IBM MQ Performance: Best Practises, and tuning.

Version 1.0 - March 2019

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

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, March 2019.

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, and best practises of IBM MQ. 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
- **Oracle Corporation :** Java

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 Introduction

This paper is a consolidation of the material previously presented in MQ performance reports on best practises and tuning.

It is intended to be used as a guide by systems programmers, and MQ application designers setting up, and running applications using IBM MQ.

Version one (March 2019) is largely the general sections from the V8 MQ on Linux performance report, which have not been included in the V9.1 performance report, as this document will be maintained alongside future performance reports.

4 Planning for Performance

There are many things that need to be taken into account when planning an MQ infrastructure. Often, performance considerations end up being given a lower priority than those of security, integrity, and even application transparency. While this is often correct (particularly in the cases of security and integrity), it's important to bear in mind that many techniques used to mitigate these concerns, if not implemented carefully, can have a negative impact on the performance of the system. In some cases, we have even seen situations where the performance of a system becomes severely degraded, rendering it unusable for the task it was designed to perform. However, with careful planning in advance, it should be possible to design a system that performs sufficiently, while being secure, scalable, transparent, and highly available.

This section illustrates some common pitfalls and best practices that should be adhered to when designing and testing an MQ infrastructure for performance.

4.1 Persistence, High Availability, & Disaster Recovery

Using persistent messaging will slow a system down. Because of this, implementations using persistent messages and high availability are some of the most common scenarios where performance issues arise. Avoiding these issues involves careful planning in advance, but can be achieved.

The first step that should always be taken is to establish whether and where persistence is actually needed. Some things to consider:

- MQ logging protects the integrity of data in transit, so that it can be recovered in the event of a failure. It is not designed to act as a historical database for audit or any other purposes.
- Since recovery often takes some time to complete in the event of a failure, it's probably not worth logging information that expires (loses its relevance) quickly.
- In a well-designed system, failures should be rare. If the in-flight information could be reconstructed and re-sent with relative ease, then this may be a preferable to logging all messages, the vast majority of which will arrive at their destination without incident. In particular, if client applications already have disaster-recovery logic built-in, then it may be possible to switch off persistence with no loss of integrity, and no additional work required.

If persistence is definitely required, then in addition to the basic recommendations given in section 5.1, there are a number of things to consider when designing an MQ architecture.

Persistence can impact performance in two ways:

1. High latency - how long it takes for a filesystem write operation to return
2. Low bandwidth - the maximum data-rate that can be achieved on the file-system

4.2 Latency

The latency of a filesystem will depend on the level of integrity it has, with the most robust set-ups generally suffering the most.

- For local persistent messaging, latency occurs due to the time taken to write logs to the disk, or its battery-backed cache. SAN systems may be able to ensure integrity before the data is synced to disk, but will instead have latency caused by the time taken to communicate with the SAN host.
- For highly-available persistent infrastructures (multi-instance), the requirement for networked file-systems introduces an additional latency in communicating with the filesystem host.
- Finally, in full disaster-recovery situations, file-systems may be synchronously replicated to a second data-centre, often geographically distant from the first. This additional network communication and disk write must be waited for before the write operation completes.

In all cases, there are 2 techniques that can be used to overcome latency:

1. Concurrency – running multiple application threads in parallel.
2. Batching – reducing the number of write operations each application thread performs.

4.2.1 Concurrency

All persistent messaging operations should be done inside syncpoint, this delays forcing messages to disk until *MQCMIT* is called by the application. If multiple applications are accessing a queue manager concurrently, this may allow the disk writes of several applications to be grouped together into a single operation, thus reducing the number of writes required.

From a client perspective, having multiple applications also means that one application can be making use of other resources (such as CPU) while another is waiting on a disk write.

4.2.2 Batching

As well as doing all persistent messaging inside syncpoint, it may be possible for applications to perform several messaging operations before each call to *MQCMIT*, thus reducing the number of write operations each thread requires.

- Responder type applications may be able to get their request and put their response inside the same syncpoint. This is how the MQI responders used to gather the results in this report are designed to work.
- Batch processing applications may be able to process multiple records before each *MQCMIT*. The time taken to reprocess a batch in the event of a failure should be weighed up against the likelihood of failure.

In more complex infrastructures, involving queue manager to queue manager communication (distributed queueing and clustering), one can also increase the batch

size on sender/receiver channels to the same effect, using the *BATCHSZ* configuration parameter.

It's worth noting that if a disk system (particularly a networked filesystem) is unreliable, then batching may have the opposite effect. Increasing the size of write packets may make the write more likely to fail, requiring the operation to be repeated, and thus *increasing* the total number of writes.

4.3 Bandwidth

A bottleneck in I/O bandwidth can only be mitigated by increasing hardware capacity. It's important to first establish which aspect of the I/O subsystem might be constrained. It could be the disks themselves, in which case adding more disks in a RAID5 array may help. Where possible, upgrading the constrained component to a higher-capacity alternative should help.

When it's not feasible to upgrade a component further, it's likely that the logs will need to be split, which will require using multiple queue managers (see section 4.6.4). How this split is implemented will depend on which aspect of the I/O system is constrained. Some possibilities:

- The local file-system on the file-system host – put the logs for each QM in a directory on a separate disk or RAID device.
- For networked filesystems, other physical constraints (CPU, RAM, network) on the file-system host – put the logs for each QM in a different remote file-system, hosted on different machines.
- For networked filesystems, network bandwidth on the queue manager host – put each QM on a separate machine.
- Synchronous data replication – any of the above, synchronising each QMs log directory with a different destination.

4.4 Slow Networks

When using MQ over slow or unreliable networks, such as satellite links and the Internet, some additional design considerations may be beneficial in achieving optimal performance. If network bandwidth is contended, and there are spare CPU cycles available on the source and destination machines, then it may be possible to obtain higher throughput by utilising header compression (*COMPHDR*)[a] and data compression (*COMPMSG*)[b].

Aside from this, it may be possible to adjust the architecture of an MQ infrastructure to make most efficient use of the network. We'll consider a number of scenarios, and some possible techniques to increase throughput.

[a] www.ibm.com/support/knowledgecenter/SSFKSJ_9.1.0/com.ibm.mq.ref.con.doc/q081880_.htm
[b] www.ibm.com/support/knowledgecenter/SSFKSJ_9.1.0/com.ibm.mq.ref.con.doc/q081840_.htm

4.4.1 Many disparate clients communicating infrequently

In this scenario, a large number of clients, all from different places on the network, connect to a queue manager, and send or receive messages at a relatively low rate. As such, there's no constraint on the network path between the queue manager and any one of these clients, but together these clients put a strain on the network interface of the QM host.

If messages are small, it could be that, although the total data flowing to and from the host is within the supposed bandwidth of the network interface, the sheer number of networking operations cannot be processed fast enough. If this is the case, one possible solution is to introduce a number of "satellite" queue managers in different locations. Clients connect to the satellite that is "closest" to it (in terms of network topology), which then forwards communication to the main QM. Following the advice in section 4.4.4 then allows us to optimise the connections between the satellites and the main host, reducing the number of networking operations performed.

