IBM MQ Streaming Queues Performance Report.

Version 1.1 - December 2021

Paul Harris
IBM MQ Performance
IBM UK Laboratories
Hursley Park
Winchester
Hampshire
United Kingdom

1 Notices

Please take Note!

Before using this report, please be sure to read the paragraphs on “disclaimers”, “warranty and liability exclusion”, “errors and omissions”, and the other general information paragraphs in the "Notices" section below.

First Edition, September 2021.

Note to U.S. Government Users

Documentation related to restricted rights.

Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule contract with IBM Corp.

DISCLAIMERS

The performance data contained in this report was measured in a controlled environment. Results obtained in other environments may vary significantly.

You should not assume that the information contained in this report has been submitted to any formal testing by IBM.

Any use of this information and implementation of any of the techniques are the responsibility of the licensed user. Much depends on the ability of the licensed user to evaluate the data and to project the results into their own operational environment.

WARRANTY AND LIABILITY EXCLUSION

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

In Germany and Austria, notwithstanding the above exclusions, IBM's warranty and liability are governed only by the respective terms applicable for Germany and Austria in the corresponding IBM program license agreement(s).

ERRORS AND OMISSIONS

The information set forth in this report could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; any such change will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time and without notice.

INTENDED AUDIENCE

This paper is intended for architects, systems programmers, analysts, and programmers wanting to understand the performance characteristics, of streaming queues, as introduced in IBM MQ V9.2.3. The information is not intended as the specification of any programming interface that is provided by IBM. It is assumed that the reader is familiar with the concepts and operation of IBM MQ.

LOCAL AVAILABILITY

References in this report to IBM products or programs do not imply that IBM intends to make these available in all countries in which IBM operates. Consult your local IBM representative for information on the products and services currently available in your area.

ALTERNATIVE PRODUCTS AND SERVICES

Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

USE OF INFORMATION PROVIDED BY YOU

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

TRADEMARKS AND SERVICE MARKS

The following terms used in this publication are trademarks of their respective companies in the United States, other countries, or both:

- **IBM Corporation :** IBM, IBM MQ

Other company, product, and service names may be trademarks or service marks of others.

EXPORT REGULATIONS

You agree to comply with all applicable export and import laws and regulations.

2 Contents

3 Version History

- Version 1.0 – September 2021
  - Original version

- Version 1.1 – December 2021
  - Message expiry tests added

4 Introduction

The streaming queues feature of IBM® MQ, introduced in V9.2.3 allows you to configure a queue to put a near-identical copy of every message to a second queue (see New Streaming Queue feature for MQ 9.2.3)

This report will illustrate the cost of the additional work by the queue manager in duplicating messages. Setting STREAMQ to point to another queue for instance will typically double the internal message rate associated with the original queue (each PUT to the original queue will trigger an additional PUT to the queue defined by STREAMQ). If STREAMQ is set to a TOPIC alias, then the increase in message rate will depend on the number of subscribers to that topic.

The additional work will show as increased CPU and (for persistent messages, or where STRMQOS is set to MUSTDUP) additional I/O to the queue manager's recovery log.

It is important to consider how these additional messages are consumed to minimise the additional load on the system:

- To achieve the best performance, duplicate messages on the streaming queue should be consumed by applications interested in the copy at the same time, to avoid deep queues building up.
- Batching multiple duplicate messages into units of work can reduce the overhead further (see below).

Previous approaches to duplicating messaging involved using Pub/Sub (though this requires the consuming application to switch to the new subscriber queue). Here we will compare the cost of duplicating messages using streaming queues vs Pub/Sub.

5 Test Scenarios

Two scenarios (with variants) were measured:

1. Single duplicate message
    a. Using streaming queues (Fig 1)
    b. Using Pub/Sub (Fig 2)
    c. Using streaming queues and messages expiry (no consumers)
2. 6 Duplicate messages.
    a. Using streaming queues with Pub/Sub (Fig 3)
    b. Using Pub/Sub only (Fig 4)

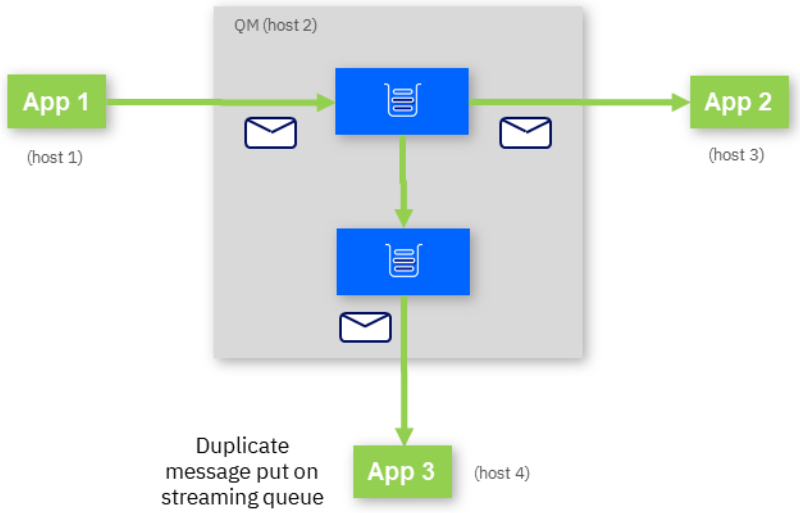

FIGURE 1 : SINGLE DUPLICATE MESSAGE PER QUEUE USING STREAMING QUEUE.

Figure 1 above shows a simple case of generating a single duplicate for each message by setting STREAMQ to point to a second queue where copies of messages are PUT. No change is needed to App1 or App2. An additional application (App3) consumes the duplicate messages from the streaming queue.

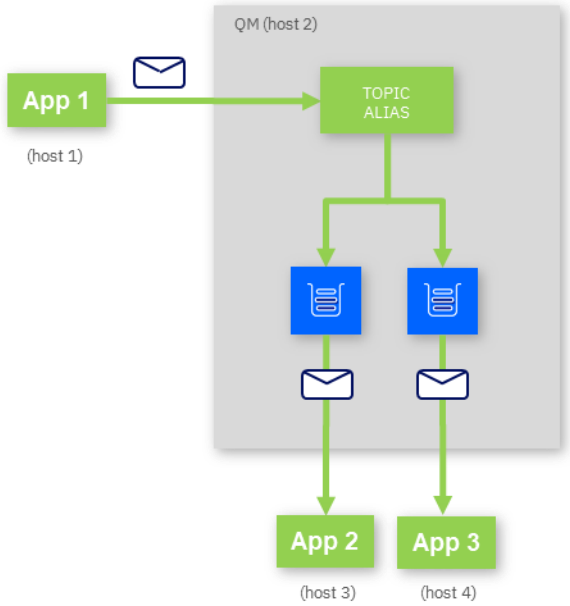

FIGURE 2 : SINGLE DUPLICATE MESSAGE USING PUB/SUB

Figure 2 shows the topology of using Pub/Sub to generate a single duplicate message. In the tests the first subscriber (App 2) is on host 3, whilst the subscriber for the duplicate message is on host 4. This pattern ensures a common topology where the primary application is on one host and secondary application, consuming the duplicate message (for logging or auditing, for example) is on a separate host. If this approach is used to create a duplicate message for an existing application then App 2 needs to be changed to consume from a subscription, rather than the original queue.

