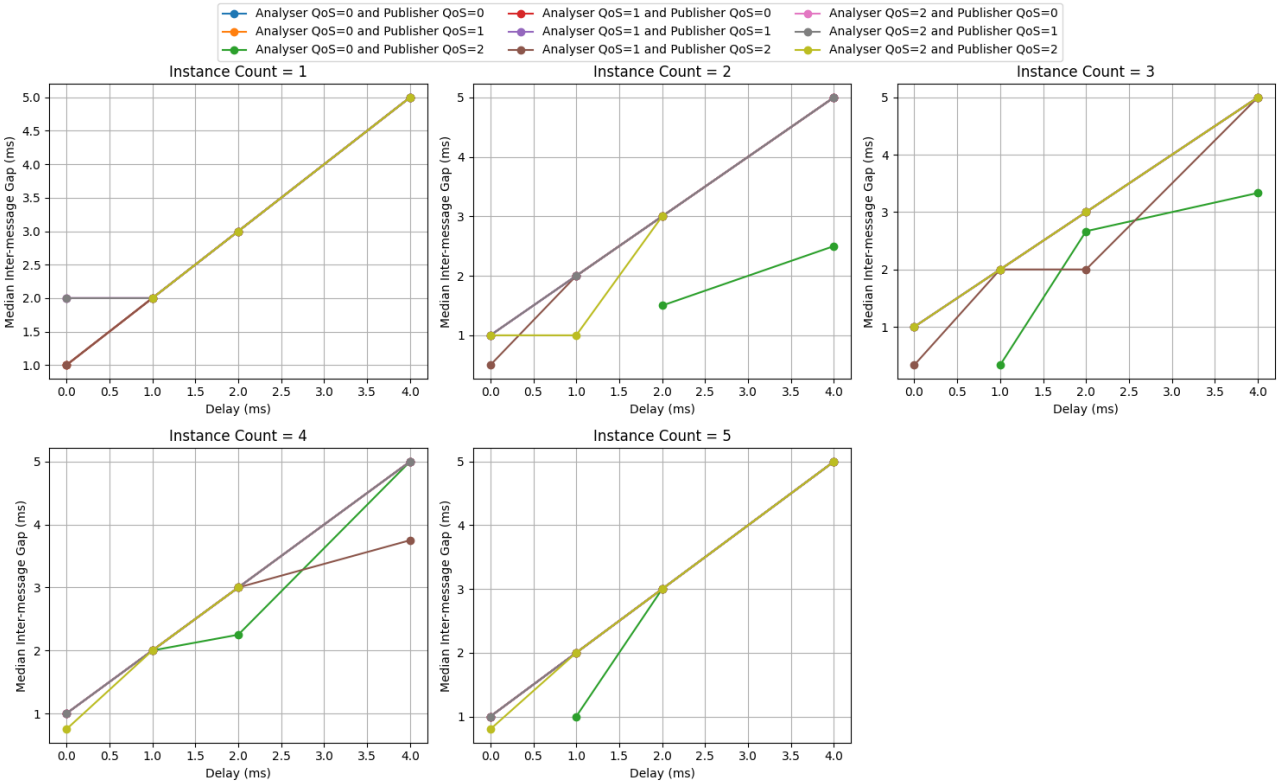


Figure 7: Average heap size vs delay for different QoS and instancecounts on local Mosquitto broker. **Note**: Only y-values greater than zero are plotted

