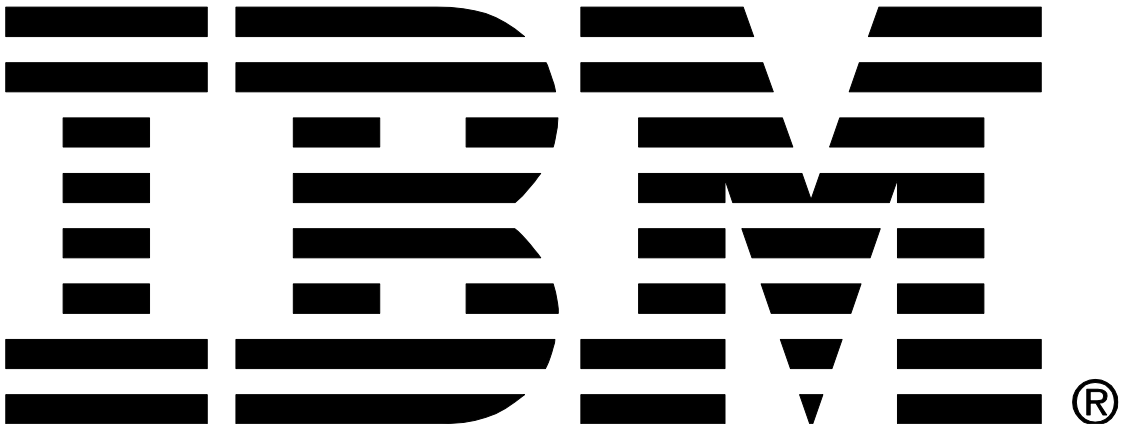


Persistent Messaging Performance in IBM MQ for Linux

Version 1.0 – November 2017

Paul Harris

IBM MQ Performance
IBM UK Laboratories
Hursley Park
Winchester
Hampshire

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, November 2017.

Notices

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 report is intended for architects, systems programmers, analysts and programmers wanting to understand the performance characteristics of *IBM MQ V8.0*. The information is not intended as the specification of any programming interface that is provided by IBM MQ. It is assumed that the reader is familiar with the concepts and operation of IBM MQ V8.0.

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
- **Intel Corporation:** Intel, Xeon
- **Red Hat:** Red Hat, Red Hat Enterprise Linux

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.

Preface

In this paper, I will be looking at MQ message persistence, and the impact it has on performance. Much of the content is generally applicable to MQ, but the data presented here is collected on the MQ V9.0.4 CD release for Linux (tested on RedHat EL).

The paper is split into three parts:

Part One – How Persistent Messaging Works and Best Practices
Part Two – Comparative Performance of File Systems
Part Three – Methodology and Tools

Part one presents an overview of how persistent messaging affects performance, how persistent messages are handled by MQ and what best practices should be employed when using persistent messaging. Some test scenarios are shown to illustrate the impact of different configurations on persistent messaging performance.

Part two presents some comparative data for a range of file systems used to host the MQ transaction log (often the key component when considering persistent messaging performance). These include local storage (HDD & SSD), and remote storage (SAN & NFS). Test results will show the possible impact some of these technologies can have, but the data is not intended to be used to size your solution. The test results illustrate what effect moving from one technology to another *can* have, and what you need to be aware of when hosting an MQ transaction log on filesystems with different characteristics.

Part three summarises the tools used to collect data for this report, along with some recommendations for performance testing persistent messaging applications.

Feedback is welcomed on this report, as it is intended to include additional data in the future, for other filesystems.

Table of Contents

1 Part One – How Persistent Messaging Works and Best Practices

1.1 Bottlenecks

When evaluating the performance of any MQ application we typically measure the message rate, i.e. the number of messages 'processed' by the queue manager in a second. A commonly used analogy for software performance is a view of the components (functional paths) being a series of pipes through which we must travel. The rate at which we can travel through the pipes from one end to the other is restricted by the diameter of the narrowest pipe; the bottleneck.

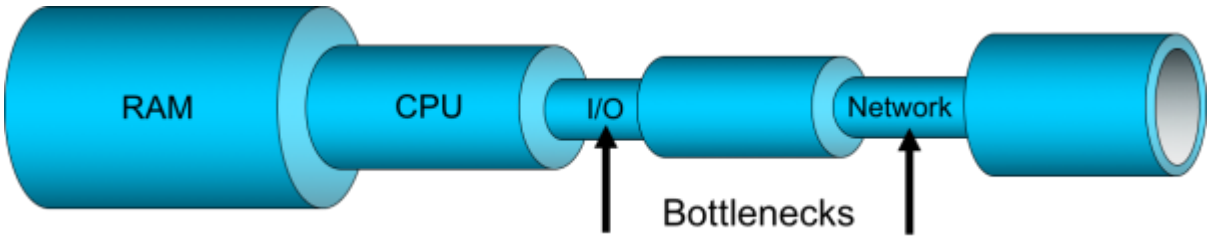


Figure 1 – The Performance Pipeline

In Figure 1 above, upgrading or adding more RAM or CPU resource won't have any effect on the end to end performance if the I/O or network remain unchanged.

It's no accident that the diagram above, used in many presentations to MQ user groups, shows (file) I/O and network (I/O), as the bottlenecks. Traditionally, moving data around, including persisting it to disk, is expensive. All sorts of caching technologies exist at various levels to mitigate this cost. Even moving data in and out of the processor has complex caching strategies (think L1,L2 & L3 caches) to avoid moving data too far, wherever possible.

Of course, MQ is fundamentally concerned with moving data, and is commonly required to do so in an absolutely dependable way (e.g. for messages involved in financial transactions). There is a trade-off between reliability and performance, however. For the best performance, we may think of storing messages in fast, volatile memory, thus avoiding, writing them to the filesystem altogether. If we suffer a machine failure however, the data will be lost, and that is often unacceptable.

The more robust and reliable we make a system, the greater the impact those features have on performance (generally).

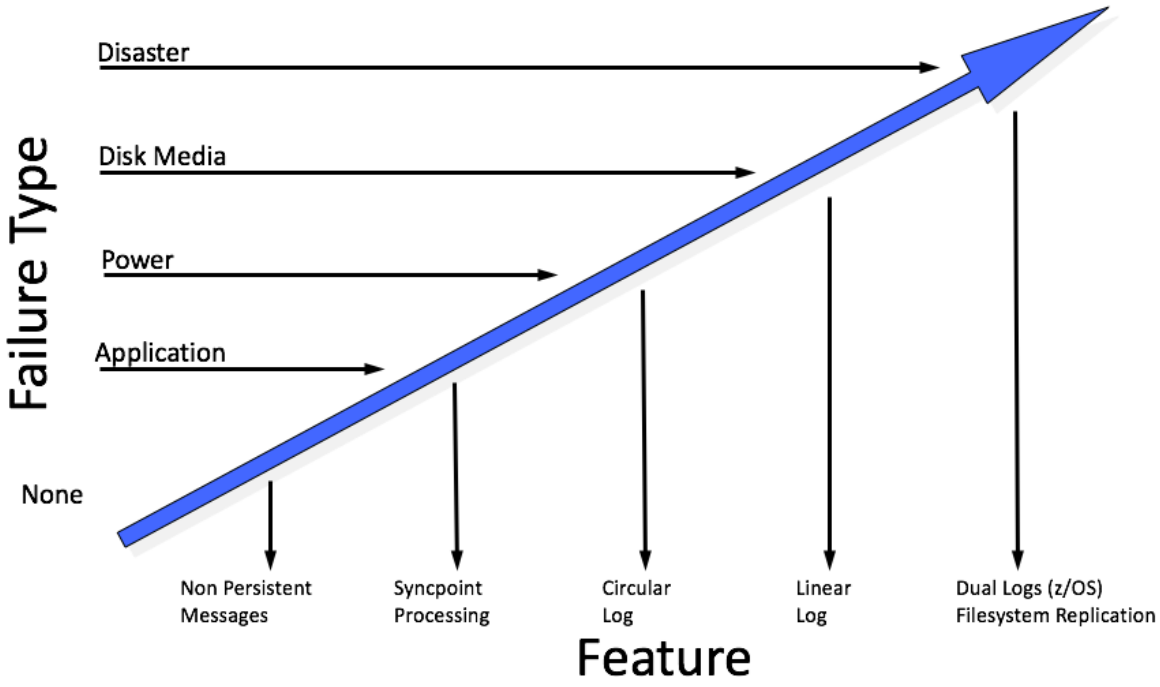


Figure 2

Figure 2 shows features that can be adopted for increasing degrees of reliability. Non-persistent messaging, at one end will be fast, but provides little, or no recovery ability. At the other end, we may have a filesystem (possibly shared, in an HA pair), synchronously replicated, to provide the ability to recover from the loss of a data centre. Unsurprisingly the two approaches will perform very differently. Simply moving to persistent messaging, can have a big impact on performance.

	#Requester Applications	Peak Round Trips	CPU Utilisation
Non-Persistent	60	325,400	100%
Persistent-1 [1]	60	70,435	72%
Persistent-2 [2]	120	87,338	93%

Table 1 - Peak 2K Round trips/Sec on 2x12 Core Server.

Table 1 shows data for a Requester/Responder scenario, where a number of 'requester' applications put 2K messages, across a range of local queues, and a corresponding set of 'responder' applications get the messages of the queues and put a similarly sized, response message onto another set of queues for the original application to get. Each round trip therefore comprises of 2 MQPUTs of a 2K message and 2 MQGETs of a 2K message. The tool used to run this scenario (CPH-MQ) is available on GitHub:
https://github.com/ibm-messaging/mq-cph

[1] Persistent-1 was defined with a 4GB transaction log file set.
[2] Persistent-2 was defined with a 1GB transaction log file set.

There is an associated introductory blog article here:
MQ-CPH Performance Harness Released on GitHub

(https://www.ibm.com/developerworks/community/blogs/messaging/entry/MQ_C_Performance_Harness_Released_on_GitHub?lang=en)

The persistent messaging application is using a circular log, persisting to local HDDs. Switching this workload from non-persistent to persistent (Persistent-1), could degrade the throughput from 325K round trips/sec to ~70K round trips/sec.

The HDD's were fronted by a RAID adapter with a large cache. Depending on the size of the MQ log required, updates to the log files may all be serviced in the RAID write cache, which will give a considerable boost. This is the case in the third row in the table above (Persistent-2), which used a smaller log file set, resulting in a higher peak rate (though this required more applications, to fully utilize the MQ logger, more on this later). This is but one example where you can apparently get different results from the same I/O infrastructure. You can imagine that if there were other processes on the host, also writing to the RAID cache, they may have an impact on MQ's I/O performance.

This may sound obvious, but persistence comes at a price, so if you don't need it, don't use it.

Best Practice #1: Only use persistent messaging when necessary for your application.

1.2 Logging Persistent Messages

From this point forward, I'm going to assume that we *do* need to use persistent, transacted messages, for the purposes of reliability, and recovery.

1.3 How MQ Uses Files to Store Persistent Messages

MQ writes to a number of files during operation. E.g.:

1. Transaction logs
2. Operational logs, traces, and diagnostics (error logs)
3. Queue files

1.3.1 Transaction logs

For persistent messages, the transaction log is used to initially persist the message. The message will be 'hardened' to the queue file during checkpoint processing (where the transaction log and queue files are reconciled), or on the normal shutdown of a queue manager. Writing to the transaction logs is typically the most performance sensitive part of

persistent messaging. The location of the logs can be defined during queue manager creation, using the '-ld' option of crtmqm. By default, they will reside at /var/mqm/log.

All of the tests in this report used circular logs, to demonstrate the performance impact of logging, but you can alos define linear logs, which have been significantly improved in terms of performance, configuration, and management in IBM MQ V9.0.2. See this blog article for details:

Logger enhancements for MQ v9.0.2

(https://www.ibm.com/developerworks/community/blogs/messaging/entry/Logger_enhancements_for_MQ_v9_0_2?lang=en)

1.3.1.1 Calculating the Size of the MQ Transaction Log

Your log needs to be large enough to accommodate your workload, with a suitable contingency for peaks, but do not make it arbitrarily large, as you may not fully benefit from such things as RAID caching.

Best Practice #2: Size your transaction log correctly.

The Knowledge Center for IBM MQ has extensive guidance on this:

Calculating the size of the log

(https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.con.doc/q018470_.htm)

1.3.1.2 Concurrency, Syncpointing, and Queue Locking

The effects of application concurrency (i.e. the number of applications accessing queues), the use of syncpointing, and the level of queue locking, combine to have an effect on the performance of MQ, particularly with regards to logging.

When a persistent message is put onto a queue (outside of syncpoint), or committed (inside of syncpoint), it will be force the write to the transaction log before any other operations are able to act on it (i.e. it becomes 'visible'), ensuring transactional integrity, and recoverability. A forced write means that MQ will not utilize operating system I/O buffers. The point at which the message is written to disk depends on whether the MQPUT is inside an MQ syncpoint, or not (whether MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT is specified in the MQPMO).

Knowledge Center: MQPMO options (MQLONG)

(https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.0.0/com.ibm.mq.ref.dev.doc/q098730_.htm)

If the message is put outside of syncpoint, i.e. MQPMO_NO_SYNCPOINT is set, (or neither option is set in the MQPMO, and the queue manger is not on z/OS), then the message is forcibly written to the transaction log as part of the MQPUT call, guaranteeing that it is on disk before responding to the application. A lock on the queue is held whilst the message is synchronously written to the transaction log.

If the message is put *inside* a syncpoint, i.e. MQPMO_SYNCPOINT is set, then the MQPUT call returns (releasing the queue lock), with no guarantee that the message has been written to disk. Internally, MQ will have copied it to a log buffer (more on that later), waiting to be written to disk. A subsequent MQCMIT call will cause the message to be forced to disk, (it may have *already* been written to disk, effectively piggy-backing on another transaction's call to MQCMIT, in that case, this MQCMIT will have less of its own data to write to the log, as the message data has already been persisted. Moreover, the log write associated with the MQCMIT is executed without holding the lock on the queue (the queue is locked, and updated, at the end of the MQCMIT call, to make the message visible, after the log write has completed).

Making sure that you always issue an MQPUT inside a syncpoint enables MQ to optimize queue locking, where there are multiple applications accessing the same queue.

Note that executing a GET on a persistent message outside of syncpoint is also affected in the same way, but GETs of a persistent message outside of syncpoint do not make any logical sense, as MQ cannot verify that a message arrived at the application after it has been destructively read. A GET of a persistent message should not complete until the application has verified receipt of the message (via a call to MQCMIT).

The following charts illustrate the impact on locking overhead by concurrency, distribution of load across queues, and the use of syncpoint control (transactions).