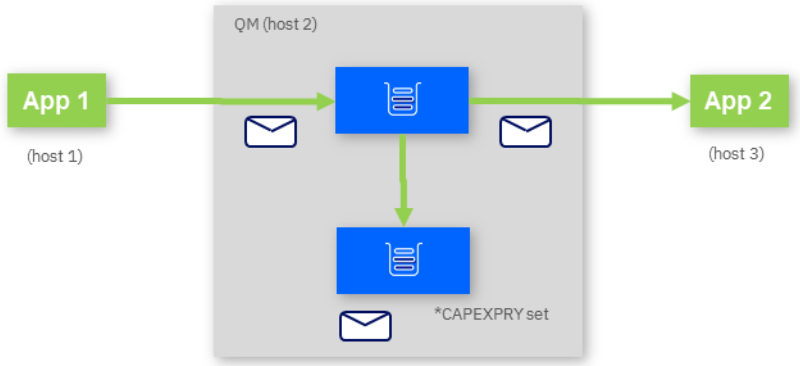

FIGURE 3 : SINGLE DUPLICATE MESSAGE USING STREAMING QUEUE WITH MESSAGE EXPIRY SET

Another use of streaming queues is for them to hold copies of messages temporarily, for contingency purposes. In this case messages put to the streaming queue have a lifetime, after which they expire, and are deleted automatically by MQ.

Message expiry is configured by setting the custom property CAPEXPRY on the streaming queue. E.g., from runmqsc:

alter ql(SQ1) custom('CAPEXPRY(6000)')

The CAPEXPRY unit is 1/10th of a second, so *subsequent* messages to that queue would have an expiry time of 600 seconds (10 minutes).

Once a message has been on the queue for longer than the expiry time, it becomes eligible for deletion, but messages are only deleted

In this case, no additional application is configured to consume the duplicate messages.

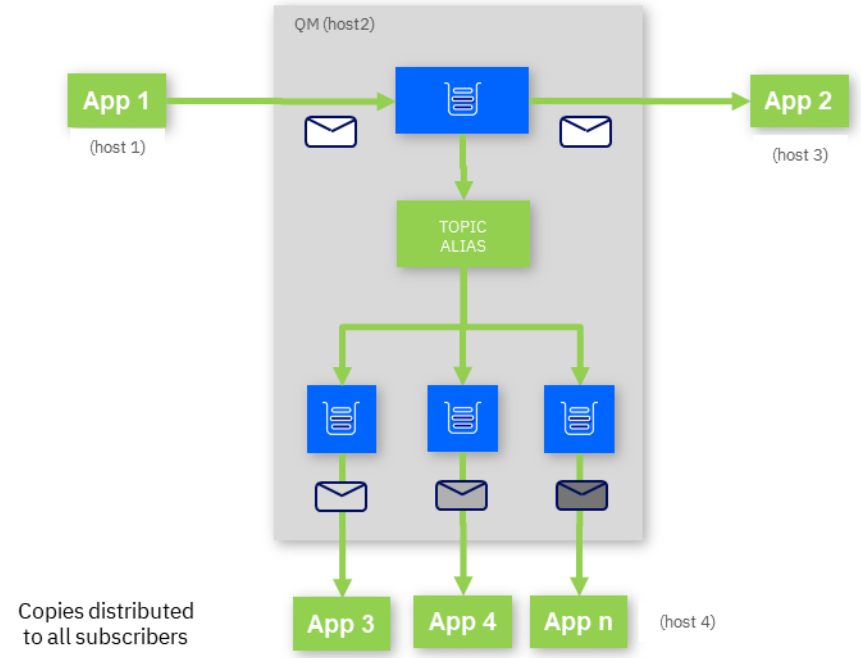

FIGURE 4: 1 TO N DUPLICATE MESSAGES USING STREAMING QUEUES WITH PUB/SUB

Figure 4 shows how multiple duplicates of messages being put onto a queue can be generated by setting STREAMQ to a topic alias and using Pub/Sub to distribute messages to

multiple subscribers. The advantages of this solution are that App2 does not need to be switched to a new subscriber queue and the existing applications do not need to be stopped to make the necessary QM changes.

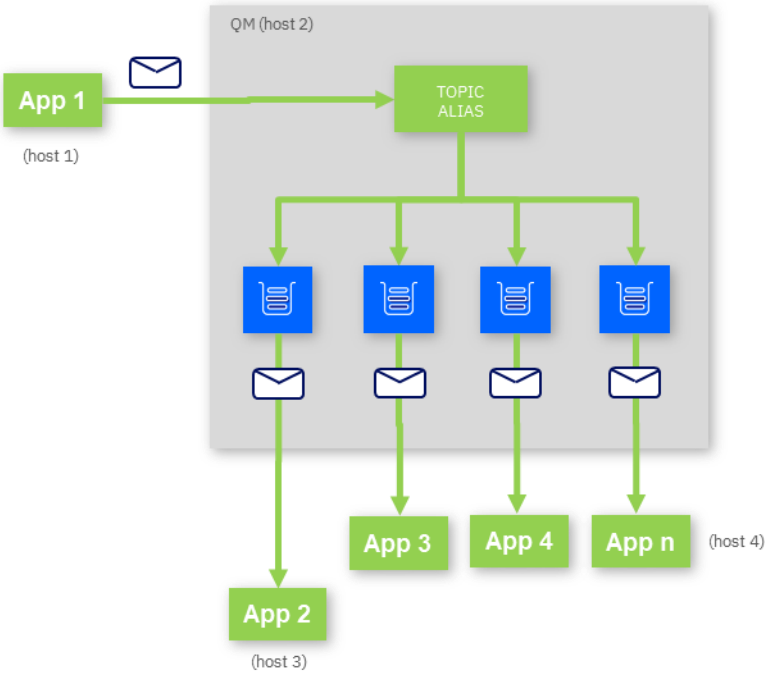

FIGURE 5 : 1 TO N DUPLICATE MESSAGES USING PUB/SUB

Figure 5 shows the topology of using Pub/Sub to generate multiple messages. In the tests the first subscriber (App 2) is on host 3, whilst other subscribers are on host 4. This pattern ensures a common topology where the primary application is on one host and secondary applications, consuming additional (duplicate) messages (for logging or auditing, for example) are on a separate host. If this approach is used to create duplicate messages for an existing application then App 2 needs to be changed to consume from a subscription, rather than the original queue.

6 Scaling the Solutions

In every scenario tested, all getters (including apps draining the stream queues, where appropriate) were started in advance (2 for every queue, ensuring there was always a waiting getter).

Tests start with 1 putter App (App1) and scale up by adding additional groups of putter apps until a limit was approached (CPU or disk bandwidth). Each instance of App1 put to its own queue.

