MQ for z/OS Performance Report (v1.0)

IBM MQ Advanced for z/OS 9.4
and
IBM MQ for z/OS 9.4

August 2024

IBM MQ Performance
IBM UK Laboratories
Hursley Park
Winchester
Hampshire
SO21 2JN

Take Note!

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

First edition, August 2024. This edition applies to IBM MQ for z/OS 9.4 and IBM MQ Advanced for z/OS 9.4 (and to all subsequent releases and modifications until otherwise indicated in new editions).

Notices

DISCLAIMERS

The performance data contained in this report were 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 for z/OS version 9.4**. The information is not intended as the specification of any programming interfaces that are provided by IBM MQ. Full descriptions of the IBM MQ facilities are available in the product publications. It is assumed that the reader is familiar with the concepts and operation of IBM MQ.

LOCAL AVAILABILITY

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

ALTERNATIVE PRODUCTS AND SERVICES

Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program,

or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

USE OF INFORMATION PROVIDED BY YOU

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

TRADEMARKS and SERVICE MARKS

The following terms, used in this publication, are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both:

- IBM®
- z/OS®
- IBM MQ®
- CICS®
- Db2 for z/OS®
- IMS™
- MVS™
- z15™
- z16™
- FICON®
- WebSphere®

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.

Summary of Amendments

Date	Changes
2024-08	Version 1.0 - Initial Version.

Table of contents

Chapter 1

Performance highlights

This report focuses on performance changes since previous versions (9.2 and 9.3) and on the performance of new function in this release.

SupportPac MP16 "Capacity Planning and Tuning Guide" will continue to be the repository for ongoing advice and guidance learned as systems increase in power and experience is gained.

In IBM MQ for z/OS 9.4, there are a number of features that can benefit the performance of the MQ queue manager:

zHyperLink for MQ active logs can increase the peak log rate of the MQ log task by up to 5.9 times over traditional asynchronous I/O.

64-bit channel initiator decouples the impact of supporting large messages over SVRCONN channels from the limited 31-bit storage. This allows up to 2.5 times the number of concurrent SVRCONN connections that previous MQ releases were able to support when using messages up to 256KB. With the 31-bit footprint of the a SVRCONN becoming independent of the message size for messages larger than 32K, the capacity of the MQ channel initiator to support SVRCONN channels with larger messages is limited by the amount of 64-bit storage available. There are also instances where the MQPUT cost may be reduced by up to 31%.

Enhancements to the data collected as part of **queue statistics**, statistics trace class(5), provide significantly more data to assist the monitoring of specific queue usage.

IBM MQ for z/OS 9.4 updates the **streaming queue** feature, with additional optimisations to MQPUT1, all of which allows the configuration of a queue to put a near-identical copy of every message to a second queue. Streaming queues can be useful in certain scenarios, where a copy of the message is required. For example:

- Performing analysis on the data going through the system.
- Storing messages for recovery at a later time.
- Capturing a set of messages to use in development and test systems.
- Consuming IBM MQ event messages from the system event queues, and sending additional copies to other queues or topics.

As part of the Connector Pack with IBM MQ Advanced for z/OS 9.4, there is support for exactly-once delivery of messages using IBM MQ Kafka Connectors.

As introduced in IBM MQ for z/OS 9.3, all new queue managers have, by default, 64-bit log RBA and are capable of having up to 310 active logs. All measurements run in this report use 64-bit RBA queue managers.

Chapter 2

Existing function

General statement of regression

CPU costs and throughput are not significantly difference in version 9.4 for typical messaging workloads when compared with versions 9.2 and 9.3.

The reader can see how that statement has been determined by reviewing details of the regression test cases in the Regression appendix.

Storage usage

Virtual storage constraint relief has not been a primary focus of this release.

CSA usage

Common Service Area (CSA) storage usage is important as the amount available is restricted by the amount of 31-bit storage available and this is limited to an absolute limit of 2GB.

The CSA is allocated in all address spaces in an LPAR, so its use reduces the available private storage for all address spaces.

In real terms, the queue manager does not have 2GB of storage to use - as there is some amount used by MVS for system tasks and it is possible for individual customer sites to set the limit even lower.

From the storage remaining of the 2GB of 31-bit storage, a large (but configurable) amount of storage may be used by the queue manager for buffer pools. This storage usage may be reduced with the use of 64-bit buffer pools.

Note: From version 9.1, buffer pools allocated in 31-bit storage are being deprecated.

The storage remaining is available for actually connecting to the queue manager in a variety of ways and using IBM MQ to put and get messages.

Initial CSA usage

CSA usage for 9.4 is similar to both 9.2 and 9.3 when similarly configured queue managers are started. On our systems this is approximately 6MB per queue manager.

CSA usage per connection

CSA usage has seen little change in the following releases: 9.2, 9.3 and 9.4.

- For local connections, MCA channels and SVRCONN channels with SHARECNV(0), CSA usage is 2.47KB per connection.
- For SVRCONN channels with SHARECNV(1), CSA usage is approximately 4.9KB per connection.
- For SVRCONN channels with SHARECNV(5), CSA usage is approximately 3KB per connection, based on 5 clients sharing the channel instance.
- For SVRCONN channels with SHARECNV(10), CSA usage is approximately 2.7KB per connection, based on 10 clients sharing the channel instance.

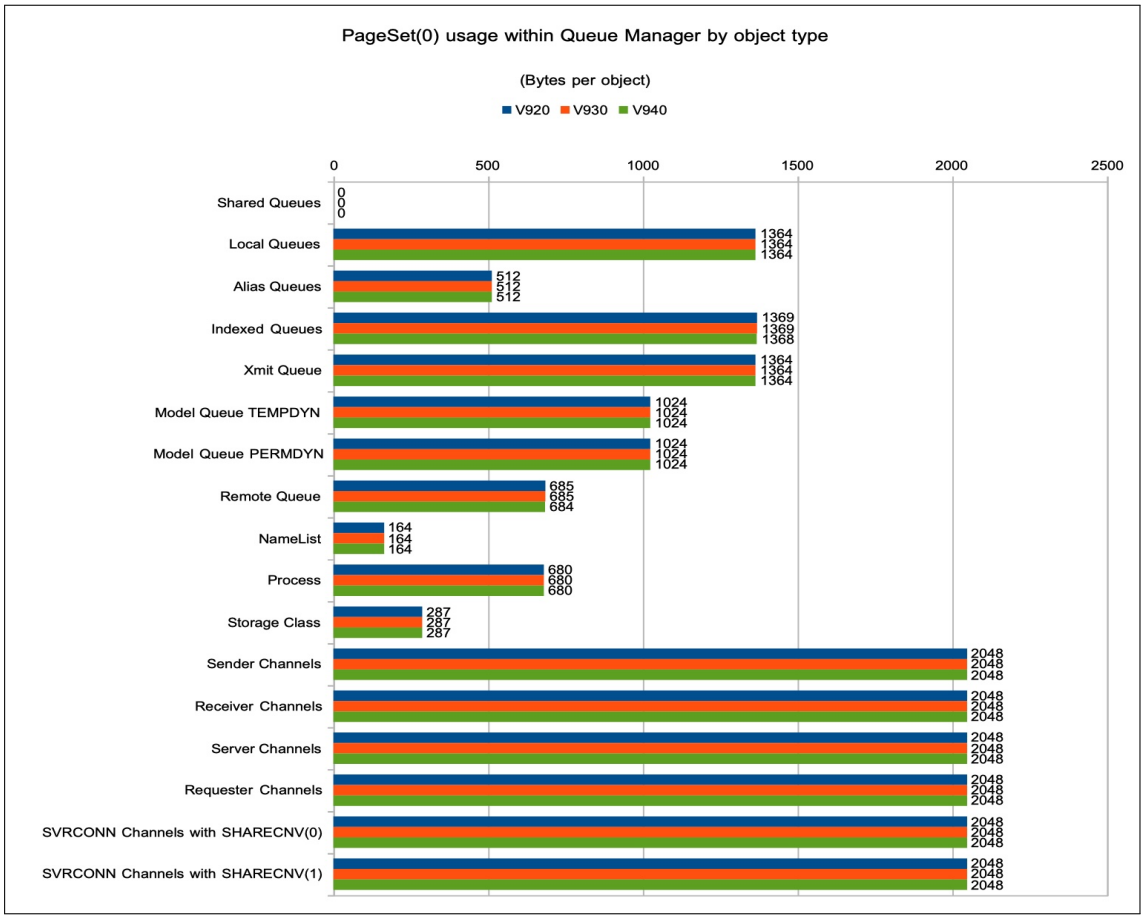
Object Sizes

When defining objects, the queue manager may store information about that object in pageset 0 and may also require storage taken from the queue manager's extended private storage allocation.

The data shown on the following 2 charts only includes the storage used when defining the objects.

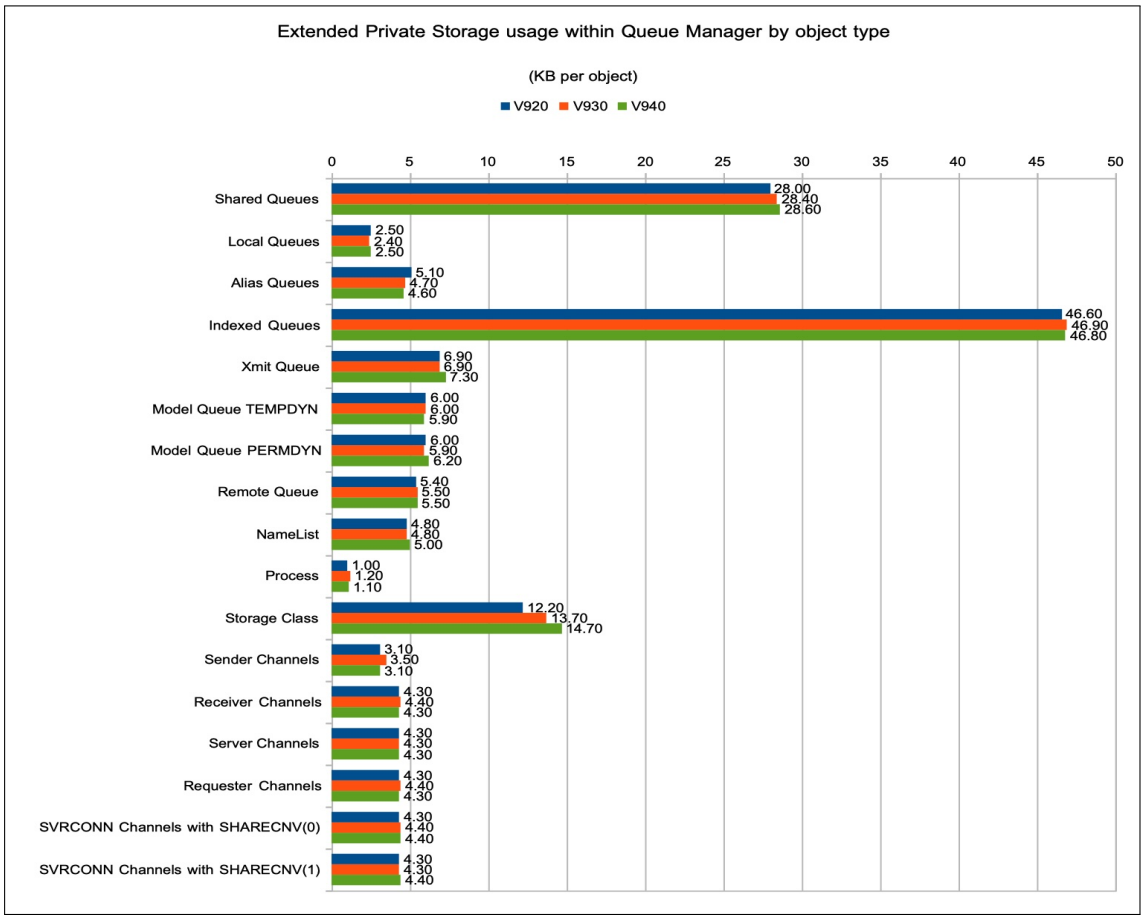
PAGESET(0) Usage

Chart: Pageset usage by object type



Virtual Storage Usage

Chart: Virtual Storage usage by object type



Capacity of the queue manager and channel initiator

How much storage does a connection use?

When an application connects to a queue manager, an amount of storage is allocated from the queue manager's available storage.

Some of this storage is held above 2GB, such as security data, and other data is storage on storage taken from that available below the 2GB bar. In the following examples, only allocations from below the 2GB bar are reported.

From v9.0-onwards, the typical storage usage is approximately 16.8KB per connection, however there is additional usage in the following (non-exhaustive) cases:

- Where connection is over a SHARECNV(1) channel, the usage increases to 27KB.
- Where connection is over a SHARECNV(10) channel and has a CURSHCNV of 10, the usage is 18KB per connection (186KB per channel).
- Where connection is over a SHARECNV(1) channel to shared queues - either CFLEVEL(4) or CFLEVEL(5) backed by SMDS, the storage is 34KB, giving an additional shared queue overhead of 7KB.

These numbers are based upon the connecting applications accessing a small number of queues. If your application has more than 32 objects open, the amount of storage will be increased.

If the number of messages held in a unit of work is large and the size of the messages is large then additional lock storage may be required.

How many clients can I connect to my queue manager?

The maximum number of clients you can connect to a queue manager depends on a number of factors, for example:

- Storage available in the queue manager's address space.
- Storage available in the channel initiator's address space.
- How large the messages being put or got are.
- Whether the channel initiator is already running its maximum number of connected clients (either 9,999 or the value specified by channel attributes like MAXCHL).

In IBM MQ for z/OS 9.4, some of the storage required by client channels has been changed to use 64-bit storage.

The 64-bit storage usage is discussed further in the "Increased channel initiator capacity for SVRCONN channel" section.

The table below shows the typical **31-bit** footprint when connecting a client application to a z/OS queue manager via the channel initiator.

The value used in the SHARECNV channel attribute can affect the footprint and consideration as to the setting should be taken. Guidance on the SHARECNV attribute can be found in SupportPac MP16 "Capacity Planning and Tuning Guide".

		31-bit Channel initiator footprint (KB / SVRCONN channel) Message size (KB)			
IBM MQ release	**SHARECNV**	**1**	**10**	**32**	**64**
9.2	**0**	86	101	159	193
9.3	**0**	86	100	159	192
9.4	**0**	86	87	93	93
9.2	**1**	168	190	245	277
9.3	**1**	168	190	247	277
9.4	**1**	164	174	201	203
9.2	**10**	223	248	312	346
9.3	**10**	223	247	313	340
9.4	**10**	219	233	265	267

Note: For the SHARECNV(10) channel measurements, the channels are running with 10 conversations per channel instance, so the cost per conversation would be the value in the table divided by 10.

Example: How many clients can I run?

When the channel initiator was started it logged the following message prior to starting channels:

"`CSQX004I Channel initiator storage usage: local storage: used 92MB, free 1379MB: above bar: used 187MB, free >10GB`".

This means that channel initiator had 1471MB of 31-bit storage available and has 1379MB available for channels to be started. To avoid letting the channel initiator run short on storage, it is advisable

to aim to keep the usage below 80% of the total available. In the example this means keeping the channel initiator storage usage below 1176MB, which in turn means that there is 1084 MB available for channels.

If the workload is expected to be clients connecting via SHARECNV(1) SVRCONN channels and using 32KB messages, we could predict that the 9.4 channel initiator could support a maximum of 5,522 running SVRCONN channels before using 80% of the available 31-bit storage.

How many MCA channels can I run to or from my queue manager?

This depends on the size of the messages flowing through the channel.

A channel will hold onto a certain amount of storage for its lifetime. This footprint depends on the size of the messages.

Message Size	1KB	32KB	64KB	4MB
Footprint (KB) per channel (channel initiator)	90	100	109	1127
Overhead of message size increase on 1KB messages		+10	+19	+1037

Chapter 3

New function for 9.4

This release has introduced a number of items that can affect performance and these include:

- Queue Manager:
  - Support for zHyperLink in MQ active logs
  - Additional data collected for queue statistics, introduced in IBM MQ for z/OS 9.3.
  - Improved performance of streaming queues, specifically using MQPUT1 with shared queues.
- Channel Initiator:
  - 64-bit storage for SVRCONN channels.
- MQ Advanced:
  - Kafka connectors supporting exactly-once delivery.

Each of these features will be detailed in subsequent chapters.

There are other new features in IBM MQ for z/OS 9.4, including promoting CAPEXPRY from a custom attribute to a "first-class" queue attribute, but as they have not resulted in any significant performance difference, they are not detailed in this report.

Chapter 4

zHyperLink support for MQ active logs

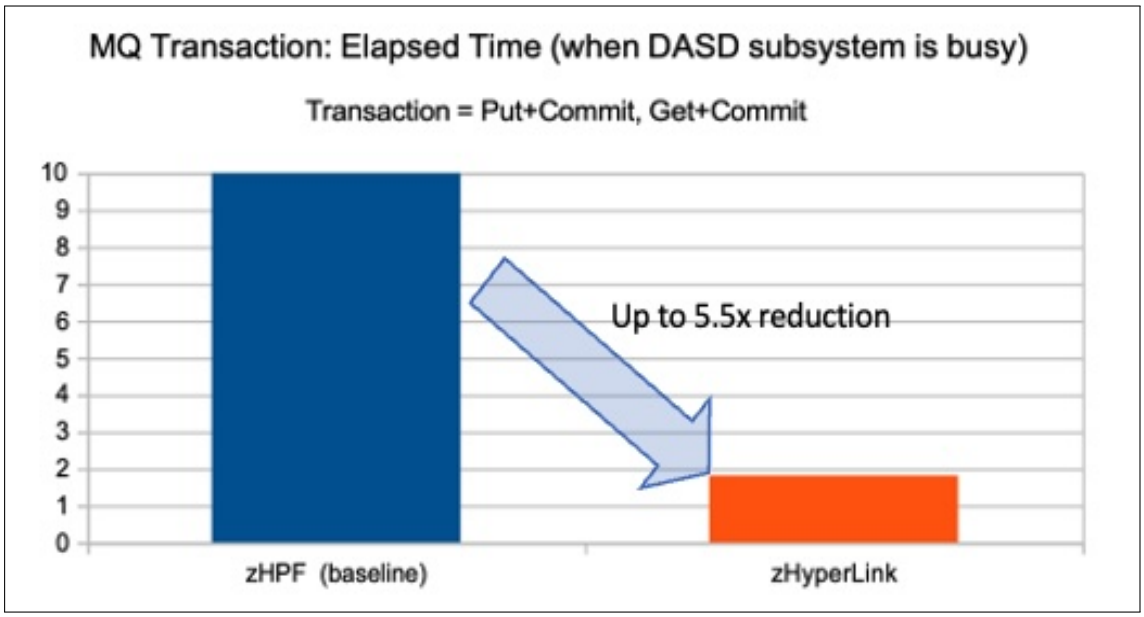
zHyperLink highlights

IBM zHyperLink is designed to provide an ultra-low latency link directly between the mainframe CPU and the I/O device.

IBM MQ for z/OS 9.4 implements support for zHyperLink when writing to MQ active logs.

In our performance environment, this improved active log throughput and reduced IBM MQ transaction times by up to 3.7 times. When running "real-world" scenarios, where the MQ queue manager and other workload are both using the disk infrastructure, enabling zHyperLink resulted in improved active log throughput by up to 5.9 times and transaction times reduced by up to 5.5 times.