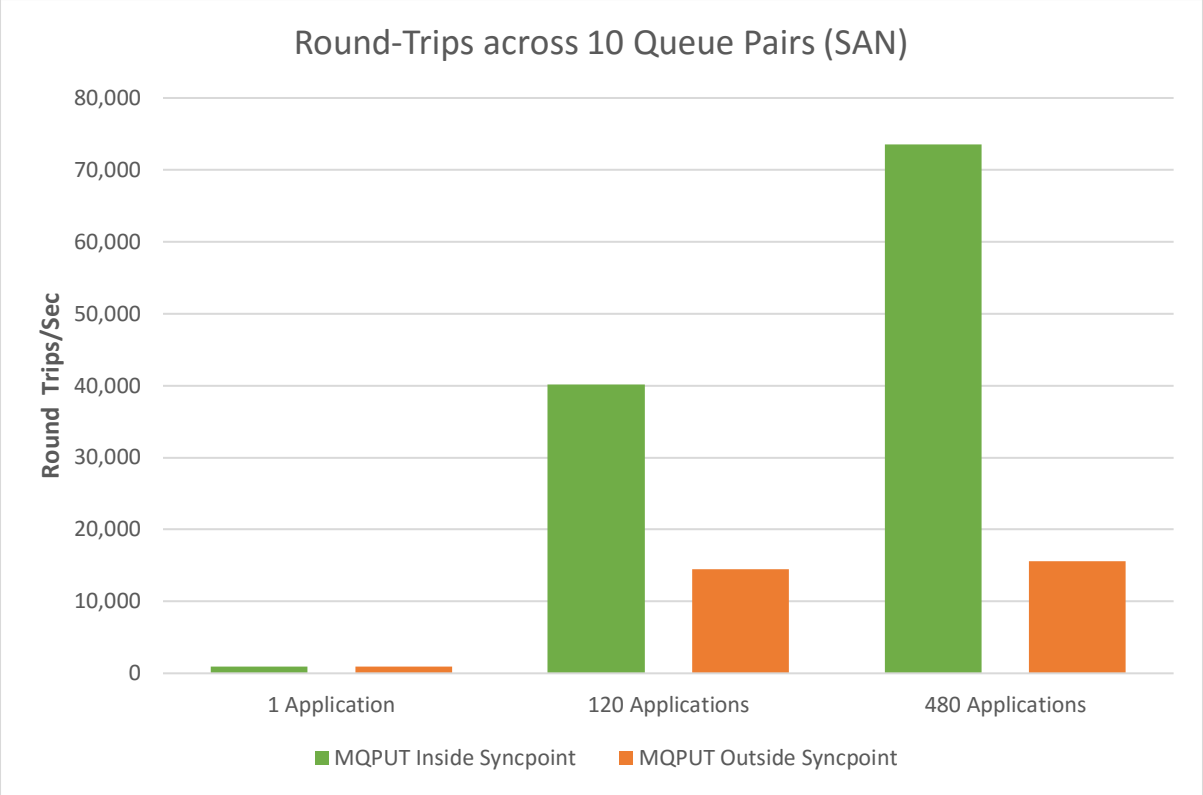


Figure 3- Effect of syncpoint on 10 queue pair workload

Figure 3 shows results for tests where 10 queues pairs are utilised, with an increasing number of requester applications running, processing 2KiB messages. When there is only one application, there will never be another MQPUT being processed alongside that of application 1, so there is little difference between executing the MQPUT inside, or outside of syncpoint, in the application. In fact an MQPUT outside of syncpoint can result in a slightly better peak throughput, because of the elimination of another flow (the MQCMIT) between the application and MQ. Once we add more MQ applications, the benefits of using syncpoints become evident, particularly with a higher latency filesystem, as MQPUTs outside of syncpoint will lock the queue while the log record is synchronously forced to disk. Using syncpoints reduces contention with the added benefit that other applications can write into the log buffer, resulting in more aggregation of log data, in a single write.

Best Practice #3: Concurrency optimizes the MQ logger throughput.
As the number of applications increase, more aggregation of data can occur in the log buffers.

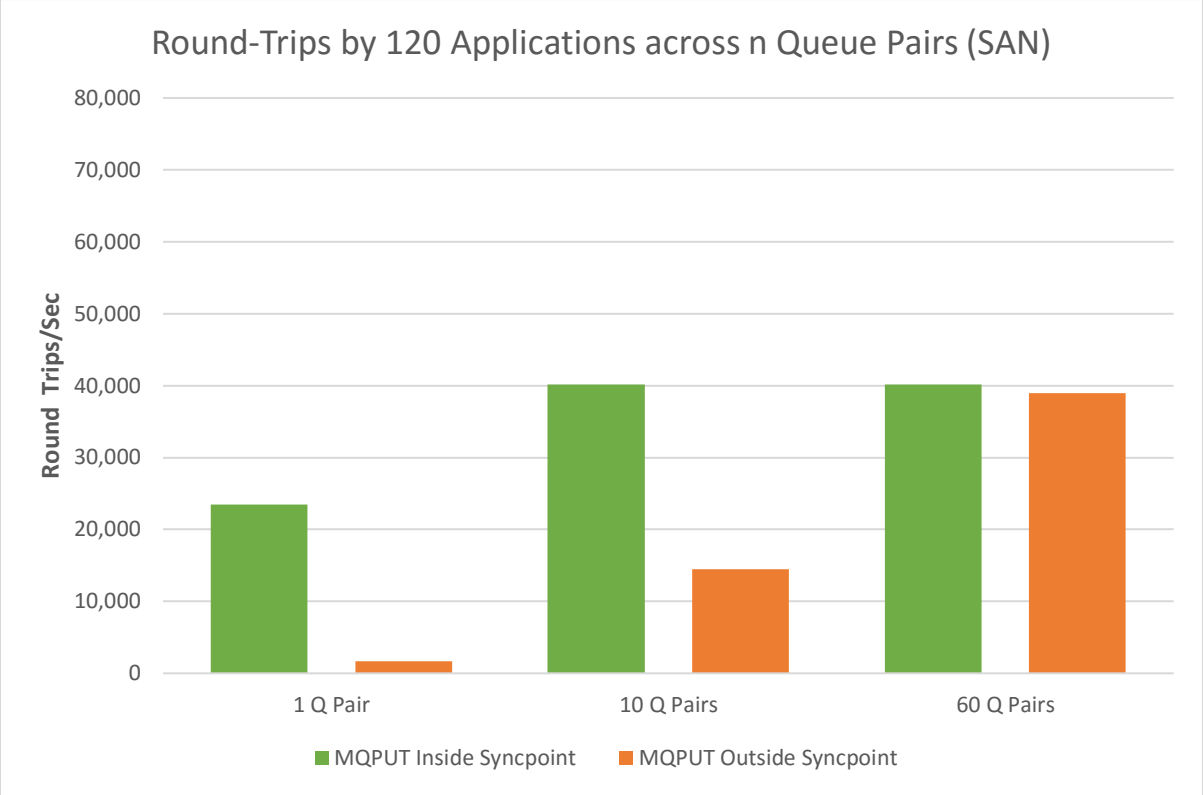


Figure 4 - Effect of syncpoint as q lock decreases

Figure 4 shows the effect of reducing queue locking by spreading the load across a number of queue pairs (REQUEST Q/REPLY Q). All tests use 120 requester applications. When the workload is driven through a single pair of queues, the non-syncpoint case has a low throughput (not much better than the test using 1 requester in chart 1), as each MQPUT queues up behind the previous one to that queue, with a forced log write being executed within the scope of the queue lock. Using syncpoints alleviated this issue, allowing for more concurrency. As we increase the number of q pairs, the locking becomes less of an issue, until, at 60 pairs of queues, where there are only 2 requester applications per queue pair, the non-syncpoint case is not much less than using syncpoints.

Best Practice #4: Spread message load across multiple queue where possible, to alleviate queue locking.

A real-world application cannot usually be designed with performance as its only criteria, but an awareness of what the best practices for performance are, can help.

From MQ V9, queue locking statistics are published to the system monitor topic. The sample program amqsrua can be used to view these (see 3.3.2.1). Using this tool for the test above, we can see the queue locking increase (for request queue REQUEST1) when testing 10 queue pairs with requester MQPUTs inside, or outside of syncpoint. The following commands were initially executed with MQPUTs inside of syncpoint (round trip rate ~4K/sec)

```
[mqperf@mqtesthost]$ /opt/mqm/samp/bin/amqsrua -m PERF0
CPU : Platform central processing units
DISK : Platform persistent data stores
STATMQI : API usage statistics
STATQ : API per-queue usage statistics
Enter Class selection
==> STATQ
OPENCLOSE : MQOPEN and MQCLOSE
INQSET : MQINQ and MQSET
PUT : MQPUT and MQPUT1
GET : MQGET
Enter Type selection
==> PUT
An object name is required for Class(STATQ) Type(PUT)
Enter object name
==> REQUEST1

Publication received PutDate:20171012 PutTime: 10052327 Interval:10.000 seconds
REQUEST1                MQPUT/MQPUT1 count 39899 3990/sec
REQUEST1                MQPUT byte count 81713152 8171132/sec
REQUEST1                MQPUT non-persistent message count 0
REQUEST1                MQPUT persistent message count 39899 3990/sec
REQUEST1                MQPUT1 non-persistent message count 0
REQUEST1                MQPUT1 persistent message count 0
REQUEST1                non-persistent byte count 0
REQUEST1                persistent byte count 81713152 8171132/sec
REQUEST1                lock contention 24.56%
REQUEST1                queue avoided puts 0.00%
REQUEST1                queue avoided bytes 0.00%

. . .

Switch to MQPUTs outside of syncpoint for requester (~14K round trips/sec)

. . .

Publication received PutDate:20171012 PutTime: 10064329 Interval:10.008 seconds
REQUEST1                MQPUT/MQPUT1 count 14605 1459/sec
REQUEST1                MQPUT byte count 29911040 2988557/sec
REQUEST1                MQPUT non-persistent message count 0
REQUEST1                MQPUT persistent message count 14605 1459/sec
REQUEST1                MQPUT1 non-persistent message count 0
REQUEST1                MQPUT1 persistent message count 0
REQUEST1                non-persistent byte count 0
REQUEST1                persistent byte count 29911040 2988557/sec
```

REQUEST1	lock contention 98.98%
REQUEST1	queue avoided puts 0.00%
REQUEST1	queue avoided bytes 0.00%

Best Practice #5: Use syncpoint with persistent messages (even if there is only one MQPUT in your transaction).

1.3.1.3 More on the MQ Logger

As we've seen above, given the right kind of usage, the MQ logger can aggregate messages, optimizing the synchronous writes required to reliably store a persistent message, when it is put on the queue. It does so by utilizing an internal buffer, made up of 4KiB blocks of memory. The size of this buffer can be controlled via the qm.ini parm:

LogBufferPages=0|0-4096

The default setting of 0 will defer the decision to MQ, and result in 512 pages (2MiB), the minimum setting is 128 (512KiB), and the maximum value of 4096 results in a 16MiB buffer. With the amount of memory typically available on modern servers, we recommend that you set this value to 4096.

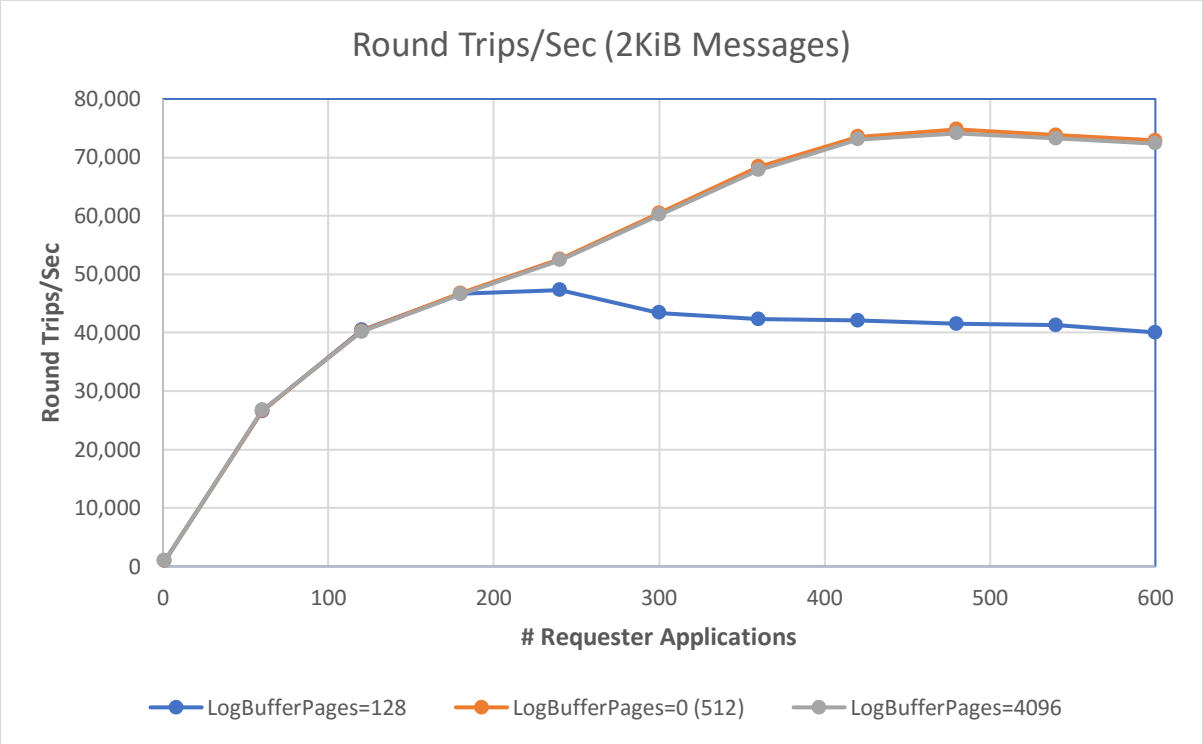


Figure 5 - Effect of LogBuffer Size on 2K Requester/Responder (logging to SAN)

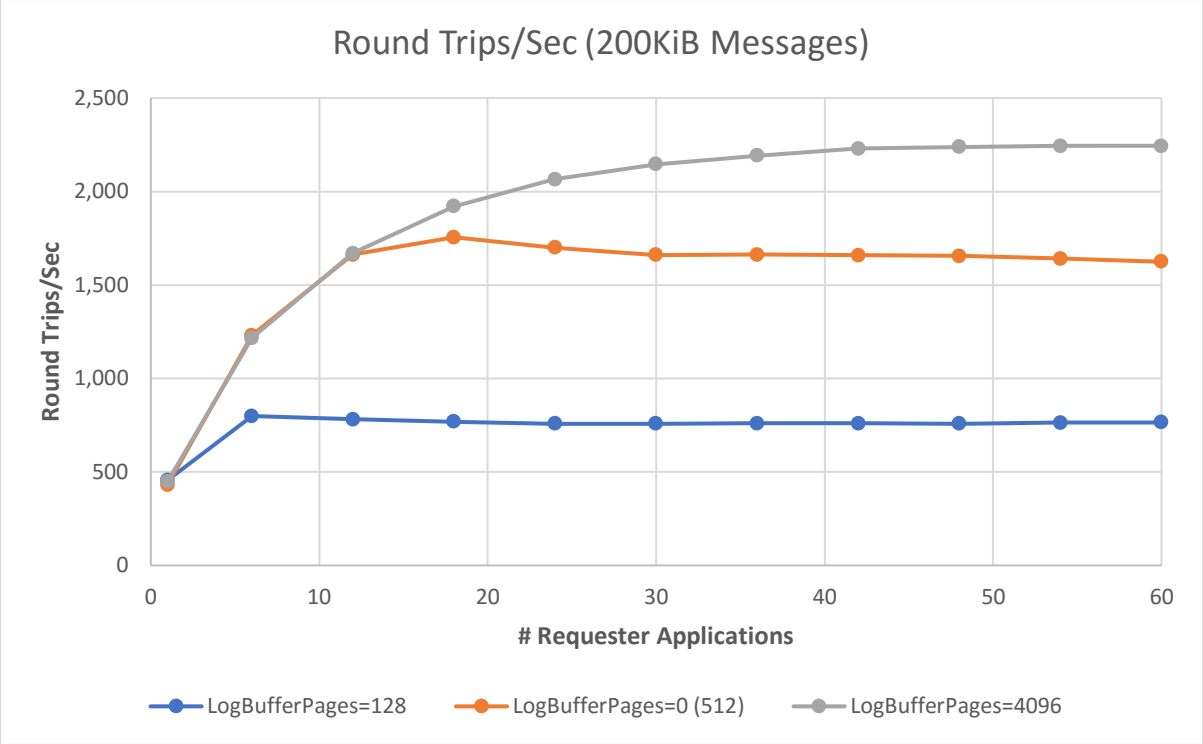


Figure 6 - Effect of LogBuffer Size on 200K Requester/Responder (logging to SAN)

Best Practice #6: Set LogBufferPages=4096 in qm.ini

Note that if you are only sending small messages, and your message rate isn't too high, you may see no benefit from setting the log buffer to the maximum, but given the relatively modest amount of storage a setting of 4096 requires, it is sensible to cater for the highest possible performance. Figure 5 & Figure 6, above show the effect of setting LogBufferPages to different values for 2KiB & 200KiB messages.