In the simple scenario in Fig 1 for example, the persistent messaging test was scaled up from 1 to 81 putters in groups of 8 putters after the first iteration, as follows:

	Iteration 1	Iteration 2	Iteration 6
# Putters (App1)	1	9	41
# Queues (actively being put to / defined)	1/41	9/41	41/41
# Primary Getters	82 (2 per q)	82 (2 per q)	82 (2 per q)
# Stream Queues (actively being put to / defined)	1/41	9/41	41/41
# Secondary getters (consuming from stream queues)	82 (2 per q)	82 (2 per q)	82 (2 per q)

All Pub/Sub scenarios used unmanaged subscriptions. The subscriber queues each had 2 getters (e.g., two instances each of App3 to App n in Fig 2).

For all tests, getter applications kept up with the putters (App1) so there was no build-up of messages on queues.

The MQI performance test client MQ-CPH was used in all cases (https://github.com/ibm-messaging/mq-cph)

7 Results

Results for the tests outlined in section 5 are presented below. For all tests the App1 rate is the measure of how fast a test is running. This is the PUT rate achieved by App1 during the test. The internal message rate will be higher where there are duplicate messages generated. For all tests, the consuming applications kept up with the PUT rates on the original queues and any additional queues (streaming queues or additional unmanaged subscriber queues) so there was no build-up of messages in the queue manager.

Full results for all tests are included in Appendix 2.

7.1 Persistent 2KB Messaging with Single Duplicate Messages.

Figure 6 below shows results for point-to point messaging with and without generating a single duplicate of each message via streaming queues or Pub/Sub. The message rate (total put rate of all App1 instances) is plotted on the y-axis as the number of putters (App1 on the x-axis) is scaled up.

Scenarios tested:

Baseline	No duplicate messages.
1 Duplicate (streaming queues - BESTEF)	Each application queue has STREAMQ set to a local queue for duplicate messages (STREAMQOS = BESTEF).
1 Duplicate (streaming queues - BESTEF) Expiring	Each application queue has STREAMQ set to a local queue for duplicate messages (STREAMQOS = BESTEF). Duplicate messages are set to expire.
1 Duplicate (Pub/Sub)	Each application queue is a topic alias with two unmanaged subscriptions.

The total 'internal' messaging rate for the three tests generating duplicates is included below (dashed lines). These will have a total internal message rate twice that of the App1.

Where duplicate messages are produced (and they are not set to expire), they are consumed individually by a 'drainer' application (i.e., each GET from the STREAMQ or additional subscription queue is a distinct transaction, as opposed to batching the GETs which is considered in section 7.2).

The expiring messages test was run without consuming duplicate messages on the streaming queues. In this case the streaming queues had a CAPEXPRY value of 6000 (10 minutes) and the qm.ini ExpiryInterval was set to 1. Each measurement of the expiring message test was run for over 10 minutes, so these values mean that at each point on the

graph there was a total of (message rate x 600) messages spread across the queues. and since each putter (App1) has its own queue, the average queue depth was (message rate x 600 / #Putters). This contrasts with the tests that consume the duplicate messages, where no build-up on the streaming queues occurred.

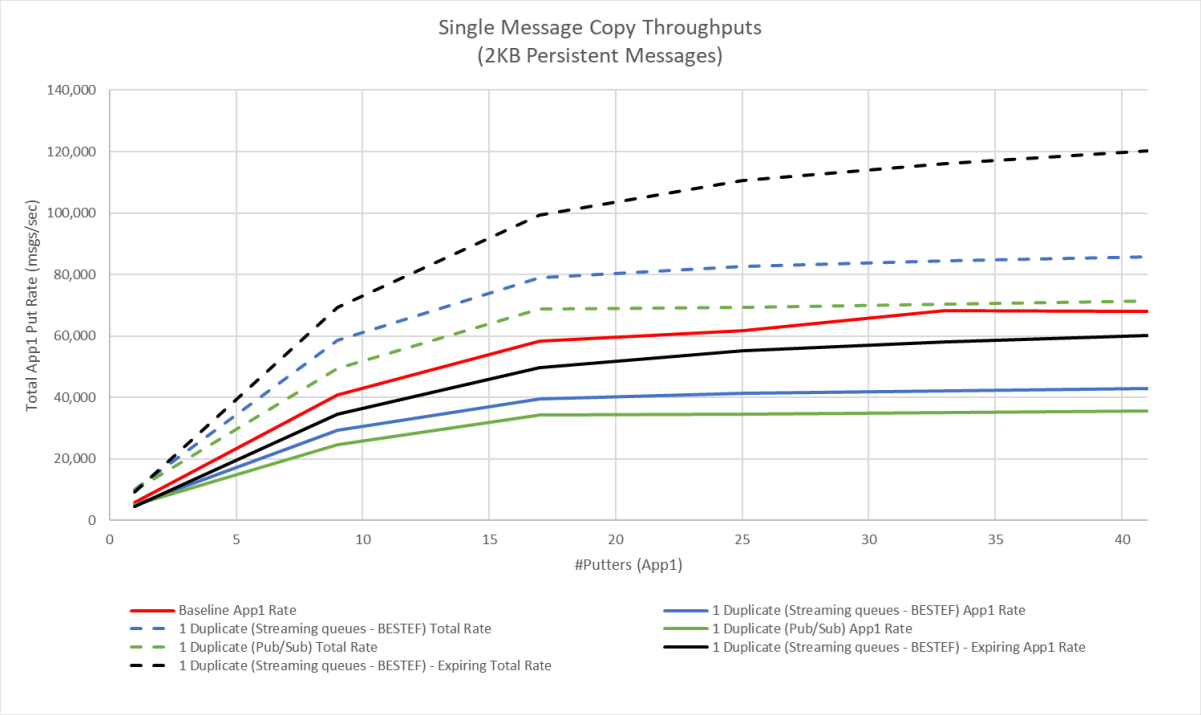

FIGURE 6: BASELINE VS SINGLE COPY THROUGHPUT (2KB PERSISTENT MESSAGING)

Using streaming queues to duplicate each message achieves a slightly higher throughput than using Pub/Sub without the need to modify the original application.

If the duplicate messages are left to expire after 10 minutes instead of consuming them, the rate achieved is significantly closer to the baseline despite messages building up (to a limit) on the streaming queues. This will cause writes to the queue files (local SSDs in this case), but these are not synchronous since the recovery log already has the data needed in the event of a disk failure for example. Having applications consume the duplicate messages can be seen to be a generally higher overhead than letting MQ delete them on expiry (though you need to consider/test the scenario where you would start consuming the duplicate messages with expiry set, for any reason). See section 7.5 for more considerations when using expiring messages with your streaming queues.

7.2 Batching Duplicate Message Consumption and Overhead of MUSTDUP

Scenarios tested:

1 Duplicate (streaming queues - BESTEF)*	Each application queue has STREAMQ set to a local queue for duplicate messages (STREAMQOS = BESTEF).
1 Duplicate (Streaming queues - drain batch=10) †	Each application queue has STREAMQ set to a local queue for duplicate messages (STREAMQOS = BESTEF). Stream queue drainer apps get in batches of 10.
1 Duplicate (Streaming queues - MUSTDUP)	Each application queue has STREAMQ set to a local queue for duplicate messages (STREAMQOS = MUSTDUP).