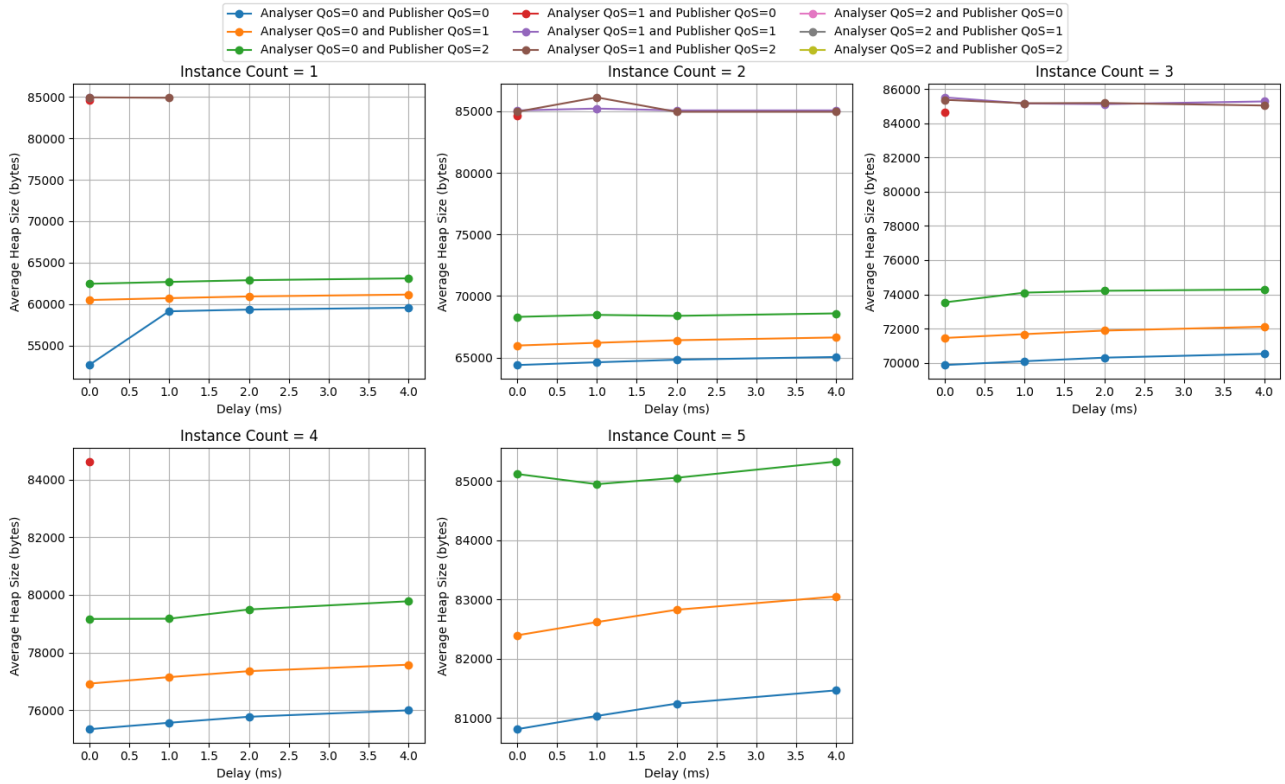


Figure 8: Number of publisher messages received vs delay for different QoS and instance-counts on local Mosquitto broker. **Note**: Only y-values greater than zero are plotted