Chart: Reduced MQ elapsed time compared to zHPF



To clarify the differences between performance environment and "real-world":

Our performance environment is a dedicated system which in order to run consistent repeatable measurements has dedicated links to dedicated DASD. This means that there is little contention on the path from CPU to disk. It also means that for the non-zHyperLink measurements, there is no significant contention over the cables or into the cache on the DASD, and so the benefit of switching from traditional I/O to zHyperLink is less significant than in a busier environment.

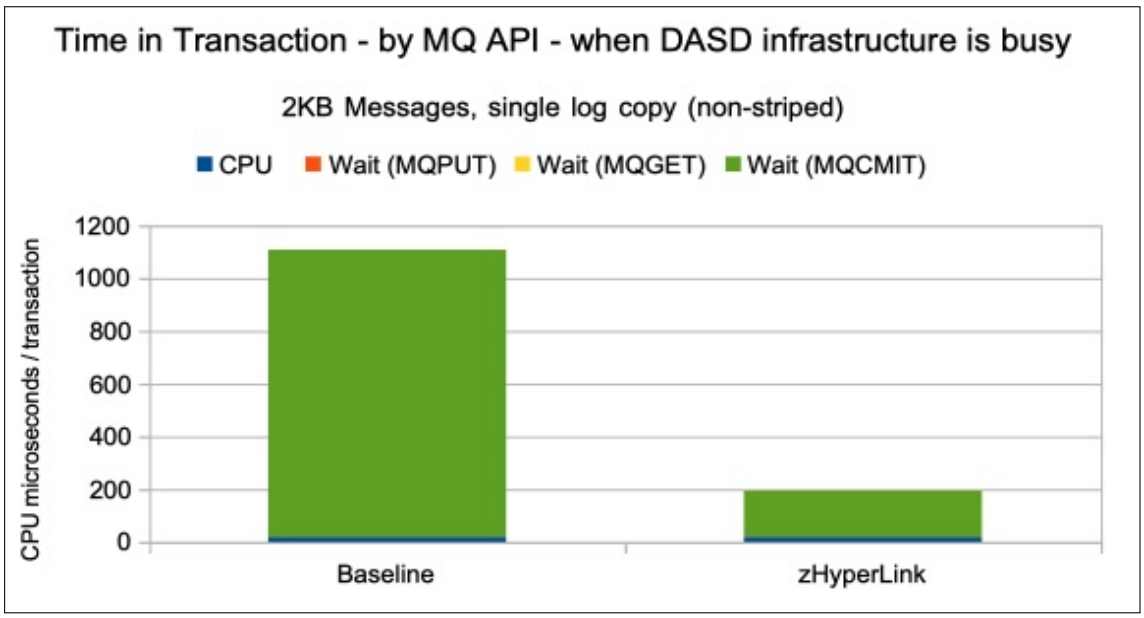
To simulate a more real-world environment, a number of simple batch tasks were run concurrently with the MQ workload, to read and write to a separate discrete set of VSAM files on the same DASD device as the MQ log data sets. This had the effect of using the existing channel paths and FICON links from the mainframe to the I/O device as well as using more disk cache. In turn, this had the effect of longer I/O times for the MQ log task. Switching MQ to the zHyperLink configuration removed the effects of contention from the channel paths and FICON links, resulting in a more significant improvement in MQ performance.

Later in this chapter, the impact of zHyperLink on MQ transactions and log rate will be compared in both the performance and the real-world environment.

With a persistent messaging workload, the elapsed time of the MQ commit is often a major component in the time spent completing a transaction. In the following two charts, a single application is using an MQPUT, MQCMIT, MQGET, MQCMIT model of processing.

The data in the chart is collected from MQ's class(3) accounting data, where the CPU cost of the MQPUT, MQGET and MQCMITs are combined and the "wait" time for each of those API's is displayed separately. The wait time is calculated from the elapsed time minus the CPU cost of the API.

Chart: Time in Transaction by MQ API for 2KB persistent workload

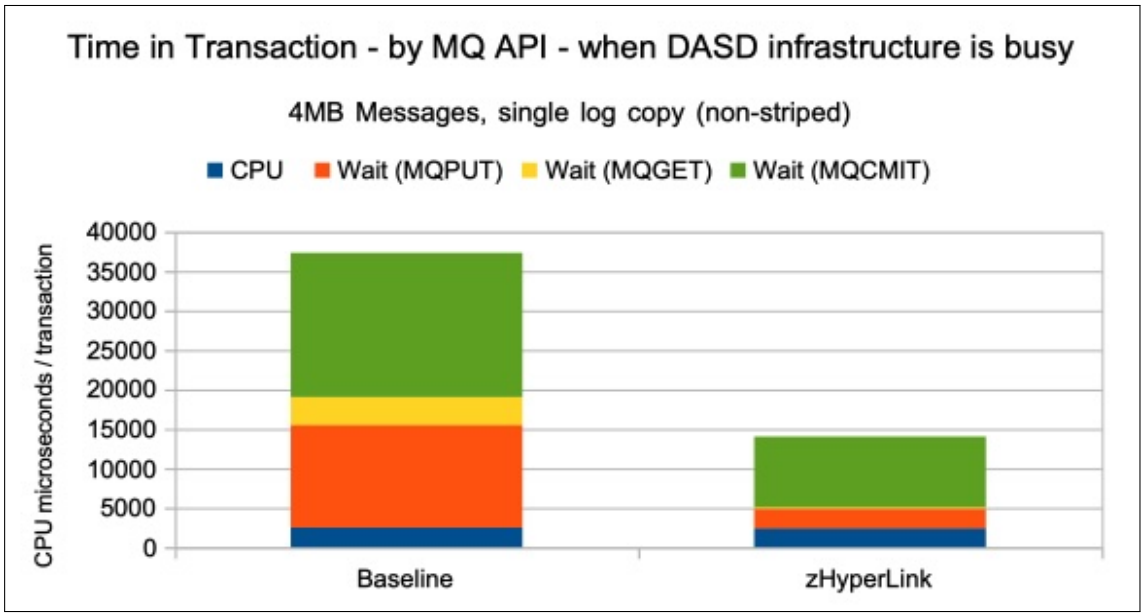


Notes on chart

- With this 2KB persistent messaging workload, the wait time for the MQPUT and MQGET is negligible.
- The API cost, as reported by class(3) accounting, is similar in the baseline and zHyperLink measurements.

- The wait time for the MQCMITs is significantly reduced in the zHyperLink configuration such that the elapsed time is 16% of the baseline measurement.

Chart: Time in Transaction by MQ API for 4MB persistent workload



Notes on chart

- The API cost, as reported by class(3) accounting, is similar in the baseline and zHyperLink measurements.
- The wait time for the MQPUT when using the zHyperLink configuration is 20% of the baseline measurement.
- The wait time for the MQGET when using the zHyperLink configuration is 5% of the baseline measurement.
- The wait time for the MQCMITs, which remains the largest component of the elapsed time even when using the zHyperLink configuration, is half that of the baseline configuration.
- The reduced wait times of the MQ APIs with zHyperLink means that the overall time in the transaction is reduced by 63%

In terms of peak logging rates when using zHyperLink, there were instances where the MQ queue manager was able to log persistent messages at a rate of 1 GB per second.

What is zHyperLink?

The "Getting Started with IBM zHyperLink for z/OS" Redbook offers a good introduction, as well as guidance on planning and deployment of zHyperLink.

IBM zHyperLink technology is designed to reduce input/output (I/O) latency by providing a fast, reliable, and direct communication path between the CPU and the I/O device.

This is achieved through zHyperLink adapters on the z/OS host, select IBM storage hardware, and connecting them using zHyperLink cables. This creates a point-to-point connection between CPU and I/O device, which reduces the I/O response time by up to 10 times compared to IBM z High-Performance FICON (zHPF). Such a low response time is achieved using synchronous I/O requests.

Asynchronous (Traditional) I/O:

Asynchronous I/O is a form of input/output processing that permits other processing to continue before the transmission on the storage pathway completes. When the operating system processes a block storage request, the CPU is released to perform other tasks, while the hardware device completes the storage operation.

Standard I/O processing that is available, such as zHPF, requires I/O operations to perform a series of time-consuming tasks, including:

- z/OS dispatching
- Interrupt handling
- CPU queue time
- Reload L1/L2 processor cache

These tasks and others that are required for I/O processing cause the I/O response time to be relatively long compared to transferring data within virtual storage, with response times of 110+ microseconds as well as interrupt handling and CPU dispatch time.

Synchronous (zHyperLink) I/O:

Synchronous I/O means that the entire path that handles an I/O request stays within the process context that started the I/O. When synchronous I/O is performed, the CPU waits (or "spins") until the I/O is completed, or the timeout value is reached.

zHyperLink can significantly reduce the time that is required to complete the I/O because the dispatching, interrupt handling, CPU queue time, and CPU cache reload activities are no longer necessary.

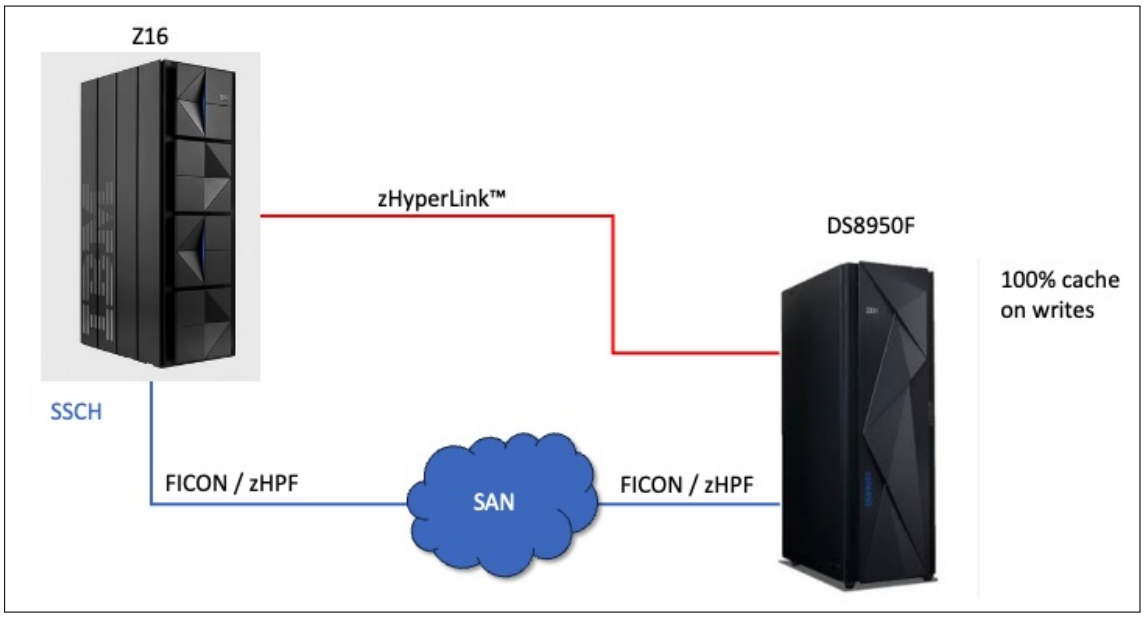
When using zHyperLink to write, DS8000 cannot meet the latency requirements of a random write because too much overhead exists in the DS8000 to acquire lock tracks, assign Non-Volatile Storage (NVS) space, and so forth. Therefore, only writes that follow a log-like pattern are supported. IBM MQ for z/OS 9.4 only uses zHyperLink to write to its active logs.

zHyperLink write operations require that z/OS establishes a zHyperLink session with the IBM DS8000, which consists of issuing a special channel program to establish zHyperLink write access. The channel program returns a token that is used for subsequent I/Os to that data set and causes the DS8000 to assign NVS space and set up descriptors to move data.

Current DS8000 technology limits the number of tokens to a single DS8000 to 64 concurrent write tokens.

The following diagram demonstrates the different routes that I/O requests will use for asynchronous I/O and zHyperLink I/O.

Diagram: Asynchronous and Synchronous I/O paths from mainframe to disk



zHyperLink and striped active logs

MQ uses Media Manager to write its I/O to disk, and this traditionally allows up to 128 4K pages to be written in a single I/O request, i.e. the maximum I/O for a single request is 512KB. Historically, the use of striping (4-stripes) would offer log rate improvements when MQ attempts to write 16K or more in a single I/O, such that a 128 page I/O request will write 32 pages (or 128KB) per stripe.

zHyperLink I/O is track-based, i.e. a zHyperLink I/O request will write no more than 56KB in a single request. In this case, 4-way striped logs would write 19KB per I/O request, which is less effective than traditional I/O's 128KB per stripe.

Db2's current recommendation is that whilst Db2 supports striped active log data sets, striping is generally unnecessary with the latest devices, and is not recommended in most cases.

MQ does not have the same recommendation but may be worth evaluating whether striped logs are still necessary when implementing zHyperLink on your MQ queue manager.

Single Page I/O Response Times

To offer an indication of the difference in response times observed on our systems, the following table details how long a single page of I/O takes in a range of configurations.

The data has been extracted from MQ's statistics trace class(3) data and formatted using tooling included in MP1B "Interpreting accounting and statistics data".

Configuration	I/O response time (microseconds)
zHPF disabled, quiet system	181
zHPF enabled, quiet system	115
zHPF enabled, busy system	200
zHyperLink, quiet system, 30 metre cable	28
zHyperLink, busy system, 30 metre cable	31

Comparing the single page I/O on quiet systems, zHyperLink offers a 4x improvement over zHPF and 6x improvement over non-zHPF response times.

On our "real world" environment, as described in the zHyperLink highlights section, for single page I/O requests zHyperLink offers a 6.4x improvement over zHPF response times (200 microseconds reduced to 31 microseconds).

Why might you use zHyperLink?

zHyperLink is intended to reduce the I/O times by providing a dedicated link between IBM z mainframe and the storage device.

On a naturally busy system, I/O's will be routed over a set of channel paths and FICON links into the storage device, and the response times of MQ active log writes can be affected by other workload's use of those common resources.

Using zHyperLink means that the I/O traffic is taken off of those common channel paths and FICON links and re-located onto the dedicated zHyperLink cable, which has 2 effects - firstly to move to a less contended path to disk and secondly to reduce the load on the shared channel paths and FICON links which may offer some relief to the workload that remains.

However the use of zHyperLink with MQ active logs does come at a cost. As the I/O is synchronous, the MQ logger task spins whilst the I/O completes. This means that once the I/O does complete, z/OS does not need to re-dispatch the MQ logger task, but whilst the I/O time is reduced, for each microsecond of I/O time, the same amount of CPU is used, i.e. a zHyperLink I/O that takes 50 microseconds to complete will use 50 CPU microseconds, and since the MQ logger task runs as a high-priority SRB, that potentially in a high CPU utilisation logical partition may prevent other tasks completing as quickly as they might when MQ uses asynchronous I/O.

The installation might determine that the potential increase in MQ SRB time that is associated with zHyperLink cannot be justified, even with the reduced latency that synchronous I/O provides.

Requirements

Both the IBM Documentation "Faster log throughput with zHyperLink" and the blog "MQ 9.4: Faster log throughput with zHyperLink" describe the following system requirements in order to use zHyperLink for MQ active logs:

Hardware:

- IBM z14 or later.
- DS8880 or later.

Software:

- zHyperLink Express is supported on z/OS 2.3 or later.
- The z/OS image must be run in an LPAR, not as a guest in a IBM z/VM.
- zHyperLink requires IBM z High-Performance FICON (zHPF) to be enabled.

If your environment will use PPRC and you intend to test the impact of zHyperLink in a loopback configuration such that the primary and secondary devices are on the same control unit, you will require APAR OA66234.

Using zHyperLink with IBM MQ active logs:

In order to use zHyperLink with the active logs of a queue manager, you need to:

- Configure IBM MQ to use zHyperLink, and
- Ensure the active logs are on zHyperLink capable volumes.

You can configure IBM MQ to use zHyperLink by using one of the following methods:

- Specify ZHYLINK(YES) in the log parameters.
- Issue the command `SET LOG ZHYLINK(YES)`

Notes:

- zHyperLink requires that zHyperWrite is switched on. This means in order to use ZHYLINK, ZHYWRITE must also be switched on in the log parameters. When only specifying ZHYLINK(YES) when ZHYWRITE(NO) is set on the queue manager, the ZHYWRITE parameter automatically overrides to YES.
- Explicitly trying to set ZHYLINK(YES) and ZHYWRITE(NO) results in an abnormal completion of the SET LOG command.
- Setting ZHYLINK=YES in the ZPRM overrides ZHYWRITE to YES.

Monitoring your environment

System commands

The "Getting Started with IBM zHyperLink for z/OS" Redbook provides information on system commands that are used for displaying and managing resources that are related to zHyperLink, but the following are the commands that we found particularly useful in our testing and evaluation of the performance.

DISPLAY PCIE

The use of the `D PCIE` command shows the information that is registered for the device type with PCIe. The example shown below shows two zHyperLink PCIe I/O cards with eight ports in total. One of the PCIe I/O cards is in standby mode.

```
PFID      DEVICE TYPE NAME          STATUS  ASID  JOBNAME  CHID VFN  PN
00000030  8GB zHyperLink            ALLC    0019  IOSAS    0150 0001 1
00000031  8GB zHyperLink            ALLC    0019  IOSAS    0150 0002 1
00000032  8GB zHyperLink            ALLC    0019  IOSAS    0150 0001 2
00000033  8GB zHyperLink            ALLC    0019  IOSAS    0150 0002 2
00000060  8GB zHyperLink            STNBY                  011C 0001 1
00000061  8GB zHyperLink            STNBY                  011C 0002 1
00000062  8GB zHyperLink            STNBY                  011C 0001 2
00000063  8GB zHyperLink            STNBY                  011C 0002 2
```

To bring the PFIDs in standby to online, the following command may be used:

```
CF PFID(60-63),ONLINE
```

DISPLAY IOS,HYPERWRITE

The display IOS command with the HYPERWRITE parameter displays the status of the HYPERWRITE option that is defined in parmlib member `IECIOSxx` or set by way of the `SETIOS HYPERWRITE` system command. HyperWrite must be enabled for zHyperLink writes.

DISPLAY IOS,ZHYPERLINK

The display IOS command with the ZHYPERLINK parameter shows if zHyperLink is `ENABLED` or `DISABLED`. When zHyperLink is `ENABLED`, it describes what I/O operation is available, i.e. read and write.

DISPLAY IOS,ZHYPERLINK,LINK=ALL,DETAIL

This display IOS command with the `LINK=ALL,DETAIL` parameter is used to display detailed information for all links.

Offline PFIDs are not included in the display.

The number of successful and unsuccessful (timeout) read and write operations is provided in the response for each active link, for example:

```
          CU  Port     ---- CU Info --- -- PFIDs --
PCHID PN Link State    Mfg.Ser or WWNN  Avail Unav
0150  1  0080 Oper     IBM.0000000TJS31    4     0
 WWNN: 5005076307FFD76D
 PFID: 00000030 S/W State:  Allocated
   Success:    Read - 0 Write - 9,395,709
   Timeout:    Read - 0 Write - 2
   Diagnostic Info:  00100000
 PFID: 00000031  S/W State:  Allocated
   Success:    Read - 0 Write - 9,395,707
   Timeout:    Read - 0 Write - 1
   Diagnostic Info:  00100000
```

As well as `Success` and `Timeout` counts, you may see `Link Busy` counts. Any of these three category of counts are not returned if both read and write counts are zero.