Messages put by App1 and read by App2 are all done so transactionally, one at a time for the persistent messaging case. This may often be the case for production applications. When reading duplicate messages from stream queues however, it may make sense to batch the reads, which can give a performance gain.

Setting MUSTDUP as the QoS on a stream queue will have a performance impact as there is additional work carried out to ensure the original message is rolled back if the PUT of the duplicate message fails.

* This is the same test as presented in section 7.1 and serves as a baseline here.

† For this test the drainer applications get 10 messages between each commit, rather than a single message per commit as for the rest of the tests.

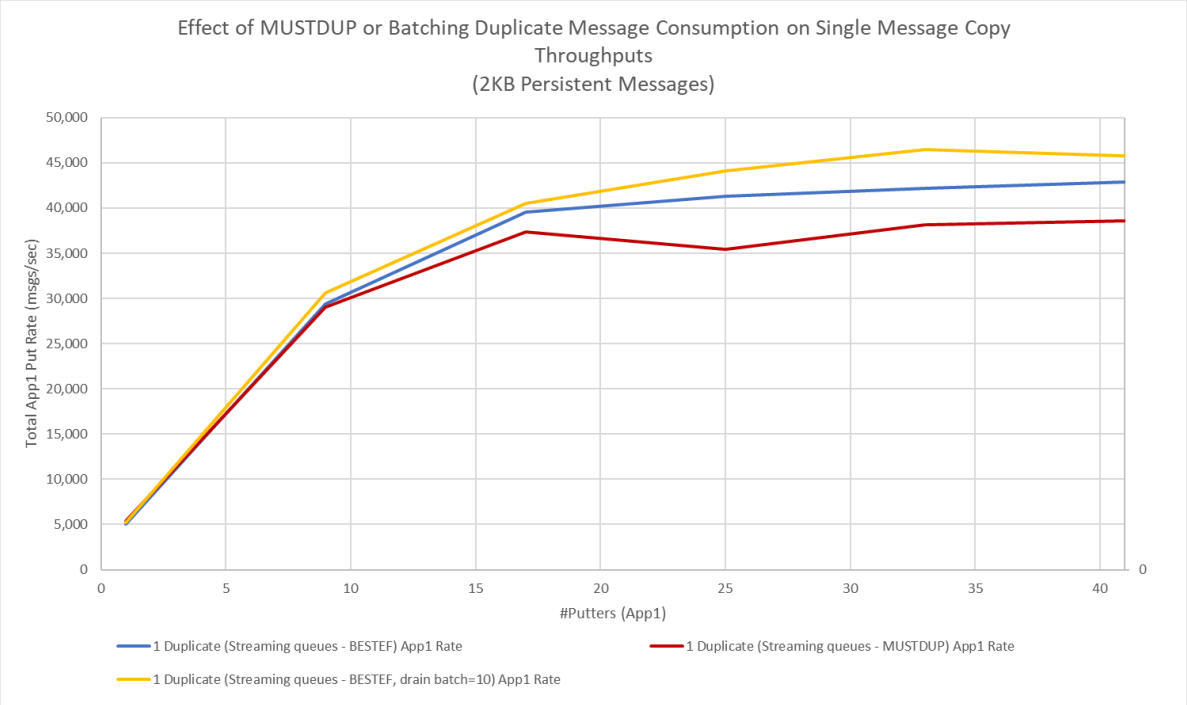

FIGURE 7: EFFECTS OF MUSTDUP OR BATCHING DUPLICATE MESSAGE CONSUMPTION

Results from the tests above are shown in Figure 7.

Batching the duplicate message consumption resulted in a significant increase in overall throughput. This gain will be more or less, depending on other factors, such as message size, performance of the recovery log file system etc. so you should experiment with different values. Bear in mind that large batch sizes may significantly delay the availability to the consuming application and result in more uncommitted data in the recovery log.

Setting MUSTDUP reduced the overall throughput significantly as expected. The overhead of MUSTDUP is significantly more for non-persistent messaging (see section 7.4 below).

7.3 Persistent 2KB Messaging with Multiple Duplicate Messages.

When more than one duplicate of a message is required STREAMQ can be set to point to a topic alias (as in Fig 3 above).

Scenarios tested were:

6 Duplicates (streaming queues – BESTEF, with Pub/Sub)	Each application queue has STREAMQ set to a topic alias with 6 unmanaged subscriptions. (STREAMQOS = BESTEF).
6 Duplicates (Pub/Sub)	Each application queue is a topic alias with 7 unmanaged subscriptions.

Note that there are 7 copies of the message in total for both solutions.

Figure 8 below show the results from this test.

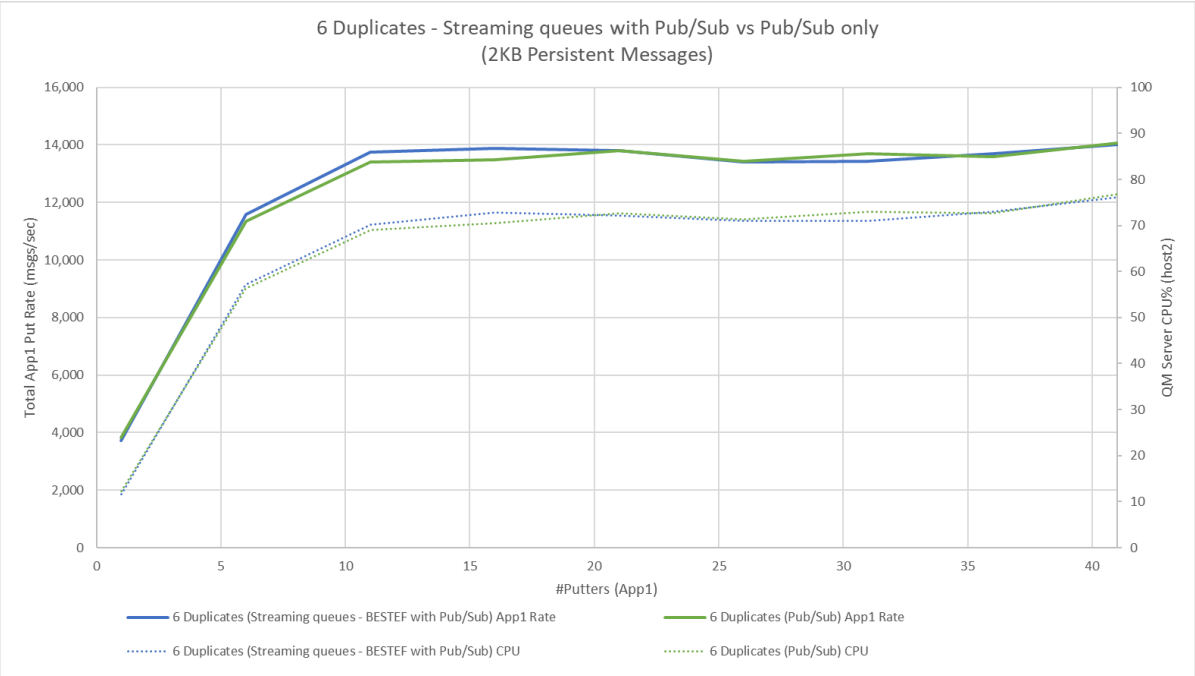

FIGURE 8: 6 DUPLICATES: STREAMING QUEUES WITH PUB/SUB VS PUB/SUB ONLY (2KB PERSISTENT MESSAGING)

Most of the work in both these solutions is being carried out by Pub/Sub in MQ, so unsurprisingly there is little difference between the two approaches in terms of performance, but once again the original applications do not need to be altered to implement the streaming queue approach.