If raw data bandwidth is constrained, there's often very little that can be done here from a design perspective; ensuring that host's network interface has enough capacity for the desired message rate is probably the simplest solution (see section 5.4). You should also ensure that the IP layer is tuned optimally. If this is not feasible, then the workload will need to be split across multiple machines. See section 4.6.4 for tips on how this can be achieved.

4.4.2 High message rate to/from a client machine

It may be that certain machines in the infrastructure will have large volumes of communication with one or more QMs elsewhere on the network. This could either be one application messaging at a high rate, or a number of applications all on the same machine. Examples could include application or web servers, databases, or mainframes.

In this situation, it may be advantageous to host an additional QM on such machines, and connect it to target QMs by making it part of a cluster, or adding distributed queuing channels. Applications then connect to this QM using local bindings, and communication can be optimised by tuning the sender/receiver channels to the target QMs (see section 4.4.4 - Optimising QM to QM channel communication). This solution has the added advantage that the MQ messaging operations performed by the applications will complete as soon as the message is queued on the local QM; they are then free to continue working without having to wait for a slow network communication to complete.

4.4.3 Communicating between regions

Often it may be the case that an MQ infrastructure spans multiple regions or geographic locations. The network within each of these regions may be fast and reliable, while communication between regions is less so, possibly travelling over the Internet. It may help for MQ inter-region messaging to be amalgamated by one or more "gateway" QMs in each region. Applications connect to their local gateway, which then handles sending messages to other regions using distributed queueing or clustering. These channels of

communication are then open to optimisation by the techniques mentioned in section 4.4.4.

4.4.4 Optimising QM to QM channel communication

For communication between queue managers, there are a number of things that can be done administratively to improve performance:

- Adjusting the *BATCHSZ*, *BATCHINT*, and *NPMSPEED* parameters of the channel definitions
  – see www.ibm.com/support/knowledgecenter/SSFKSJ_9.1.0/com.ibm.mq.ref.con.doc/q081660_.htm
- Setting *PipeLineLength=2* in the *CHANNELS* stanza of the QM's *qm.ini* file
  – see www.ibm.com/support/knowledgecenter/SSFKSJ_9.1.0/com.ibm.mq.ref.con.doc/q080700_.htm
- Using multiple channels – see section 5.5

Please note, however, that the finaloption may result in messages arriving on their destination queues in a different order than they were originally put. If message ordering must be maintained, then while multiple channels may still be used, each subset of messages that must keep their order (e.g. all those from the same client application) must be configured to use the same channel for communication.

4.5 Ultra-High Message Rates

If other hardware factors (such as network and I/O) are not the constraint, but message rates are very high, then eventually the maximum rate at which messaging operations can be performed on a single queue will be reached. The exact rate achievable in a set-up will depend on several factors, including the pattern in which messages are put/got, the type of message selection, if any, and the size of messages. This limitation occurs because certain parts of most messaging operations require exclusive access to certain data about the queue. We call this "queue lock" (see section 5.3).

Queue lock can be mitigated in certain ways by upgrading hardware capacity and altering the configuration of the queue manager:

- Faster processor cores (or more exclusive access to them by MQ) may often help, but more cores often won't, unless there is other work being done by the machine that can be offloaded onto these additional engines.
- If queue buffers are already occupying all the available physical memory on the machine (see section 6.1.1.1), then adding more RAM may help. However, it would be advisable to reconsider the design of the infrastructure, as MQ performs best with short queues: it is not intended to be used as a message store, except for brief periods in exceptional circumstances.

4.6 Maximising the throughput of a single queue

There are several measures that can be taken to ensure that a queue operates as efficiently as possible.

4.6.1.1 Ensure there are "waiting getters"

If an application requesting to get a message off of a queue makes this request before a suitable message arrives, then when one does arrive, it is passed straight to the waiting

application without being written to the queue. As well as reducing the total path length required in handling the message, this has the added advantage that the getting application doesn't have to take the queue lock; all required data are updated by the application that puts the message on the queue, so less expensive lock negotiation is required. This only applies to messages outside of syncpoint control.

Unless it is necessary to process all messages from a queue in a single thread (for order integrity reasons), it is therefore advantageous to have a number of application threads receiving messages from a queue. It is far better to have "too many" (applications waiting in *MQGET* and not ever doing anything) than too few (messages waiting on a queue for getters to become available).

4.6.1.2 Be selective about message selection

Sometimes, receiving applications may only be interested in a subset of the messages that might appear on a queue. MQ caters for this with message selection. However, message selection can dramatically increase the amount of time a receiving application spends browsing a queue, holding the queue lock, and thus preventing other applications from accessing it.

If message selection can instead be replaced by putting different categories of message to different queues, such that receiving applications can get messages from a queue in an indifferent manner, then this should be preferred. It is often easier to make this design decision in the first place, rather than trying to retro-fit it to an infrastructure that's become overloaded, especially since the performance impact of having additional queues is minimal.

If message selection must be used, try to rely mainly on MQ's built in *Correlation ID* field. Selection by Correlation ID has been optimised within the queue manager, so that getters spend far less time than usual finding a suitable message. It also allows for the advantages afforded to "waiting getters", as described in section 4.6.1.1.

4.6.2 Adding queues to increase throughput

The current trend for hardware, or cloud-based systems, to get wider (more cores) rather than taller (faster cores) means can you cannot rely on hardware upgrades to satisfy future increases in demand on your MQ infrastructure. Once the throughput limit of a queue has been reached, there is little that can be done to increase capacity further. It is therefore important to design systems and applications where high demand is anticipated to be able to use multiple queues if necessary. Splitting messages by their purpose, as described in 4.6.1.2, is probably the first option that should be considered.

If the purpose (or "topic") of messages that an application will produce or consume is not known in advance, or if the distinction between message purposes is complex, then using publish-subscribe messaging, rather than simple queueing, may be advantageous. Then, messages of different topics can be optimised administratively in MQ, and consuming applications can still share the workload of a particular topic by attaching them to the subscriber queue of a single administrative subscription object.

Furthermore, it may be possible to reorganise the topic space to handle increased demand without requiring a change to the applications.

Beyond these options, scaling a system across multiple queues transparently (without the need to manually assign queues to applications) is difficult, but can be achieved in certain ways (see section 4.6.6).

4.6.3 Transparent Scalability

Designing an MQ infrastructure that scales to accommodate future (or present) demand is something that needs careful planning from the beginning. The hardware factors that normally constrain MQ infrastructures (disks and network) are often infeasible to upgrade further (make faster), and adding more of them to split workload presents administrative challenges. Traditional methods of load-balancing networked communication by using intermediaries is often not feasible when the assured delivery guarantees that MQ makes are required: the contention just shifts from the original servers to the load-balancer.