For non-zero `Link Busy` counters, this means that z/OS has to retry the I/O operation on another PFID.

For non-zero `Timeout` counters, this means that z/OS has had to retry the I/O operation asynchronously.

DISPLAY M=DEV(devno),ZHYPERLINK

The display system configuration command `M=DEV(devno)` with the ZHYPERLINK parameter displays whether the specified device has zHyperLink capabilities. It also shows the reason why a device is not enabled for zHyperLink.

If the device displayed is enabled for zHyperLink, it provides information about the type of zHyperLink (READ|WRITE) and the number of zHyperLinks that are available for accessing the device.

Whether zHyperLink is entirely disabled for READ and WRITE or partially disabled for READ or WRITE, it displays why it is disabled. These include the following reasons:

- Processor does not support zHyperLink
- z/OS reasons (IECIOSxx, SETIOS), including zHyperLink is disabled for the system.
- zHyperLink reasons, for example there are no zHyperLinks available.
- Secondary device reasons when using Metro Mirror, for example there are no zHyperLinks available for the secondary device.

DEVSERV QDASD,TYPE=ALL

This command may be abbreviated to `DS QD,TYPE=ALL`.

The command displays diagnostic information about the status of direct access storage devices (DASD) and storage control units (SCU).

It can be used to correlate the specific volume(s) used by the MQ active logs and the storage control unit.

Use of the `TYPE` option will cause an I/O operation for each DASD in the system.

DEVSERV QDASD,MACH=<serial number>,ZHL

This command may be abbreviated to `DS QD,MACH=<serial number>,ZHL`.

The command displays zHyperLink write tokens that are allocated to datasets on the named disk subsystem. The `MACH=` parameter is the 10-byte SCU serial number of the disk subsystem that was obtained when using the `DS QD,TYPE=ALL` command.

One token represents a zHyperLink write session with the DS8000. As mentioned in the Synchronous I/O explanation, current DS8000 technology limits the number of tokens to a single DS8000 to 64 concurrent write tokens.

DISPLAY SMS,DSNAME(<MQ active log data set name>)...

The command in full is `DISPLAY SMS,DSNAME(<MQ active log data set name>),STATS(ZHLWRITE)[,RESET]`.

This display SMS command with the `STATS(ZHLWRITE)` parameter is used for displaying zHyperLink write statistics for the data set that is specified in the `DSNAME` parameter.

For example:

```
D SMS,DSNAME,STATS(ZHLWRITE) Start of Report
DATA SET MQMDATA.VTS1.LOGCOPY1.DS01.DATA
STATISTICS Since 07/19/2024 21:32:12.043383
SUMMARY
       TOTAL     %SYNC -------------%ASYNC---------------
 WRITE REQUESTS WRITES   SKIP LNKBSY   EST    MISC DISABL
          34791  99.88   0.01   0.00  <0.01   0.02   0.00
                       ----%ASYNC----
                         MISS  DELAY   DUAL
                         0.02   0.00   0.04
DEVICE STATISTICS
            TOTAL  %SYNC ---------------%ASYNC---------------------
SSID DEVNO WRITES WRITES   SKIP LNKBSY   !EST   MISC   MISS  DELAY
3110 0782B 294863  99.96  <0.01   0.00  <0.01   0.02 <0.01   0.00
D SMS,DSNAME,STATS(ZHLWRITE) End of Report
```

The output from the above display command of the "MQMDATA.VTS1.LOGCOPY1.DS01" data set reveals the following:

- The vast majority of I/Os were synchronous, with 0.12% of being asynchronous.
- The IGW289I message provides an explanation as to why there were synchronous writes to this data set.
- The "device statistics" section reports a much higher number of writes (294,863) compared to the data set total of 34,791. This is simply because multiple of the queue managers' active log data sets were defined on the same device, i.e. device number 0782B.

Adding the `RESET` parameter to the display command will clear the statistics that follow the display. The timestamp of the last `RESET` is displayed at the top of the output. If a `RESET` for the data set was never done, the time stamp is from when the data set was opened.

RMF

The z/OS Resource Measurement Facility (RMF) can be used to review the I/O characteristics of your MQ active log data sets. When the MQ active log data sets have been enabled for zHyperLink access, there are a few differences to be aware of:

- Channel Path Activity report - there may be a reduction in the utilisation and number of operations performed by the defined channel paths. How significant this reduction is, will depend on what else is using the channel paths.
- Direct Access Device Activity report - the storage group and device number used by the MQ active log data sets may see a significant decrease in the Activity Rate.
- Synchronous I/O Device Activity report - this report will show the synchronous I/O for each device using zHyperLink.

Channel Path Activity report

RMF Monitor I gathers data for this report automatically.

To produce the report, specify: `REPORTS(CHAN)`

Traditional asynchronous I/O to MQ active log data sets will use channel paths, and the I/O activity is included in the utilisation percentages and either the FICON or ZHPF operation rates. When successfully using zHyperLink to write the data to MQ active logs, the I/O is no longer passed through the channel paths and as a result, there may be a decrease in the usage.

Direct Access Device Activity report

RMF Monitor I gathers data for this report automatically with the default option `DEVICE(DASD)`.

To produce the report, specify: `REPORTS(DEVICE(DASD)`

The report provides an activity summary grouped by device number.

```
                              D I R E C T   A C C E S S   D E V I C E   A C T I V I T Y

                               RPT VERSION 3.1 RMF        TIME 21.35.57             CYCLE 0.200 SECONDS

TOTAL SAMPLES =   300   IODF = 03   CR-DATE: 06/05/2024   CR-TIME: 11.42.43     ACT: POR

                                              DEVICE   AVG  AVG   AVG  AVG  AVG   AVG  AVG  AVG    %      %      %
STORAGE  DEV   DEVICE   NUMBER  VOLUME PAV  LCU  ACTIVITY RESP IOSQ  CMR  DB   INT   PEND DISC CONN   DEV    DEV    DEV
 GROUP   NUM   TYPE     OF CYL  SERIAL           RATE     TIME TIME  DLY  DLY  DLY   TIME TIME TIME   CONN   UTIL   RESV
MQMLOG1  0782B 33909     65520  AALG12 1.0H 0006   4.117S .420 .008  .000 .000 .000  .038 .147 .227   0.09   0.15    0.0
```

Notes on DASD report:

The storage group `MQMLOG1` on storage number `0782B` appears to have a low activity rate of 4.117 per second, which in isolation appears to indicate that there is little activity to the MQ data set(s) stored on that device. The activity rate does however have an "S" appended to the rate i.e. 4.117S, which indicates that the device performed synchronous I/O requests and that detailed synchronous I/O performance measurements for this device are available in the Synchronous I/O Device Activity report section.

Synchronous I/O Device Activity report

The Synchronous I/O Device Activity report is only produced if at least one DASD device actively performed synchronous I/O requests using IBM zHyperLink technology.

For DASD devices actually used for synchronous I/O, the Synchronous I/O Device Activity report shows detailed IBM zHyperLink activity data:

```
                                S Y N C H R O N O U S   I / O   D E V I C E   A C T I V I T Y

          z/OS 3.1                  SYSTEM ID MVAA             DATE 07/19/2024           INTERVAL 00.59.999

                                    RPT VERSION 3.1 RMF        TIME 21.35.57             CYCLE 0.200 SECONDS


TOTAL SAMPLES =    300   IODF = 03   CR-DATE: 06/05/2024   CR-TIME: 11.42.43     ACT: POR

                                     - DEVICE ACTIVITY RATE - -- AVG RESP TIME --  AVG SYNCH I/O   %       %     %     %
STORAGE  DEV   DEVICE   VOLUME  LCU  -- SYNCH I/O --   ASYNCH  -SYNCH I/O - ASYNCH  TRANSFER RATE  REQ     LINK  CACHE --REJECTS--
 GROUP   NUM   TYPE     SERIAL          READ   WRITE      I/O   READ  WRITE    I/O   READ   WRITE  SUCCESS  BUSY  MISS  READ WRITE
MQMLOG1  0782B 33909    AALG12  0006   0.000 13567.3    4.117  0.000  0.115  0.420  0.000   592.9    99.98  0.00  0.00  0.00  0.02
..
                        LCU     0006   0.000 13567.3    4.117  0.000  0.115  0.420  0.000   592.9    99.98  0.00  0.00  0.00  0.02
```

Notes on Synchronous I/O Device Activity report:

The storage group `MQMLOG1` on storage number `0782B` reports the same low asynchronous activity rate of 4.117 per second as the earlier Direct Access Device Activity report. However in this report, details of the synchronous I/O are presented.

For the selected interval, there were 13567.3 synchronous write requests per second against device number 0782B. This resulted in 592.9 MB per second being written synchronously with the average I/O being 0.115 milliseconds. Synchronous I/O accounted for 99.98% of I/O requests on this device.

SMF 42 (DFSMS Statistics and Configuration)

SMF 42 subtype 6, specifically DASD data set level I/O statistics now includes two new Synch I/O sections and provides data on the many attributes including number of SyncIO write requests at a data set level.

Please refer to the SMF 42 subtype 6 documentation for further details.

The RMF post processing application `ERBRMFPP` does not provide a report option to format the SMF 42 subtype 6 records.

MQ Statistics

The MQ class(3) statistics trace has been updated to include zHyperLink I/O statistics.

The additional data can be formatted using the MQSMF application, part of MP1B "Interpreting accounting and statistics data", where the results will be found in the `LOG` DD card. Additionally a zHyperLink summary in CSV format may be generated to the `LOGHLCSV` DD card.

The following is an example of the `LOG` output for a queue manager with a single MQ active log copy that was running a simple put/get workload using 4MB persistent messages:

```
From 2024/07/17,05:38:31 to 2024/07/17,05:39:31, duration    60 seconds.
 Wait for buffers (should be 0):         6779 out of  14968773, 0%
 Total Number of pages written:      14984701
 Number of pages written/sec:          249745
 Amount of data written/sec:              975 MB/Sec
 Total Number of write requests:       168887
 Number of write requests/sec:           2814
 Pages written per I/O:                    88
 Total number of read requests:             0
 Total number of copy1 new logs used during interval:         14
 Total number of copy2 new logs used during interval:          0
 Total number of new logs used which are zHyperWrite capable:          0
 Total number of new logs used which are zHyperWrite enabled:         14
 Total number of new logs used which are zHyperLink capable:          14
 Total number of new logs used which are zHyperLink enabled:          14
 Total number of new logs used which are encrypted:          0
 Write_Wait 0, Write_Nowait 57669209, Write_Force 196, WTB 6779
 Read_Stor          0, Read_Active          0, Read_Archive         0, TVC         0
 BSDS_Reqs      15067, CIs_Created   14968773, BFWR           1054442, ALR         0
 ALW                0, CIs_Offload          0, LLCheckpoints       28
 Read_delayed       0, Tape_Lookahead       0, Lookahead_Mount      0
 Write_Susp     68504, Write_Reqs      168887, CI_Writes     14984701
 Write_Serl         0, Write_Thrsh     985938, Buff_Pagein          0
 Copy1_Used        14, Copy2_Used           0,
 Logs_ZHWC          0, Logs_ZHWE           14, Logs_encr            0
 _____,__  write requests,        CIs, Average I/O,     After I/O, pages/IO
                                      time in uSec,  time in uSec,
 Log 1, 1 page      21218,      21218,          29,             0,        1
 Log 1,>1 page     147669,   14963483,         370,             0,      101
 Standard deviation of first log, 1 page per I/O, response time +-       29
 Log 1, 1 page Longest I/O         852 at 2024/07/17,04:38:34.043256 UTC
 Log 1, 1 page Longest Request     194 at 2024/07/17,04:39:16.060978 UTC
 Log 1,>1 page Longest I/O       21610 at 2024/07/17,04:39:03.320261 UTC
```

```
  Log 1,>1 page Longest Request   21611 at 2024/07/17,04:39:03.320261 UTC
  Log write rate     975MB/s per copy
  Logger I/O busy :  0.35%
  Logger task busy:  96.6%
  zHyperLink write stats
   _____________________1 page,_______>1 page,_______Total pages
   Write requests         21218,        147669,          14984701
   Sync writes            21209,        147484,          14965521
   Async writes               9,           185,             19180
   ...switch                  1,            26,              2604
   Average async writes per log switch:  1.9
   Average async pages written per log switch:  186.0
  zHyperLink I/O stats
   __Pages,_______Longest I/O,_______Shortest I/O,_______Total I/O_______Avg I/O
         1,          852 uSec,            24 uSec,     629572 uSec,    29.7 uSec
        >1,         1978 uSec,            29 uSec,   54497499 uSec,   369.5 uSec
Standard deviation of first log, 1 page per I/O, response time +- 29
```

Notes on LOG extract:

- Text in blue is zHyperLink specific.
  - There is a count of capable and enabled zHyperLink logs used during the interval. The values for capable and enabled may differ, for example if the zHyperLink environment is disabled during the interval.
  - zHyperLink write statistics
    - This section provides counts of the number of write requests, both single and multi-page requests whilst providing a breakdown of synchronous and asynchronous requests
    - Additionally, at log switch MQ must relinquish the write token(s) on the current MQ active log and establish a new token on the next MQ active log, so there is a brief interval where synchronous requests are delayed. The impact of this is indicated in the row labelled "switch" and the subsequent data reporting average async writes. To minimise the impact of log switch, the maximum sized MQ active logs should be used (4GB).
  - zHyperLink I/O statistics
    - This section provides the shortest, longest and average I/O times for both single and multi-page I/Os that successfully used zHyperLink to complete the write to the MQ active log.
    - The section also provides the total I/O time spent when performing zHyperLink writes to the MQ active logs data sets. This is a useful number as it is a good indication of how much time the MQ logger task has spent spinning (using CPU) whilst performing zHyperLink I/O. This is additional CPU cost that is not incurred when using asynchronous I/O and further evidence of this cost can be found in the RMF workload report, where it is recorded as SRB time.

- Text in red: has been affected by zHyperLink
  - With traditional asynchronous I/O, the "logger I/O busy" and "logger task busy" percentages are typically close in utilisation values.
  - When the logger task is successfully using zHyperLink, the "logger I/O busy" percentage is generally a much lower value as the logger task is no longer waiting for the I/O to complete.

Getting best performance out of your zHyperLink environment

Whilst the enabling of zHyperLink with MQ active logs is relatively straightforward, there are some things that can be done to give MQ the best opportunity to benefit from zHyperLink.

1. Ensure there are sufficient write (token) sessions available on your system.
2. Use the shortest possible zHyperLink cable.
3. Ensure there are sufficient zHyperLink paths to the DASD.

These topics are expanded in the following pages.

Do I have enough write sessions?

When using zHyperLink, one or more zHyperLink write sessions are established with the DASD.

Current DASD supports a maximum of 64 concurrent write sessions so you should carefully consider which queue managers you enable zHyperLink on, and whether other subsystems, such as Db2 are also using zHyperLink for writing to the same DASD. If you run out of available write sessions then the queue manager automatically switches back to using traditional asynchronous I/O.

You can calculate the number of zHyperLink write sessions as follows:

```
Number of log copies (either 1 or 2) * number of stripes per log copy

Multiply this number by 2 if Metro Mirror (PPRC) is used
```

Note: While Metro Mirror results in twice as many write sessions being used, unless using a loopback to single device, those write sessions are split evenly between the two mirrored DASDs.

For example, a queue manager with single logging with four stripes uses 4 write sessions.

To view how many write sessions are in use, you can use the command:

```
DEVSERV QDASD MACH=<device serial number>,ZHL
```

Where the device serial number is a ten character serial number specifying either the storage control unit or the DASD device about which DEVSERV will display information.

This command will return the number of tokens and the names of the dataset with the write token, for example:

```
IEE459I 23.32.01 DEVSERV QDASD 089
UNIT  VOLSER SCUTYPE DEVTYPE       CYL  SSID SCU-SERIAL DEV-SERIAL EFC
07800 AAATS4 2107996 2107900     65520  3110 0175-TJS31 0175-TJS31 *OK
ZHYPERLINK WRITE TOKENS ASSIGNED
TOKEN DATE    TIME  UNIT  DATA SET NAME
1BE0 07/19/24 23:30 07861 MQMDATA.VTS8.LOGCOPY1.DS01.DATA
1CE1 07/19/24 23:30 07860 MQMDATA.VTS8.LOGCOPY1.DS01.DATA
1DE2 07/19/24 23:30 0782B MQMDATA.VTS8.LOGCOPY1.DS01.DATA
1EE3 07/19/24 23:30 078C4 MQMDATA.VTS8.LOGCOPY2.DS01.DATA
1FE4 07/19/24 23:30 078C9 MQMDATA.VTS8.LOGCOPY2.DS01.DATA
20E5 07/19/24 23:30 078C3 MQMDATA.VTS8.LOGCOPY2.DS01.DATA
21E6 07/19/24 23:30 0785F MQMDATA.VTS8.LOGCOPY1.DS01.DATA
22E7 07/19/24 23:30 078C7 MQMDATA.VTS8.LOGCOPY2.DS01.DATA
**** 8 WRITE TOKEN(S) MET THE SELECTION CRITERIA
**** 1 DEVICE(S) MET THE SELECTION CRITERIA
**** 0 DEVICE(S) FAILED EXTENDED FUNCTION CHECKING
```

In the above example, there is a single queue manager with dual logging, where each log is striped 4-ways. This results in 8 write tokens being used.

What is the impact of zHyperLink cable length?

The length of the cable used for zHyperLink between the IBM z mainframe and the DASD is limited to 150 metres, and the length used may be predicated by the distance between the two.

However, there is a benefit to having the shortest possible cable in terms of both I/O response times and the additional cost incurred by the MQ logger task when using zHyperLink synchronous writes.

The Getting Started with IBM zHyperLink for z/OS Redbook in section 2.3.6 discusses the standard cable lengths (40 and 150 metres) but also states that other length options are available.

In our original testing of zHyperLink, there was a 150 metre cable connecting our IBM z16 and DS8950F as the DASD was located on the far side of our data centre. For a single page I/O request with the 150 metre cable, MQ reported the I/O taking 35 microseconds to complete.

Following discussions with the zHyperLink team, we moved our DASD to be located much closer to the IBM z16, and replaced the 150 metre cable with a 30 metre cable. In the "What is zHyperLink" section, a table was provided that reports the subsequent single page I/O times. For zHyperLink on a quiet system with a 30 metre cable, we saw I/O times of 28 microseconds.

For a single page I/O request, reducing the cable length from 150 to 30 metres, reduced the I/O time by 7 microseconds which is the equivalent of a 20% improvement in response times.

Since the synchronous I/O results in the MQ logger task spinning whilst the I/O completes, reducing the length of the cable in this way also reduces the I/O cost by 7 CPU microseconds.

A longer zHyperLink cable also offers more opportunity for the I/O request to timeout, which is more noticeable with multi-page I/O requests. Such a request would initially be attempted as a synchronous I/O request and then may fall back to an asynchronous I/O request.