7.4 Peak Rates Achieved for All Scenarios

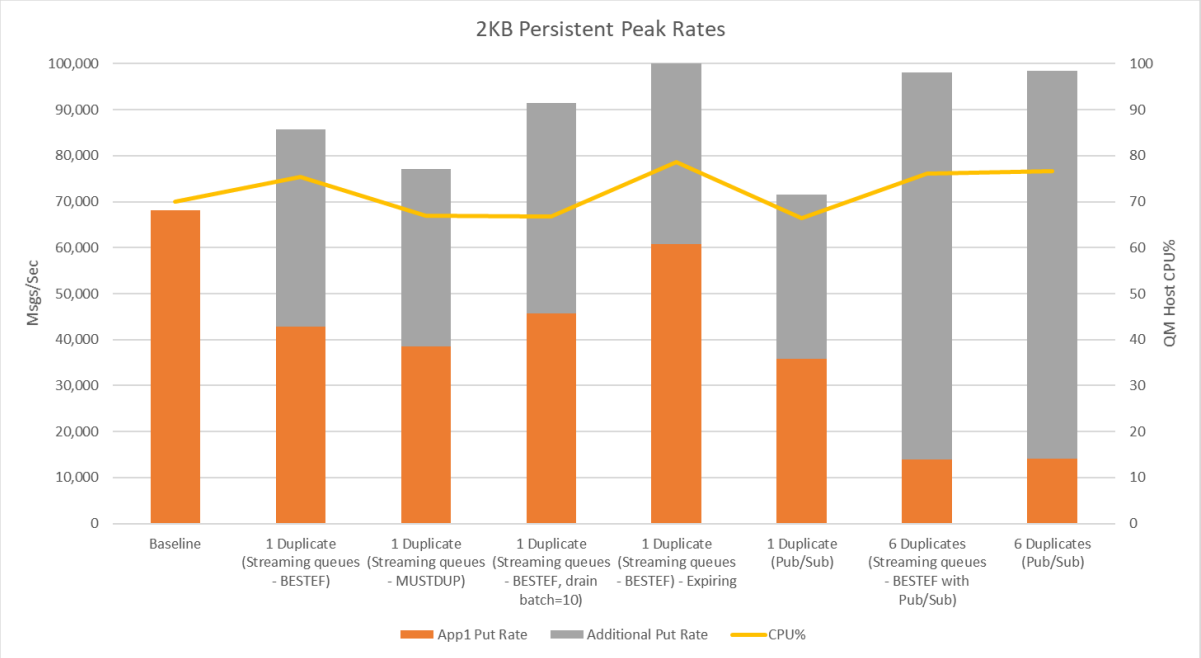

FIGURE 9: PEAK RATES FOR PERSISTENT MESSAGE TESTS

Figure 9 above, shows the peak rates achieved for all the persistent tests presented in the previous sections. The height of each bar represents the peak App1 PUT rate achieved plus the additional internal PUTS caused by message duplication.

Typically, recovery log and locking are the limiting factors for persistent messaging, which is evident here by the CPU consumption not approaching 100% when the peak rate is achieved (compare this to the CPU consumption for non-persistent, below). An existing system already have the CPU capacity to accommodate duplicating messages, but you need to consider the additional data being written to the recovery log for persistent messaging (alongside any additional network bandwidth being consumed by duplicate message consumption).

A tool (MQLDT) is available to assess the recovery log's file system performance/capacity (see section 0 below).

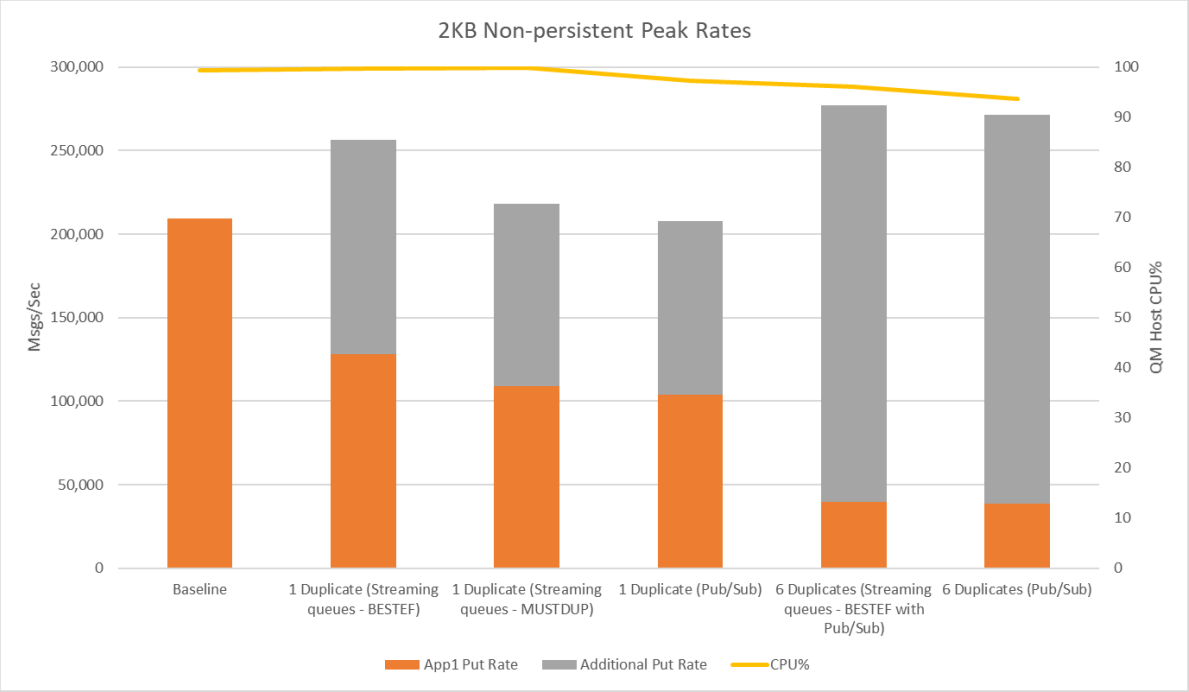

FIGURE 10: PEAK RATES FOR NON-PERSISTENT MESSAGE TESTS

Tests were also run for non-persistent messaging. Figure 10 above shows the peak rates achieved for those tests.

Non-persistent results show a similar pattern to persistent results. Specifying MUSTDUP has a proportionally larger effect on non-persistent messaging (a 15% reduction in throughput compared to BESTEF versus a 10% reduction for persistent messaging in this case).

7.5 Further Considerations for Using Expiring Messages.

In section 7.1 a scenario with expiring messages was presented (illustrated in Figure 3). You can use streaming queues with expiring messages to hold a buffer of duplicate messages, held for a pre-determined amount of time. There are a few things that need to be configured and understood using this approach.