When increasing demand requires additional instances of the objects involved in an infrastructure (machines, queue managers, queues), as discussed in the previous sections of this chapter, it is often desirable to do this in the most transparent way possible. Briefly, we'll discuss some of the available techniques to do this, requiring the smallest possible change to the peripheral parts of a network, and instead focusing alterations on the core, so that changes can be administered in as few places as possible.

4.6.4 Scaling across machines & queue managers

From the perspective of an MQ client, there is little distinction between a queue manager and the machine it's hosted on. Hence, this section will cover options for scaling across either as of they were the same thing.

4.6.5 Clustering

One technique that's often attempted to achieve transparent scalability is to put the relevant queues and queue managers into an MQ cluster. The hope is that then the demands on networking or I/O will be spread across all the machines and log file-systems involved in the cluster, and allow additional capacity to be added simply by adding more queue managers to the cluster.

This can certainly help with relieving queue lock (see 4.6.6.1 below), or CPU contention caused by heavy processing of messages by locally-bound receiving applications. It can also relieve the load placed on systems by MQ itself, but only if application connections are spread across the queue managers in the cluster. Having all input applications connect to a single "gateway" queue manager, as is often desired to simplify the configuration of applications across an organisation, simply shifts contention to this gateway. The gateway queue manager will have the same requirements on its network and I/O interfaces as would a single server handling all messages itself.

It's also worth noting that using clusters increases the latency of messages, and often results in a higher overall use of hardware resources than pushing the same number of messages through a single QM (if possible). Network bandwidth is consumed for each QM that a message passes through, and persistent messages will need to be written to the logs of each QM they touch as well.

4.6.5.1 CCDTs and application logic

An alternative (or complement) to using a cluster is to have applications connect using a *Client Channel Definition Table*[c]. This allows applications to select a QM to connect to from a defined group at connection time. Thus, application connections will be spread across all queue managers in that group, which could be a cluster, or could just be a set of queue managers, each having the required queues defined individually.

CCDTs are not suitable when XA distributed transactions will be involved, and may not always result in the best balancing of workloads. For this, a small amount of application logic, which can be redistributed across an organisation as part of a library, is probably preferable. The 3-part IBM developerWorks article found at the following URLs discusses these options in detail, and provides some example code:

1. www.ibm.com/developerworks/websphere/library/techarticles/1303_broadhurst/1303_broadhurst.html
2. www.ibm.com/developerworks/websphere/library/techarticles/1308_broadhurst/1308_broadhurst.html
3. www.ibm.com/developerworks/websphere/library/techarticles/1403_broadhurst/1403_broadhurst.html

4.6.6 Scaling across queues

If splitting a workload across multiple queues is sufficient to relieve contention (normally because queue lock is the contending factor), then in addition to the techniques mentioned in section 4.6.4 (which will work for queues by defining an instance of the queue on multiple queue managers), a few additional options are available.

4.6.6.1 Clustering

Clustering can more easily be used to relieve queue contention than other constraints (see section 4.6.5). Although the use of multiple queue managers is still required (possibly all on the same machine), a single gateway queue manager can be used to simplify application connection, provided this gateway is at MQ v7.5 or later: using multiple cluster transmission queues[d] on the gateway (i.e. one per destination QM) allows workload to be split across multiple queues as soon as it enters the cluster.

4.6.6.2 API exits

An alternative to using a cluster to spread workload across queues is to write a small custom MQ API Exit program. This is a piece of code attached to the queue manager that can intercept MQI calls. For example, to could listen for calls to *MQOPEN* a particular queue. When such a call is intercepted, the queue being requested is silently swapped for an alternative from a pre-defined pool. As such, applications can all make identical

[c] www.ibm.com/support/knowledgecenter/SSFKSJ_9.1.0/com.ibm.mq.dev.doc/q027490_.htm
[d] www.ibm.com/support/knowledgecenter/SSFKSJ_9.1.0/com.ibm.mq.pla.doc/q118600_.htm
www.ibm.com/developerworks/community/blogs/messaging/entry/Multiple_cluster_transmission_queues_what_why_and_how?lang=en

requests to open the queue, while being transparently balanced across a number of them.

Although this requires more work to implement than setting up a cluster, it has the advantage that everything is done with a single queue manager, making maintenance simpler. It also requires less hardware resource to run, will have slightly lower latency, and since the logic of how the load balancing occurs is up to the code in the API exit, it can be made as simple or as complex as necessary.

The details of writing API exists are beyond the scope of this document, but as a place to start, one can study the API exit sample program, *amqsaxe*, and read the associated documentation at www.ibm.com/support/knowledgecenter/SSFKSJ_9.1.0/com.ibm.mq.dev.doc/q027920_.htm.

4.7 Testing for Performance

It is important to thoroughly test any new MQ infrastructure prototype for performance before going into production. This section mentions a few best practices and common pitfalls to avoid to make your testing more effective.

4.7.1 It's a dress rehearsal, not an audition

Performance testing should be done in an environment as close to that of production as possible. This means the same – or very similar – hardware, operating systems, storage systems, network topology, bandwidth, congestion, message size, message rate, protocol, MQ configuration, application messaging profile, etc.

It is easy to underestimate the impact small changes to configuration can have, or think that one can "compensate" for the lack of a feature in the test environment by requiring better-than-actual performance without it. This is usually not the case, and if an infrastructure can perform as well as required in a test environment with a constraining feature missing, then any further improvements may have no impact on the complete environment. Obviously, if a cut-down environment *doesn't* perform as required, then adding a constraining factor is unlikely to improve performance, so there may be some merit in investigating and attempting to widen the bottlenecks that already exist.

However, in some cases, apparently simpler configurations actually degrade performance, leading to unnecessary investigations of an issue that would never have occurred in production. Two of the most common examples of this are:

1. Persistent messaging outside of syncpoint (when it would be inside in production) (see section 4.2.1).
2. Putting a batch of messages all at once, then getting them, or relying on message expiry to remove them from the queue (in production, putting and getting would happen concurrently). This prevents optimisations for "waiting getters" from being used (see section 4.6.1.1) and could also lead to slower operation due to having to access disks instead of memory (see section 6.1.1.1).

4.7.2 Know your tools

When testing a messaging infrastructure, production applications are often swapped out for performance-oriented simulators or "harnesses". This is fine, but it's important to know how the application you've decided to use works. Where possible, use one of the following:

- An IBM-produced tool, such as the freely available JMSPerfHarness [e] or MQ-CPH [f]
- An application developed in-house, designed to behave similarly to the applications it's simulating.
- An open-source project

With all of these, the key factor is the ability to know exactly how the application is behaving internally. This is important, as we have discovered that many such applications use MQ in a sub-optimal way. For instance, some tools cannot make proper use of syncpoint transactions, which seriously impacts the performance of persistent messaging scenarios.