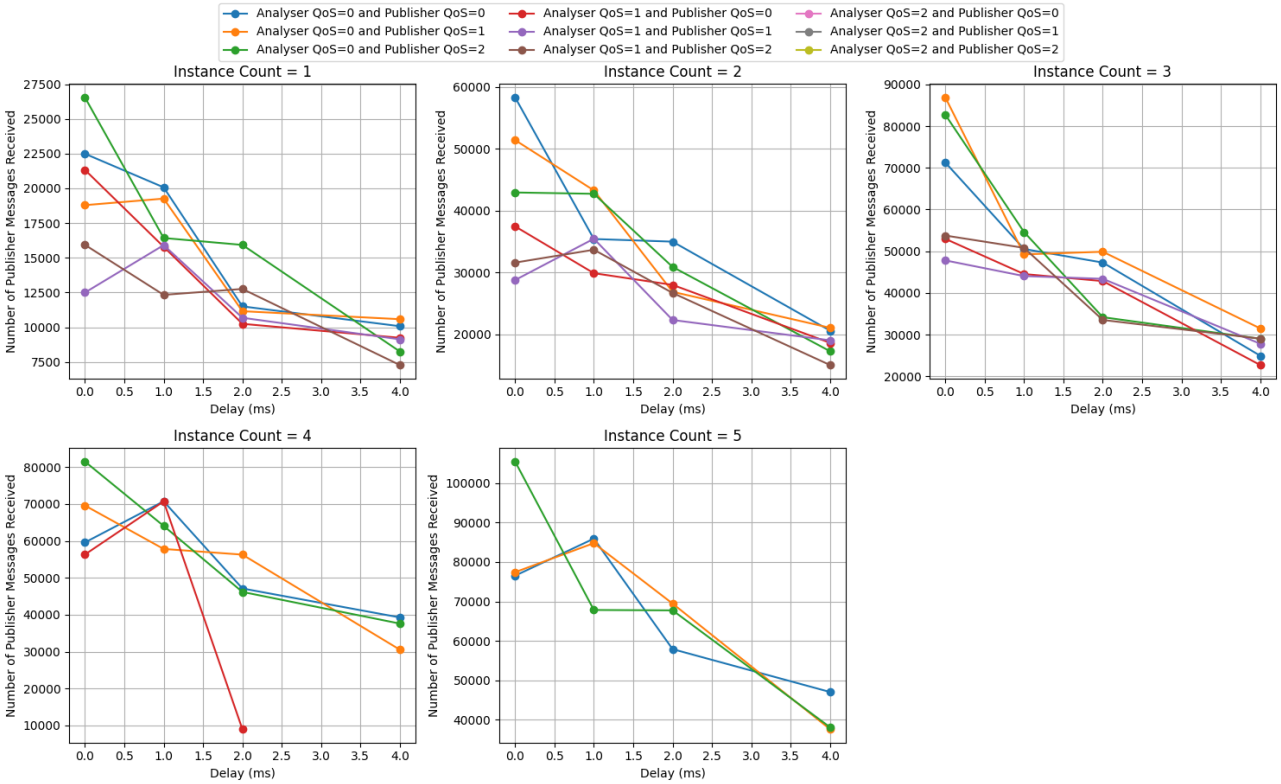


Figure 9: Number of publisher messages sent vs delay for different QoS and instancecounts on local Mosquitto broker. **Note**: Only y-values greater than zero are plotted