1.3.1.4 LogWriteIntegrity

This parameter should generally be left to TripleWrite. It does *not* mean that all log records are written three times, and SingleWrite requires a very specific guarantee from the I/O hardware. Generally, you will not see any benefit from using SingleWrite, as it only helps in some edge cases.

1.3.2 Operational logs, traces, and diagnostics.

These files contain a mix of data reported during normal, or exceptional periods of processing e.g.

- Errors logs (AMQERR<nn>.log) : These contain informational, warning, and error messages regarding MQ
  See: Error log directories
  (https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.tro.doc/q039570_.htm)

- Trace files: (/var/mqm/trace). Typically created under direction of IBM MQ service.

During normal operation, a queue manager's operational logs, diagnostics files etc are not going to present a large I/O workload. Turning trace on will generate much more data, but this would typically only be used for diagnostic purposes, and can have a big impact on performance.

It's a good idea, to host the error logs on a filesystem mounted on a different device to the queue and transaction log data, so that they do not interfere with each other. Imagine a scenario, for instance, where something unexpected caused MQ to allocate additional secondary logs, which filled the capacity of the filesystem. MQ would be unable to write the associated error records, if the error logs are hosted on the same filesystem.

Best Practice #7: Host a queue manager's error logs on a different location to the transaction log, to avoid being unable to write errors.

See the –md and –ld parameters of crtmqm on how to control this.

crtmqm - Create a queue manager.

(https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.ref.adm.doc/q083120_.htm)

1.3.3 Queue Files

Every queue has an associated file to hold its messages. Whether a message ever gets written to the file is another matter. If a persistent message is put onto a queue, and subsequently retrieved by another application before a checkpoint has occurred, it will only have been written to the log (by the time MQ decides to 'harden' messages to the queue file, it is no longer on the queue). Messages are written to a queue file in the following circumstances:

1. The message is persistent, and resides on the queue at checkpoint time*
2. The message is persistent and the queue manager is shut down in a controlled way.
3. The message is non-persistent, NPMCLASS is set to 'HIGH', and the queue manager is shut down in a controlled way.
4. The queue buffer space has been exhausted for a queue.

*MQ synchronises the queue files and logs at checkpoint time, to provide a point of consistency from which forward recovery can be carried out using the queue file and subsequent log entries. This also enables log space to be freed up, by moving the recovery point forward in the log. There is a level of optimization in the checkpoint process, which means that not *all* live persistent messages are written to the queue file at checkpoint time. if a message is very recent, it may be left on the log file only, in the hope that it has been read by an application before the next checkpoint, saving MQ writing it to the queue file.

You can read some more detail about the checkpointing process in the knowledge center:

Restart recovery

(https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.con.doc/q018450_.htm)

Note that long running transactions can stop MQ moving the recovery point forward in the log, and eventually cause problems when the log becomes full. MQ will attempt to recover from this situation by rolling back any long running transactions that are causing the log to become full (see message 'AMQ7469: Transactions rolled back to release log space'). The application will receive a 2003 MQ error, when it subsequently attempts to commit the transaction.

Where MQ is unable to roll back a transaction (e.g. in the case of an XA transaction, where MQ is not the coordinator), it will attempt to move the transaction details forward in the log, re-writing the details of the live XA transaction to the tail of the log, and freeing up older extents, as a result.

Although long running transactions are primarily an operational issue, they can also impact performance (secondary logs may need to be created, formatted and later deleted, transactions are being rolled back or moved forward in the log, etc).

Increasing the number of log files to accommodate long running transactions may seem like a good solution, but this can also be detrimental to performance. A large log set may not make best use of a RAID cache and a lot of transactions in flight will cause any MQ recovery processing to take longer. It is generally better to avoid long running transactions, rather than cater for them, if possible.

Best Practice #8: Avoid Long Running Transactions

The checkpointing process can potentially cause regular writes to the queue files, but with a well-behaved set of applications, these should not be excessive (as many messages will come and go, without ever needing to be written to the queue files).

Queue files are also written to if the associated queue buffer for a queue is full. These buffers are in-memory representations of the queue, and there is one each for non-persistent and persistent messages. Persistent *and* non-persistent messages that 'spill' these buffers will be written to a queue file.

Generally, the queue files are not written to forcibly, MQ writes to them via the operating system buffers, only flushing to disk at the end of a syncpoint, to ensure consistency.

1.4 Messaging vs Queueing

Queue files do not consistently represent the current state of a queue. Their primary function is to store messages that cannot be held in memory due to the depth of a queue, and to store the state of a queue at checkpoint, or shutdown.

MQ is optimal when it is used in messaging mode, rather than queueing. That is, if the messages are retrieved in a timely manner, and queue depth are subsequently small, then only a small proportion of messages (even persistent messages) ever reach the queue file.

1.5 Persistent Messaging, and Applications

Section 3 of this paper will discuss some performance methodologies, but a key point is that any performance test scenario must mimic the proposed production environment as closely as possible. The numbers of application and queues, can affect the performance as much, as the machine a test is being run on.

If there is one message I want this paper to convey it is this; understand your applications and requirements. As has just been shown, in my environment, one size of transaction log will perform differently to another, and there are many other factors that can affect the performance of the MQ logger, including concurrency, syncpoint control, and message size. All of this is aside from the technology used to host the filesystem (whether it be HDD, SSD, SAN or NFS). A good understanding of the application will enable you to:

1. See how you can benefit from the best practices with regards to persistent messaging.
2. Clearly convey what you are trying to achieve when you talk to the administrators of your filesystems.
3. Create valid test scenarios to establish performance capabilities.

Best Practice #9: Understand your application and requirements.

Understanding your application will enable to you to design good tests, whilst understanding your requirements will enable you to know when the performance is ‘good enough’.

2 Part Two – Comparative Performance of File Systems

2.1 Where Should I Host the MQ Transaction Log Files?

In part one, I've covered some of the main impactors on performance that effect persistent messaging in MQ, resulting in some best practices:

To Recap:

- **Only use persistent messaging when necessary for your application.**
- **Size your transaction log correctly.**
- **Concurrency optimizes the MQ logger throughput.**
- **Spread message load across multiple queue where possible, to alleviate queue locking.**
- **Use syncpoint with persistent messages (even if there is only one MQPUT in your transaction).**
- **Set LogBufferPages=4096 in qm.ini**
- **Host a queue manager's error logs on a different location to the transaction log, to avoid being unable to write errors.**
- **Avoid Long Running Transactions**
- **Understand your application and requirements.**
- **Establish infrastructure capabilities outside of MQ to better understand possible bottlenecks.**

Given reasonable behaviour of applications, and a fast network (or local applications), the limiting factor in terms of throughput is likely to be how fast MQ can write to its transaction log. The choice of where to locate the log is not usually based exclusively on performance however (otherwise we'd always locate them locally, fronted by as large a RAID cache as possible). HA and recovery may dictate that the log files are remotely hosted and even replicated synchronously. All of this will have an impact on performance. In part 2 of this paper, I will compare some different filesystems for hosting the MQ transaction logs.

In this section, I will compare some file systems as illustrative examples of how different technologies behave when being used to host the MQ transaction log, namely:

- Local HDD, fronted by a 4GB cached RAID controller
- Local 'small' SSD, fronted by a 4GB cached RAID controller
- SAN (with SVC)
- NFS (through a 40Gb network switch) to a host with HDD fronted by 4GB cached RAID controller
- NFS (through a 10Gb network switch) to a host with HDD fronted by 4GB cached RAID controller
- NFS (through a 1Gb network switch) to a host with HDD fronted by 4GB cached RAID controller
- NFS (through a 10Gb network switch) to a host with HDD fronted by 4GB cached RAID controller and simulated fibre optic latency across large distances

The purpose of these tests is to show the kind of impact that may be seen when moving from one technology to another. I will not be providing benchmarking data for specific models of hardware.

The scenario used is initially somewhat unrealistic, as all applications are connected to MQ using bindings mode, but this enables us to eliminate the potential network bottleneck, to directly compare file I/O performance. The workload is driven by the MQ-CPH test tool, available on Git Hub (https://github.com/ibm-messaging/mq-cph).

There is a blog article describing it's use on developer works.

MQ-CPH Performance Harness Released on GitHub

(https://www.ibm.com/developerworks/community/blogs/messaging/entry/MQ_C_Performance_Harness_Released_on_GitHub?lang=en)

MQ-CPH Configuration:

- All requesters/responders are connected in bindings mode.
- A fixed number of responders are started (200 or 600, depending on the maximum number of requesters)
- Load is evenly distributed across 10 pairs of request/reply queues.
- 1 requester is started followed by increments up to a maximum required to saturate the capability of the MQ logger / filesystem.
- All best practices and tunings are applied from part one of this paper.

In addition to the MQ-CPH workloads, a tool (MQLDT) is used to establish a theoretical write speed limit of the filesystem (MQLDT writes to a set of similarly sized files, using the same flags as MQ (e.g. opening with O_DIRECT, O_DSYNC etc.). Note that MQLDT cannot show what write speed you will get from MQ, as that depends on many factors, including the applicability of some of the best practices shown in part one of this paper. It measures the maximum speed of writing to the filesystem, in a similar way to MQ, when there is no work to do, other than writing to the disk. As we shall see, this is sometimes close to what we see with MQ, when the disk latency is high, and sometimes very divergent (as is the case in the first set of results below), when the latency is very low.

2.1.1 MQ Transaction Log File-Sets.

The MQ transaction log file-set was configured at 1GB, 1.5G or 4GB, as specified in the results that follow, i.e.

- 1GB Transaction log:
  LogFilePages=16384
  LogPrimaryFiles=16
  LogSecondaryFiles=2

- 1.5GB Transaction Log:
  LogFilePages=16384
  LogPrimaryFiles=24
  LogSecondaryFiles=2

- 4GB Transaction Log:
  LogFilePages=16384
  LogPrimaryFiles=64
  LogSecondaryFiles=2

2.2 Test Results

In the charts below, MQ vs MQLDT comparisons show the Mbytes/sec logged to the filesystem as the number of requester application was increased (increasing the concurrency and delivery rate of messages). The block sizes reported by amqsrua for each number of MQ applications were then used by MQLDT to ascertain the raw performance of the filesystem at that block size.

All tests use 2KiB messages unless otherwise specified.

2.2.1 Local Storage

2.2.1.1 HDD Results

For the HDD results, we used 2 HDDs in a RAID 0 pair. This might be considered to perform similarly to a 4 disk RAID 10 configuration for some RAID technologies. As we shall see however, the RAID cache makes a big difference.

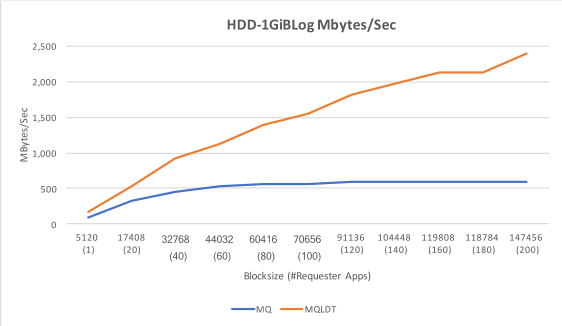


Figure 7 - HDD Log Writes for 1GiB MQ Log, RAID WB Enabled

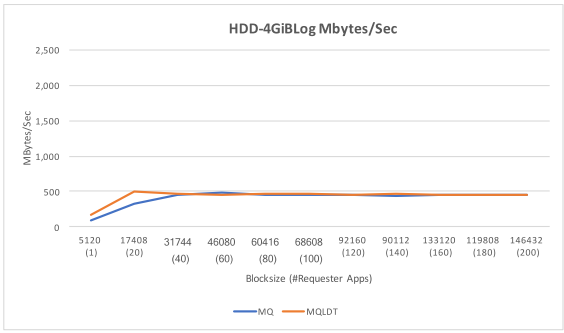


Figure 8 - HDD Log Writes for 4GiB MQ Log, RAID WB Enabled

Figure 7 shows results when stressing MQ, which is logging to a 1.5GB log, hosted on an HDD backed RAID volume, via a RAID controller with a 4GB, battery backed cache. The RAID controller was configured with 'Write Back' enabled. 2GB of the RAID cache was eligible for Write caching, so the set of log files for the first test could be written to, entirely within the RAID cache, resulting in optimum performance. The server was approaching CPU saturation at this point however. The MQLDT plot shows how much further we can go in terms of bandwidth, writing to a RAID cache with Write Back policy set (and the plot is still increasing at ~100K block size).

For the second test (Figure 8), the MQ log was increased from 1GiB to 4GiB, such that the MQ log files no longer reside entirely within the RAID cache. The cache *does* still aggregate writes, resulting in higher block size writes to the disk however. Both MQ and MQLDT now show similar limits, i.e. we have hit the bandwidth limit of writing to the disk with these application block sizes, in both cases (rather than MQ exhausting CPU resources).

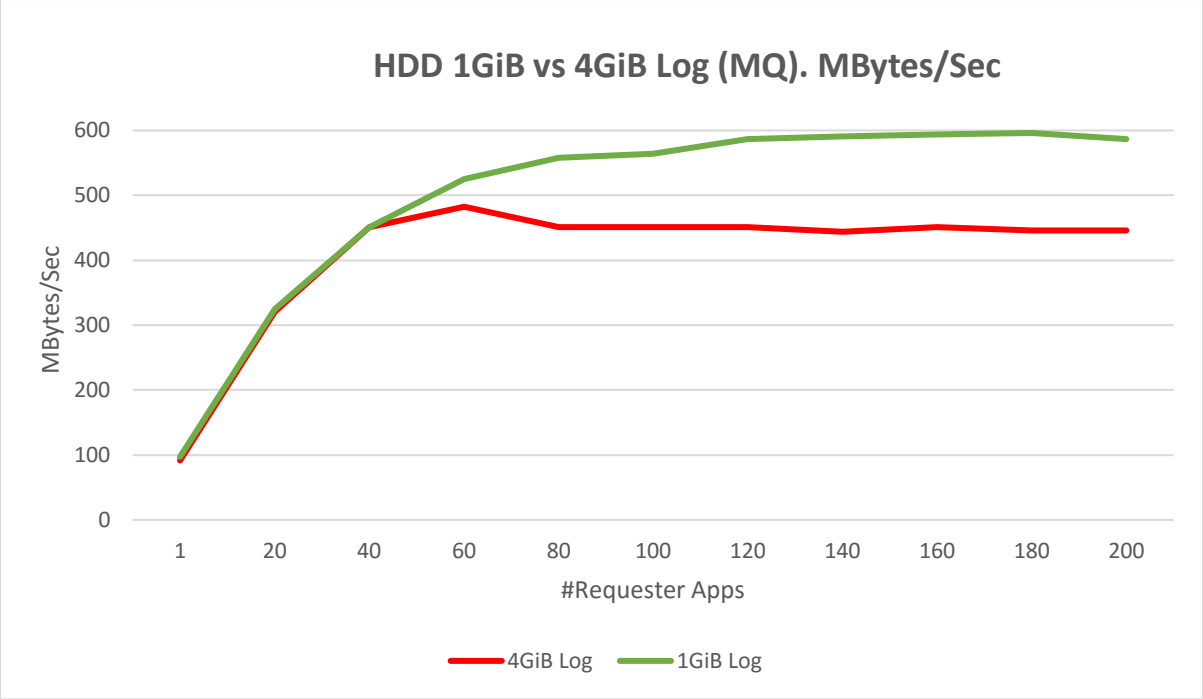


Figure 9 - HDD Log Writes for 1Gib vs 4GiB MQ Log, RAID WB Enabled

Figure 9 shows the 1GiB and 4GiB log test for the MQ runs, compared (block sizes are not shown as they will not be the same, being determined by the differing latencies of the filesystem in the two tests which causes MQ to aggregate data more, in the case of the 4GiB log file set.