4.7.3 When it doesn't work, simplify

Despite what was said in section 4.7.1, when an infrastructure doesn't perform as required, your first question should not be:

> *"What can I change to try to make things better?"*

Instead, you should ask:

> *"How can I determine what's constraining my system?"*

Often, the easiest way to do this is to start *removing* features of your system to see which is acting as a bottleneck to performance. Even if those features are necessary for functional reasons, knowing where the problem lies will help to tackle the first question more efficiently.

For instance:

- If using persistent messaging, try non-persistent. If things don't improve, then the issue is not with the logging or disks, so there's no need to investigate those.

---

e www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=1c020fe8-4efb-4d70-afb7-0f561120c2aa

f www.ibm.com/developerworks/community/blogs/messaging/entry/MQ_C_Performance_Harness_Released_on_GitHub?lang=en

- If using networked file-systems, switch to local disks. If an improvement is seen, then the problem likely lies in the latency or bandwidth of the networked file-system.
- If using a cluster, drop back to a single QM. If performance divided by the number of QMs improves dramatically, then something about the cluster configuration is holding the system back.

Keeping detailed, comparable records of how your system performs in various configurations can often help lead to a swifter resolution in the event that IBM assistance is needed.

4.7.4 Test your environment

When trying to establish the cause of a problem, it often helps to know how the performance of MQ compares to the theoretical limits of the system on which it's running. Stress and latency testing tools exist for CPU, I/O, and networking on most operating systems. These should be used to test all aspects of an infrastructure before you waste time trying to "tune" your way out of a hardware limitation.

4.8 Memory Requirements.

The memory requirements of MQ are heavily dependent on the nature of the workload and the objects defined to the queue manager. Modern server class machines should not encounter memory constraints but injudicious setting of very large queue buffers across multiple heavily used queues might cause a problem, for instance.

Monitor your system to ensure that you are not paging. Paging *out* of memory is not necessarily a problem, but does indicate that the system is nearing memory exhaustion. Once the 'working set' of the system (i.e. the pages of allocated memory that are frequently accessed) exceeds the physical memory available then performance will be significantly impacted.

Factors affecting memory use of MQ:

- Number of objects in use on the queue manager
- Number of client channels attached to the queue manager
- Queue depth, up to the buffer size for each queue

5 How Am I Constrained?

A common question that is often asked of the MQ Performance team is:

"What is limiting my messaging rate?"

Understanding what might be restricting your messaging rate can help to improve your messaging performance (by eliminating or reducing the constriction) or provide information to assist in planning future migration/expansion of your messaging infrastructure.

5.1 Disk Contention

For persistent messaging the likely reason that you are limited is due to the performance of the disk subsystem. This could be due to the raw speed (or lack of it) of the disk system, disk configuration, configuration of MQ logging or other system usage of the disk subsystem.

Use this checklist to assist in checking your disk contention:

- Increase the number of producers/consumers to increase concurrent messaging workload
- Use disk tools (iostat/TaskManager) to determine disk utilisation and increase speed/capacity/cache size as required to provide increased disk capacity
- Check other processes are not utilising the same disk systems
- Check log configuration (http://www.ibm.com/developerworks/websphere/library/techarticles/0712_dunn/0712_dunn.html)

It can often be useful to run a simple scenario (put/get across single queue) to understand the typical maximum persistent throughput through your disk system before deploying a more complex production scenario.

This test from a locally bound request/responder scenario (MQ V8), illustrates the maximum throughput across a pair of request/reply queues on the server hardware under test.

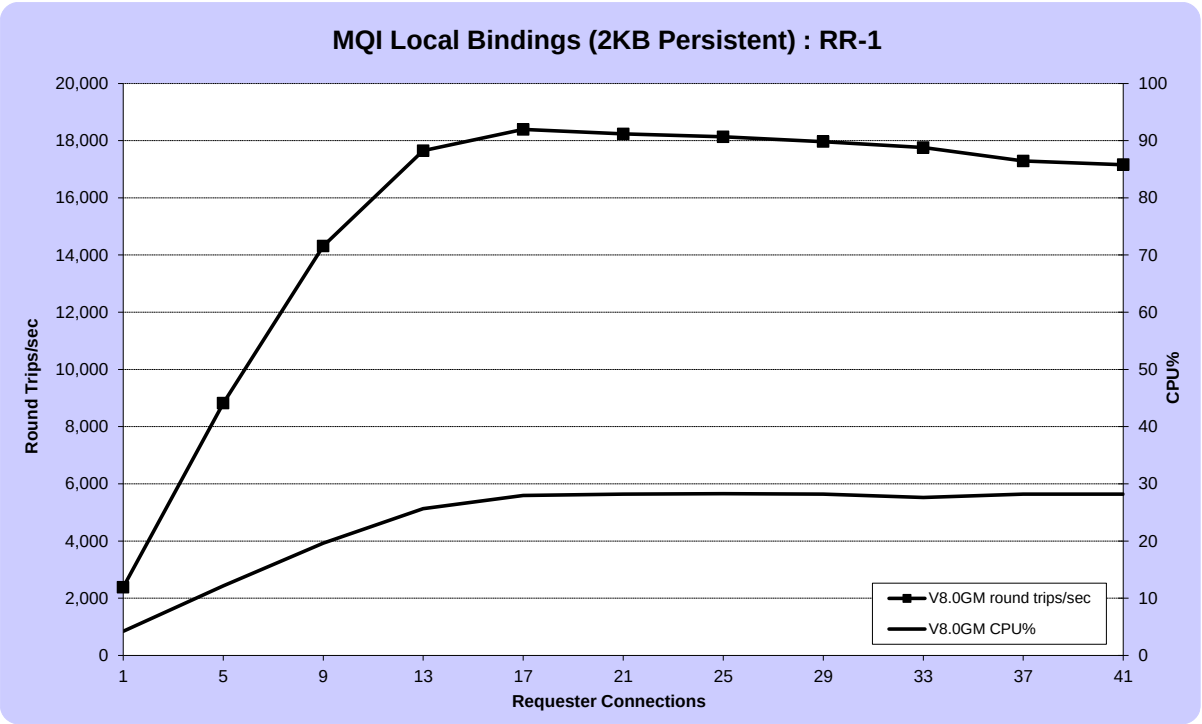


FIGURE 1 - PERFORMANCE RESULTS FOR MQI LOCAL BINDINGS (2KB PERSISTENT)

Here are some suggestions to consider when looking to improve persistent performance:

- Use fast disk sub-systems
- Locate log file on its own filesystem/disk
- Use battery-backed write back cache where possible
- Use circular logs unless linear logs are required for auditing/recovery
- Send/receive messages within syncpoint
- Use separate queues for persistent and non-persistent workloads
- Use separate channels for persistent and non-persistent workloads
- Use separate queues for out-of-band message sizes

5.2 CPU Saturation

For any type of messaging, if you have saturated the CPU, there is often little you can do to increase performance without changing hardware (or allocating more Cores in an LPAR/VM environment). Most operating systems provide a simple way of monitoring CPU usage (vmstat/top/TaskManager). If MQ processes are shown as the highest CPU consumers, it is likely that you have encountered the messaging limit on this piece of hardware.