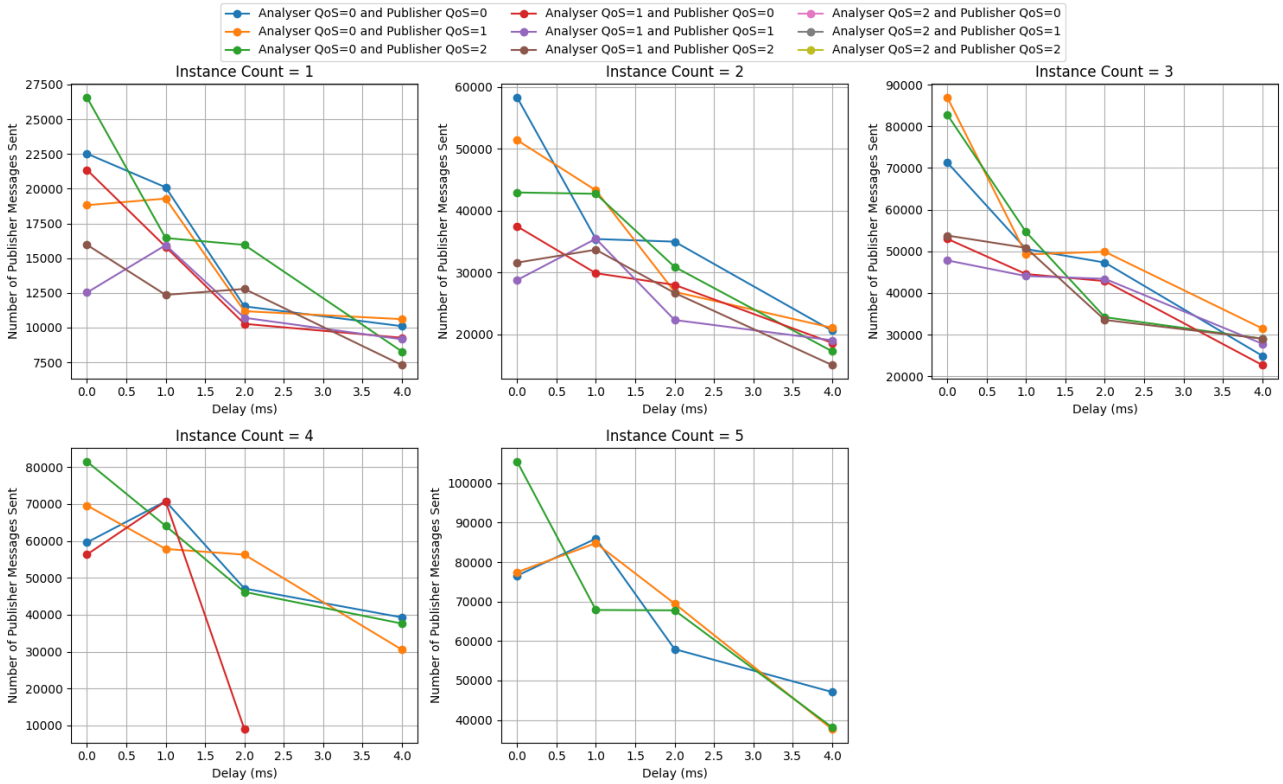


Figure 10: Message rate vs instance count on local Mosquitto broker with 10 publishers, `qos=0`, and `delay=0`. **Note**: Only y-values greater than zero are plotted

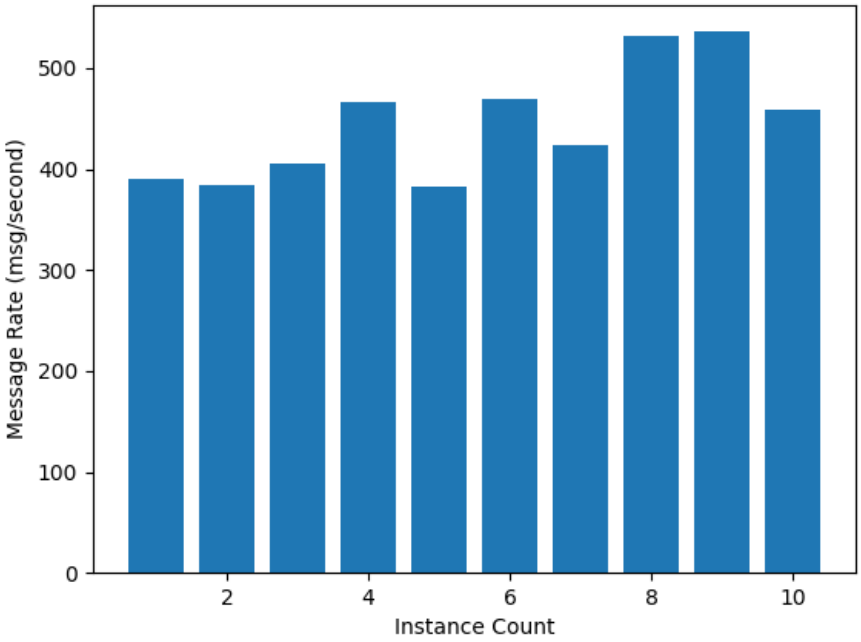


Figure 11: Heap size vs instance count on local Mosquitto broker with 10 publishers, `qos=0`, and `delay=0`. **Note**: Only y-values greater than zero are plotted