Whilst the 4GiB log test shows an impact of being unable to serve the log writes out of the RAID cache in their entirety, the cache does still enable the RAID adapter to aggregate writes, and the backing storage may also have a suitable cache that can be utilized.

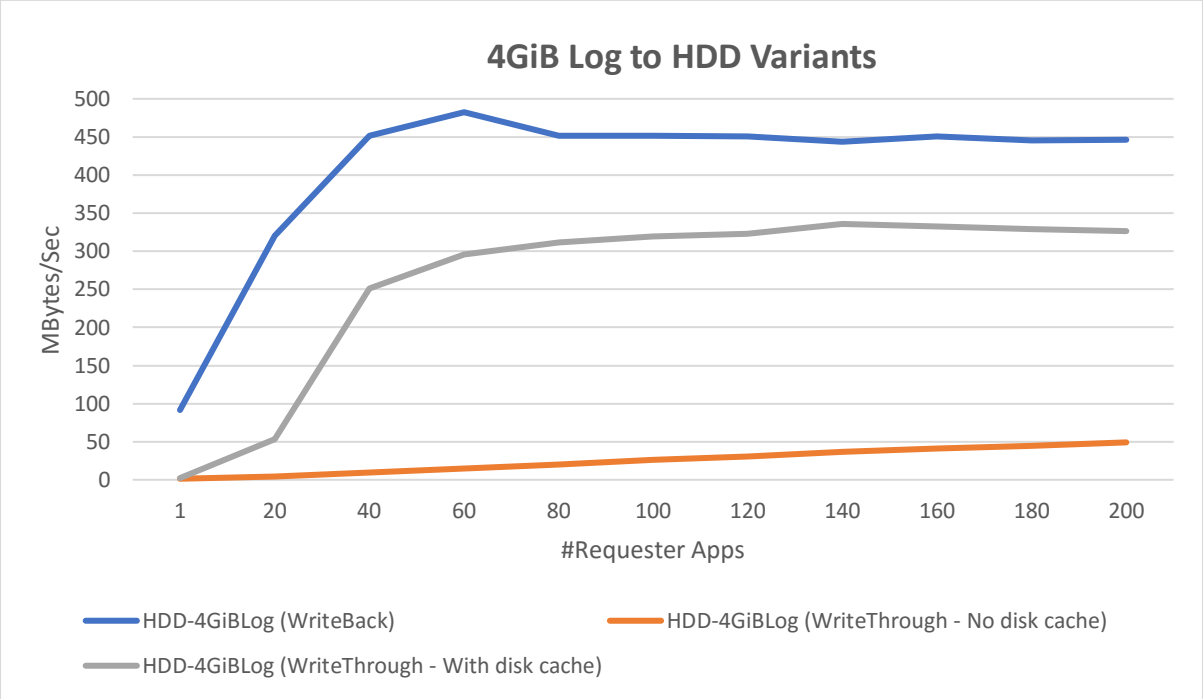


Figure 10 - HDD Throughput for 4GiBLog

The latency of the filesystem will be higher for the 4GiB file set (see Table 2 below), this causes MQ to aggregate the writes into larger blocks.

	Block Size	Latency
4GiBLog (WriteBack – No disk cache)	146807	207 µs
4GiBLog (WriteThrough – With disk cache)	206019	415 µs
4GiBLog (WriteThrough – No disk cache)	215842	5012 µs

Table 2 - Write latency at peak throughput of test (from amqsrua)

The results above, demonstrate how MQ will usually benefit significantly from a write cache, be it on the RAID controller, the disk, an SSD, or (as we shall see later), on the SAN volume controller. You need to think carefully about the applicability of any RAID cache however. If persistent messages should be considered mission critical. If they are not, then you need to think about whether these messages should in fact be non-persistent. Such data cannot be lost in the event of a power failure, so caches need to be persistent too. RAID caches are typically battery backed, and the RAID controller should be set to switch to 'Write Through' mode in the event of a battery failure. SSDs can have capacitor backed caches, and some HDDs (such as the ones used in these tests), can use the EMF energy of the spinning disk to persist the volatile cache, on power failure. Be aware of the capabilities and limitations of caches, once MQ forces a log write to the RAID controller, SSD, SAN etc. we trust the device to commit it.

2.2.1.2 SSD Results

SSD drives are becoming more prevalent, and better performance is typically expected of this type of device, when compared to more traditional, spinning disks. As we have seen, MQ transaction logging has a very specific use of an I/O device however, which does not hit the typical sweet spot for an SSD (random read/writes). This section measures some basic server class SSD devices.

For these result, 2 SSD cards were configured identically to the HDD tests above. The SSD is a low-end server model.

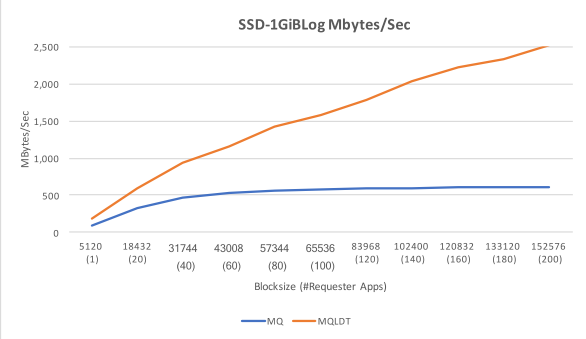


Figure 11 - SSD Log Writes for 1GiB MQ Log, RAID WB Enabled

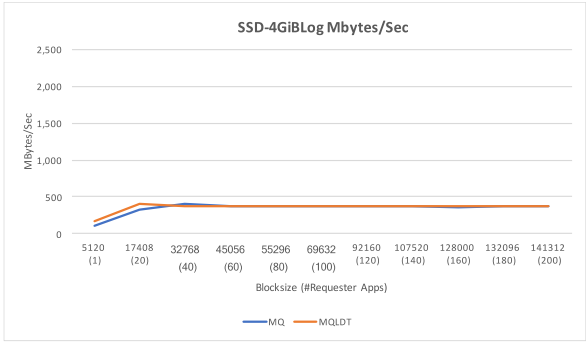


Figure 12 - SSD Log Writes for 4GiB MQ Log, RAID WB Enabled

With the RAID controller write cache in use with a 1GiB log, the striking thing is that the performance of the HDD and SDD is very close, peaking at around 600MB/s (comparing Figure 8 & Figure 11 above). This is not surprising, as both tests are measuring the speed of the RAID cache (up to the limit), rather than the backing storage. This illustrates the difficulty in predicting the performance characteristics based on the storage device alone.

With the MQ log file set increased to 4GiB, the write bandwidth is reduced, as was the case for the HDD, being affected by the write speed of the SSD in this case. What is interesting is that the write rate is less than that attained in the same test using an HDD (compare Figure 8 with Figure 12).

Both the HDD, and SSD have on-board write caches, though the SSD cannot be disabled via the RAID controller. The write cache on the HDD was disabled in the RAID controller for all WriteBack tests.

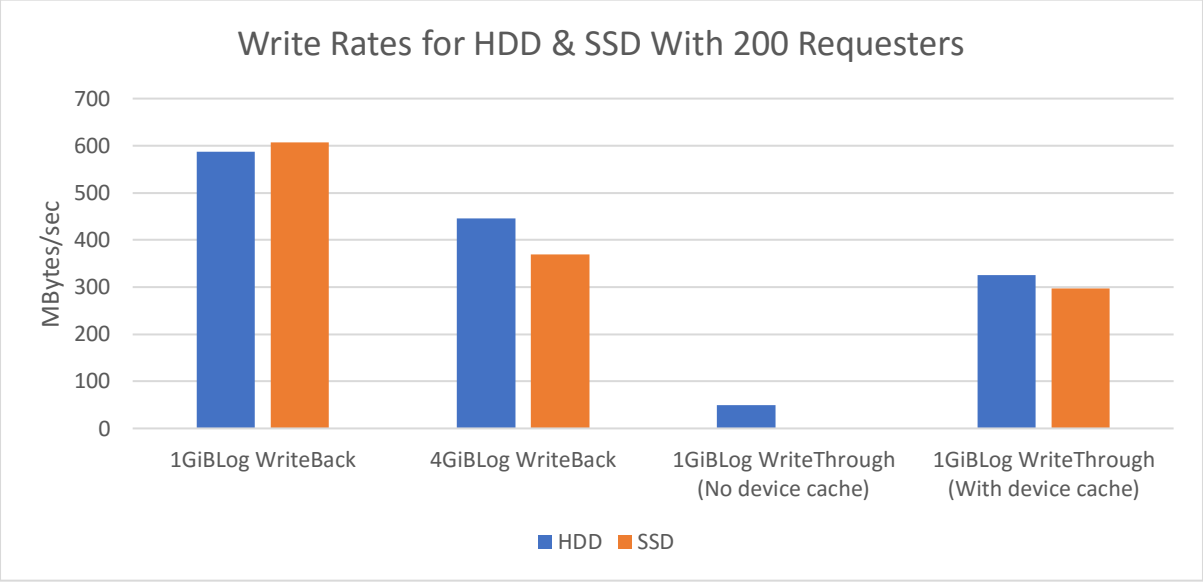


Figure 13 - Comparison of caching on SSD & HDD

Figure 13 shows a comparison of logging to an SSD and an HDD with WriteBack mode (i.e. RAID cache utilised), or WriteThrough. In WriteThrough mode, the HDD was tested with and without the device based write cache enabled, the SSD cache could not be disabled through the RAID controller.

- Writes directly to the HDD with no device cache are very slow.
- With the device caches on, the HDD outperformed the SSD with direct writes.
- MQ logging benefits from write caching, whether it be on the RAID controller, or device.
- The RAID cache can compensate for the lack of a device cache, exploiting better bandwidth for very large writes to the device. *

*The last point is not obvious. The 4GiB log test, with WriteBack enabled, shows the HDD to be faster than the SSD. But the HDD's device cache is disabled in this test, and the WriteThrough test shows the HDD performs poorly without the device cache. Whilst the 4GiB test cannot be served completely from the RAID cache, the cache *will* aggregate the write to the disk, exploiting any benefit of doing so. If these large writes to the HDD are the reason for it out-performing the SSD, then running large block size tests using a tool such as MQLDT should also demonstrate this, and indeed MQLDT showed the HDD, without the device cache enabled, outperforming the SSD for block sizes of ~4MiB and above.

In Figure 10, the write rate for the WriteThrough test to the HDD without a device cache is still increasing at the end of the test. If we continue to add more applications, the logger will aggregate more messages in a single write. On extending the test, the peak throughput was found to be 310MB/sec at a write size of 5.7MB (MQ effectively acting as the write cache), but required 6800 requester applications to be running.

2.2.1.3 Local Device – Conclusions

Testing against two types of device, configured in a RAID 0 array has shown that the overall bandwidth can be very sensitive to the size, and availability of any write cache. MQ transaction logging is predominantly comprised of sequential writes and as such, is 'cache friendly'. This sequential write pattern also means the using an SSD may not provide any benefit over an HDD.

Where a filesystem is slow, MQ will aggregate writes, increasing the bandwidth, but this is also dependent on MQ application concurrency.

2K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	97					
	Latency (µs)	26					
	Write size (KB)	5					
60 Requesters	MB/s	525					
	Latency (µs)	39					
	Write size (KB)	44					
180 Requesters	MB/s	596					
	Latency (µs)	111					
	Write size (KB)	119					

Table 3 - 2KiB Message Results (HDD)

*20K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	202					
	Latency (µs)	27					
	Write size (KB)	13					
60 Requesters	MB/s	2054					
	Latency (µs)	93					
	Write size (KB)	388					
180 Requesters	MB/s	2320					
	Latency (µs)	267					
	Write size (KB)	1252					

Table 4 - 20KiB Message Results (HDD)

*200K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	653					
	Latency (µs)	26					
	Write size (KB)	85					
18 Requesters	MB/s	2629					
	Latency (µs)	154					
	Write size (KB)	768					
60 Requesters	MB/s	1628					
	Latency (µs)	806					
	Write size (KB)	3768					

Table 5 - 200KiB Message Results (HDD)

*Note that for the 20K & 200K message tests the number of primary log files was increased to accommodate the high logging rate, so these tests used a 1.5GiB log.

2.2.1.4 Local Storage Specifications

- MQ Host:
  - x3550, 2x14 Cores: Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz.
  - Red Hat Enterprise Linux Server release 7.2 (Maipo).
  - 128GB RAM

- HDD:
  - 2 x 300GB, (12Gbs SAS 3.0 link), 15,000 RPM HDDs configured as a RAID 0 array, fronted by a RAID controller with 4GB, battery backed cache (Write Back enabled).
  - 128 MiB on-board write cache (and can be disabled via the RAID adapter).

- SSD:
  - 2 x 120GB, (6Gbs SATA link) SSDs configured as a RAID 0 array, fronted by a RAID controller with 4GB, battery backed cache (Write Back enabled).
  - The SSDs have a nominal sequential write speed of 200MB/s (block size unspecified), but this is with the write cache disabled. The 2GiB on-board, write cache is capacitor backed, and cannot be disabled via the RAID controller.

2.2.2 Remote Storage

There are many reasons why the filesystem hosting an MQ transaction log may reside off-box. Generally, pure performance is *not* one of them. Moving the MQ transaction log from a local filesystem to SAN, NFS or any other remote hosted storage will generally degrade the performance characteristics of the queue manager, whether this *matters*, will depend on the requirements of your application. It may be, that an application drives logging at a rate of 5MB/sec. Moving the logs to NFS will increase the latency of the log writes, but MQ may then aggregate those writes such that the application can still log at a rate of 5MB/sec, but the application will experience higher latencies on MQPUT, or MQCMIT calls.

2.2.2.1 SAN Storage

The SAN tests used an IBM Storwize V7000 populated with 10,000 rpm disks configured in a RAID 10 array, and fronted by an IBM SAN Volume Controller (SVC).

The svc was connected to the MQ server via a dual-port 8Gb fibre channel adapter.