It is possible that a badly performing application can utilise large amounts of CPU whilst interoperating with MQ (i.e. constantly polling the QM, or simply performing extensive business logic), and you would likely see non-MQ applications high on the list of CPU consumers. These processes should be investigated to determine if their behaviour is expected.

There are also a number of strategies to help reduce CPU when developing your MQ applications:

- Don't poll the QM and use sensible timeouts when receiving messages from MQ
- MQ performs optimally when the queue depth is 0 and there are consumers waiting to consume messages that are delivered to that queue
- Use *FASTPATH* channels/applications where there is no (or low) risk of QM corruption

5.3 Queue Lock Contention

There can be scenarios where you cannot use all of the CPU available and increasing the numbers of producers/consumers has no affect. If the scenario is only using a single or small number of queues, you may find that you are encountering queue lock. This can occur in scenarios where extremely high messaging throughput is distributed across a small set of queues You can see how this scenario is encountered in the non-persistent version of the request/responder scenario (peaking at just over 120,000 messages/sec across a pair of request/reply queues):

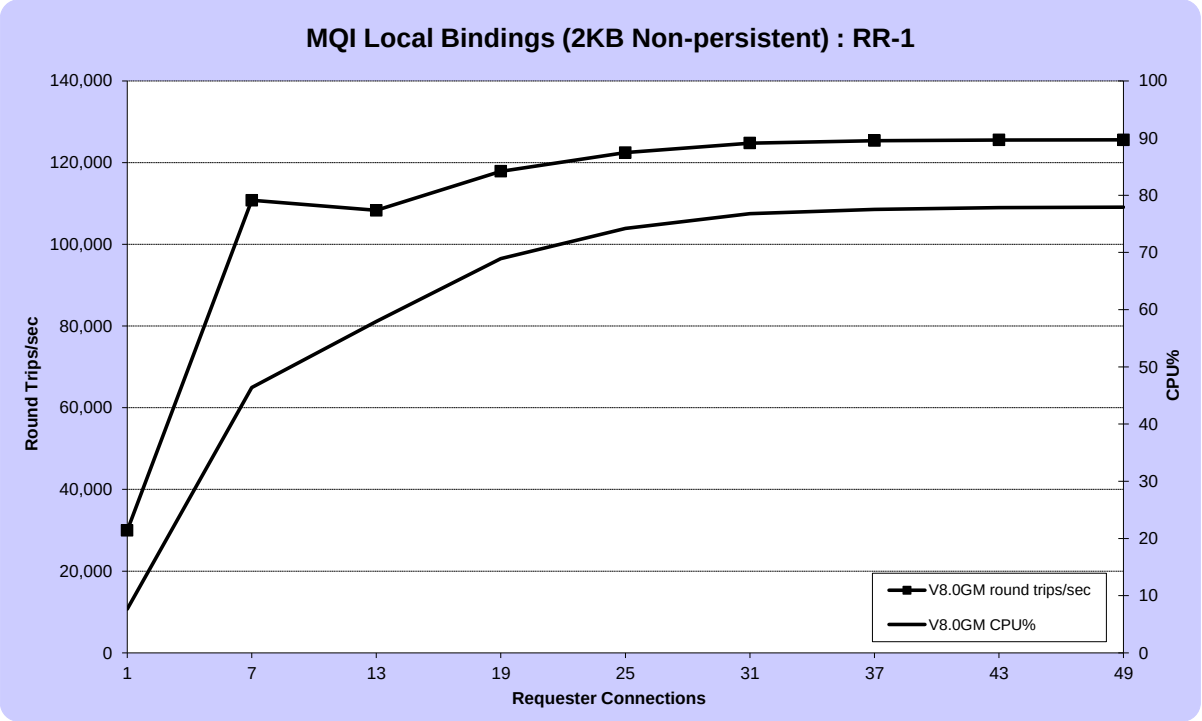


FIGURE 2 - PERFORMANCE RESULTS FOR MQI LOCAL BINDINGS (2KB NON-PERSISTENT)

Queue lock contention is more likely to manifest itself on machines with more cores due to the single-threaded nature of the bottleneck.

To confirm that you are encountering queue lock and not an effect of your client applications or QM configuration, the 'MQ Local Queue Manager Debug Utility' (amqldmpa) tool can identify this behaviour:

```
amqldmpa -m <QM Name> -q <Q Name> -s <interval secs> -n <interval count> -f
<output file> -c A -d 3

i.e. amqldmpa -M QMTEST -q INPUTQ -s 10 -n 6 -f C:\ldmpa.out -c A -d 3
```

From the ldmpa output, check the following values during the period of collection:

```
hmtx.RequestCount        102954894
hmtx.WaitCount           22227930
```

The *WaitCount*, as its name suggests, is the number of times that access to the queue was initially denied and a thread had to wait before processing its work. As soon as this value reaches 10% (or higher) of the *RequestCount*, application performance will start to suffer. Distributing this work over a set of queues would alleviate this contention.

5.4 Network Saturation

Network saturation can be diagnosed by the use of monitoring tools such as dstat/TaskManager. If these tools show that you are encroaching on the bandwidth limits of the network, you may need to increase your networking capacity to 1Gb/10Gb/40Gb as appropriate or add additional network cards.

If you do not have access to these tools or direct access to the hardware, you can calculate approximate messaging limits for a requester/responder scenario with local responders given your networking environment. We can estimate how many 20KB messages we can flow through a scenario per second in a 10Gb environment with a remote producer and local consumer[g]:

10 Gigabit network = 10 x 1024 * 1024 * 1024
= 10,737,418,240 bits
/ 8 = 1,342,177,280 bytes
* 80% = ~ 1,073,741,824 bytes (approx maximum switch throughput)
/ 20680 = ~ 51,922 msg (each message approx 20480 + 200 byte hdr)

[g] This calculation assumes an operational limit of 80% of the nominal bandwidth of the network adapter. The achievable throughput is limited by additional transport layer overheads and will vary by platform and network adapter.