1. Expiry behaviour is configured by the custom CAPEXPRY queue property and the ExpiryInterval qm.ini property.
2. Duplicate messages will build up on the streaming queue(s) until they start to be deleted automatically by MQ. The number of messages on a queue (queue depth) is a function of the messaging rate and the expiry configuration parameters above.
3. Additional file system storage may be required to accommodate the queue files holding the duplicate messages. The amount of storage required is a function of the average message size and the maximum queue depth reached.

7.5.1 Message Expiry Configuration

Message expiry is configured on a per queue basis, using the CAPEXPRY custom property.

E.g., from runmqsc:

alter ql(SQ1) custom('CAPEXPRY(6000)')

The CAPEXPRY unit is $1/10^{th}$ of a second, so *subsequent* messages to that queue would have an expiry time of 600 seconds (10 minutes). Messages already on a queue will not be affected by a new setting for CAPEXPRY.

Expired messages are deleted by MQ periodically, with a frequency determined by the ExpiryInterval setting in the TuningParameters stanza of the mq.ini file for the queue manager. This is specified in seconds and defaults to 300 seconds, if not set explicitly.

7.5.2 Queue Depth of Streaming Queues with CAPEXPRY set.

Assuming a streaming queue has been configured to hold a buffer of duplicate messages with expiry time set and messages are being put onto the queue at a rate of N messages per second, then the queue will grow to a maximum depth of:

N x (CAPEXPRY /10) + N x ExpiryInterval

When the Expiry task is scheduled by MQ there will always be N x (CAPEXPRY/10) messages that are too young to be expired. By the time the next expiry task is scheduled there will be an additional N X ExpiryInterval messages on the queue.

7.5.3 Additional File system Storage Requirements for Streaming Queues with CAPEXPRY set

As messages arrive on the streaming queue, the queue depth will increase if there are no consumers. Messages will be written to the underlying queues files as the internal queue buffers are filled and at checkpoint times (when MQ establishes a consistent state between the queues files and recovery log, for restart purposes).

We can ignore the complexities of queue buffers and checkpointing and safely assume that at some point we may have to hold all the messages on a queue when it is at its deepest. The size of a message on the queue is more than just the bytes in the application however, there is additional metadata stored, including message headers. In practise a safe overhead to use is 50% (though this will be smaller for large messages) i.e., if we store 100 x 2KB messages on disk, this will require 300KB of disk space at most. Note that the minimum message size this is reasonable for, is 2K. If your messages are much less than 2K in size, the metadata component becomes very significant, so size for 2K as a minimum.

From the previous calculations of the maximum queue depth of a queue with CAPEXPRY set, then we can estimate that such a queue will require the following disk space for its file:

Msg Size x 1.5 x N x (CAPEXPRY/10 + ExpiryInterval)

Where N = PUT rate to the queue.

7.5.4 Message Expiry Tests

Using the understanding and calculations above we can make some predictions of the behaviour and requirements for some example tests using streaming queues with expiring messages.

In the tests below, persistent messages were PUT to 5 queues, each with a streaming queue defined with STRQOS(MUSTDUP). Consumers drained the 'original' 5 queues whilst the streaming queues were set up to expire messages. The messages were PUT onto the queues at a fixed rate and the queue depth and file system requirements monitored.

Test #	Msg Size	Total Msg Rate * (PUTS/sec)	CAPEXPRY	Expiry Interval	Max Queue Depth of Streaming Queues		Total Streaming Queue File System Usage (MB)	
					Predicted	Actual	Sizing	Actual
1	2KiB	10,000	3,000	300	1,200,000	1,188,584	18,432	15,005
2	2KiB	10,000	36,000	300	7,800,000	7,795,602	119,808	97,440
3	20KiB	10,000	6,000	1	1,202,000	1,202,876	184,627	123,566

TABLE 1 - FIXED RATE TESTS WITH MESSAGE EXPIRY RESULTS

***The total message rate is the PUT rate for the test. As there are 5 queues being used, the PUT rate per queue was 2,000/sec.**

The results in Table 1 above show three tests. The first 2 are for smaller (2KiB) messages, both run the message expiry task every 5 minutes (ExpiryInterval=300). Test #2 has the larger CAPEXPRY, so results in deeper queues, before the messages start expiring. Test#3 is for a larger message size and whilst the resultant queue depths are less, the amount of storage on disk is the largest as CAPEXPRY is still set relatively high (10mins).

Using the formulas discussed previously, it can be seen that the measured maximum queue depth is very close to that predicted (the measured queue depth is for one of the 5 queues, but the other 4 showed similar depths). The file system sizing is a calculation that has some contingency built in. The actual file system space is well within the sizing.

Test #3 has the minimal ExpiryInterval value of 1sec, so once the messages start to expire, the queue depth should remain fairly static. Figure 11 shows the recorded queue depth of one of the streaming queues in Test #3 and it can be seen that the behaviour is as expected. The graph is for a single streaming queue, but the remaining streaming queues showed the same behaviour.

Test #2, in contrast to Test #3, has a much longer CAPEXPRY value and ExpiryInterval is set to 5 minutes, so we expect a much larger maximum queue depth and a pronounced saw-tooth effect on queue depth thereafter, as messages have more time to accumulate on the queue before the next expiry task is triggered. This can be seen in Figure 12.