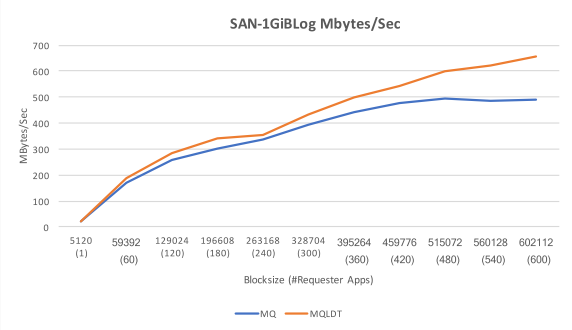


Figure 14 - SAN Log Writes for 1GiB MQ Log

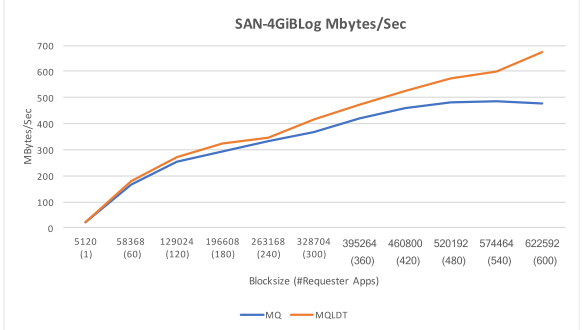


Figure 15 - SAN Log Writes for 4GiB MQ Log

Figure 14 & Figure 15, show results for 2K persistent message test logging to a SAN volume. Note that the results from MQLDT and the actual MQ tests are now very similar. Due to the increased latency of the SAN volumes, a larger number of applications are required to reach the limit. At 600 requester applications, the MQ logger is aggregating log writes, averaging around 622K each for the 4GiB log test. For 2K writes, MQ is still the limiting factor in terms of bytes/sec. If we increase the message size to 200K, MQ can log at a rate approaching the limit of the 8Gb fibre connection itself, at only 60 requester applications (see tables below). MQLDT, can get even closer to the limit, writing at a rate of ~1GB/sec, but this requires a write size of 64MB (which is the size of each log file).

Results for 1GiB and 4Gib tests are now virtually identical. We are no longer assisted by the RAID controller cache, but the SVC volumes are configured with caching enabled, and writes to both the 1GB and 4GB log files sets were found to be serviced directly by the SVC cache, with no downstream writing to the 'MDisks' (the SVC logical unit of physical storage), during normal operation. Once again, MQ transaction logging is seen to be very write cache friendly, such that testing against three separate RAID array servers (V7000 populated with

10K RPM HDDs, V7000 populated with 15K RPM HDDs, and V900 populated with SSDs), all showed identical performance due to the caching performance of the SVC.

2K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	97	24				
	Latency (µs)	26	172				
	Write size (KB)	5	5				
60 Requesters	MB/s	525	171				
	Latency (µs)	39	302				
	Write size (KB)	44	59				
180 Requesters	MB/s	596	302				
	Latency (µs)	111	589				
	Write size (KB)	119	197				

Table 6 - 2KiB Message Results (HDD & SAN)

*20K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	202	52				
	Latency (µs)	27	171				
	Write size (KB)	13	12				
60 Requesters	MB/s	2054	515				
	Latency (µs)	93	693				
	Write size (KB)	388	414				
180 Requesters	MB/s	2320	796				
	Latency (µs)	267	1367				
	Write size (KB)	1252	1272				

Table 7- 20KiB Message Results (HDD & SAN)

*200K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	653	187				
	Latency (µs)	26	319				
	Write size (KB)	85	86				
18 Requesters	MB/s	2629	774				
	Latency (µs)	154	755				
	Write size (KB)	768	1051				
60 Requesters	MB/s	1628	894				
	Latency (µs)	806	3263				
	Write size (KB)	3768	3704				

Table 8 - 200KiB Message Results (HDD & SAN)

2.2.2.1.1 SAN Tuning

There are a host of tuning options for the Linux I/O subsystem. Various dispatchers were tried, for example, with no effect on results. Disabling write merges did have a positive impact on tests at some rates for larger messages (20KiB for example), so this was set for the active multipath device (mpatha, see below) used for san storage.

Executing lsblk, shows the san block device topology.

```
NAME          MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
sdd             8:48   0   128G  0 disk
└─mpatha        253:3    0   128G  0 mpath
  └─mpatha1     253:6    0   128G  0 part  /var/san1
sdh             8:112  0   128G  0 disk
└─mpatha        253:3    0   128G  0 mpath
  └─mpatha1     253:6    0   128G  0 part  /var/san1
```

With the default value of nomerges=0 (i.e. write merges are enabled), a round trip rate of ~13,350/sec was achieved. Using iostat (iostat –xN 2) we can see the log write rates, through the devices supporting /var/san1 (the mount-point of the MQ transaction log).

iostat output (abridged, to show only relevant devices)

Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sdd	0.00	0.00	0.00	737.50	0.00	284926.00	772.68	0.77	1.04	0.00	1.04	0.93	68.75
sdh	0.00	0.00	0.00	737.00	0.00	287310.00	779.67	0.83	1.13	0.00	1.13	1.02	75.35
mpatha	0.00	1128.50	0.00	1474.50	0.00	572236.00	776.18	1.63	1.11	0.00	1.11	0.62	91.15
mpatha1	0.00	0.00	0.00	2603.00	0.00	572310.00	439.73	3.06	1.18	0.00	1.18	0.35	90.75

The iostat output shows us writing at a rate of 2603/sec to mpatha1. This is reduced to 15474/sec on mpatha due to write merging. Column ‘wrqm/s’ shows us the number of write merges/sec that are occurring (1128.58). The new write rate (which will now be for a larger write size - see column avgrq-sz), is then split across the two paths to the fibre adapter (sdd & sdh).

We can turn off write merging, with a value of nomerges=2. E.g.;
echo 2 > /sys/block/dm-3/queue/nomerges
Where dm-3 is the system device for mpatha (multipath –l will show this).

After turning off write-merges, a round trip rate of ~16,300/sec was achieved.

Running iostat again, confirms that we have disabled write merges:

Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sdd	0.00	0.00	0.00	1578.50	0.00	347464.00	440.25	1.37	0.87	0.00	0.87	0.48	75.20
sdh	0.00	0.00	0.00	1579.00	0.00	349980.00	443.29	1.47	0.93	0.00	0.93	0.51	81.00
mpatha	0.00	0.00	0.00	3159.50	0.00	697956.00	441.81	2.92	0.92	0.00	0.92	0.28	88.70
mpatha1	0.00	0.00	0.00	3157.00	0.00	697380.00	441.80	2.91	0.92	0.00	0.92	0.28	88.45

If you notice significant write merges occurring on a heavily utilised file system, support MQ transaction logging it is worth testing whether disabling merges will benefit you. Note that merging writes is attempting to increase your bandwidth, so this it is not a general recommendation to disable this function, test your own environment first.

2.2.2.2 NFS

A network file system (NFS) can be hosted in many ways, and the performance you will see, will depend a lot on the network configuration, predominantly

For the tests that follow, the transaction logs were hosted on a secondary machine, of the same specification as the one hosting the MQ server and applications, and connected to the MQ host via local network switches, supporting 1Gb, 10Gb, and 40Gb links.

Basic NFS tuning was applied, in addition to the settings necessary to support MQ transaction logging, and MIQM operation (though MIQM is not configured in the tests that follow).

On the nfs server RPCNFSDCOUNT was set to 28 in /etc/sysconfig/nfs, to equal the number of cores on the machine.

On the nfs client machine (the MQ host) being tested, wsize and rsize nfs mount options both defaulted to 1048576 (you can run 'nfsstat –m' to check this on your system). Reducing the buffer sizes to 512KB resulted in a small increase in bandwidth, though you should test for what is optimal in your own environment.

All nfs mount lines were similar to the one below (for the 40Gb link).

nfsserver40:/mqm_hdd /var/nfs40 nfs4 rw,soft,auto,rsize=524288,wsize=524288

The *async* mount option is also recommended (not to be confused with the export option on the nfs server, which must be *sync*), but this is the default in nfs anyway (see *man nfs*).

Sample Setup from /etc/exports on nfs host:

/export 9.1.aaa.bbb/24(rw,fsid=0,sync,no_wdelay,anonuid=30000,anongid=30000)
/export/mqm_hdd 9.1.aaa.bbb/24(rw,sync,no_wdelay,anonuid=30000,anongid=30000)
/export 9.10.aaa.bbb/24(rw,fsid=0,sync,no_wdelay,anonuid=30000,anongid=30000)
/export/mqm_hdd 9.10.aaa.bbb/24(rw,sync,no_wdelay,anonuid=30000,anongid=30000)

Where 9.1.aaa.bbb is the 1Gb IP address of the nfs host, and 9.10.aaa.bbb is the 10Gb address of the nfs host.

IBM has published the requirements for a shared file systems used by MQ in the knowledge center.

Requirements for shared file systems

(https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.0.0/com.ibm.mq.pla.doc/q005810_.htm)

You can use the supplied MQ tool amqmfsck to check a shared filesystem.

Verifying shared file system behaviour

(https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.pla.doc/q005820_.htm)

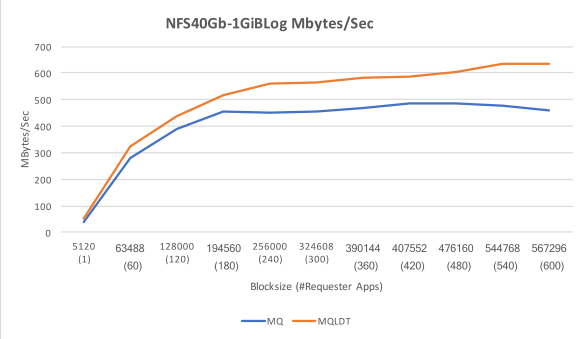


Figure 16 – NFS (40Gb) Log Writes for 1GiB MQ Log

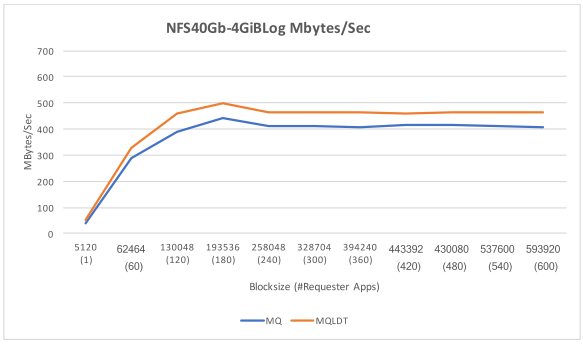


Figure 17 – NFS (40Gib) Log Writes for 4GiB MQ Log

For the initial tests, a 40Gb network link was used to ensure the network was not the limit, and establish the capability of NFS.

Figure 16 & Figure 17 above, show nfs results across a 40Gb link to an nfs server hosting the file system on an HDD, accessed via a RAID adapter with a 4GB cache. The HDD configuration is the same as for the local disk tested earlier, we're now accessing it via nfs though. Similarly to previous results, the larger MQ log results in a lower peak throughput, as the effects of the RAID cache being filled occur, before either the network or nfs reach their limits.

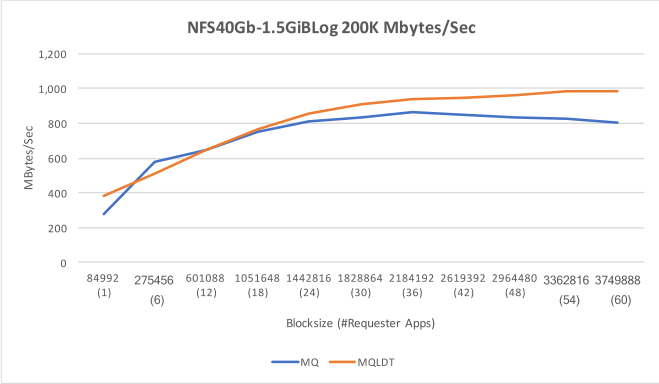


Figure 18 - NFS (40Gib) Log Writes for 200KiB Messages

Increasing the message size to 200KiB (**Error! Reference source not found.**), r esults in an increase in throughput but MQLDT peaked at around 1GB/sec. Given that we can write at 1.6/GB sec to a local HDD through the RAID cache, and the network link has a nominal rating of 40Gb/sec (5GB/sec), this would seem to indicate a limit in the nfs layer. Further testing confirmed that an nfs mount using the loopback adapter showed a similar throughput, discounting a network limitation. Tests were also carried out independently on the network link , to establish the actual limit that could be handled.

Best Practice #10: Establish infrastructure capabilities outside of MQ to better understand possible bottlenecks.

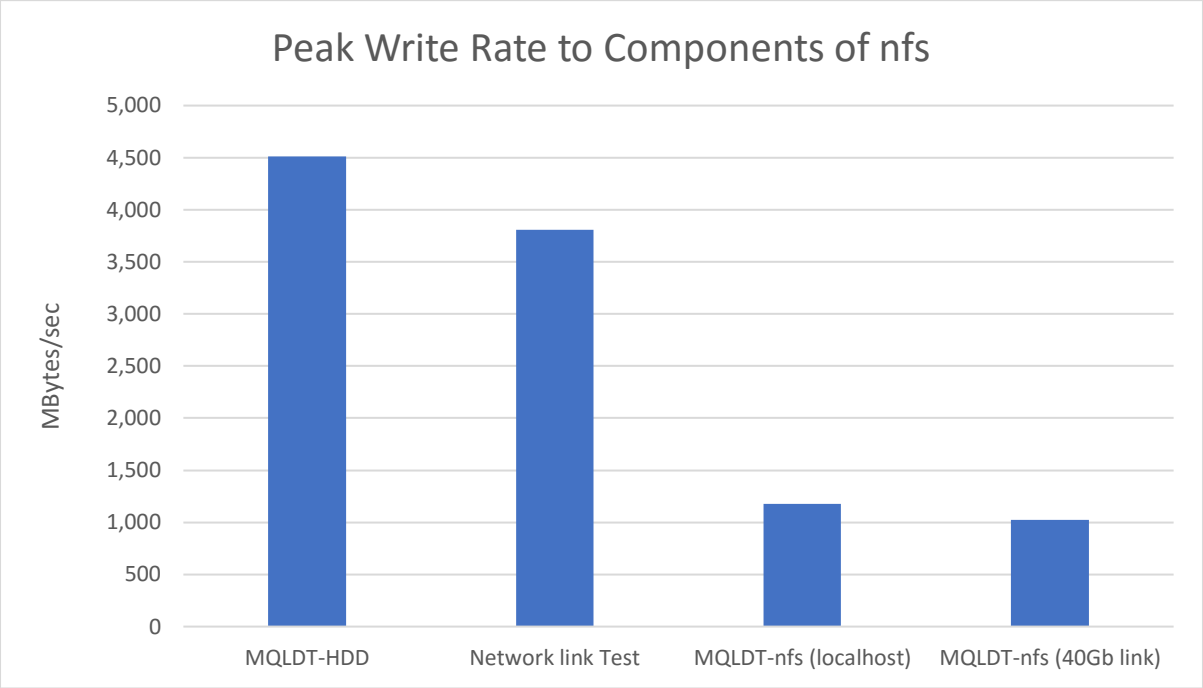


Figure 19 - NFS Bandwidth breakdown

Figure 19 shows the peak write rates for MQLDT to HDD (directly, via a localhost nfs mount, and via a 40Gb link nfs mount), using a 3MB write size. A network link test is also shown for comparison. This indicates the nfs layer as being the limiting factor.

Whilst the 40Gb results are interesting, nfs is not typically deployed across a 40Gb end to end network. The following charts show results using nfs across 1Gb and 10Gb network links.