Even when the I/O remains synchronous, the cable length can affect the I/O response times for multi-page I/Os. For example, a multi-page I/O writing 16 pages or 64KB of data with a 150 metre cable reported a response time of 167 CPU microseconds. Using a 30 metre cable resulted in the I/O time taking 105 CPU microseconds, a reduction of 62 CPU microseconds or an equivalent of a 37% reduction in I/O and MQ logger SRB time.

If you are not in the fortunate position of having the DASD and the IBM z mainframe located adjacent, or are unable to move them closer together, it is still beneficial to check that the zHyperLink cable is not unnecessarily long as this will add both latency to the I/O response times and CPU cost to the MQ logger task. Additional cost to the MQ logger task may be of importance as the logger task runs as a high priority SRB, which when spinning may be preventing other, potentially high importance, tasks from fulfilling their own CPU requirements.

Are there sufficient zHyperLink paths to the DASD?

In our original testing environment there were 8 PFIDs enabled over a single PCIe adapter, which is 4 PFIDs per zHyperLink port for each logical partition sharing the DS8000. The IBM z16 was configured with a second adapter and a further 8 PFIDs but these were reserved.

When using zHyperLink to write to MQ active logs, the I/O requests are shared equally between the available PFIDs, using a round-robin technique.

We were advised that for most reliable performance, the PFIDs usage should not exceed 25% else there was an increased risk of the I/O changing to asynchronous, with the subsequently longer I/O times.

To determine the PFID utilisation requires reviewing the RMF Synchronous I/O Device Activity report. If we review the data in the earlier Synchronous I/O Device Activity report, we need to use the following attributes:

- Synch I/O writes
- Average Response Time for Synch I/O

If there are multiple devices, perhaps from striping, or the MQ active logs being located on separate devices, it may be easier to use the values in the LCU summary, shown at the bottom of the diagram.

Multiplying these two values together i.e. `13567.3 * 0.115`, the sum is the number of CPU milliseconds used per second, i.e. 1560 CPU milliseconds per second. This equates to a single PFID being 156% busy for the interval.

Provided the number of active PFIDs is known, this can be used to work out the approximate utilisation of the PFIDs, i.e. on a system with 8 PFIDs that is 156% busy, each PFID is 19.5% utilised, which is below the 25% busy guideline value.

A symptom of the PFIDs being over capacity is a decrease in synchronous I/O and an increase in asynchronous I/O.

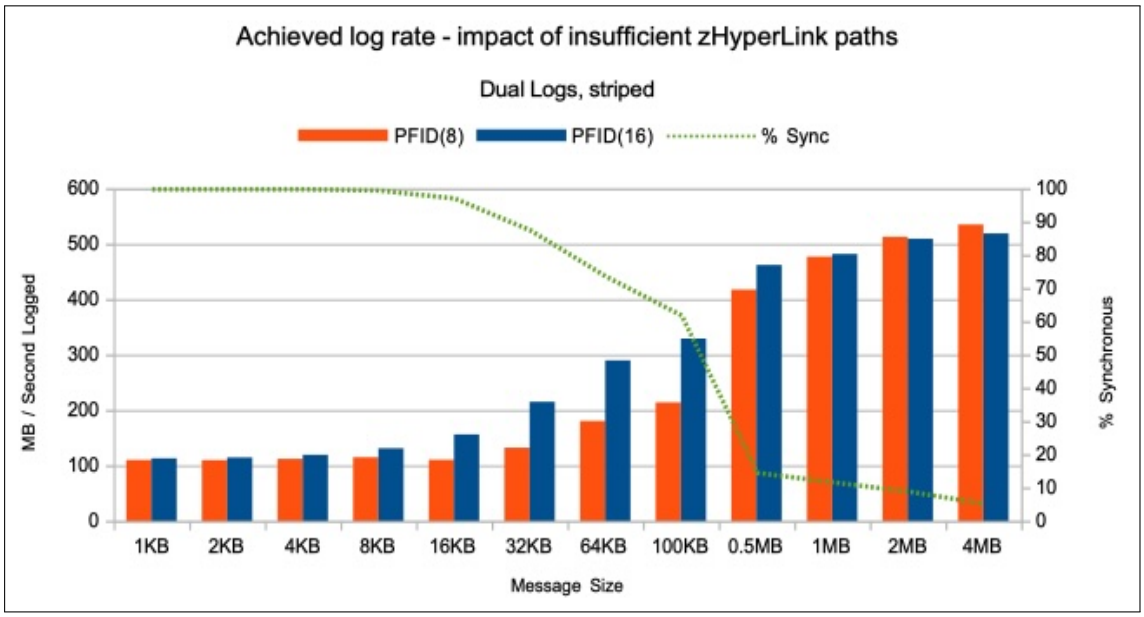
By using the second adapter and allocating a total of 16 PFIDs per logical partition, we found that for all of the configurations detailed in this section, there was minimal asynchronous I/O.

It should be noted that having insufficient paths for zHyperLink may not always cause a performance issue, as it can depend on the configuration of the MQ queue manager and the workload.

The following chart compares the MQ log rate achieved on an MQ queue manager configured with dual log copies that are each striped 4-ways, when there is both insufficient and sufficient zHyperLink paths to the DASD.

The workload used to collect the data in the chart is based upon that used for "Upper bounds of persistent logging rate" in the Regression section of this document.

Chart: Impact of insufficient zHyperLink paths



Notes on chart:

- For messages up to 8KB, 8 PFIDs was sufficient to keep the workload synchronous and was largely able to sustain throughput rates similar to the 16 PFID configuration.
- As the message size increased to 16KB to 100KB, the percentage of synchronous I/O decreased from 97 to 62% and throughput rates dropped by up to 38% with 8 PFIDs.
- For messages of 0.5MB to 4MB with the 8 PFID configuration, the percentage of synchronous I/O continued to drop compared to the 16 PFID configuration. However as MQ switched to asynchronous I/O, the achieved logging rate improved due to the benefits of striping the active logs.
    - With the 16 PFID configuration using messages of 0.5MB to 4MB, the striped nature of the logs does provide the same benefits when using zHyperLink to perform the I/O.

Rule of Thumb

Try to keep the zHyperLink path utilisation to a maximum of 25% busy.

Performance Measurements

This section compares the performance of 4 queue manager configurations running workloads with increasing message sizes, with the configurations being:

1. Single active log copy, non-striped
2. Single active log copy, striped
3. Dual active log copies, non-striped
4. Dual active log copy, striped

Each of those configurations compares the workload run with traditional asynchronous I/O with zHPF-enabled against synchronous I/O.

For all measurements, there are 16 PFIDs enabled, to ensure the maximum chance of zHyperLink success. The zHyperLink cable used is 30 metres.

Additionally, for each configuration of queue manager, there are two environments used. The first environment is our performance environment where the only workload on the logical partition is the MQ workload. The second workload named "real-world" has other non-MQ workload driving I/O to and from the same DASD device. This is intended to simulate a more realistic environment which has the benefit of showing zHyperLink to best effect.

Each section contains comparisons of the achieved rate that the MQ logger task was able to sustain for a range of message sizes from 1KB to 4MB. For the performance section, a comparison of the cost per MB logged is also included.

The cost per MB comparison charts can make the impact of zHyperLink look quite significant, particularly in these micro-benchmark workload where the MQ cost forms 95% or more of the total cost, and because of that the section "Cost of zHyperLink" attempts to put the additional cost into context.

In every measurement in this section, the queue manager was configured with 4GB active logs.

The workload used to collect the data in this section is based upon that used for "Upper bounds of persistent logging rate" in the Regression section of this document.

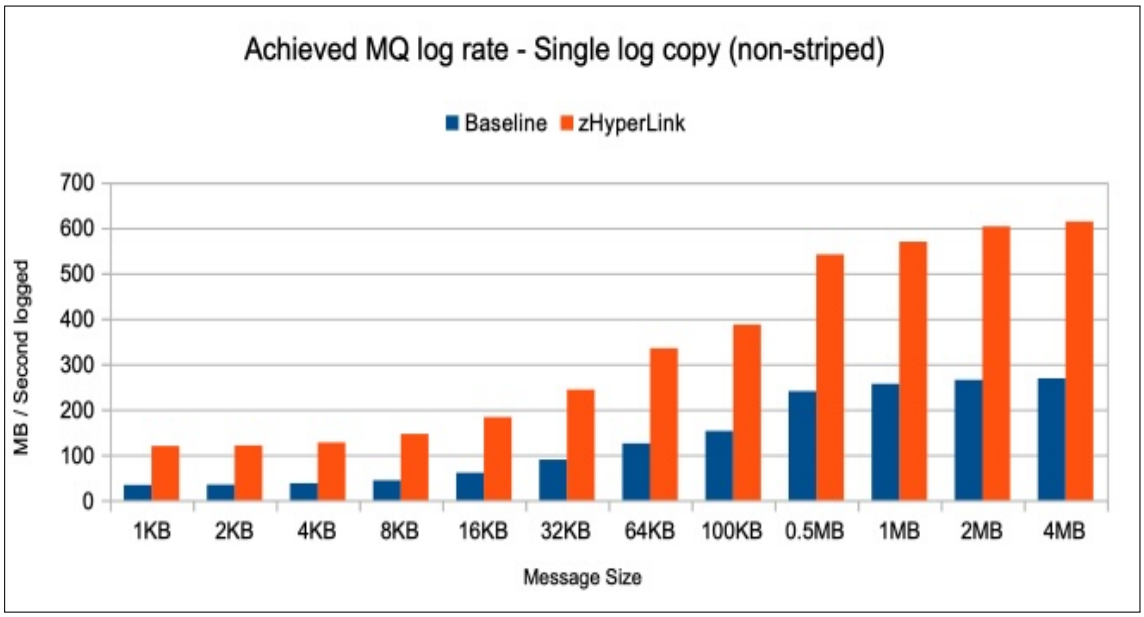
Note: The costs reported are based on the total cost charged to the MQ queue manager and the light-weight applications divided by the total MB logged.

Measurements in performance environment

Single active log copy, non-striped

The following charts compare the performance of the MQ queue manager when configured with a single non-striped log copy.

Chart: Achieved Log Rate using single log copy (non-striped)



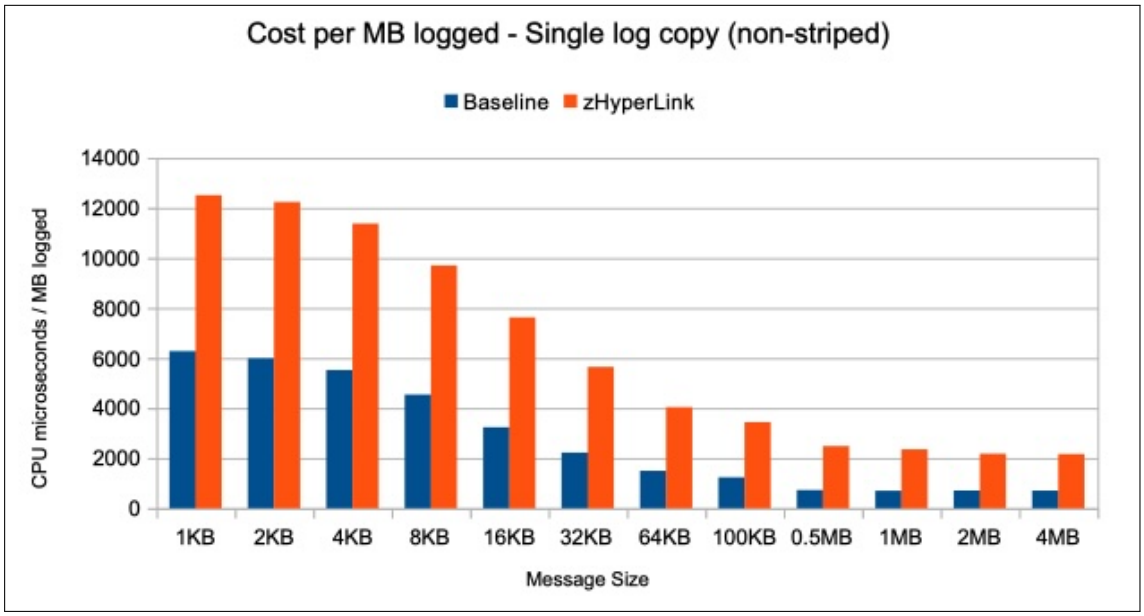
When using a single log copy with zHyperLink enabled, where the logs were not striped, MQ was able to achieve improved log rate for all of the measured message sizes.

For smaller messages, the log rate was up to 3.6 times that of the baseline configuration, whilst with larger messages the improvement was up to 2.3 times that of the baseline measurement.

For example, with 1KB messages the baseline workload achieved a log rate of 33 MB per second, but when the MQ active log was zHyperLink-enabled, the log rate reached 120 MB per second.

With larger messages, the baseline measurement peaked at 268 MB per second, whilst the zHyperLink configuration achieved 614 MB per second.

Chart: Cost per MB logged when using single log copy (non-striped)



The cost per MB for the MQ workload when using a single log copy with zHyperLink ranged between 2 to 3.5 times that of the baseline measurement.

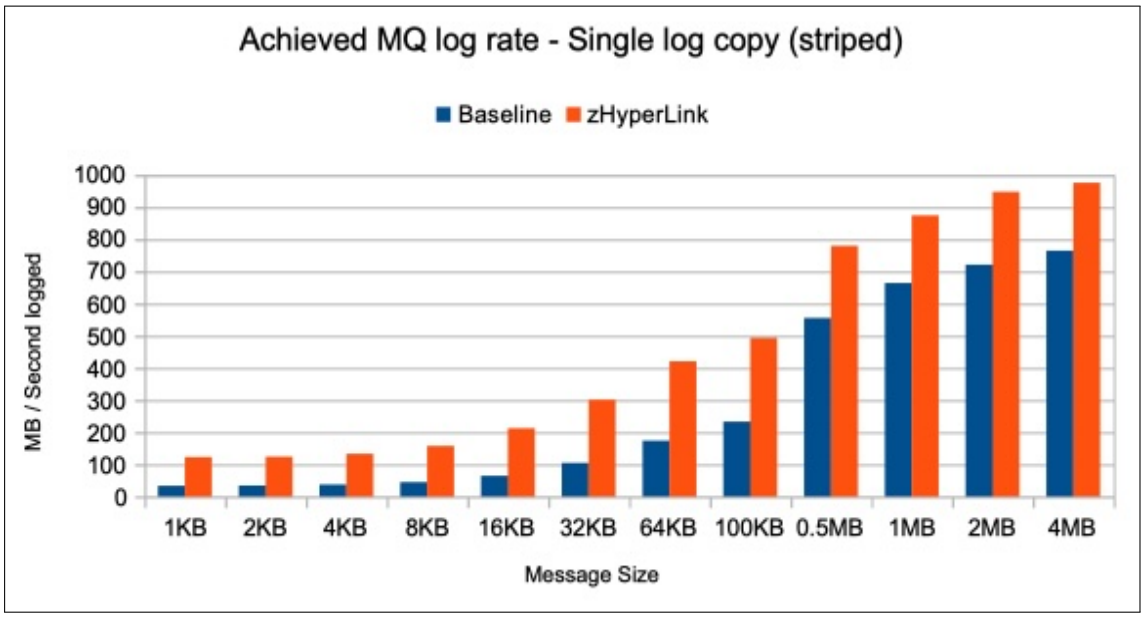
The additional cost is incurred in the MQ queue manager address space and is attributed to SRB in the RMF CPU report.

For smaller messages the cost was typically 2 to 2.4 times that of the baseline, whereas larger message workloads saw a more significant increase in cost per MB logged.

Single active log copy, striped

The following charts compare the performance of the MQ queue manager when configured with a single log copy that is striped 4-ways.

Chart: Achieved Log Rate using single log copy (striped)



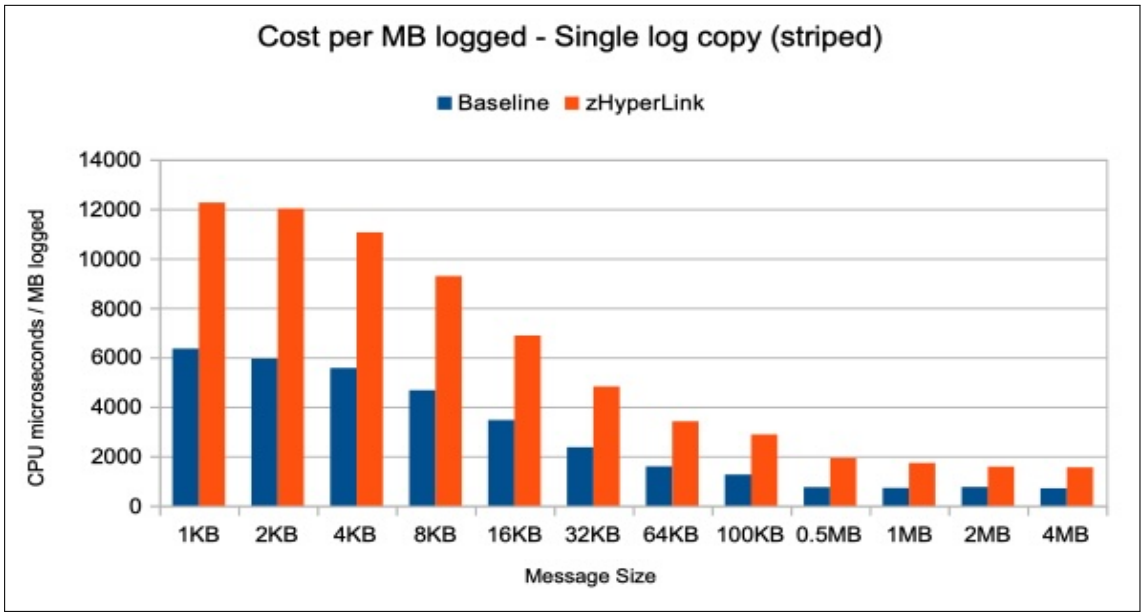
When using a single striped log copy with zHyperLink enabled, MQ was able to achieve improved log rate for all of the measured message sizes.

For smaller messages, the log rate was up to 3.7 times that of the baseline configuration, whilst with larger messages the improvement was up to 1.3 times that of the baseline measurement.

The baseline workload achieved a log rate of 33 MB per second with 1KB messages, but when the MQ active log was zHyperLink-enabled, the log rate reached 122 MB per second.

With larger messages, there was less benefit for zHyperLink as striping the logs in the non-zHyperLink environment gave a significant improvement to the log capacity, increasing from 268 to 764 MB per second. By contrast, striping the zHyperLink single logs resulted in the 4MB workload increasing from 614 to 976 MB per second.

Chart: Cost per MB logged when using single log copy (striped)



The cost per MB for the MQ workload when using a single log copy that was striped with zHyperLink ranged between 1.9 to 2.6 times that of the baseline measurement.