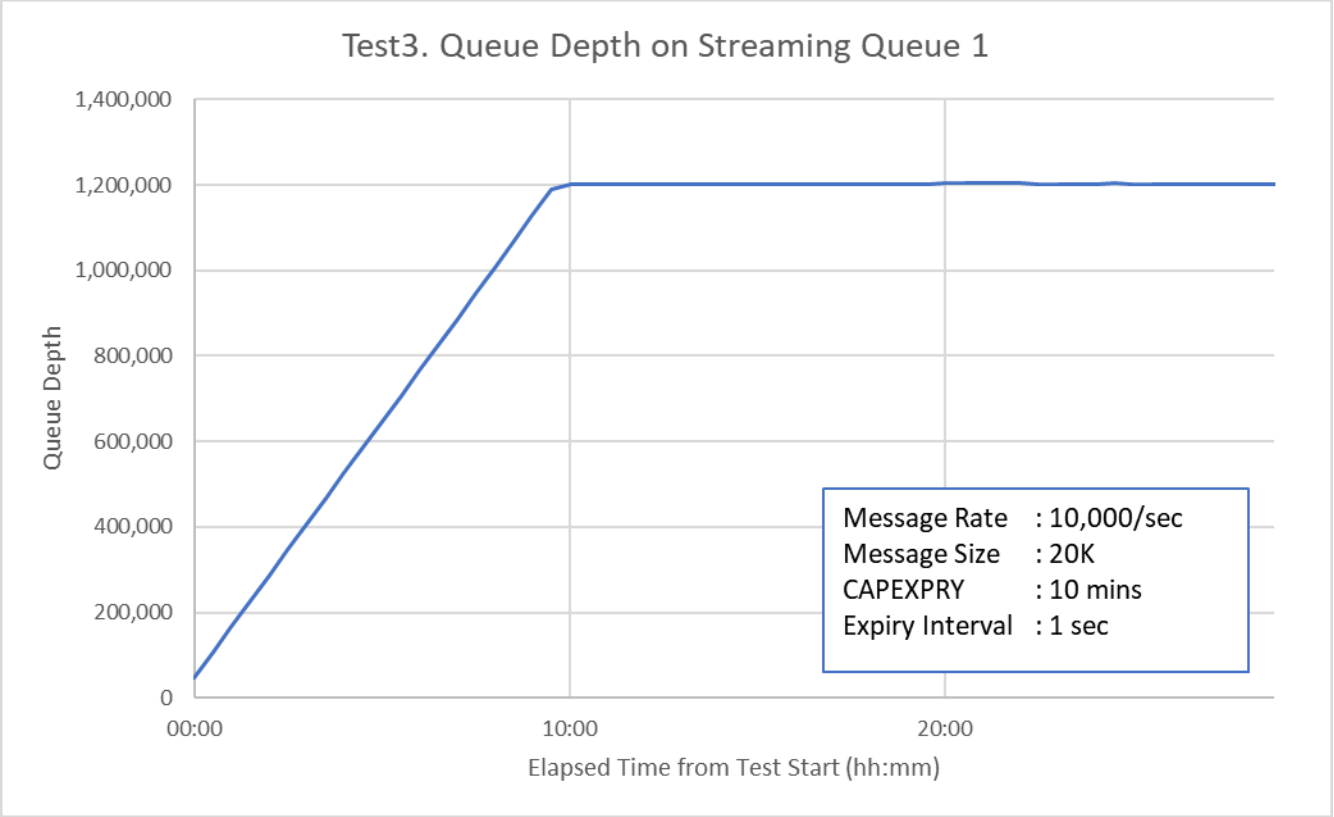

FIGURE 11 - STREAMING QUEUE DEPTH FOR TEST #3

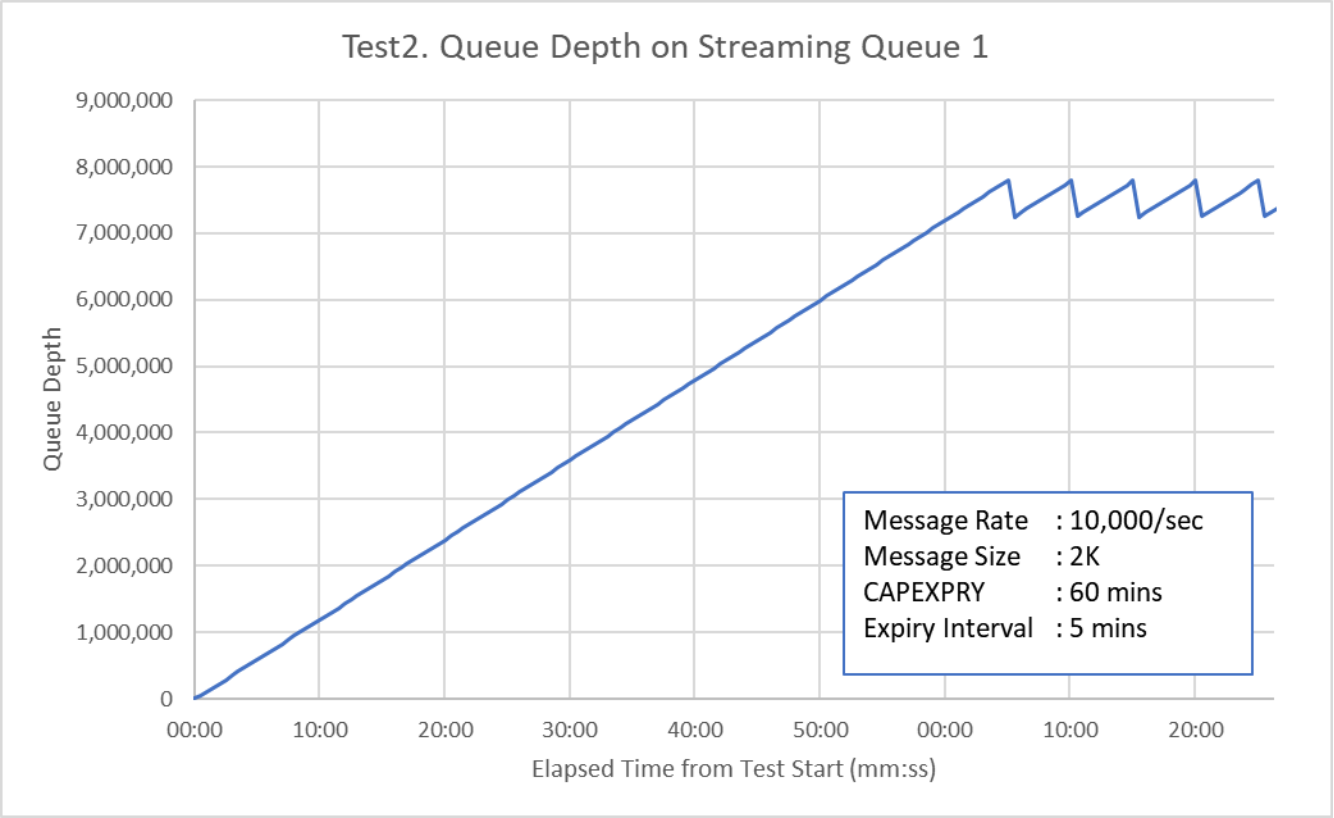

FIGURE 12 - STREAMING QUEUE DEPTH FOR TEST #2

The default value for ExpiryInterval is 300 (5 minutes). By reducing that value you can reduce the maximum queue depths of the streaming queues and the additional storage required for the queue files, as seen above. What effect does this have on performance though? If we run more frequent expiry tasks (but consequently processing less eligible messages on each invocation), how does this compare with running the expiry less frequently?

Table 2 shows results from a range of tests, measuring the CPU of the MQ server for different values of message expiry. In each test a fixed load of 55,000 2KiB persistent messages/sec was run across 40 queues, configured as described by Figure 3. CAPEXPRY was set to 3000 for all tests, so the maximum messages across the queues was (55,000 x 300 + 55,000 x ExpiryInterval). All non-srteaming queues were kept drained by applications, and the rate of 55,000/sec was maintained in all cases.

ExpiryInterval	Queue manager Host CPU%
300 (default)	65%
150	66%
120	65%
50	66%
30	66%
10	67%
5	66%
1	66%

TABLE 2 - EFFECT OF EXPIRYINTERVAL ON CPU CONSUMPTION AT LOAD

CPU consumption was measured for a period including 2 or more expiry task after the test had settled (i.e., expiry tasks were starting to delete messages). For this test altering ExpiryInterval showed no discernible impact on CPU consumption.

Another consideration in using message expiry is that in the event of the queue manager being re-started or recovering from a power failure for example, processing deep queues has an impact on the time it takes for the queue manager to be available again. A report on re-start times is available on the MQ performance GitHub site: [Queue Manager Restart Times for IBM MQ V9.1 & V9.1.1]

8 Conclusions

- Streaming queues introduces a new approach to message duplication which does not require changes to existing applications whilst matching or surpassing the performance of previous options (e.g., using Pub/Sub).