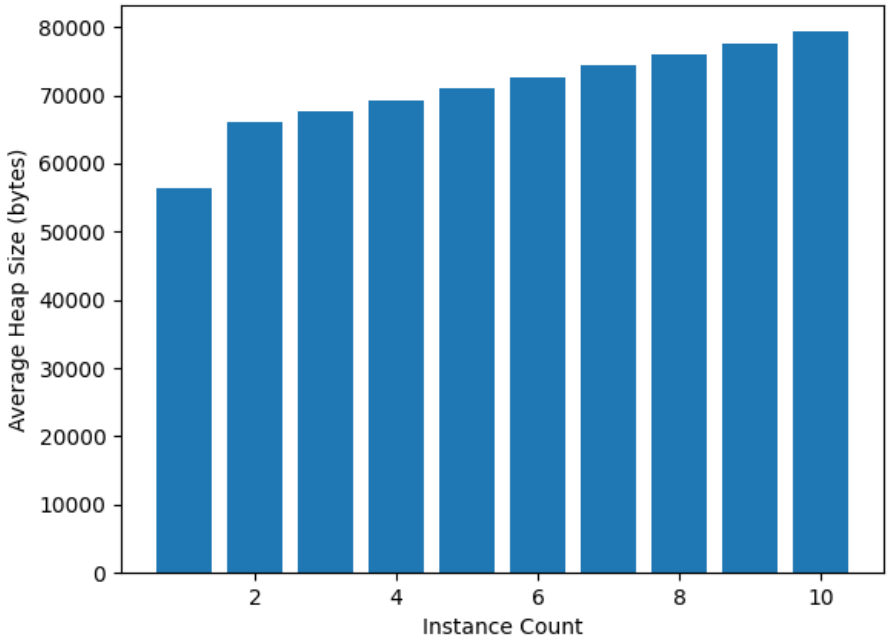


Figure 12: Message rate vs delay for different QoS and instancecounts on `broker.hivemq.com` using `delay = 0`. **Note**: Only y-values greater than zero are plotted