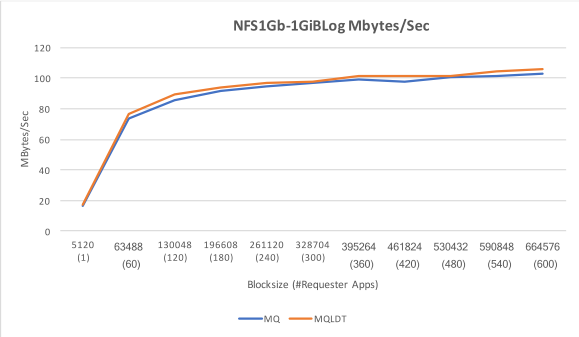


Figure 20 – NFS (1Gb) Log Writes for 1GiB MQ Log

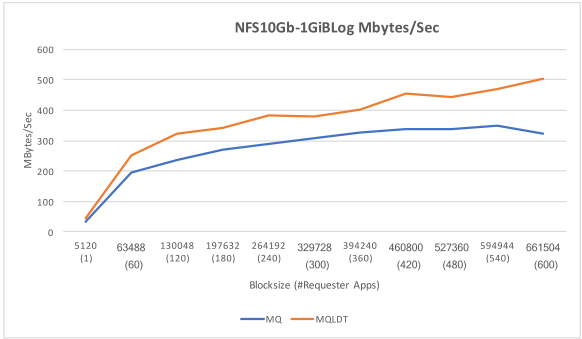


Figure 21 – NFS (10Gib) Log Writes for 1GiB MQ Log

Figure 20 & Figure 21 above show throughput using nsf across a 1Gb or 10Gb network link. Both MQ and MQLDT are constrained by the network, with the 1Gb results being almost identical. At 10Gb the MQ test reaches a peak, similar to the 2K test to SAN. A 200K message size makes larger aggregations possible once again (see tables below)

2K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	97	24	17		35	
	Latency (µs)	26	172	270		97	
	Write size (KB)	5	5	5		5	
60 Requesters	MB/s	525	171	74		196	
	Latency (µs)	39	302	793		262	
	Write size (KB)	44	59	63		63	
180 Requesters	MB/s	596	302	92		271	
	Latency (µs)	111	589	2089		717	
	Write size (KB)	119	197	196		196	

Table 9 - 2KiB Message Results (HDD, SAN, NFS-1G & NFS-10G)

*20K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	202	52	30		73	
	Latency (µs)	27	171	358		122	
	Write size (KB)	13	12	13		13	
60 Requesters	MB/s	2054	515	100		366	
	Latency (µs)	93	693	4036		1073	
	Write size (KB)	388	414	410		413	
180 Requesters	MB/s	2320	796	105		499	
	Latency (µs)	267	1367	11577		2006	
	Write size (KB)	1252	1272	1260		1255	

Table 10 - 20KiB Message Results (HDD, SAN, NFS-1G & NFS-10G)

*200K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	653	187	71		190	
	Latency (µs)	26	319	1055		303	
	Write size (KB)	85	86	85		86	
18 Requesters	MB/s	2629	774	107		493	
	Latency (µs)	154	755	9401		2378	
	Write size (KB)	768	1051	1017		941	
60 Requesters	MB/s	1628	894	109		511	
	Latency (µs)	806	3263	32854		7012	
	Write size (KB)	3768	3704	3662		3716	

Table 11 - 200KiB Message Results (HDD, SAN, NFS-1G & NFS-10G)

2.2.3 Remote link tests.

All of the test scenarios here are single queue manager, non-MIGM scenarios. When MIQM is used, an NFS filesystem may be utilised which can be in another data centre to one of the queue managers (or data replication may occur, to a remote site).

We can simulate the sort of additional latency that may be seen across a remote link to an NFS filesystem, by injecting a delay into the network interface, using the Linux network traffic controller program *tc*, e.g.

Taking some common values for a fibre optic cable, might results in the following additional latency across a 10KM link:

Refractive Index*	Distance/time	Latency for 10KM
1.470	203.94m/µs	49.03 µs

*The refractive index of a fibre optic cable affects the speed of the transmission of information, from one end to the other. It's beyond the scope of this paper to discuss the

performance characteristics of fibre optic cables, the value chosen has been given here for information only.

A 10KM fibre optic cable with a refractive index of 1.47 would introduce *at least* an additional 49.04µs each way to the packets sent in writing a log record to NFS down that link. The subsequent tests round that value to 50µs, and this delay was added to the 1GB and 10GB network interfaces on the queue manager host and the machine hosting the NFS server.

Using tc, we can set this delay on interface ens2f0 as follows:

```
tc qdisc add dev ens2f0 root netem delay 50us
```

Results are in the NFS1-1GR and NFS-10GR columns below (where the R denotes the 'remote' 10KM delay added).

Please bear in mind that these results are for illustrative purposes to show the kind of effect a latency can have. Every network, and storage solution will be different, so tests on what is being proposed are essential.

2K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	97	24	17	12	35	18
	Latency (µs)	26	172	270	395	97	189
	Write size (KB)	5	5	5	5	5	5
60 Requesters	MB/s	525	171	74	61	196	156
	Latency (µs)	39	302	793	846	262	348
	Write size (KB)	44	59	63	49	63	63
180 Requesters	MB/s	596	302	92	86	271	255
	Latency (µs)	111	589	2089	2229	717	761
	Write size (KB)	119	197	196	195	196	197

Table 12 - 2KiB Message Results (HDD, SAN, NFS-1G, NFS-1GR, NFS-10G & NFS-10GR)

*20K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	202	52	30	23	73	38
	Latency (µs)	27	171	358	504	122	280
	Write size (KB)	13	12	13	12	13	13
60 Requesters	MB/s	2054	515	100	96	366	327
	Latency (µs)	93	693	4036	4127	1073	1165
	Write size (KB)	388	414	410	408	413	405
180 Requesters	MB/s	2320	796	105	104	499	428
	Latency (µs)	267	1367	11577	11812	2006	2186
	Write size (KB)	1252	1272	1260	1268	1255	1261

Table 13 - 20KiB Message Results (HDD, SAN, NFS-1G, NFS-1GR, NFS-10G & NFS-10GR)

*200K Message		Local HDD	SAN	NFS-1G	NFS-1GR	NFS-10G	NFS-10GR
1 Requester	MB/s	653	187	71	64	190	150
	Latency (µs)	26	319	1055	1182	303	449
	Write size (KB)	85	86	85	85	86	84
18 Requesters	MB/s	2629	774	107	105	493	403
	Latency (µs)	154	755	9401	9823	2378	3022
	Write size (KB)	768	1051	1017	1046	941	941
60 Requesters	MB/s	1628	894	109	109	511	321
	Latency (µs)	806	3263	32854	34309	7012	10442
	Write size (KB)	3768	3704	3662	3726	3716	3724

Table 14 - 200KiB Message Results (HDD, SAN, NFS-1G, NFS-1GR, NFS-10G & NFS-10GR)

2.2.3.1 Remote Storage Specifications

- MQ Host:
  - x3550, 2x14 Cores: Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz.
  - Red Hat Enterprise Linux Server release 7.2 (Maipo)
  - 128GB RAM
  - 1Gb, 10Gb & 40Gb network adapters

- SAN
  - Dual-port, 8Gb, Fibre Channel Host Adapter, connected to SVC.
  - IBM SAN Volume Controller (SVC), with 128GB cached volume, from MDisk pool (backed by Storwize V7000).
  - IBM Storwize V7000 populated with 10,000 rpm disks configured in a RAID 10 array.
- NFS
  - NFS (v4) mapped via 1Gb, 10Gb and 40Gb links from MQ host, to NFS host
  - NFS host:
    - x3550, 2x14 Cores: Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz.
    - Red Hat Enterprise Linux Server release 7.2 (Maipo).
    - 128GB RAM
    - 1Gb, 10Gb & 40Gb network adapters

2.3 More on Logger Aggregation

What is noticeable across all sets of results, is that the average write size for a particular combination of message size and number of requesters, remains very similar, regardless of the technology being used to host the filesystem. This is because, our requester applications are running unrated (we do not throttle the rate of MQPUT calls), so deliver requests *very* fast. As soon as one reply is received, the next request (MQPUT) is sent. An unrated test, particularly where the applications are running on the same machine as the queue manager is likely to cause the MQ logger to reach the same average write size, regardless of the filesystem being used.

The MQ logger writes in 4K pages to the transaction log files, this is partly why the 2KB message case with only one requester results in an average 5KB write size. A single requester test means that we never aggregate data in log writes, moreover, each log page write will be partially populated, requiring the 'TripleWrite' mechanism to ensure data integrity. Triple Write does not mean that MQ logs every message three times, but does, in extreme cases (like the 2KB message scenario with only one driving application) result in significantly more writes to the log.

You should always use TripleWrite, as in most cases, the level of concurrency in a typical MQ workload will ameliorate the performance impact of it.

Developer works article: Bitesize Blogging: LogWriteIntegrity.... should I pick SingleWrite or TripleWrite?

(https://www.ibm.com/developerworks/community/blogs/messaging/entry/Bitesize_Blogging_LogWriteIntegrity_should_I_pick_SingleWrite_or_TripleWrite?lang=en)

Every workload will perform *some* 'triple write' logic, as there is always a partial page to be written. Most of the time, however, there will be n full pages + 1 partial page. As n increases (the amount of data aggregated in the log buffer is larger), the impact of additional writes to the log files (by the triple write logic) decreases.

You can see whether you are writing a lot of partial pages, by reviewing the output from amqsrua.

1 requester test for 2K message – amqsrua output:

```
Publication received PutDate:20171023 PutTime:12302392 Interval:10.000 seconds
Log - bytes in use 1610612736
Log - bytes max 1744830464
Log file system - bytes in use 1675640832
Log file system - bytes max 234127560704
Log - physical bytes written 1159884800 115987019/sec
Log - logical bytes written 290741505   29073784/sec
Log - write latency 25 uSec
Log - write size 5122
```

Log - current primary space in use 4.61%

Log - workload primary space utilization 9.35%

The output above shows a big discrepancy between the physical bytes written, and the logical bytes, indicating a lot of partial 4KB pages being written. At the other end of the scale, 60 requesters sending and receiving 200K messages shows very few partial page writes (i.e. the physical bytes written are close to the logical bytes written)

60 requester test for 200K message – amqsrua output:

Publication received PutDate:20171023 PutTime:12332393 Interval:10.000 seconds

Log - bytes in use 1744830464

Log - bytes max 1744830464

Log file system - bytes in use 1833332736

Log file system - bytes max 234127560704

Log - physical bytes written 26274746368 2627408952/sec

Log - logical bytes written 26274623928 2627396708/sec

Log - write latency 181 uSec

Log - write size 3673301

Log - current primary space in use 67.21%

Log - workload primary space utilization 85.96%

Note that you don't have large messages and lots of applications to see the benefit of efficient log buffer page use. A test with only 12 requesters, and 2KB messages is already showing the logical and physical byte count to have converged to a more efficient point.

Publication received PutDate:20171023 PutTime:12352393 Interval:10.000 seconds

Log - bytes in use 1610612736

Log - bytes max 1744830464

Log file system - bytes in use 1685499904

Log file system - bytes max 234127560704

Log - physical bytes written 2544381952 254433488/sec

Log - logical bytes written 2361596215 236155253/sec

Log - write latency 23 uSec

Log - write size 10076

Log - current primary space in use 24.48%

Log - workload primary space utilization 24.70%

2.4 How Fast is Fast Enough – Bandwidth, or Latency?

In all of the tests so far, we have run unrated requesters, so we optimise the effects of log data aggregation for file writes. This *also* means that we see the effect on round trip rate as soon as we move from a fast filesystem to a slower one, or increase the latency on an NFS network link. In this way, we see that NFS over a 1Gb Ethernet link is slower than a SAN disk over an 8Gb fibre channel for instance. Often the real question is 'can I drive my workload at x requests/sec, with my filesystem', or 'can I drive my workload at x requests/sec, and maintain a response time of < y milliseconds'.

As we have seen, the MQ logger aggregates data in log writes, if certain criteria are fulfilled:

1. There are multiple applications accessing the queue manager.
2. These applications either
    a. Do not all update the same queue (i.e. multiple queues are being updated in a workload)
    b. Issue MQPUTs/MQGETs to a queue inside a syncpoint
    c. Access the QM using a combination of a & b (this is the ideal pattern)
3. During the time between two log writes, more than one update is logged (though not necessarily committed, see below).

It's important to understand the third point. Let's imagine an application with the following usage pattern (where all MQPUTs are inside syncpoint control):

Time	Action
10.01.02.034	Application A puts message1 on Q1
10.01.04.000	Application A commits message1
10.01.04.002	Application B puts message2 on Q1
10.01.04.005	Application B commits message2
10.01.04.010	Application C puts message3 on Q2
10.01.04.015	Application D puts message4 on Q1
10.01.04.020	Application E puts message5 on Q2
10.01.04.025	Application F puts message6 on Q2
10.01.04.027	Application C commits message3
10.01.04.030	Application D puts message7 on Q3

At 10.01.04.00, the commit, issued by application A, will cause a log write (containing at least message1 and the commit record). While MQ waits for the write to complete, it continues to write data into the log buffer, from other applications, and the commit by application at 10.01.04.005 means that there is now another write outstanding, which cannot be executed until the write associated with the first commit has completed.

Whilst the MQ logger waits for the first write to commit, additional log records are being written into the buffer. These can be message data and/or commit records.

Depending on the time it takes for the initial log write, the subsequent log write will aggregate more or less data:

Write Response time	Subsequent log write
<10 ms	Message2 + commit(msg2)
12ms	Message2 + commit(msg2) + message3
31ms	Message2 + commit(msg2) + message3 + message4 + message5 + message6 + commit(msg3) + message7

Note that the amount of data written to the log will depend on the latency of the filesystem, *and* the rate of delivery of new requests (as well as the message size).

If we fix the rate of delivery, then the write size is more dependent on the latency of the file system, or to put it another way, MQ is able to increase the write size to the filesystem, if latency increases, maintaining the throughput (up to a point).

Taking an NFS scenario, we can run unrated vs rated tests to see this in action. In the charts below, three scenarios are shown with MQ logging to an NFS filesystem across a 10Gb network link, but with increasing network delays (0µs to 4000 µs, i.e. 4ms), injected into the nfs network link, using the Linux network traffic controller program *tc*. Note that this delay is *each way,* so a 50µs delay will result in at least an additional 100µs on an nfs write.

The three scenarios tested, all used 20K messages with:

1. Single unrated requester. This is the same test presented in the tables above.
2. Sixty unrated requesters. This is the same test presented in the tables above.
3. Sixty rated requesters. In this scenario, the requesters were rated at ten requests/sec, for an overall rate of 600 round trips/sec.

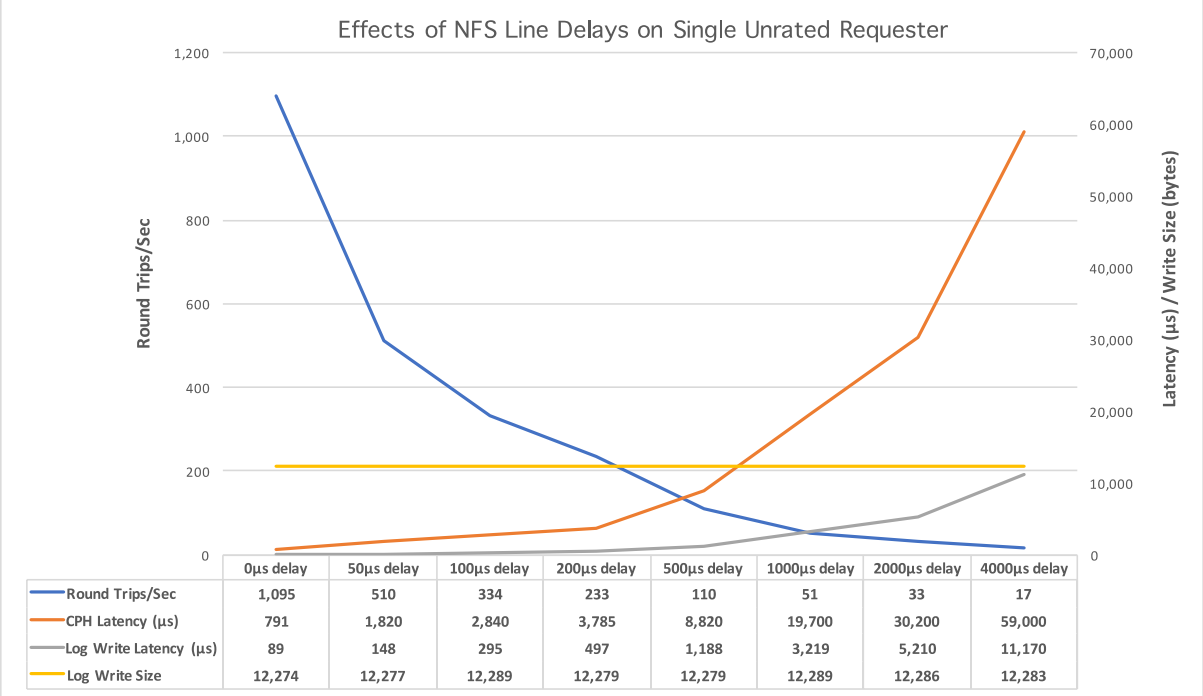
Note that in the following charts the following data is plotted.