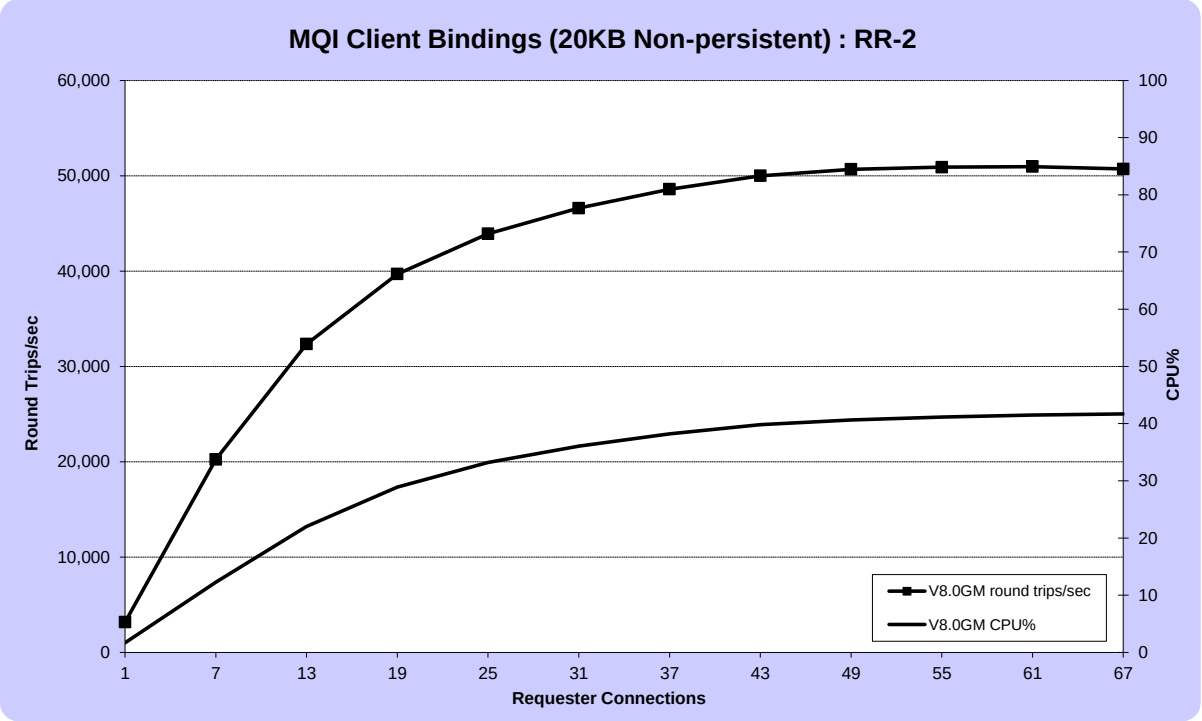
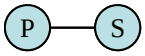


FIGURE 3 - PERFORMANCE RESULTS FOR MQI CLIENT BINDINGS (20KB NON-PERSISTENT)



Note that most switches support full duplex mode so the stated switch capacity is for each direction at the network level; if the consumers were located remotely, then the maximum theoretical throughput as calculated above would need to be halved since network traffic has been doubled.

5.5 Channel Saturation

Channel saturation can occur when high throughput messaging takes place between two Queue Managers connected by a single (or small) number of channels. To determine if you are approaching channel saturation, use the technique in section 5.3 to evaluate the wait for the queue lock of the transmission queue.

Some things to try before adding additional channels:

- Increase batch size to reduce the ratio of acknowledgement flows to transmitted messages.
- Use channel attribute *NPMSPEED=FAST* for non-persistent messaging to transfer messages without the use of transactions.

If you are sending a mix of message sizes, you might consider sending very large messages on their own channel to avoid incurring any delays in the transmission of the smaller size messages.

If you find that you are encountering channel saturation, adding more channels can provide increased performance. Figure 4 below shows a 2KB distributed queuing non-

persistent test (MQ V8), when using a single and 10 channel configuration. It can be seen that in this case the use of multiple server channels enabled MQ to process significantly more messages and more fully utilise the server CPU.

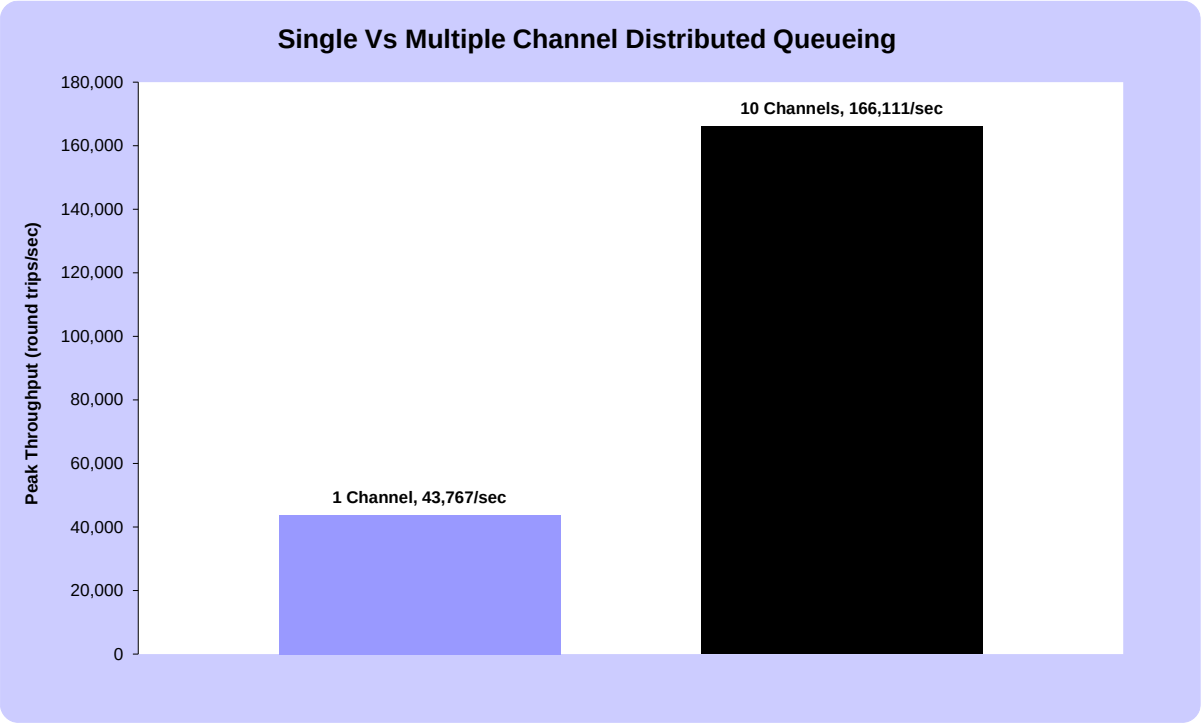
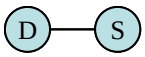


FIGURE 4 - SINGLE CHANNEL VS MULTI CHANNEL DISTRIBUTED QUEUING



Multiple server channels are not a panacea in distributed queuing however. In our test environment they did not provide a significant benefit for persistent messaging as the i/o subsystem was already the bottleneck in that case. Larger message sizes did not gain as significantly either (another reason to consider the separation of messages across different channels based on message size)

6 Tuning Recommendations

6.1 Tuning The Queue Manager

This section highlights the tuning activities that are known to give performance benefits for MQ

V8.0. Note that the following tuning recommendations **may not** necessarily **need** to be applied, especially if the message throughput and/or response time of the queue manager system already meets the required level.