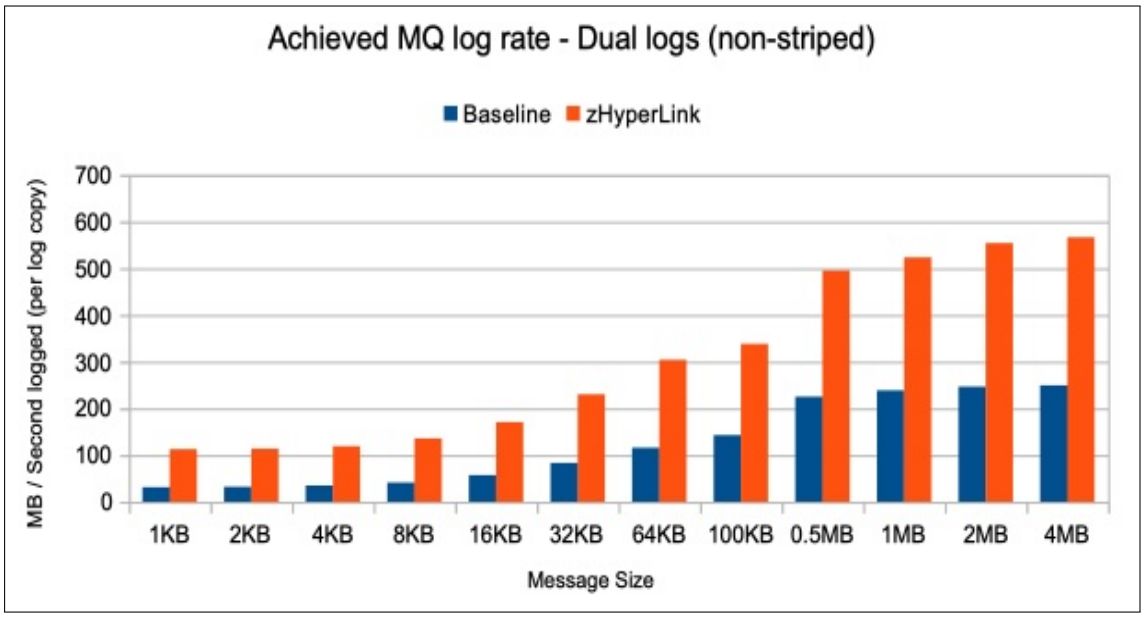
For smaller messages the cost was typically up to double that of the baseline, whereas larger message workloads saw a more significant increase in cost per MB logged.

Striping the MQ active log in the baseline configuration made little difference to the cost per MB logged but using striped logs in the zHyperLink configuration with larger messages reduced the cost by up to 30% over the non-striped equivalent measurement.

Dual active log copies, non-striped

The following charts compare the performance of the MQ queue manager when configured with a dual log copies without striping.

Chart: Achieved Log Rate using dual log copies (non-striped)



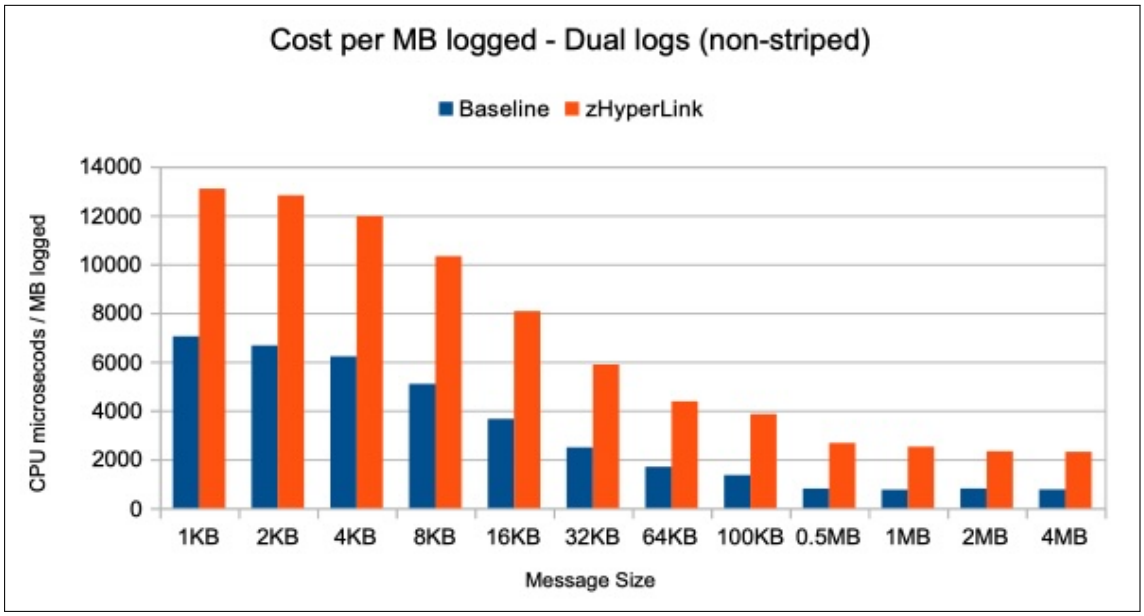
When using a dual log copies with zHyperLink enabled, where the logs were not striped, MQ was able to achieve improved log rate for all of the measured message sizes.

For smaller messages, the log rate was up to 3.6 times that of the baseline configuration, whilst with larger messages the improvement was up to 2.3 times that of the baseline measurement.

For example, with 1KB messages the baseline workload achieved a log rate of 31 MB per second, but when the MQ active log was zHyperLink-enabled, the log rate reached 113 MB per second.

With larger messages, the baseline measurement peaked at 249 MB per second, whilst the zHyperLink configuration achieved 566 MB per second.

Chart: Cost per MB logged when using dual log copies (non-striped)



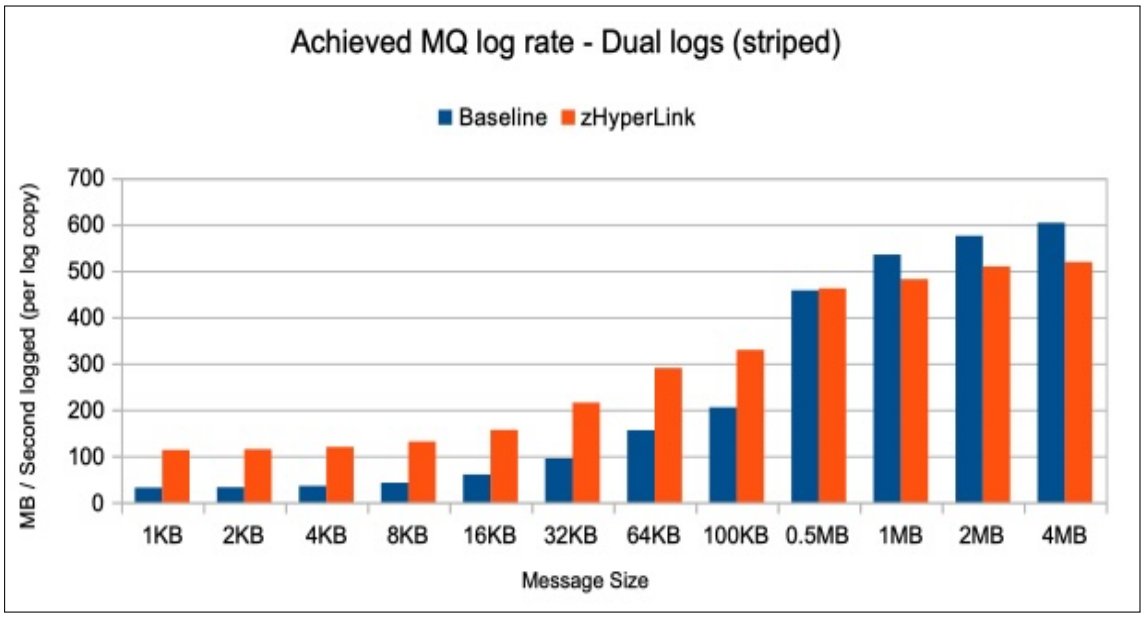
The cost per MB for the MQ workload when using dual log copies with zHyperLink ranged between 1.9 to 3.4 times that of the baseline measurement.

For smaller messages the cost was typically up to double that of the baseline, whereas larger message workloads saw a more significant increase in cost per MB logged.

Dual active log copies, striped

The following charts compare the performance of the MQ queue manager when configured with a dual log copies that are striped 4-ways.

Chart: Achieved Log Rate using dual log copies (striped)



When using dual striped log copies with zHyperLink enabled, MQ was able to achieve improved log rate for messages up to 0.5 MB, but for larger messages the rate achieved was 90% of the baseline configuration.

For smaller messages, the log rate was up to 3.6 times that of the baseline configuration, whilst with larger messages there was a 10% decrease in peak log rate when using zHyperLink.

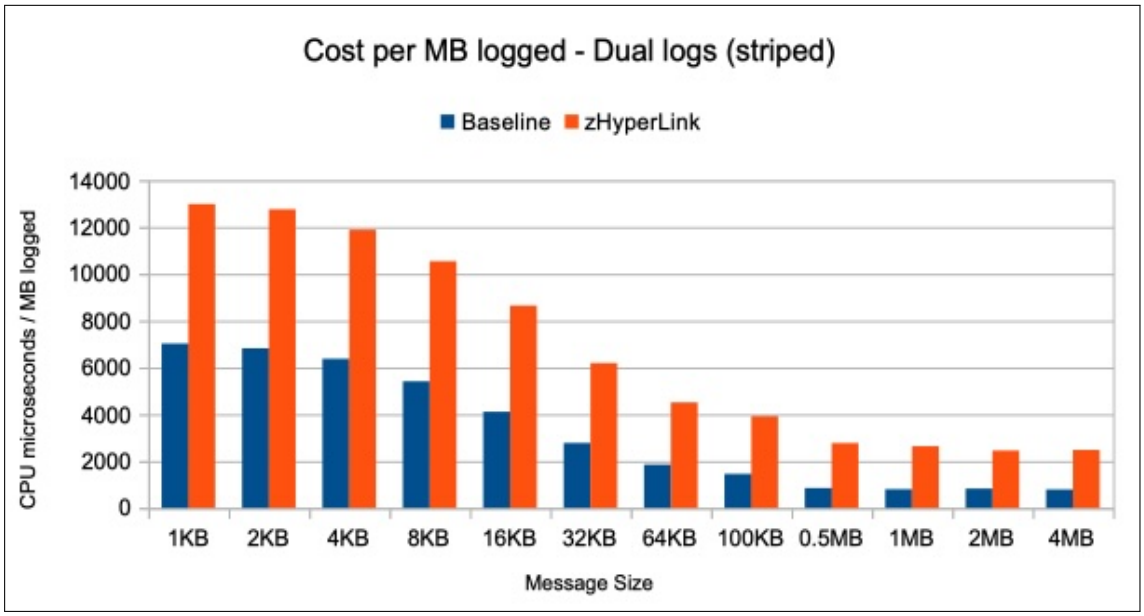
The baseline workload achieved a log rate of 31 MB per second with 1KB messages, but when the MQ active log was zHyperLink-enabled, the log rate reached 112 MB per second.

With larger messages, the baseline configuration was able to exploit the striped logs to best effect and was able to sustain up to 600 MB per second per log copy. This was a significant increase in capacity over the non-striped configuration where the peak log rate was 249 MB per second.

In the zHyperLink configuration with striped dual log copies, there was sufficient zHyperLink capacity, i.e. paths to disk, that the I/O remained synchronous for all message sizes but this resulted in the peak log rate actually being lower than the non-striped equivalent measurements, such that non-striped achieved 566 MB per second and striped logs achieved 519 MB per second.

With less PFIDs (paths to disk), the zHyperLink configuration would have resulted in more time-outs and used asynchronous I/O, giving a slightly improved log capacity.

Chart: Cost per MB logged when using dual log copies (striped)



The cost per MB for the MQ workload when using dual log copies that are striped with zHyperLink ranged between 1.8 to 3.3 times that of the baseline measurement.

For smaller messages the cost was typically up to double that of the baseline, whereas larger message workloads saw a more significant increase in cost per MB logged.

Once more, striping the MQ active log in the baseline configuration made little difference to the cost per MB logged. Using striped logs in the zHyperLink configuration saw a small increase in cost per MB logged over the non-striped equivalent measurement of up to 7.5%.

With fewer zHyperLink resources, i.e. 8 PFIDs rather than 16, the I/O for the larger messages would have gone predominantly asynchronous, such that the costs would have been 60% of the 16 PFID measurement. Note, that the 8 PFID measurements going asynchronous with larger messages still incur a cost over the baseline measurements as synchronous I/O is still attempted.

Measurements in "real-world" environment

The measurements in this section are a repeat of those in the performance environment, with a small but significant difference.

For these measurements, the workload includes a set of 30 medium-priority batch tasks that are each reading and writing to their own RRDS (relative record data set) file. These batch tasks randomly read and write records of 32KB, with the purpose of increasing the utilisation of the channel paths to the storage device and to increase cache usage on the disk.

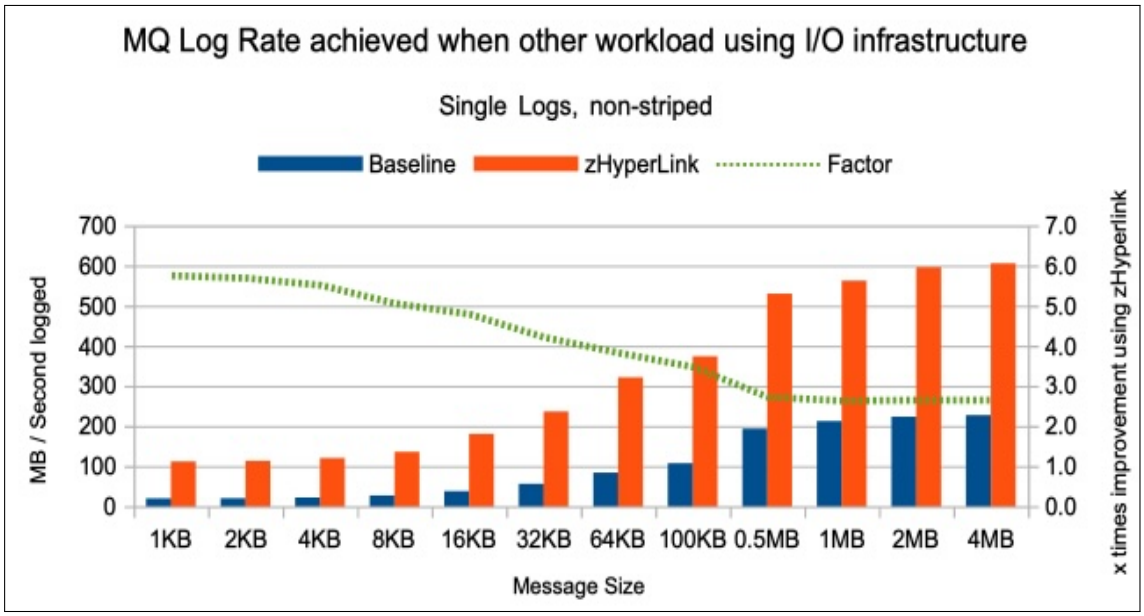
To offer an indication of the impact of the additional workload on the 1KB single active log copy (non-striped) baseline measurement, the average of the 4 channel paths utilisation increased from 2% to 94%.

As mentioned in the introduction to the performance measurements, the "real world" measurements in the following section will not include charts with the cost per MB logged, as the costs for the MQ workload is not significantly different to those reported in the performance environment.

Single active log copy, non-striped

The following chart compares the performance of the MQ queue manager when configured with a single log copy that is not striped.

Chart: Achieved Log Rate using single log copy (non-striped) when disk infrastructure is busy



When using a single non-striped log copy with zHyperLink enabled in an environment where the common I/O infrastructure was in use, MQ was again able to significantly improve log rate for all of the measured message sizes.

For smaller messages, the log rate was up to 5.8 times that of the baseline configuration, whilst with larger messages the improvement was up to 2.7 times that of the baseline measurement.

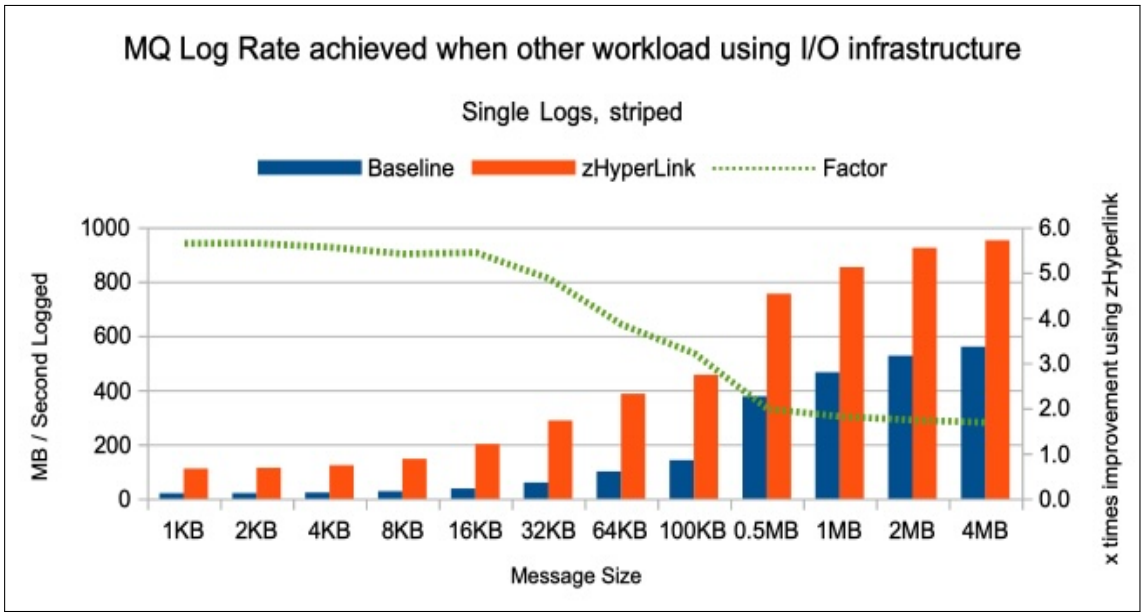
The baseline workload achieved a log rate of 19 MB per second with 1KB messages, but when the MQ active log was zHyperLink-enabled, the log rate reached 112 MB per second.

With larger messages, there was still a significant benefit for zHyperLink as the I/O no longer used the same congested channel paths as the non-MQ workload, such that the baseline configuration achieved a peak of 227 MB per second whereas the zHyperLink measurement achieved 605 MB per second.

Single active log copy, striped

The following chart compares the performance of the MQ queue manager when configured with a single log copy that is striped 4-ways.

Chart: Achieved Log Rate using single log copy (striped) when disk infrastructure is busy



When using a single striped log copy with zHyperLink enabled in an environment where the common I/O infrastructure was in use, MQ was able to significantly improve log rate for all of the measured message sizes.

For smaller messages, the log rate was up to 5.7 times that of the baseline configuration, whilst with larger messages the improvement was up to 1.8 times that of the baseline measurement.

The baseline workload achieved a log rate of 19 MB per second with 1KB messages, but when the MQ active log was zHyperLink-enabled, the log rate reached 110 MB per second.