Round Trips/Sec: Where a round trip comprises the MQPUT onto the request queue, and the MQGET from the reply queue (which by definition, includes the MQGET off the request queue and MQPUT onto the reply queue, by the responder application).

CPH Latency: The response time in µs of a round trip.

Log Write Latency: The average latency in µs of a log write to the nfs file system, as reported by amqsrua.

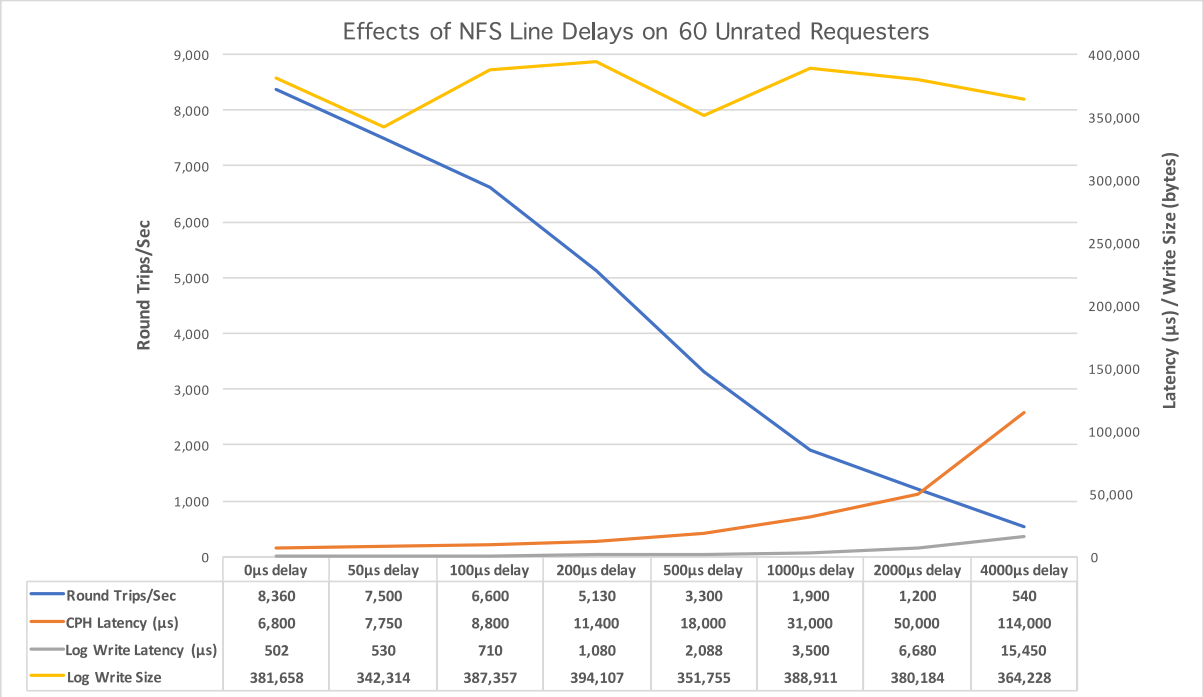
Log Write Size: The *average* size of a log write to the nfs file system, as reported by amqsrua (MQ V9.0.4 onwards). Note that some writes (e.g. logging the MQPUT) will be larger than others.



	0µs delay	50µs delay	100µs delay	200µs delay	500µs delay	1000µs delay	2000µs delay	4000µs delay
Round Trips/Sec	1,095	510	334	233	110	51	33	17
CPH Latency (µs)	791	1,820	2,840	3,785	8,820	19,700	30,200	59,000
Log Write Latency (µs)	89	148	295	497	1,188	3,219	5,210	11,170
Log Write Size	12,274	12,277	12,289	12,279	12,279	12,289	12,286	12,283

Figure 22 - Single Unrated Requester Logging to Delayed NFS

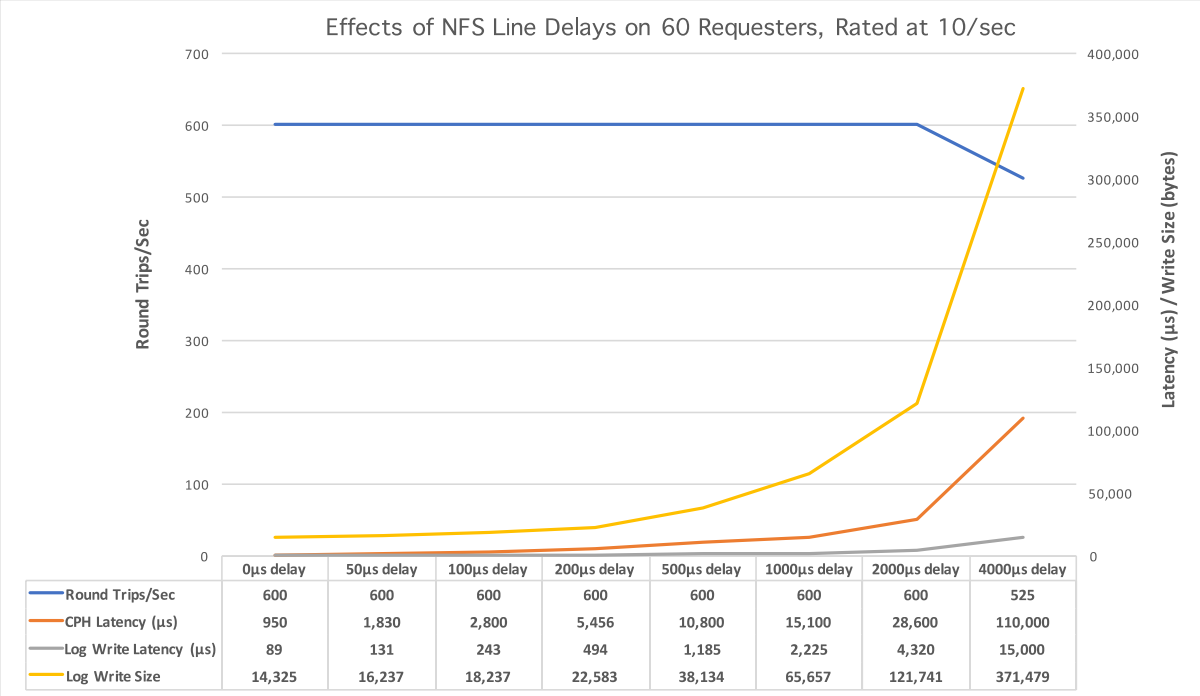
A single, unrated requester results in serialised logging, as there is no concurrency in the application, so the average log write size remains at around 12.2KB. Since the test is running as fast as it can, logging at this size, an increase in the latency of the log writes, caused by the network delays introduced, results in a corresponding, and immediate decrease in the total throughput (round trips/sec), and an increase in application latency (CPH latency).



	0µs delay	50µs delay	100µs delay	200µs delay	500µs delay	1000µs delay	2000µs delay	4000µs delay
Round Trips/Sec	8,360	7,500	6,600	5,130	3,300	1,900	1,200	540
CPH Latency (µs)	6,800	7,750	8,800	11,400	18,000	31,000	50,000	114,000
Log Write Latency (µs)	502	530	710	1,080	2,088	3,500	6,680	15,450
Log Write Size	381,658	342,314	387,357	394,107	351,755	388,911	380,184	364,228

Figure 23 - Sixty Unrated Requesters Logging to Delayed NFS

With sixty unrated requesters (Figure 23), the log write size is much larger (around 390MB in these tests), but the requesters are running so fast that MQ is unable to aggregate any more data in the log writes with this number of applications as the latency of the filesystem increases (in much the same way as we see similar write sizes for the same number of requesters, across different file systems in the data from the previous sections). Once again, the test is running as fast as it can (logging at this size), and an increase in the latency of the log writes results in a corresponding, and immediate decrease in the total throughput (round trips/sec), and an increase in application latency (CPH latency).



	0µs delay	50µs delay	100µs delay	200µs delay	500µs delay	1000µs delay	2000µs delay	4000µs delay
Round Trips/Sec	600	600	600	600	600	600	600	525
CPH Latency (µs)	950	1,830	2,800	5,456	10,800	15,100	28,600	110,000
Log Write Latency (µs)	89	131	243	494	1,185	2,225	4,320	15,000
Log Write Size	14,325	16,237	18,237	22,583	38,134	65,657	121,741	371,479

Figure 24 - Sixty Rated Requesters Logging to Delayed NFS

Running the same test with sixty *rated* requesters (Figure 24), results in a very different profile. This is arguably more like the real world, with a rate determined by the business application rather than immediately affected by the limitations of the file system.

With a workload running at a steady rate of 600 round trips/sec, the application (CPH) reports a 0.95ms response time. As the latency of the nfs writes increase, MQ is able to maintain the application rate of 600 round trips/sec as there will be more data in the log buffer to write, so with an additional delay of 0.5ms on the wire (a total additional latency of 1ms on the round trip of an nfs write), the rate remains at 600/sec, but the latency seen by CPH is now ~11ms. This trend continues until the amount of data cannot be aggregated any further. When the delay is 4ms, the write size has reached the same value as the unrated test cases (i.e. the maximum) then the rate starts to drop. If we wanted to maximise the potential bandwidth of the file system, more concurrency would be needed.

2.5 Client Bound Latency

In part one of this paper, we showed the model of performance as a pipe, with the narrowest part, representing the limiting factor. The initial, unrated results presented in part 2, demonstrated the relative capabilities of some file systems, where applications are on the same host as the queue manager. This enabled us to demonstrate the limitations of the file system. Some other factors were also introduced, such as injecting delays into nfs links, demonstrating the impact of the network on I/O performance. All of the tests so far, are not limited by the delivery rate of the applications however (except in terms of how many application threads are running). In the real world, our application is often client bound programs, residing on another host and introduce their own levels of throttling.

Whenever you consider the performance of a filesystem, you must also consider the capability of the network being used to connect in the client bound applications.
The charts below demonstrate the potential impact of moving the applications 'off box'.

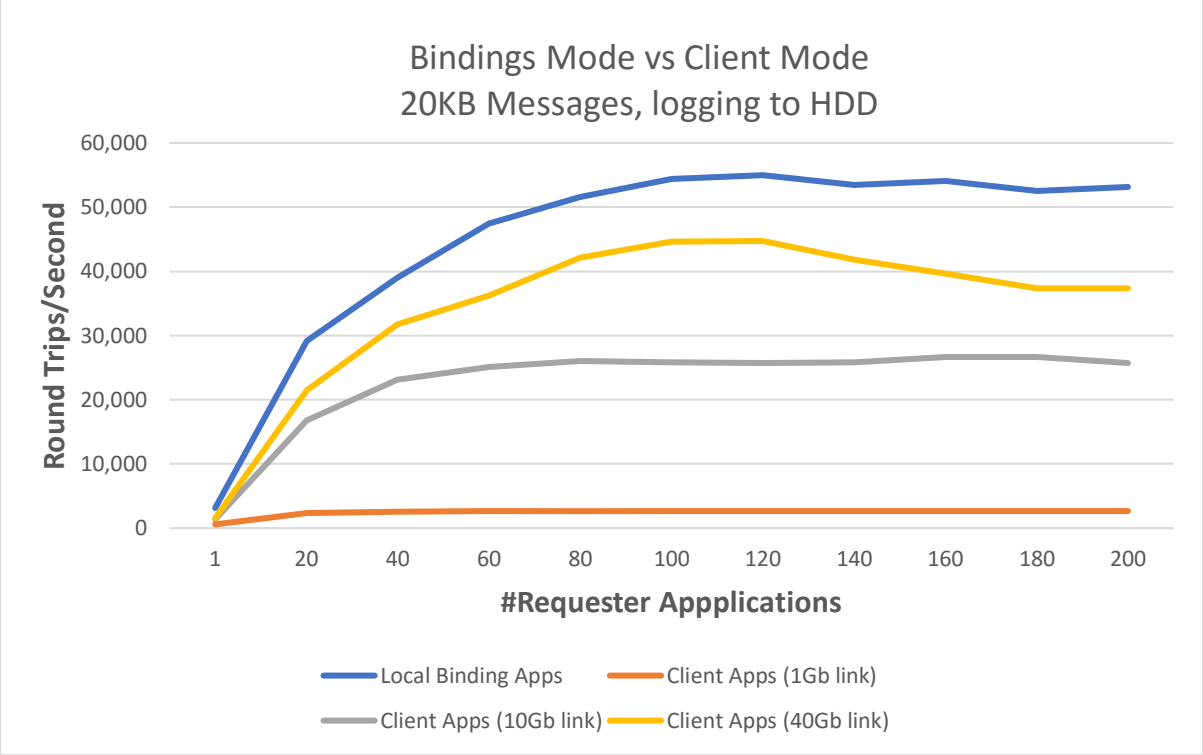


Figure 25 - Logging to HDD, with Bindings vs Client Mode Apps

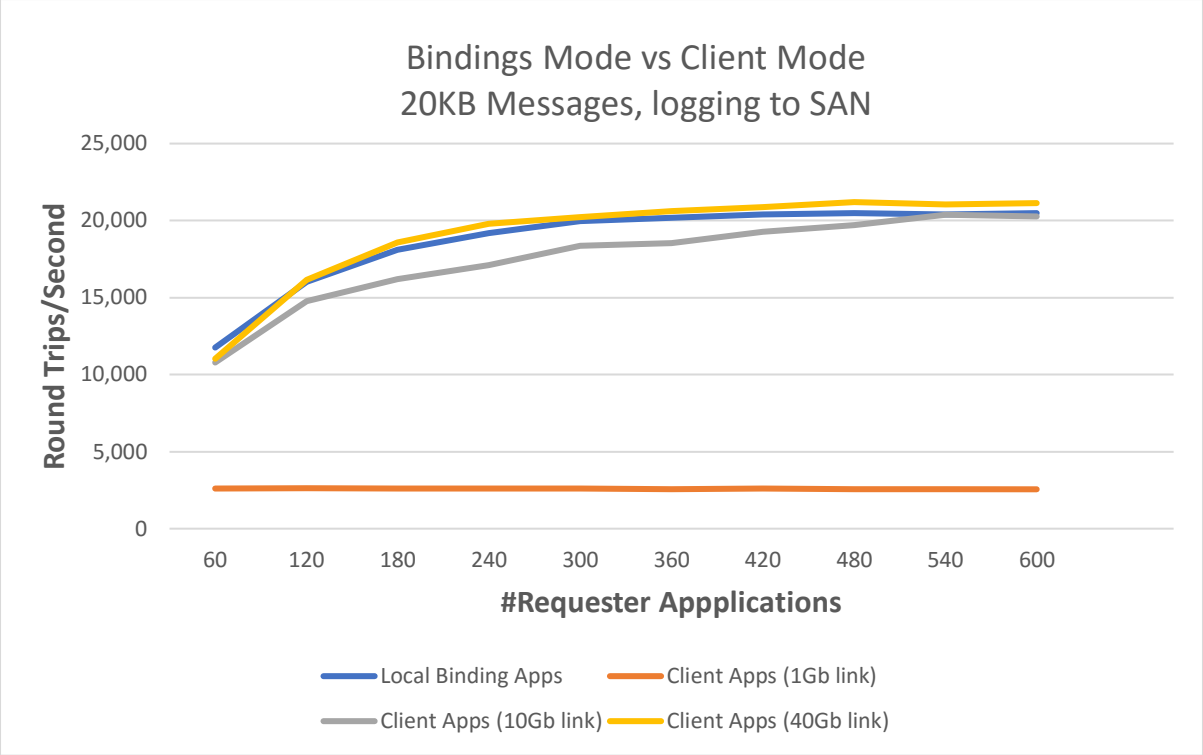


Figure 26 - Logging to SAN, with Bindings vs Client Mode Apps

Figure 25 & Figure 26 above show the impact of moving the local, bindings mode applications off the MQ host onto two additional, dedicated machines, connected via 1Gb, 10Gb, or 40Gb dedicated Ethernet links. The first comparison uses the local RAID cached disk, with a log that fits in the cache, the fastest I/O on test. The client bound applications now slow the rate of the test as we can't move data across the network as fast as we can log it in MQ.