Some tuning recommendations that follow may degrade the performance of a previously balanced system if applied inappropriately. Carefully monitor the results of tuning the queue manager to be satisfied that there have been no adverse effects.

Customers should test that any changes have not used excessive real resources in their environment and make only essential changes. For example, allocating several megabytes for multiple queues reduces the amount of shared and virtual memory available for other subsystems, as well as over committing real storage.

If several changes are to be made, it would be prudent to test the impact of each change individually. This allows one to establish exactly which changes are providing a benefit (some may even degrade performance), which may also in turn provide useful information about how the system is constrained, and so how it may be improved further.

Note: The 'TuningParameters' stanza in the queue manager's qm.ini, or the MQ installation's mq.ini, is not a documented external interface and may be changed or be removed in future releases.

6.1.1 Queue disk, log disk, and message persistence

Non-persistent messages are held in main memory, spilt to the file system as the queues become deep and lazily written to the Queue file. Persistent messages are synchronously written to the log and are also periodically flushed to the Queue file.

To avoid potential queue and log I/O contention due to the queue manager simultaneously updating a queue file and log extent on the same disk, it can help if queue files and logs are located on separate and dedicated physical devices. Storage requirements may be fulfilled by the use of a Storage Area Network (SAN), but multiple high volume queue managers can require different logical volumes, and potentially individual interface pathways to avoid congestion.

With the queue and log disks configured in this manner, careful consideration must still be given to

message persistence: persistent messages should only be used if the message needs to survive a queue manager restart (forced by the administrator or as the result of a power failure, communications failure, or hardware failure). In guaranteeing the recoverability of persistent messages, the path length through the queue manager is significantly longer than for a non-persistent message. This overhead does not include the additional time for the message to be written to the log which will depend on the write speed of the i/o device (including any caching at the hardware level).

6.1.1.1 Non-persistent and persistent queue buffer

The default non-persistent queue buffer size is 128K per queue and the default persistent is 256K per queue for MQ on a 64-bit operating system. These can be increased up to 100MB using the *DefaultQBufferSize* and *DefaultPQBufferSize* parameters in the *TuningParameters* stanza of the *qm.ini* file. (For more details see SupportPac MP01: MQSeries – Tuning Queue Limits). Increasing the queue buffer provides the capability to absorb spikes in message throughput at the expense of real memory. Once these queue buffers are full, the additional message data is written to the file system that will eventually find its way to the disk (operating system buffers will be utilised for queue files).

Defining large queue buffers is *not* a solution to avoiding disk I/O when the input message rate to MQ exceeds the output rate, for whatever reason, for sustained periods. The buffer will fill and spilling will occur.

Defining queues using large non-persistent or persistent queue buffers can degrade performance if the system is short of real memory either because a large number of queues have already been defined with large buffers, or for other reasons - e.g. large number of channels defined.

Note: The queue buffers are allocated in shared storage so consideration must be given to whether the agent process or application process has the memory addressability for all the required shared memory segments.

Queues can be defined with different values of *DefaultQBufferSize* and *DefaultPQBufferSize*. The value is taken from the *TuningParameters* stanza in use when the queue manager was last started.

6.1.1.2 Logging

6.1.1.2.1 Log type

The log component is often the bottleneck when processing persistent messages. Sufficient information is stored in the log to restart the queue manager after failure.

Both circular and linear logging are sufficient to recover from application, software, or power failure whilst linear logging can also recover from media failure resulting in the loss of a queue file (assuming some historical copy of the queue file exists along with the linear logs to perform a forward recovery from that point).

Circular logging was historically preferable from a performance perspective as no time is required to allocate and format new log extents or to delete or archive them with circular logging[h]. Performance of linear logging has been improved however, from V9.0.2 (rolled up into V9.1) through the introduction of automatic management of log extents and image copies[i]. With these enhancements, performance is not a significant factor in deciding between circular or linear logging.

6.1.1.2.2 Log buffer

Log records are written to the log buffer at each *MQPUT* and *MQGET* of messages outside of syncpoint, and each *MQCMIT*. This information is synced onto the log disk. Periodically the checkpoint process will decide how many of these log-file extents are in the active log and need to be kept online for recovery purposes. Those extents no longer in the active log are available for archiving when using linear logging or available for reuse when using circular logging. There should be sufficient primary logs to hold the active log plus the new log extents used until the next checkpoint, otherwise some secondary logs are temporarily included in the log set and they have to be instantly formatted which is an unnecessary delay when using circular logging.

The log buffer is a circular piece of main memory where the log records are concatenated so that multiple log records can be written to the log file in a single I/O operation. The default values used for *LogBufferPages* are probably suitable for most installations. The default size of the log buffer is 512 pages with a maximum size of 4096 pages.

To optimise the throughput of large persistent messages (> 1MB) *LogBufferPages* could be increased to improve the likelihood of messages only needing one I/O to get to the disk. Environments that process under 100 small (< 10KB) persistent messages per second can reduce the memory footprint by using smaller values like 32 pages without impacting throughput.

Changes to the queue manager *LogBufferPages* stanza take effect at the next queue manager restart. The number of pages can be changed for all subsequent queue managers by changing the *LogBufferPages* parameter in the product default *Log* stanza.

[h] www.ibm.com/developerworks/websphere/techjournal/0904_mismes/0904_mismes.html

[i] developer.ibm.com/messaging/2018/08/28/logger-enhancements-mq-v9-0-2/

6.1.1.2.3 Log files

LogFilePages (or *crtmqm –lf <LogFilePages>*) defines the size of one physical disk extent (default 4096 pages). The larger the disk extent, the longer the elapsed times between changing disk extents. It is better to have a smaller number of large extents but a long running transaction can prevent checkpointing from efficiently freeing the disk extent for reuse.

Larger extents reduce the frequency of log switching (permitting a greater amount of log data to be written into one extent) and allow more time to prepare new linear logs or recycle old circular logs (especially important for long running units of work). As an initial target, one log extent should hold at least 10 seconds of log data streaming.

The number of *LogPrimaryFiles* (or. *crtmqm -lp <LogPrimaryFiles>*) can be configured to a large number (the maximum number of Primary plus Secondary extents is 255 for Windows and 511 for UNIX). For circular logging you should configure enough primary logs to cope with expected peak load as secondary logs are formatted each time they are used, so incur a performance penalty,

The active log set is the number of extents that are identified by the checkpoint process as being necessary to be kept online. As additional messages are processed, more space is taken by the active log.

As transactions complete, they enable the next checkpoint process to free up extents that now become available for archiving with linear logging or re-use with circular logging.

Some installation will use linear logging and *not* archive the redundant logs because archiving impacts the run time performance of logging. Instead, they will periodically (daily or twice daily) use '*rcdmqimg*' on the main queues thus moving the 'point of recovery' forward , compacting the queues, and freeing up log disk extents. This approach prevents the continuous build-up of log extents (assuming the old ones are deleted).