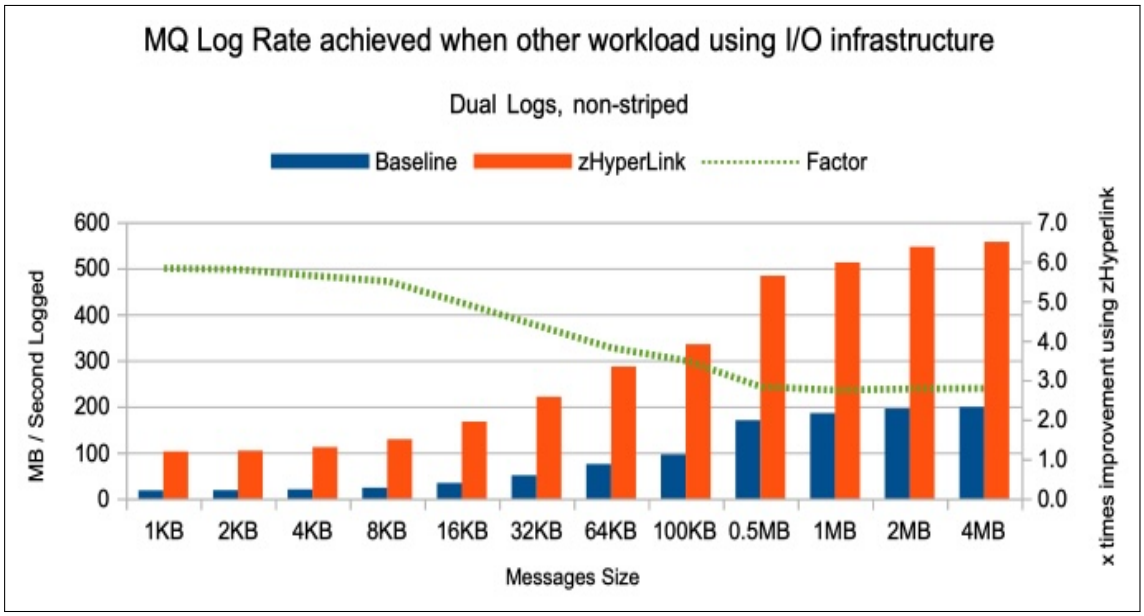
With larger messages, even with striped logs, there was still a significant benefit for zHyperLink as the I/O no longer used the same congested channel paths as the non-MQ workload, and whilst that the baseline configuration achieved a peak of 560 MB per second, the zHyperLink measurement achieved 952 MB per second.

The benefits of striping the logs for both the baseline and zHyperLink configurations equated to improving the peak log rates by 330 to 350 MB per second over the non-striped equivalent measurement.

Dual active log copies, non-striped

The following chart compares the performance of the MQ queue manager when configured with dual log copies that are not striped.

Chart: Achieved Log Rate using dual log copies (non-striped) when disk infrastructure is busy



When using dual non-striped log copies with zHyperLink enabled in an environment where the common I/O infrastructure was in use, MQ was again able to significantly improve log rate for all of the measured message sizes.

For smaller messages, the log rate was up to 5.9 times that of the baseline configuration, whilst with larger messages the improvement was up to 2.8 times that of the baseline measurement.

The baseline workload achieved a log rate of 17 MB per second with 1KB messages, but when the MQ active log was zHyperLink-enabled, the log rate reached 102 MB per second.

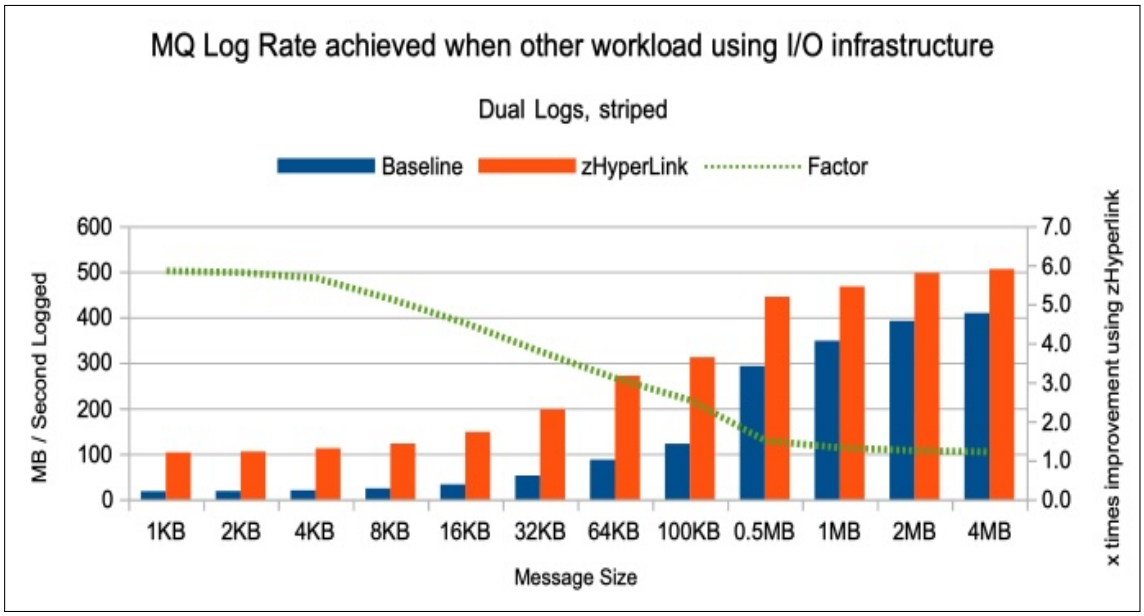
With larger messages, there was still a significant benefit for zHyperLink as the I/O no longer used the same congested channel paths as the non-MQ workload, such that the baseline configuration achieved a peak of 198 MB per second whereas the zHyperLink measurement achieved 557 MB per second.

It is worth recalling that in the equivalent performance environment measurement, the baseline configuration achieved a peak MQ log rate of 249 MB per second and this rate has dropped to 198 MB per second (-21%) due to the additional load on the channel paths. By contrast the zHyperLink configuration changes by less than 2%.

Dual active log copies, striped

The following chart compares the performance of the MQ queue manager when configured with dual log copies that are striped 4-ways.

Chart: Achieved Log Rate using dual log copies (striped) when disk infrastructure is busy



When using dual striped log copies with zHyperLink enabled in an environment where the common I/O infrastructure was in use, MQ was able to significantly improve log rate for all of the measured message sizes.

For smaller messages, the log rate was up to 5.9 times that of the baseline configuration, whilst with larger messages the improvement was up to 1.2 times that of the baseline measurement.

The baseline workload achieved a log rate of 17 MB per second with 1KB messages, but when the MQ active log was zHyperLink-enabled, the log rate reached 103 MB per second.

With larger messages using striped logs there was a less significant benefit to zHyperLink, similar to the performance environment, but due to the baseline performance being impacted by non-MQ workload using the channel paths, MQ's active logs using zHyperLink were able to achieve higher peak log rates of up to 505 MB per second compared to the baseline of 409 MB per second.

The benefits of striping the logs for the baseline configuration resulted in the achieved log rate increasing by 210 MB per second over the equivalent non-striped measurement, however the zHyperLink configurations saw a drop of 52 MB per second on the peak log rate achieved when compared to the dual logs non-striped configuration.

Cost of zHyperLink

CPU in the MQ queue manager address space is largely attributed to administration-type tasks, for example: active log writes, deferred write processing, message expiry and archiving. Of these 4 examples, the first 3 perform their processing using SRB and archiving uses a TCB.

How much cost is charged to the queue manager for each of these tasks will depend on many factors and will be unique to each MQ installation.

For a non-persistent type MQ workload, there may be very little cost attributed directly to the MQ queue manager, but a persistent workload would typically see a higher cost, particularly associated with active log writes and at end of log, writing the active log to archive.

In the measurements for zHyperLink thus far, the impact has been to increase the cost per MB logged from between 1.8 to 3.7 times the cost per MB of traditional asynchronous I/O. These calculations are based on the total CPU used for the MQ queue manager and the very simplistic applications putting and getting messages.

Given synchronous I/O results in the MQ logger task (SRB) spinning CPU until the I/O completes, I/O that has an elapsed time of 100 microseconds will cost 100 microseconds of CPU.

As such, it is clearly beneficial to reduce the I/O time, and therefore the cost as much as possible. This can be achieved in two ways:

1. Increase the number of paths available to zHyperLink.
2. Reduce the length of the connection between the IBM z mainframe and the DASD.

In our initial investigation, we increased the number of zHyperLink paths from 8 to 16. This gave mixed results, the I/O was spread evenly over the additional paths and did in some cases reduce the I/O time, particularly when MQ was attempting to write many pages per I/O request. However, as the paths were more difficult to saturate, there were instances where large message workloads remained synchronous and achieved lower throughput than the asynchronous equivalent.

Reducing the length of the zHyperLink connection meant re-organising the data centre to physically locate the DASD next to the IBM z mainframe and replacing the 150 metre cable with a 30 metre cable. This had the effect of decreasing a single page I/O from 35 to 28 microseconds, which for synchronous I/O also meant the cost of the I/O was reduced by 7 CPU microseconds (-20%).

By ensuring both of the above actions are applied will ensure that MQ with zHyperLink-enabled active logs will have the best opportunity for minimal I/O times and therefore minimal time spent spinning CPU while the I/O completes.

Cost of zHyperLink is all relative

Having ensured that the zHyperLink environment is optimised, the next step is to look at the relative cost of zHyperLink.

Users of MQ may be interested in the CPU increase for their MQ estate, for example they may be aware of their MQ CPU usage from RMF workload reports, and a significant increase in MQ CPU utilisation may raise concerns.

To demonstrate the impact of zHyperLink against the MQ MSTR address space, the following compares the costs as reported by RMF Workload reports for 2 message sizes, 2KB and 1MB, from different perspectives. In both instances, the queue manager is configured with a single log copy that is not striped.

The values given are the number of times higher the zHyperLink cost per MB is, compared to the baseline (asynchronous I/O) measurement.

2KB workload

- Total cost (applications plus MQ): 1.9 times
- MQ (TCB plus SRB): 3.2 times
- MQ SRB (active logging only): 3.2 times

1MB workload

- Total cost (applications plus MQ): 3.1 times
- MQ (TCB plus SRB): 10.4 times
- MQ SRB (active logging only): 11.1 times

Even with the improvements to the MQ log rate from zHyperLink, it might be difficult to justify a cost increase to your MQ estate that approaches 11 times the existing cost, which is it why it is crucial to consider your own processing requirements.

For example if the depth of the queues on a queue manager is high for periods of time, then the MQ queue manager may incur costs from writing data from buffer pools to page set using the Deferred Write Processor (DWP). Similarly if there is a high usage of message expiry, there will be additional non-logger related cost attributed to the MQ queue manager.

The cost of using the DWP and message expiry will increase the baseline SRB cost in the queue manager and will not be affected by implementing zHyperLink support for MQ active logs, but may reduce the impact of zHyperLink as a proportion of the total MQ CPU utilisation.

Equally if the system is set up to combine MQ queue manager and channel initiator costs as a single value, the impact of enabling zHyperLink will be a smaller proportion of the total MQ cost.

For the measurements in this chapter, the application costs are small, simply because the applications are doing nothing other than MQ messaging, i.e. there are no complex calculations, database or file interactions etc.

Factoring in a more complex environment

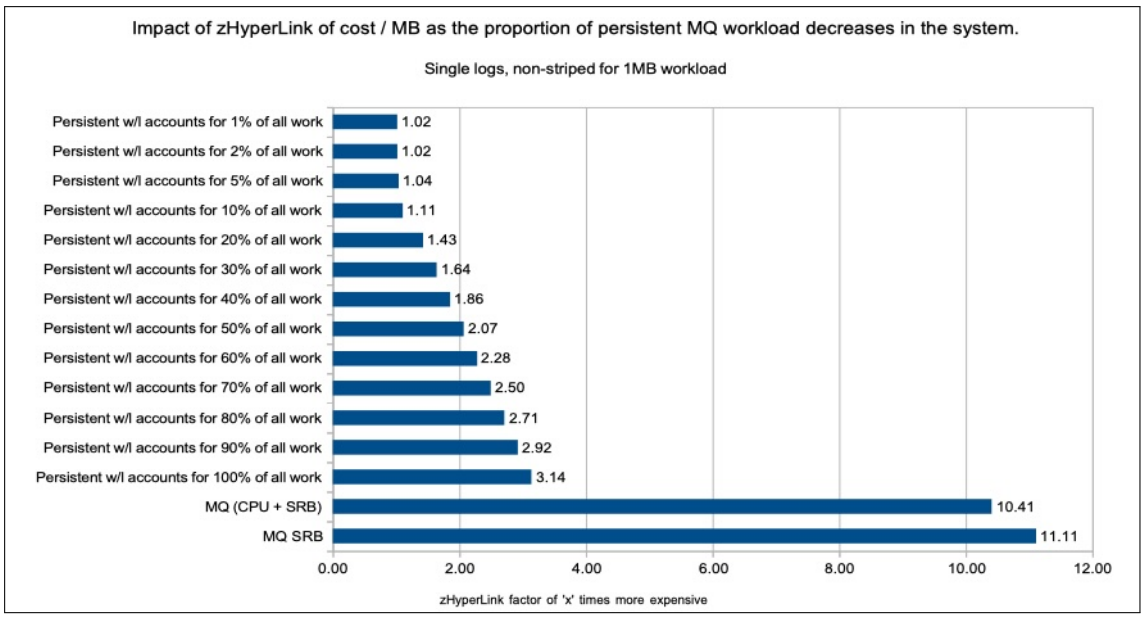
Throughout all of this zHyperLink section, the workloads have used simplistic applications with no complex processing.

The total costs of the workloads have been based on the total of those applications and the MQ queue manager address space, but this is not a realistic environment for MQ.

In this section, a comparison of the impact of zHyperLink to the MQ costs is offered, which differs from previous comparisons as a higher cost from those systems that interact with the MQ queue manager will diminish the relative impact to CPU usage of zHyperLink.

The following chart shows the number of times more expensive enabling zHyperLink is to the total cost as the application (or MQ channel initiator) cost increases.

Chart: Impact of zHyperLink on cost per MB of an MQ workload



Notes on chart:

- The MQ SRB costs consist entirely of MQ logger task costs - there is no deferred write nor message expiry.
- When comparing the impact of zHyperLink solely on MQ SRB costs, the impact for the 1 MB workload is that zHyperLink was 11.11 times as expensive as the asynchronous I/O workload.
- When comparing the impact of zHyperLink, when the MQ persistent workload accounts for 80% of the entire CPU cost, zHyperLink increases the overall CPU cost by a factor of 2.71 times.
- When comparing the impact of zHyperLink, when the MQ persistent workload accounts for 20% of the entire CPU cost, zHyperLink increases the overall CPU cost by a factor of 1.43 times.
- When comparing the impact of zHyperLink, when the MQ persistent workload accounts for 5% of the entire CPU cost, zHyperLink increases the overall CPU cost by a factor of 1.04 times, or 4%

There is clearly a cost impact to a system when MQ active logs are configured to use zHyperLink, but the overall impact will depend on how large a proportion of the workload uses persistent messages.

Checklist

- Be aware if already using zHyperLink for other subsystems e.g. Db2 - additional load on those paths may impact the response times of all subsystems using the links.
- Monitor resources, including CPU, Synchronous I/O Device Activity, MQ log statistics.
- Use the shortest zHyperLink cable between IBM z mainframe and storage device.
- Ensure there are sufficient paths (PFIDs) to the storage device.
- Consider whether you need striped logs and zHyperLink, particularly when using MQ with dual logs. The performance characteristics can change depending on whether there are *sufficient* zHyperLink resources.
- **Measure in your own environment** to ensure the additional cost is not prohibitive to your installation and to understand whether there are sufficient zHyperLink resources for your workload.
- Ensure sufficient CPU is available to address the additional SRB costs incurred with zHyperLink.
- Be aware that in a Metro Mirror (PPRC) environment, zHyperLink being enabled results in zHyperWrite being enabled too.

Whilst the additional cost of zHyperLink may be of concern to the installation, be aware that once zHyperLink has been configured and enabled in the MQ queue manager, zHyperLink can be enabled/disabled dynamically on the MQ queue manager using `SET LOG ZHYLINK(YES | NO)` to address periods of high-volume persistent workloads.

Chapter 5

Statistics Trace Class(5)

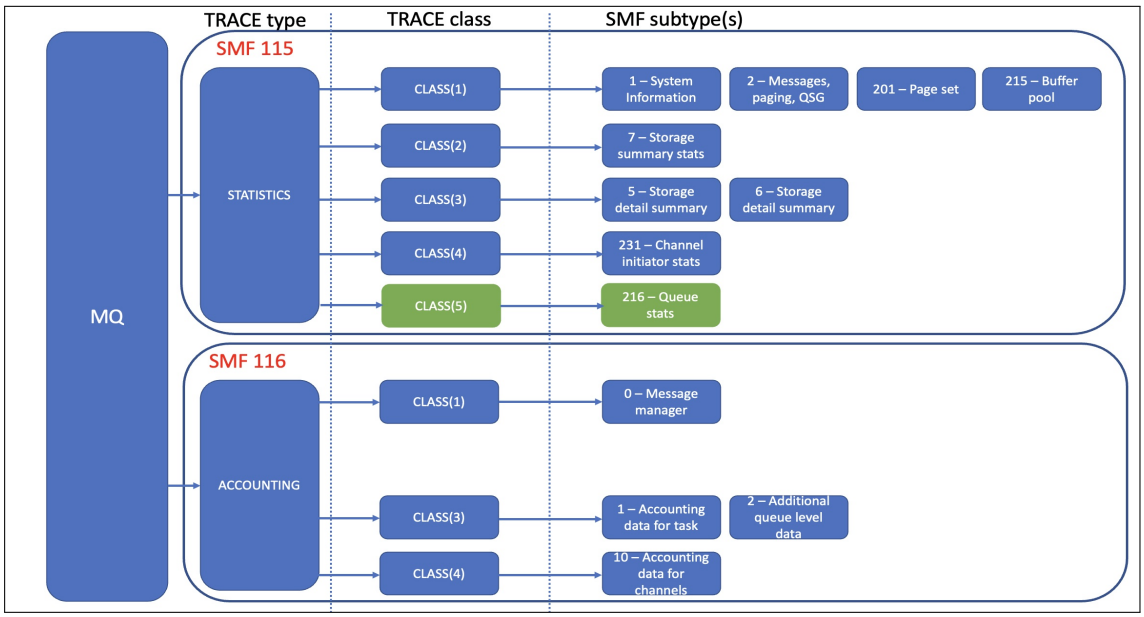
Queue Statistics - additional data collected

IBM MQ for z/OS 9.4 significantly extends the range of data collected in queue statistics records that are collected when enabling MQ for z/OS' `TRACE(S) CLASS(5)`. The latest list of collected data in available in IBM documentation section "Queue data records".

The previous long-term service release, IBM MQ for z/OS 9.3, added the collection of queue statistics to the existing SMF information available at your enterprise. This new queue statistics data provided relevant data for each queue without the burden of collecting accounting information, or having to write an application to issue the `DISPLAY QSTATUS` command. In turn this allows you to better monitor the performance of your queues over time, using your existing SMF tools.

The blog "MQ and SMF - What, when and how much?" discusses what SMF records were available prior to IBM MQ for z/OS 9.3 and includes a diagram showing what SMF records and sub-types are written for each MQ Accounting and Statistics trace. The following updated diagram shows where the class(5) statistics data fits.

Diagram: MQ Trace type and class with resulting SMF subtype(s)



How do I enable queue statistics?

In order to ensure queue statistics data is collected, it is necessary to start `TRACE(S) CLASS(5)` and enable the `STATQ` option on all of the queues for which queue statistics are required.