- Whilst duplicating each message once will double the internal work rate (assuming the duplicates are also being consumed), the original message rate will not be halved, even when resources on the host are exhausted (see figures 6 & 7).

- Any form of message duplication will involve additional work by the queue manager, so testing and planning for capacity is essential.
    - Additional CPU.
    - Additional data written to the recovery log for persistent messages or if MUSTDUP is specified.
    - Additional network bandwidth utilised/required if remote clients are consuming duplicate messages.

- Specifying MUSTDUP will incur an additional cost, which should be evaluated.

- To minimise additional load on the system and to achieve the best performance, duplicate messages should not be left to build up on the streaming queue. Instead, they should be consumed by applications interested in the copy as they arrive.

- It may be possible to optimise the consumption of duplicate messages by batching gets of persistent messages for instance, where this approach was not suitable for the original application.
- The message expiry capability of MQ can be used in conjunction with streaming queues to hold a temporary store of messages, but care must be taken to understand how deep the queues will grow, and how much additional file space may be required.

Note that you should test your own environment and applications where possible as there may be factors not present in these tests that alter the behaviour of the products, operating system, network etc. in your shop.

9 Resources

Streaming queues blog article :	New Streaming Queue feature for MQ 9.2.3
Streaming queues documentation :	https://www.ibm.com/docs/en/ibm-mq/9.2?topic=scenarios-streaming-queues
IBM MQ C Performance Harness (MQ-CPH) :	https://github.com/ibm-messaging/mq-cph
MQ Log Disk Tester (MQLDT) tool :	https://github.com/ibm-messaging/mqldt
Queue Manager Restart Times Paper :	https://ibm-messaging.github.io/mqperf/Queue%20Manager%20Restart%20Times.pdf

Appendix A: Software and Hardware

MQ Server (host 2):

System x3550 M5 -[8869AC1]

CPU 2x14 Cores: Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz

Memory: 128GB

Network: 40Gb via dedicated switch.

O/S : RedHat Enterprise Linux Server V7.9 (Maipo)

MQ : V9.2.3

Primary Client Hosts (host 1 & host3)

System x3550 M5 -[8869AC1]

CPU 2x14 Cores: Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz

Memory: 128GB

Network: 40Gb via dedicated switch.

O/S : RedHat Enterprise Linux Server V7.9 (Maipo)

MQ-CPH Performance Harness with MQ V9.2.3 client libraries.

Secondary Client Host Running Duplicate Message Drainer Apps (host 4)

ThinkSystem SR630 - [7X02CTO1WW]

CPU 2x12: Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz

Memory: 192GB

Network: 40Gb via dedicated switch.

O/S : RedHat Enterprise Linux Server V7.9 (Maipo)

MQ-CPH Performance Harness with MQ V9.2.3 client libraries.

Appendix B: Additional Data

The tables below present the raw data used in the charts throughput this report, including the CPU% of the machine hosting the MQ queue manager. The CPU consumption of client application hosts is not show as this was negligible, did not present a bottleneck and will not be representative of any specific application in production.

# Clients (App1 Putters)	1	9	17	25	33	41
Baseline App1 Rate	5,872	40,810	58,265	61,644	68,222	68,094
Baseline CPU	4.46	27.74	42.82	55.2	69.33	69.94
1 Duplicate (Streaming queues - BESTEF) App1 Rate	5,019	29,367	39,561	41,332	42,218	42,882
1 Duplicate (Streaming queues - BESTEF) CPU	5.37	45.73	66.92	71.89	72.55	75.43
1 Duplicate (Streaming queues - BESTEF) Total Rate	10,038	58,733	79,121	82,663	84,437	85,765
1 Duplicate (Streaming queues - MUSTDUP) App1 Rate	5,421	29,050	37,371	35,474	38,172	38,585
1 Duplicate (Streaming queues - MUSTDUP) CPU	5.05	44.01	62.28	59.89	64.61	66.92
1 Duplicate (Streaming queues - BESTEF, drain batch=10) App1 Rate	5,205	30,577	40,477	44,106	46,459	45,745
1 Duplicate (Streaming queues - BESTEF, drain batch=10) CPU	4.55	37.24	56.93	62.74	65.92	66.73
1 Duplicate (Pub/Sub) App1 Rate	4,956	24,741	34,462	34,694	35,234	35,755
1 Duplicate (Pub/Sub) CPU	5.21	41.18	62.31	63.64	63.68	66.38
1 Duplicate (Pub/Sub) Total Rate	9,912	49,483	68,925	69,387	70,468	71,510
1 Duplicate (Streaming queues - BESTEF) - Expiring App1 Rate	4,593	34,634	49,627	55,313	58,029	60,116
1 Duplicate (Streaming queues - BESTEF) - Expiring CPU	3.57	29.29	46.14	58.35	69.15	73.04
1 Duplicate (Streaming queues - BESTEF) - Expiring Total Rate	9,187	69,268	99,253	110,626	116,059	120,231

TABLE 3: SINGLE DUPLICATE RESULTS (SECTIONS 7.1 & 7.2)

# Clients (App1 Putters)	1	6	11	16	21	26	31	36	41
6 Duplicates (Streaming queues - BESTEF with Pub/Sub) App1 Rate	3,712	11,597	13,754	13,891	13,809	13,416	13,423	13,704	14,006
6 Duplicates (Streaming queues - BESTEF with Pub/Sub) CPU	11.6	57.17	70.23	72.8	72.21	70.94	70.96	73	76.18
6 Duplicates (Pub/Sub) App1 Rate	3,817	11,336	13,405	13,494	13,797	13,430	13,704	13,603	14,060
6 Duplicates (Pub/Sub) CPU	12.31	56.27	69.01	70.45	72.68	71.26	73.02	72.63	76.71

TABLE 4: MULTIPLE DUPLICATE RESULTS (SECTION 7.3)

	App1 Put Rate	Additional Put Rate	CPU%
Baseline	68,094	0	69.94
1 Duplicate (Streaming queues - BESTEF)	42,882	42,882	75.43
1 Duplicate (Streaming queues - MUSTDUP)	38,585	38,585	66.92
1 Duplicate (Streaming queues - BESTEF, drain batch=10)	45,745	45,745	66.73
1 Duplicate (Pub/Sub)	35,755	35,755	66.38
6 Duplicates (Streaming queues - BESTEF with Pub/Sub)	14,006	84,033	76.18
6 Duplicates (Pub/Sub)	14,060	84,361	76.71

TABLE 5: PERSISTENT MESSAGING PEAK RATES (SECTION 7.4)

	App1 Put Rate	Additional Put Rate	CPU%
Baseline	209,257	0	99.29
1 Duplicate (Streaming queues - BESTEF)	128,130	128,130	99.7
1 Duplicate (Streaming queues - MUSTDUP)	108,973	108,973	99.85
1 Duplicate (Pub/Sub)	103,840	103,840	97.25
6 Duplicates (Streaming queues - BESTEF with Pub/Sub)	39,568	237,407	96.11
6 Duplicates (Pub/Sub)	38,790	232,740	93.56

TABLE 6: NON-PERSISTENT MESSAGING PEAK RATES (SECTION 6.4)