6.1.1.2.4 LogWriteIntegrity: SingleWrite or TripleWrite

The default value is *TripleWrite*. MQ writes log records using the *TripleWrite* method because it provides full write integrity where hardware that assures write integrity is not available.

Some hardware guarantees that, if a write operation writes a page and fails for any reason, a subsequent read of the same page into a buffer results in each byte in the buffer being either:

- The same as before the write, or
- The byte that should have been written in the write operation

On this type of hardware (for example, SSA write cache enabled), it is safe for the logger to write log records in a single write as the hardware assures full write integrity. This method provides the highest level of performance.

Queue manager workloads that have multiple streams asynchronously creating high volume log records will not benefit from *SingleWrite* because the logger will not need to rewrite partial pages of the log file.

Workloads that serialize on a small number of threads where the response time from an *MQGET*, *MQPUT*, or *MQCMIT* inhibits the system throughput are likely to benefit from *SingleWrite* and could enhance throughput by 25% but in practice, we see very few customer deployments that have gained significantly from changing this parameter.

Measurements in this report used *LogWriteIntegrity=TripleWrite*

6.1.2 Channels: process or thread, standard or fastpath?

Threaded channels are used for all the measurements in this report ('*runmqlsr*', and for server channels an *MCATYPE* of '*THREAD*') the threaded listener '*runmqlsr*' can now be used in all scenarios with client and server channels. Additional resource savings are available using the '*runmqlsr*' listener rather than 'inetd', including a reduced requirement on: virtual memory, number of processes, file handles, and System V IPC.

Fastpath channels can increase throughput for both non-persistent and persistent messaging. For persistent messages, the improvement is only for the path through the queue manager, and does not affect performance writing to the log disk.

It is not recommended to use fastpath channels when channel exits are being used as any problem with the exit code has the potential to bring the queue manager down.

Note: Since the greater proportion of time for persistent messages is in the queue manager writing to the log disk, the performance improvement for fastpath channels is less apparent with persistent messages than with non-persistent messages.

6.2 Applications: Design and Configuration

6.2.1 Standard (shared or isolated) or fastpath?

There are issues associated with writing and using fastpath applications—described in the 'MQSeries Application Programming Guide'. Although it is generally recommended that customers use fastpath channels, it is not recommended to use fastpath applications. If the performance gain offered by running fastpath is not achievable by other means, it is essential that applications are rigorously tested running fastpath, and never forcibly terminated (i.e. the application should always disconnect from the queue manager).

6.2.2 Parallelism, batching, and triggering

An application should be designed wherever possible to have the capability to run *multiple instances* or *multiple threads* of execution. Although the capacity of a multi-processor (SMP) system can be fully utilised with a small number of applications using non-persistent messages, more applications are typically required if the workload is mainly using persistent messages. Processing messages inside syncpoint can help reduce the amount of time the queue managers takes to write a group of persistent messages to the log disk. The performance profile of a workload will also be subject to variability through cycles of low and heavy message volumes, therefore a degree of experimentation will be required to determine an optimum configuration.

Queue avoidance is a feature of the queue manager that allows messages to be passed directly from an MQ putter to an MQ getter without the message being placed on a queue. This feature only applies for processing messages outside of syncpoint. In addition to improving the performance of a workload with multiple parallel applications, the design should attempt to ensure that an application or application thread is always available to process messages on a queue (i.e. an MQ getter), then messages outside of syncpoint do not need to ever be physically placed on a queue.

Note that as more applications are processing messages on a single queue there is an increasing likelihood that queue avoidance will not be maintainable. The reasons for this have a cumulative and exponential effect, for example, when messages are being placed on a queue quicker than they can be removed. The first effect is that messages begin to fill the queue buffer—and MQ getters need to retrieve messages from the buffer rather than being received directly from an MQ putter. A secondary effect is that as messages are spilled from the buffer to the queue disk, the MQ getters must wait for the queue manager to retrieve the message from the queue disk rather than being retrieved from the queue buffer. While these problems can be addressed by configuring more MQ getters (i.e. processing threads in the server application), or using a larger queue buffer, it may not be possible to avoid a performance degradation.

Processing persistent messages inside syncpoint (i.e. in batches) can be more efficient than outside of syncpoint. As the number of messages in the batch increases, the average processing cost of each message decreases. For persistent messages the queue

manager can write the entire batch of messages to the log disk in one go whilst outside of syncpoint control, the queue manager must wait for each message to be written to the log before returning control to the application.

Only one log record per queue can be written to the disk per log I/O when processing messages outside of syncpoint. This is not a bottleneck when there are a lot of different queues being processed. When there are a small number of queues being processed by a large number of parallel application threads, it *is* a bottleneck. By changing all the messages to be processed inside syncpoint, the bottleneck is removed because multiple log records per queue can share the same log I/O for messages processed within syncpoint.

A typical triggered application performs in the same way as a 'short session' program. The '*runmqlsr*' program has a much smaller overhead compared to inetd of connecting to and disconnecting from the queue manager because it does not have to create a new process so is more suitable to triggered applications.

When programming a triggered application it may be worth exposing a disconnect interval as an input parameter to the application program. This can provide the flexibility to make tuning adjustments in a production environment to establish the best balance between reducing connection costs or freeing up queue manager and operating system resources.

6.3 Virtual Memory, Real Memory, & Paging

6.3.1 BufferLength

The *amqrmppa* process contains a thread per connected client. The *BufferLength* parameter of the *MQGET* on the client application is also used to allocate a long-term piece of storage of this size in the *amqrmppa* process, in which the message is held before being retrieved by the client. If the size of the arriving messages cannot be predicted then the application should provide a buffer than can deal with 90% of the messages and re-drive the *MQGET* after return code "*2080 (X'0820') MQRC_TRUNCATED_MSG_FAILED*" by providing a larger buffer for retrieving this particular message. There is a mechanism to gradually reduce the size of the storage in *amqrmppa* if the recent *BufferLength* size is significantly smaller than previous *BufferLength*.

For messages encrypted with AMS it is advisable to set a buffer *larger* than the plaintext version of the largest expected message as the encrypted form will be longer.

6.3.2 MQIBindType

MQIBindType=FASTPATH will cause the channel to run 'trusted' mode. Trusted applications do not use a thread in the agent (*amqzlaa*) process. This means there is no IPC between the channel and agent because the agent does not exist in this connection. If the channel is run in *STANDARD* mode then any messages passed between the channel and agent will use IPC memory (size = *BufferSize* with a maximum size of 1MB) that is dynamically obtained and only held for the lifetime of the *MQGET*. Standard channels each require an additional 80KB of memory. As the message rate increases, there will be more IPC memory used in parallel.

6.3.3 Paging

The memory available on a machine needs to handle the peaks in throughput. It is important to prevent the queue depths increasing if possible, especially if large queue buffers have been set as these will occupy memory and can cause paging in the worst circumstances.

Queue buffers will grow up to the maximum specified but do not shrink back until the queue manager is restarted.