Queue statistics can be configured either at the queue level using `STATQ(ON | QMGR)`, or if the "QMGR" option is selected then the queue manager must also set `STATQ(ON)`. Note that this will enable queue statistics for **all** queues that have `STATQ(QMGR)`.

It is worth highlighting that the default option for queues is `STATQ(QMGR)`, therefore enabling `STATQ(ON)` at the queue manager may result in all application queues and some `SYSTEM.*` queues collecting queue statistics data.

What is the cost of enabling queue statistics?

The data collected for queue statistics in IBM MQ for z/OS 9.4 now includes more than 60 data points for each queue being monitored.

As in IBM MQ for z/OS 9.3, MQ collects the queue statistics data including queue depth at the time when the SMF data is captured, i.e. at SMF interval end.

To minimise the amount of data written to SMF in IBM MQ for z/OS 9.4, a single SMF 115 subtype 216 record may contain details of up to 59 queues, where each queue statistics record comprises of 552 bytes of data.

Even with the number of data points collected per queue in conjunction with a high frequency of data capture (STATIME of 1 second), the cost of enabling queue statistics is minimal when monitoring up to one thousand queues. For our micro-benchmark workloads we have observed that this impact is no more than 3 microseconds on our simple CICS/COBOL transactions.

We compared the cost of reporting the queue depth of a set of queues using PCF against the cost of reporting the queue depth using the new queue statistics trace and found that when collecting data for 4 or more queues, the queue statistics was more cost effective than using PCF.

Queue statistics data - example output from MP1B's MQSMF

The application program MQSMF, available as part of MP1B "Interpreting accounting and statistics data", has been updated to support additional output data formats named: `QSTAT`, `QSTATCSV`, `QPUTSCSV` and `QGETSCSV`.

An example of the output from report `QSTAT` for a single private queue:

```
MVAA,VTS1,2024/06/12,12:20:16,VRM:940,
From 2024/06/12,12:20:15 to 2024/06/12,12:20:16, duration 1 seconds.

MVAA,VTS1,2024/06/12,12:20:16,VRM:940,
Queue Name.............................LQ1001
Disposition............................Private
Pageset ID.............................1
Bufferpool ID..........................1
Current Depth..........................60
Open Output Count......................1
Open Input Count.......................1
QTIME Short............................0
QTIME Long.............................0
Last Put Time..........................2024/06/12,12:20:15.625491
Last Get Time..........................
Uncommitted Changes....................No
Oldest Message Age.....................59
Highest Depth..........................60
Lowest Depth...........................59
Total Puts.............................1
MQPUTs.................................1
Persistent MQPUTs......................0
Non-Persistent MQPUTs..................1
MQPUT1s................................0
Persistent MQPUT1s.....................0
Non-Persistent MQPUT1s.................0
Total Put Bytes........................1024
MQPUT Bytes............................1024
Persistent MQPUT Bytes.................0
Non-Persistent MQPUT Bytes.............1024
MQPUT1 Bytes...........................0
Persistent MQPUT1 Bytes................0
Non-Persistent MQPUT1 Bytes............0
Fast Puts..............................0
Fast Puts Bytes........................0
Rolled Back Puts.......................0
Failed MQPUTs..........................0
Failed MQPUT1s.........................0
Successfully Streamed Messages.........0
Minimum Message Size Put...............1024
Maximum Message Size Put...............1024
Average Message Size Put...............1024
```

```
MQGETs.....................................0
Persistent MQGETs..........................0
Non-Persistent MQGETs......................0
MQGET Bytes................................0
Persistent MQGET Bytes.....................0
Non-Persistent MQGET Bytes.................0
MQGET Browses..............................0
Persistent MQGET Browses...................0
Non-Persistent MQGETs Browses..............0
MQGET Browse Bytes.........................0
Persistent MQGET Browse Bytes..............0
Non-Persistent MQGET Browse Bytes..........0
Failed MQGETs..............................0
Failed MQGETs with MQRC 2033...............0
Failed MQGETs with MQRC 2080...............0
Failed MQGET Browses.......................0
```

Notes on report:

- Queue LQ1001 is a private queue on page set 1, buffer pool 1 and had a message depth of 60 when the SMF record was captured at 12:20:16.
- During the 1 second interval, a single 1KB non-persistent message was put to the queue and there were no messages gotten (destructively) or browsed.

For a shared queue, the `QSTAT` report would have the following differences:

- `Disposition` would be `Shared` rather than `Private`
- `QSG Name` is reported.
- `CF Structure Name` replaces `Pageset ID` and `Bufferpool ID`.

Chapter 6

Streaming Queues - improved performance relating to streaming

Overview

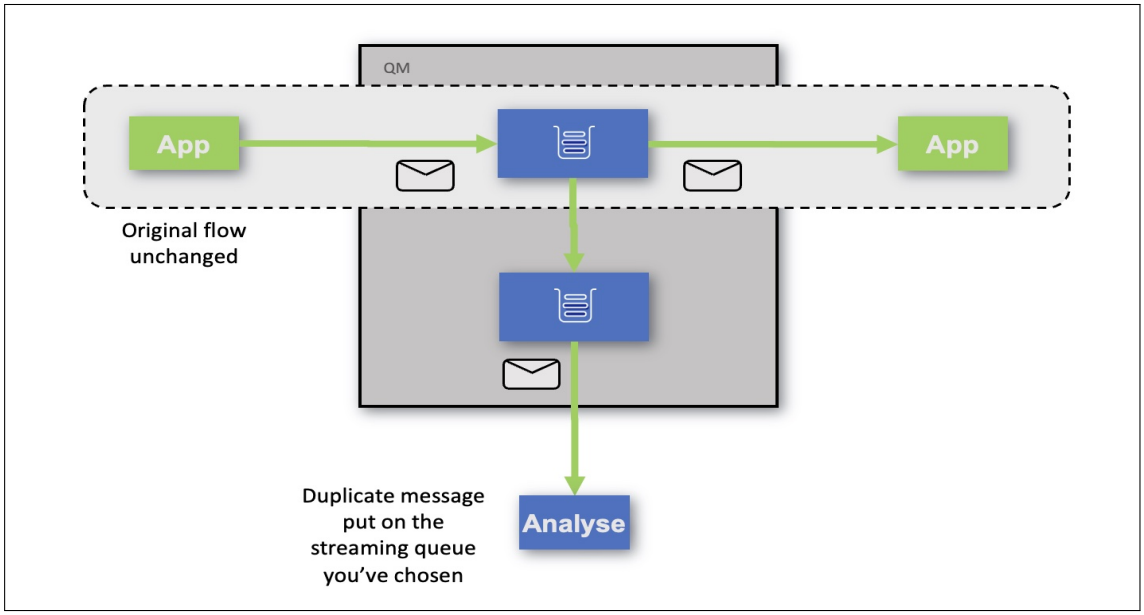
What are Streaming Queues?

IBM MQ for z/OS 9.3 introduced the Streaming Queues feature. Streaming queues are designed to provide a way for the MQ queue manager to take a copy of messages put to local queues, and deliver the copies to a separate queue or topic of your choosing.

The initial feature on MQ for z/OS only supported private queues. Subsquently APAR PH49686 provided support for shared queues.

In IBM MQ for z/OS 9.4, the performance of streaming to shared queues has been improved when using MQPUT1.

Diagram: Streaming queue overview



With streaming queues configured, when a putting application puts a message to the original queue,

a near-identical duplicate message is delivered to the stream queue. They are designed to provide a convenient way of capturing a duplicate stream of messages in order to:

- Stream messages to Apache Kafka using the Kafka Connect source connector for IBM MQ.
- Perform analysis on the data going through the system.
- Store messages for recovery at a later time.
- Capture a set of messages to use in development and test systems.
- Consume IBM MQ event messages from the system event queues, and sending additional copies to others queues or topics.

Because duplicate messages are delivered to the stream queue at put time, messages on the original queue when the stream queue is defined are not sent to the duplicate queue.

The messages sent to the stream queue are almost identical with a small number of exceptions. For example, the following aspects of the stream queue are identical to the original message:

- Message ID
- Correlation ID
- Message properties
- Application data

The following parts of the stream queue message may be different to the original:

- Report options that cause the queue manager to generate new messages are removed.
- Expiry is set to unlimited regardless of the expiry of the original message. If the stream queue has a CAPEXPRY value configured, then that value will be applied to the streamed message.

Configuration

Configuration is done on a queue-by-queue basis. For example if messages are currently delivered to QUEUE1 and it is desired that duplicates of those messages are streamed to QUEUE2, the STREAMQ attribute would be set as follows:

```
ALTER QL(QUEUE1) STREAMQ(QUEUE2) STRMQOS(BESTEF)
```

The `STRMQOS` attribute allows the choice of two streaming queue operating modes:

- **Best Effort (BESTEF)**
  - This option is the default and is the best option for scenarios where you do not want the original application to be affected by the streaming queue feature. Messages will be delivered to the original queue (i.e. QUEUE1) regardless of any problems encountered when streaming messages to QUEUE2. So if QUEUE2 is full when a message is put by the application it will be delivered to QUEUE1 for processing even though the queue manager is not able to stream a copy to QUEUE2.
  - Set using ALTER QL(QUEUE1) STRMQOS(BESTEF)
- **Must duplicate (MUSTDUP)**
  - This option is best for scenarios where it is important that each message delivered to the original queue is also delivered to the stream queue. If there is a problem streaming a message to QUEUE2 then the message will not be delivered to QUEUE1 and the putting application will receive an appropriate error code. The application will need to retry the MQPUT just as it would if QUEUE1 had a problem.

- Set using ALTER QL(QUEUE1) STRMQOS(MUSTDUP)

Why might I use streaming queues rather than a publish/subscribe model?

Prior to the implementation of streaming queues, if you required a near identical copy of a message, it was possible to configure a pub/sub model using topic alias. This however had several disadvantages over streaming queues, for example:

- Switching from using local queues to alias-to-topic could be complicated.
- Additional MQ objects to manage.
- Loss of message context.

By contrast using streaming queues rather than pub/sub means:

- You have the ability to leave your original application untouched and still create additional copies of messages.
- No need to stop applications while you reconfigure.
- No need to drain the original queue of messages.
- No need to delete queues and replace with topic aliases.
- Message ID, context etc remain unchanged.

What if I want multiple copies?

Streaming queues allow you to stream the duplicate messages to alias queues as well as local queues. If you define a queue alias to a topic, and configure a queue to stream messages to that topic alias, the duplicate messages are delivered to any subscribers to that topic based on MQ's existing publish/subscribe semantics. Importantly the messages delivered to subscribers will not be identical to the original because MQ's existing publish/subscribe behaviours are retained. One example of this difference is that each message delivered to subscribers will have a new message ID.

It is also worth nothing that if there are no subscribers to the topic the duplicate messages will not be delivered to any applications even when the original queue is set with `STRMQOS(MUSTDUP)`.

What do MQ statistics and accounting traces show?

Statistics trace

Class(1)

The data manager (QIST) record includes a count of messages put to streaming queues.

The message manager (QMST) record includes both the count of opens of streaming queues and the count of messages put to streaming queues.

Since class(1) statistics are generated at the scope of the queue manager, enabling streaming queue may not result in a significant change in the counts reported and as a result there may be little insight gained.

Class(5)

Enabling class(5) statistics in parallel with enabling `STATQ(ON)` on the streaming queue will show the relevant queue statistics data.

An extract of the MQSMF QSTAT report is shown below. This report is available as part of MP1B "Interpreting accounting and statistics data".

Queue with STREAMQ attribute set		Streaming Destination queue	
Queue Name	LQ1000	Queue Name	STRM000
Disposition	Private	Disposition	Private
Pageset ID	3	Pageset ID	5
Bufferpool ID	3	Bufferpool ID	5
Current Depth	10	Current Depth	10
..		..	
Last Put Time	..,07:06:17.097465	Last Put Time	..,07:06:17.097470
..		..	
Successfully Streamed Messages	10	Successfully Streamed Messages	0
Minimum Message Size Put	1024	Minimum Message Size Put	1024

Notes on table:

- Page set and buffer pool for the streaming destination queue is different from the base queue.
- Depth of queue is consistent between base queue and streaming destination queue.
- Last put time of streaming destination 5 microseconds later than the base queue.
- Baseline queue reports that 10 messages were successfully streamed.
- Minimum message sizes are consistent between queues.

Accounting trace

Class(1)

The class(1) accounting records (QMAC) **do not** include counts for messages put to streamed queues, but do include the additional CPU time used on MQPUT and MQPUT1 when using streaming queues.

Since the data reported by class(1) accounting includes the information for tasks *ending* in that interval and therefore may contain information spanning and summarising many hours of activity, the additional cost of enabling streaming queues may not result in a significant change in the CPU time reported for any particular interval.

Using class(1) accounting trace in conjunction with streaming queues is unlikely to provide any significant insight as to the performance of streaming queues on your systems.

Class(3)

The class(3) accounting records (WTID and WTAS) now include counts for messages put to streamed queues, plus some insight can be gained when comparing the data from tasks before and after streaming queues are configured.

The following sample data from MQSMF's TASK report, which is part of MP1B "Interpreting accounting and statistics data", shows the differences as a result of enabling streaming queues when the application puts 10 non-persistent messages of 1KB.

Example MQSMF TASK output for batch task performing 10 MQPUTs:

Without streaming queue	With streaming queue
VTS1 Batch Jobname:R001 Userid:PERFTASK	VTS1 Batch Jobname:R001 Userid:PERFTASK
...	...
Pages old 32	Pages old 64
Pages new 11	Pages new 22
...	...
Open name LQ1000	Open name LQ1000
Queue type: QLocal LQ1000	Queue type: QLocal LQ1000
Queue indexed by NONE LQ1000	Queue indexed by NONE LQ1000
First opened 2024/06/23,07:26:49 LQ1000	First opened 2024/06/23,07:38:51 LQ1000
Last closed 2024/06/23,07:26:49 LQ1000	Last closed 2024/06/23,07:38:51 LQ1000
Page set ID 3 LQ1000	Page set ID 3 LQ1000
Buffer pool 3 LQ1000	Buffer pool 3 LQ1000
Current opens 0 LQ1000	Current opens 0 LQ1000
Total requests 12 LQ1000	Total requests 12 LQ1000
Open count 1 LQ1000	Open count 1 LQ1000
Open avg elapsed time 34 uS LQ1000	Open avg elapsed time 39 uS LQ1000
Open avg CPU time 34 uS LQ1000	Open avg CPU time 39 uS LQ1000
Close count 1 LQ1000	Close count 1 LQ1000
Close avg elapsed time 7 uS LQ1000	Close avg elapsed time 11 uS LQ1000
Close avg CPU time 7 uS LQ1000	Close avg CPU time 10 uS LQ1000
Put count 10 LQ1000	Put count 10 LQ1000

Put avg elapsed time	10	uS	LQ1000	Put avg elapsed time	17	uS	LQ1000
Put avg CPU time	10	uS	LQ1000	Put avg CPU time	16	uS	LQ1000
...				...			
Put + put1 valid count	10		LQ1000	Put + put1 valid count	10		LQ1000
Put size maximum	1024	bytes	LQ1000	Put size maximum	1024	bytes	LQ1000
Put size minimum	1024	bytes	LQ1000	Put size minimum	1024	bytes	LQ1000
Put size average	1024	bytes	LQ1000	Put size average	1024	bytes	LQ1000
Streamed messages	0		LQ1000	Streamed messages	10		LQ1000
...				...			
Total queue elapsed time	149	uS	LQ1000	Total queue elapsed time	226	uS	LQ1000
Total queue CPU used	148	uS	LQ1000	Total queue CPU used	217	uS	LQ1000

Notes on table:

- Pages used - both new and old have doubled as a consequence of configuring a stream queue.
- Open, put and close counts remain the same across configurations.
- Average elapsed and CPU times increase for open, put and close.
- Total queue CPU used increases in-line with the sum of the open, put and close CPU times.
- Streamed messages count increases from zero to 10.

Who pays for Streaming Queues?

The putting application is charged for the additional work performed by the configuration of streaming queues.

This may include the channel initiator for messages arriving via MQ channels with a queue destination configured with `STREAMQ`.

The additional time taken for the MQOPEN, MQPUT, MQPUT1 and MQCLOSE APIs will increase the usage of the channel initiator adaptor tasks.

Should you be planning to use streaming queues, it may be beneficial to review your adaptor (`CHIADAPS`) usage using the SMF 115 subtype 231 records generated from enabling MQ's `TRACE(S) CLASS(4)`. This is of particular importance if streaming persistent messages.

When using shared queues for streaming, there will be additional load in both the Coupling Facility (application structure) and the MQ MSTR address space.

Which MQ APIs are affected by Streaming Queues?

Generally, only MQOPEN, MQPUT, MQPUT1, and MQCLOSE are affected by enabling streaming queues, but when using streaming to a shared queue in a separate CF structure to the base queue, there can also be increased cost to the MQ commit.

Primarily this section will concentrate on the costs as reported by class 3 accounting data.

The numbers reported by class 3 accounting data may vary depending on a number of factors - in our test environment, MQ is the focus of the workload whereas in your environment, MQ may be just a small part of the entire workload.

This can affect the reported costs of each particular API - for example in our environment we may be driving the MQOPEN / MQCLOSE APIs at a high rate using a small number of tasks whereas a more real-world configuration might be that many applications are connecting, opening the queue once and performing many MQPUTs and MQGETs.

As such, often called MQOPENs in our environment may cost 1-2 CPU microseconds, but the same MQOPEN called occasionally may cost tens of microseconds - which means it is important for the user to look at the values as reported on their own system.

Additionally, and as a side effect of the type of measurements being run for this report, the overhead of streaming queues may be a 1 or 2 microseconds but this may equate to doubling or more the cost of the MQOPEN API. In a different environment where the MQOPEN costs for example 30 microseconds, the impact of streaming queues may be different and as such should be measured on your own system.

Private Queue: Impact of Streaming Queues on MQOPEN and MQCLOSE

It is difficult to separate MQOPEN and MQCLOSE simply because an MQOPEN should be paired with an MQCLOSE, so they will be discussed in parallel.

In our environment the impact of opening a queue where the `STREAMQ` attribute was set, doubled the cost of the MQOPEN. Whilst doubling the cost of the MQOPEN appears high, it is worth noting that this meant the cost increased by 1 CPU microsecond to a total of 2 CPU microseconds.

The following table compares the cost of an MQOPEN of a private queue against a private queue with a streaming queue configured as well as the cost of opening a topic with 0, 1 and 2 subscribers.

MQOPEN	Private queue)	Private queue and streaming queue	Alias-to-topic 0,1,2 subscribers
Cost (CPU microseconds)	1	2	3
Elapsed time (microseconds)	1	2	6

Notes on table:

- Increasing the number of subscribers does not affect the MQOPEN costs of Alias-to-topic.
- Pub/sub also accrues cost under SRB in the MQ queue manager - for MQOPEN this equates to an additional 1.4 CPU microseconds per MQOPEN.

For MQCLOSE the impact was less - the CPU cost was reported as zero whether a `STREAMQ` was configured or not - but the elapsed time increased by 1 microsecond.