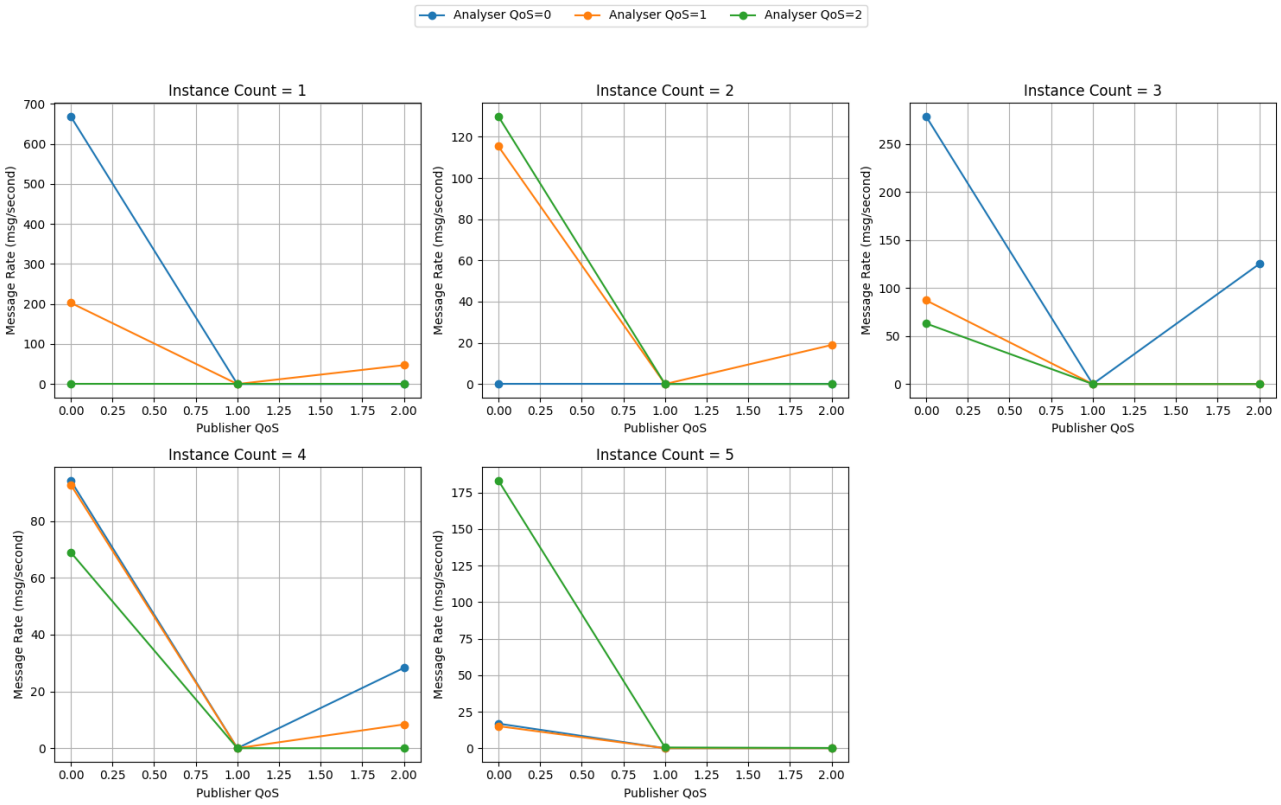
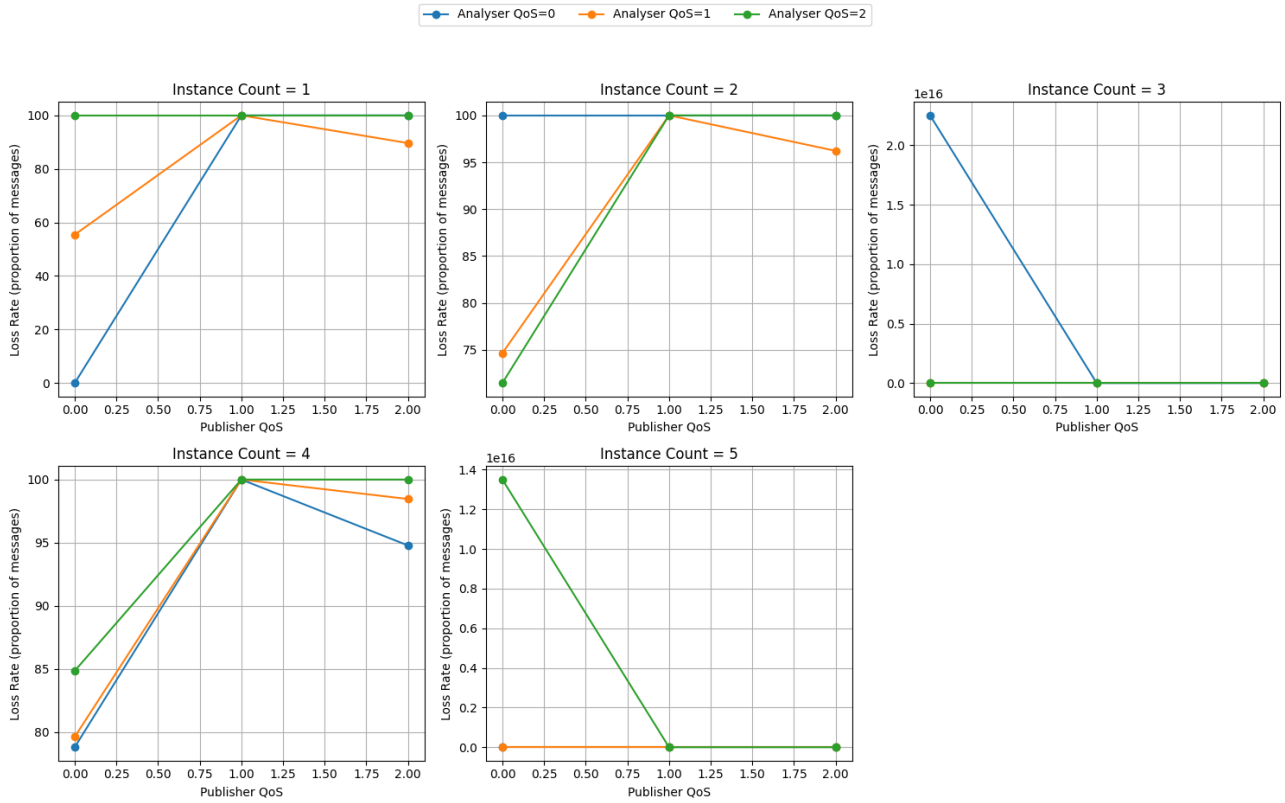


Figure 13: Loss rate vs delay for different QoS and instancecounts on `broker.hivemq.com` using `delay = 0`.



References

[1] HiveMQ Team. What is mqtt quality of service (qos) 0,1, 2? – mqtt essentials: Part 6. https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/. Accessed: 21-05-2024.

[2] Ivan Zahariev. Mqtt qos level between publisher and subscribers. https://blog.famzah.net/2022/10/03/mqtt-qos-level-between-publisher-and-subscribers/. Accessed: 21-05-2024.