The second scenario uses the higher latency SAN filesystem to log transactions. Now the impact of using client bound applications is not as significant, as the links can be fast enough for the applications to send and receive messages at the same rate that MQ is logging to SAN. The 40Gb scenario is even showing a slight improvement as out dedicated network links are faster than the 8Gb fibre channel links to the SAN, and we have off-loaded some of the CPU load to the remote machines. In general, you are likely to see a slowdown when anapplication is connected in client mode, in comparison to bindings mode applications.

2.6 So How Fast Will *My* Application Run?

The results presented here show that there are numerous factors dictating how fast a persistent application will run, including:

- Log size
- Write cache availability
- Concurrency
- Message Size
- Network speed
- Use of syncpoints
- Log Buffer Size
- Location of applications

It is very difficult to predict how fast a persistent application will run, as the performance is often dictated by the filesystem, and those environmental factors that affect the rate at which MQ can log its transactional data. We *can* predict how fast MQ is *capable* of processing messages of a given size, with a certain number of applications, in a laboratory environment, but real world performance requires performance testing to validate those numbers and extend them into usable metrics for your own application. The next section details a methodology for performance testing MQ persistent messaging applications, and the tools which can be used to drive, monitor, and measure those performance tests.

3 Part Three – Methodology and Tools

In this section, I will detail some of the tools used to record the data presented in this paper, and talk about performance testing methodology

3.1 Performance Testing Methodology: Divide and Conquer

Let's imagine a simple, scenario that requires testing:

200 JMS applications are typically expected to be connected to a queue manager hosted on a 4-core machine, logging to NFS. They will put messages onto a single queue, to be consumed by 4 JMS applications, which will get the messages and put a reply onto a second queue.

Each message is 20K in size, and we envisage the messages arriving at a total rate of 10,000/second.

We might be tempted to set up a test that closely matched the final scenario, using JMSPerfHarness, to simulate the 200 JMS Requesters, and 4 Responders. When we run the test, we see that the rate achieved is only 2,500/second. What is the bottleneck? At this point is could be:

1. Machine(s) hosting the JMS application is starved of CPU
2. Machine hosting the QM is starved of CPU
3. Queue locking across the single pair of queues is limiting throughput.
4. The network bandwidth between the applications and the queue manager host is not sufficient.
5. The network bandwidth between the MQ applications and the queue manager host is not sufficient.
6. There are not enough consumers processing the messages.

We can dive in with monitoring tools, but if we are in charge of the environment, then it's best to validate the capacity of the system starting from a simple scenario first, moving to the more complex solution, once we have proved the core performance. You can imagine the layers of an onion, with the simplest, fastest test being at the core. For MQ this would be a non-persistent, binding mode test. If your performance test does not meet its objectives running locally with, non-persistent measurements, then no amount of tuning of the filesystem is going to help (equally, if the filesystem is too slow, adding more CPU resource is not going to help, you'll know from your non-persistent tests, that this is not the case).

So how might we envisage a series of tests for the scenarios above?

1. Test with bindings mode CPH applications, and non-persistent messages
    - CPH will consume less CPU than JMSPerfHarness, so this is recommended, even if the final application will be JMS, keep things simple.

- Check CPU utilisation, is the machine big enough (you can monitor CPU with tools such as TOP, to see the resource usage of CPH, to factor that out).
- Test other factors, like spreading the load across multiple queues, increasing the number of responders etc.

2. Add in persistence
    - Monitor with amqsrua
    - Test filesystem outside of MQ with a tool like mqldt (see below).
    - Test the nfs network link for bandwidth.
3. If target is being met, then move applications off box onto a network with the same bandwidth as the production system
4. Change the applications from CPH to JMS

At each stage, we want to keep things simple, change one thing at a time, and at the same time match what is expected in production. Sometime this is not negotiable (perhaps the expected size of the messages), and sometimes your findings may need to be fed back into the design of the application (e.g. spreading the load across multiple pairs of queues).

There may be cases where an application is presented as a fait accompli. The methodology above may still need to be used, but in reverse (though I often turn to a non-persistent test as the first measure of performance).

Avoid trying to test MQ in an over-simplistic mode. Although the methodology above starts very simple, we want to run in an efficient mode from the start, so spreading the load across multiple queues is very likely to give performance gains. Another common test scenario seen is queue fill and drain. If we want to see how fast MQ can process messages we could see how long it takes to put 10,000 messages on a queue, and then how long it takes to drain the queue. This approach does not exercise MQ in an optimal way however, building deep queues may involve writing messages out to disk, to accommodate them, and reading them back when draining the queue. Unless this is a test emulating something specific that is likely to happen in production (in which case, it is likely to be a supplemental test) don't measure MQ performance in this way.

3.2 Tools Useful for Assessing Performance

3.2.1 MQI Workload Driver

All the results in this paper were measured whilst MQ was under load, using the MQ-CPH workload tool.

This is an MQI native a API tool that can simulate many application threads. There is an MQ blog article on the tool here: MQ-CPH Performance Harness Released on GitHub

MQ-CPH is a lightweight driver, and the recommended tool for establishing initial numbers, especially where MQ native application are being assessed. You can start multiple MQ-CPH processes from a number of driver machines if required, but you will need to collate the statistics from each, to arrive at the total message rate, for instance (as we do here).

3.2.2 JMS Workload Driver

The PerfHarness workload driver is available on GitHub here: https://github.com/ot4i/perf-harness

Running PerfHarness in its JMSPerfHarness mode can drive JMS messaging scenarios in a very similar way to MQ-CPH (the MQ-CPH interface was modelled on JMSPerfHarness).

Other third party JMS drivers are available but check that they support persistent messaging inside syncpoint (-pp -TX flags in JMSPerfHarness) to ensure best practices for performance (i.e. it uses JMS transacted sessions).

As MQ-CPH and JMSPerfHarness have an almost identical interface, it is often useful to test with MQ-CPH first. Every JMSPerfHarness application runs in a JVM, which may require a large heap to support the number of threads and message sizes being tested.

3.3 MQ Monitoring and Statistics

IBM MQ has a number of tools for monitoring and collecting information pertinent to performance.

See the main knowledge centre section here:

Monitoring and performance

3.3.1 Real Time Monitoring

Whilst none of the data presented in this paper required the use of real time monitoring, some of these metrics can be useful (e.g. nettime on a channel object, when a queue is remote). See 'Real-time monitoring'

3.3.2 Monitoring and Statistics

MQ can generate many statistics that are useful in performance evaluations. This functionality has been significantly enhanced in V9 (see Statistics published to the system topic in MQ v9).

3.3.2.1 AMQSRUA – Logger Statistics

In IBM MQ V9 onwards, a supplied sample statistics reporting program, amqsrua can be used to monitor the queue manager, and objects such as queues, channels etc. It was amqsrua that was used in this paper to establish log write latency, and log write size.

Invoke the tool with the DISK/Log options to display statistics including the number of bytes/sec being written to the log, the average log write latency, and (from V9.0.4), the average write size to the log.

```
[mqperf@mqperfm1]$ /opt/mqm/samp/bin/amqsrua -m PERF0
CPU : Platform central processing units
DISK : Platform persistent data stores
STATMQI : API usage statistics
STATQ : API per-queue usage statistics
Enter Class selection
==> DISK
SystemSummary : Disk usage - platform wide
QMgrSummary : Disk usage - running queue managers
Log : Disk usage - queue manager recovery log
Enter Type selection
==> Log
Publication received PutDate:20171004 PutTime:13555170 Interval:51.098 seconds
Log - bytes in use 1610612736
Log - bytes max 1744830464
Log file system - bytes in use 5327290368
Log file system - bytes max 21071134720
Log - physical bytes written 3904667648 76414183/sec
Log - logical bytes written 3898039989 76284480/sec
Log - write latency 364 uSec
Log - write size 322288
Log - current primary space in use 42.06%
Log - workload primary space utilization 56.82%

Publication received PutDate:20171004 PutTime:13560170 Interval:10.001 seconds
Log - bytes in use 1610612736
Log - bytes max 1744830464
Log file system - bytes in use 5327290368
```

```
Log file system - bytes max 21071134720
Log - physical bytes written 3660836864 366039835/sec
Log - logical bytes written 3655298220 365486037/sec
Log - write latency 306 uSec
Log - write size 304454
Log - current primary space in use 49.58%
Log - workload primary space utilization 62.59%
```

Further resources on MQ Resource Monitoring:
[IBM Knowledge Center - System topics for monitoring and activity trace]

Prior to V9.0.4 amqsrua did not report the average write size for the MQ logger. This *can* be retrieved in versions earlier than V9.0.4, using the amqldmpa service tool, though the output from this tool is subject to change, as it in an IBM internal tool, with no supported use by customers.

The following commands will retrieve the current writeSize statistic using amqldmpa (this was valid at the time of writing this document, but is subject to change, without notice).

Make sure that the file specified by $LDMPA_FILE does not exist, as this set of commands will append data to the file specified (unless this is what you want, to collect multiple datapoints).

```
QM=PERF1
LDMPA_FILE=/var/mqm/errors/ldmp.lggr.out

/opt/mqm/bin/amqldmpa -m $QM -c H -f $LDMPA_FILE -n 1
cat $LDMPA_FILE | grep WriteSizeLong

WriteSizeLong   :       388111
```

3.4 FileSystem Tools

It is often useful to test, and monitor performance of the filesystem outside of MQ. Whilst these tools cannot predict what write speeds can be obtained by a specific MQ setup (which is dependant, in great part, as we have seen, on concurrency, best practices etc), they *can* indicate capability, and flag problems. If you know for instance, the write size that the MQ logger is achieving in your performance test (through amqsrua), you can test the same write size with a tool such as MQLDT, to see if it is also constrained around the same value. If it is, then the filesystem needs to be re-evaluated for your needs.

3.4.1 MQ Log Disk Tester (MQLDT)

As we have seen, MQ writes to a file system in a particular way, to ensure data integrity. MQLDT is a Linux utility designed to test a filesystem by writing to it in the same fashion. As its use is specialised, very few parameters need to be set, and some of these can be derived from your own queue manager (using the qm.ini file).

MQLDT is available on GitHub here:
https://github.com/ibm-messaging/mqldt

Sample output:

```
Options (specified or defaulted to)
========================================================================
Write blocksize             (--bsize)              : 128K
Directory to write to       (--dir)                : /var/san1/testdir
Test file prefix            (--filePrefix)         : mqtestfile
Number of files to write to (--numFiles)           : 24
Size of test files          (--fileSize)           : 67108864
Test duration               (--duration)           : 20

Creating files...
Executing test for write blocksize 131072 (128k). Seconds elapsed -> 20/20

Total writes to files          :          46,790
Total bytes written to files   :   6,132,858,880
Max bytes/sec written to files :     311,689,216
Min bytes/sec written to files :     299,630,592
Avg bytes/sec written to files :     306,754,931

Max latency of write (ns)      :       4,669,311
Min latency of write (ns)      :         375,204
Avg latency of write (ns)      :         415,704
```

3.4.2 fio

fio is a popular, and more generalised, 3rd party file-system tester. It can produce very different results to MQLDT, if the parameters are not set correctly, so can be misleading.

An fio jobfile that will configure fio to test a filesystem in a similar way that MQ writes to it, can be downloaded here:

https://ibm-messaging.github.io/mqperf/samp/fio/fio-jobfile

The jobfile has sections to test different write sizes, to test 64K write for example:

fio fio-jobfile --section=write-64K

3.4.3 iostat

iostat is part of the sysstat package. It can be used to display detailed information about filesystem I/O.

Invoked with the –x (extended) option, it provides a host of metrics (see below, for sample output).

Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
sdb	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
sdc	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
sdd	0.00	0.00	0.00	2040.00	0.00	498480.00	488.71	4.40	2.15	0.00	2.15	0.39	79.15
sde	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
sdf	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
sdg	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
sdh	0.00	0.00	0.00	2040.00	0.00	521520.00	511.29	5.22	2.55	0.00	2.55	0.45	91.85
sdi	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
sdj	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
sdk	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
dm-0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
dm-1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
dm-2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
dm-3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
dm-4	0.00	0.00	0.00	4080.00	0.00	1020000.00	500.00	10.09	2.47	0.00	2.47	0.24	98.10
dm-5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
dm-6	0.00	0.00	0.00	4080.50	0.00	1020092.00	499.98	10.07	2.46	0.00	2.46	0.24	97.65
dm-7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
dm-8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
dm-9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
dm-10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	

Whilst this is a useful, and powerful tool, care must be taken interpreting some of the numbers. It is not advised to use svctm (deprecated), for instance, as this is a calculated number that does not always accurately represent the actual service time for modern devices. If you are using iostat to monitor latency, then use await numbers. The scope of await is broader (total time taken for an i/o operation, including time spent in o/s code), but is accurate. In the line above, await is 2.46ms, which reflected the time reported by MQ in amqsrua, whereas svctm is reporting 0.24ms.

3.5 Network

The network has the potential to throttle throughput as it may constrain the rate at which the applications can send or receive data, and it is part of the pipeline for NFS logging.

Tools such as iperf can be used to test the bandwidth of the network. Remember that if you are using NFS, the message data and the logging data may all be contending the same bandwidth of the network. Netstat can provide good data on network usage, use netstat –I to monitor the interface traffic. Combined with a knowledge of what bandwidth your network is capable of, you can use this to see if the host is approaching its limit.

3.6 Other System Monitoring Tools

There are various tools that can be used to monitor system performance. On Linux, you can use, amongst others:

sar	Part of the sysstat package, can be useful in recording performance metrics particularly.
vmstat	CPU monitoring
iostat	Covered above
top	General performance monitor

NMon	Open source performance monitor (and part of AIX) http://nmon.sourceforge.net/pmwiki.php
dstat	Alternative for vmstat, iostat, netstat and ifstat

These tools can be used in combination, to understand what resources are being used by your messaging host/queue manager(s). Aside from investigating resource use in a performance test environment, it is particularly useful to have metrics at hand for a production environment when everything is operating normally. If you have log write latency records for such times, for example, you'll easily spot if an increase in write latency looks likely to be causing a slowdown, if it occurs. Data from a lot of these tools can be very detailed, it's simpler to establish the root of a possible problem, if you know what the metrics *should* look like, when your application is running smoothly.