Private Queue: Impact of Streaming Queues on MQPUT and MQPUT1

On our performance systems, the impact of streaming queues on MQPUT and MQPUT1 was similar to that of MQOPEN, i.e. in the range of **2 to 2.5 times** cost as reported by class 3 accounting data.

It must be emphasised that this is **not** doubling the application cost by enabling streaming queues - but it is 2-2.5 times the MQ cost of the MQPUT when putting to a queue configured with a streaming queue.

Private Queue: How much does message size affect streaming costs?

Message size does affect the overhead of streaming queues to a certain extent.

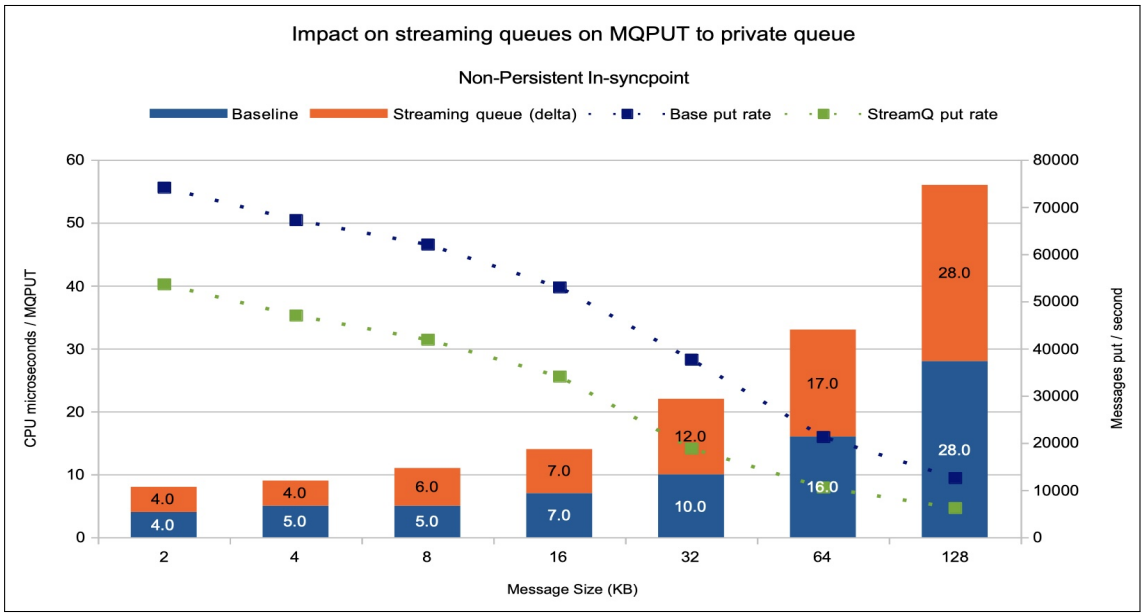
The following table shows the cost of a batch application putting messages of increasing sizes to both a private queue and then to a private queue where a streaming queue is configured. The values reported are class 3 accounting data based on thousands of messages being put to queues where the buffer pool is sufficiently large to avoid the MQPUT being delayed whilst offloading to page set.

Message size	Private queue)	Private queue and streaming queue	Overhead of streaming queue
2KB	4	8	4
4KB	5	9	4
8KB	5	11	6
16KB	7	14	7
32KB	10	22	12
64KB	16	33	17
128KB	28	56	28

Whilst the overhead of streaming queues grows from 4 to 28 microseconds when increasing messages from 2KB to 128KB, the factor increase remains approximately double that of the private queue without streaming.

The following chart repeats the data in the table, whilst including the rate at which the single batch application was able to put the messages. The application uses a model of [put-then-commit] repeated thousands of times.

Chart: Impact of streaming queues on MQPUT



Notes on chart:

- The stacked chart shows that for 128KB messages, the cost of putting to a private queue cost 28 microseconds - and configuring a streaming queue added 28 microseconds to the MQPUT for a total of 56 microseconds.
- The achieved rate of the streaming queue configuration was between 50 and 27% lower than the baseline configuration. This is because the batch application is in a tight loop of MQPUT-then-Commit and does not have any business logic. As a result, any impact to MQAPIs causes a significant impact to the achievable performance. Applications with less intensive MQ operations, i.e where MQ does not account for 95% of the application cost, would see a smaller impact from streaming queue.

Shared Queue: Impact of Streaming Queues on MQOPEN and MQCLOSE

The impact of streaming queue on shared queues can be far more significant than for private queues, particularly for the MQOPEN verb.

In our measurements, a typical MQOPEN and MQCLOSE where the queue is shared, costs of the order of 18 CPU microseconds.

With the `STREAMQ` attribute configured to use a shared queue, the cost increased to 32 CPU microseconds. This was regardless of whether the streaming queue was on the same or a different application structure in the same geographically located and configured Coupling Facility.

The "first-open and last-close" effects as discussed in MP16 "Capacity Planning and Tuning Guide" section "Frequent opening of shared queues" can make a significant difference to the cost of both the MQOPEN and MQCLOSE when accessing shared queues.

Avoiding CF access for the open and close of a queue configured *without* a streaming queue reduces the cost from 18 to 5.4 CPU microseconds.

Avoiding CF access for the open and close of a queue configured *with* a streaming queue reduces the cost from 32 CPU microseconds, but the scale of the impact will depend on how that CF access is avoided.

The following table illustrates the impact of avoiding first-open and last-close effects on the queue referenced by the `STREAMQ` attribute.

Configuration	Effect on streaming queue	Cost of MQOPEN and MQCLOSE (CPU microseconds)
First-open and last-close effect	CF Access every time	32
Hold base queue open for input wrong type	CF Access every time	20
Hold base queue open before streamQ configured	CF Access every time	20
Hold base queue open when streamQ in separate structure	CF Access every time	20
Hold only streamQ open	No CF Access	20
Hold base queue open with streamQ in same structure	No CF Access	7.5
Hold base queue and stream queue open (different structures)	No CF Access	7.5

Shared Queue: Impact of Streaming Queues on MQPUT and MQPUT1

On our performance systems, the impact of using shared queues when streaming on MQPUT results in the costs increasing by approximately **2 to 3 times**, as reported by class 3 accounting data. The impact of streaming queues on MQPUT1 can be more significant, with the costs as reported by class 3 accounting data increasing by up to 3 times.

As with private queues, it must be emphasised that the impact of streaming queues on MQPUTs to shared queues is not necessarily doubling the application cost.

Shared Queue: How much does the type of MQPUT affect the cost?

With shared queues, the impact of adding a streaming queue can vary depending on a number of factors, including whether the message is put in syncpoint. The following table offers example class 3 accounting costs for MQPUT and MQPUT1 in a range of configurations when putting a 1KB non-persistent message.

Configuration	STRMQOS	Baseline put cost (CPU microseconds)	StreamQ put cost (CPU microseconds)
MQPUT out-of-syncpoint	BESTEF	7	15
MQPUT out-of-syncpoint	MUSTDUP	7	23
MQPUT in-syncpoint	Either	7	15
MQPUT in-syncpoint with StreamQ on separate structure	Either	8	15
MQPUT1 in-syncpoint	Either	10	21

Notes on table:

- The high cost of MUSTDUP on the MQPUT out-of-syncpoint measurement is due to effectively enforcing an MQ commit such that the put is successful for both or neither queue. The blog "MQ for z/OS - CF statistics" discusses the additional CF statistics data included as part of MQ's task records, and reviewing this data shows that the additional cost arises from MQ having to issue 2 additional "New", 1 "Write" and 1 "MoveEnt" calls to the Coupling Facility.
- In IBM MQ for z/OS 9.3, there was a three-times cost increase incurred when using MQPUT1 with shared queues which was due to 1 additional "new" and 2 "write" requests. In IBM MQ for z/OS 9.4, the cost has been reduced as there is 1 additional "new" and a much lower ratio of "write" requests.

Shared Queue: How much does message size affect streaming costs?

Message size does affect the overhead of streaming queues to a certain extent.

The following table shows the cost of a batch application putting messages of increasing sizes to both a shared queue and then to a shared queue where a streaming queue is configured. The values reported are class 3 accounting data based on thousands of messages being put to queues.

Message size	Shared queue	Shared queue and streaming queue	Overhead of streaming queue
2KB	8	16	8
4KB	9	19	10
8KB	11	23	12
16KB	16	32	16
32KB	27	54	27
64KB	25	51	26
128KB	35	70	35

Whilst the overhead of streaming queues grows from 8 to 35 microseconds when increasing messages from 2KB to 128KB, the factor increase remains approximately double that of the shared queue without streaming.

The following table shows the cost of a batch application using MQPUT1 to put messages of increasing sizes to both a shared queue and then to a shared queue where a streaming queue is configured. The values reported are class 3 accounting data based on thousands of messages being put to queues.

Message size	Shared queue	v9.3 Shared queue and streaming queue	v9.4 Shared queue and streaming queue
2KB	10	33	21
4KB	12	35	24
8KB	14	40	28
16KB	18	49	38
32KB	27	65	58
64KB	28	68	56
128KB	39	89	77

Notes on table:

- In IBM MQ for z/OS 9.4, the cost of MQPUT1 has been reduced by 11-12 CPU microseconds compared to IBM MQ for z/OS 9.3.

Shared Queue: Impact of Streaming Queues on MQCMIT

When the `STREAMQ` is configured on a separate application structure to the base queue, there is additional cost in both the z/OS LPAR and the Coupling Facility, and this is due to the impact of additional CF work at the time of the commit.

The following table compares an MQPUT and MQCMIT when using streaming queues, where in one instance the streaming queue uses the same application structure as the base queue and in the second instance the streaming queue uses a separate application structure.

Once more, we are using the CF statistics as discussing in the blog "MQ for z/OS - CF statistics".

	Queues on same structure	**Queues on separate structures**
MQPUT to base and streaming queue	2 x "New"	2 x "New"
MQCMIT	1 x "New" 1 x "Write" 1 x "MoveEnt"	2 x "New" 1 x "Write" 2 x "MoveEnt"
z/OS cost per MQPUT	15 CPU microseconds	15 CPU microseconds
z/OS cost per MQCMIT	8 CPU microseconds	18 CPU microseconds
CF cost per MQPUT	8 CPU microseconds	9 CPU microseconds
CF cost per MQCMIT	12 CPU microseconds	20 CPU microseconds

Notes on table:

- The MQPUT performance is not notably impacted by the streaming queue being on a separate application structure.
- The MQCMIT performance is affected by having to commit across the two structures, with additional calls to the CF in the form of 1 "New" and 1 "MoveEnt". These additional CF calls result in both increased load on the z/OS LPAR and the CF.

How much will streaming queues cost?

Throughout this chapter, we have reported that according to class 3 accounting data, the impact of using queues with `STREAMQ` configured is to effectively double the cost of the affected MQ APIs.

How much enabling streaming queues will affect your workload will very much depend on how you measure the cost of the MQ workload. For example, if the costs are measured using the RMF Workload report and based on the CPU used by the MQ queue manager and the applications, the impact may be much less than observed from the class 3 accounting data.

Similarly if using class 1 accounting data, the impact may be much less than reported by class 3 accounting data.

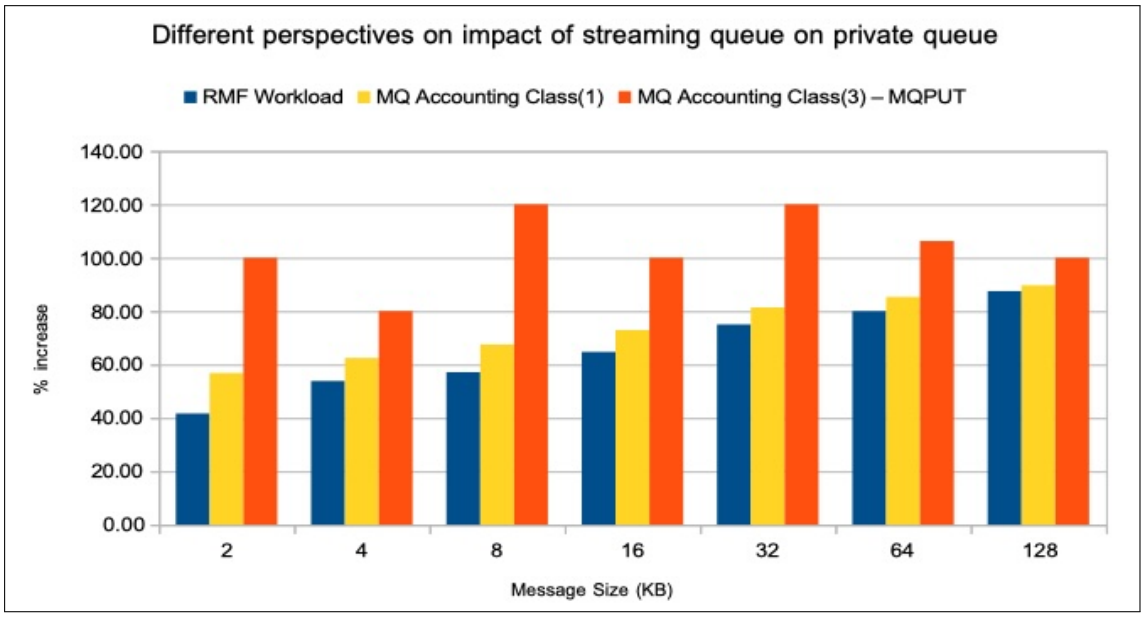
In these measurements, we use a simple batch application that opens the queue and then repeatedly put messages to the base MQ queue, before closing the queue and exiting. The measurements are then repeated with streaming queues configured. A more complex application that performed MQGETs, and performed database requests or file I/O would see a much smaller impact from streaming queues as the proportion of MQPUT work compared to the total workload would be much less.

The measurements compare the impact of streaming queues based on comparing the costs from the:

- RMF Workload report - calculating the transaction cost based on the CPU used by MQ MSTR address space and the application.
- MQ class 1 accounting data - based on the total cost of an MQOPEN, multiple MQPUT and MQCMIT calls and MQCLOSE, divided by the number of MQPUTs.
- MQ class 3 accounting data - based on the average cost of the MQPUT.

The first chart shows that the impact of streaming queues on a private queue workload can vary significantly depending on which data is compared, and the second chart presents the data from the test using shared queues.

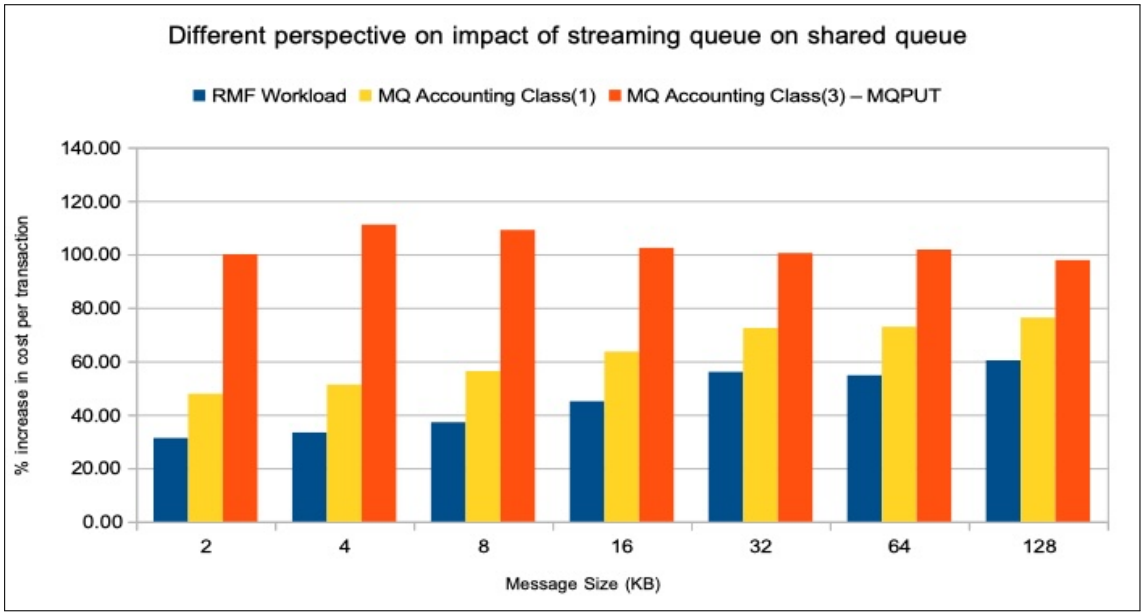
Chart: Impact of streaming queues on private queue - comparing different sources of cost



Notes on chart:

- With a very simple application primarily calling MQ APIs, the impact of streaming queues according to the RMF Workload report is to add between 40 and 90%.
- Class 1 accounting data shows an impact of between an additional 55 to 90%.
- Class 3 accounting data shows the impact to be approximately doubling the MQPUT cost.

Chart: Impact of streaming queues on shared queues - comparing different sources of cost



Notes on chart:

- With a very simple application primarily calling MQ APIs, the impact of using shared queues for streaming according to the RMF Workload report is to add between 30 and 60%.
- Class 1 accounting data shows an impact of between an additional 45 to 75%.
- Class 3 accounting data shows the impact to be approximately doubling the MQPUT cost.

Must Duplicate or Best Efforts?

As discussed in the configuration section, the **must duplicate** option ensures that either both the original message and the streaming queue message are delivered or neither are delivered.

By contrast, the **best effort** option will allow the original message to be put, even if the streaming queue message is not able to be delivered.

Should an error occur when putting the streaming message, regardless of streaming quality of service (QoS), a message will be written to the queue manager job log in the form:

```
CSQM587E VTS1 CSQMPUT Unable to put to streaming queue STRMQ000 for queue LQ1000, mqrc=2192 (MQRC_PAGESET_FULL)
```

This message should only be logged once for the life time of the application handle, unless the error changes.

In the event of the `CSQM587E` message being logged, the MQPUT API may get a non-zero return code, depending on the `STRMQOS` value.

Consider the following:

- Queue manager has queue LQ1000 defined on page set with 2 million pages.
- Queue manager has queue STRMQ000 defined on page set with 20,157 pages and `EXPAND(NONE)`.
- Queue LQ1000 configured to use STREAMQ(STRMQ000) with STRMQOS configured as either BESTEF or MUSTDUP.

A single batch application is configured to put 500,000 non-persistent messages of 1KB to queue LQ1000.

The following is an example of the different outcomes based on the value of `STRMQOS`.

STRMQOS(BESTEF):

- Application reports 500,000 messages put to queue LQ1000 and exits with return code 0.
- Queue manager logs `CSQM587E` with return code 2192 (MQRC_PAGESET_FULL).
- LQ1000 depth reported as 500,000.
- STRMQ000 depth reported as 19,980.

STRMQOS(MUSTDUP):

- Application reports MQPUT failed with mqrc=2192 MQRC_PAGESET_FULL, on iteration 19,881.
- Queue manager logs `CSQM587E` with return code 2192 (MQRC_PAGESET_FULL).
- LQ1000 depth reported as 19,980.
- STRMQ000 depth reported as 19,980.

What else differentiates MUSTDUP and BESTEF?

Setting MUSTDUP as the QoS on a stream queue has a small performance impact as there is additional work carried out to ensure that the original message is rolled back if the MQPUT of the duplicate failed.

How much does MUSTDUP affect performance?

The impact of MUSTDUP is relatively small when compared to BESTEF, and a comparison is offered in the following scenario.

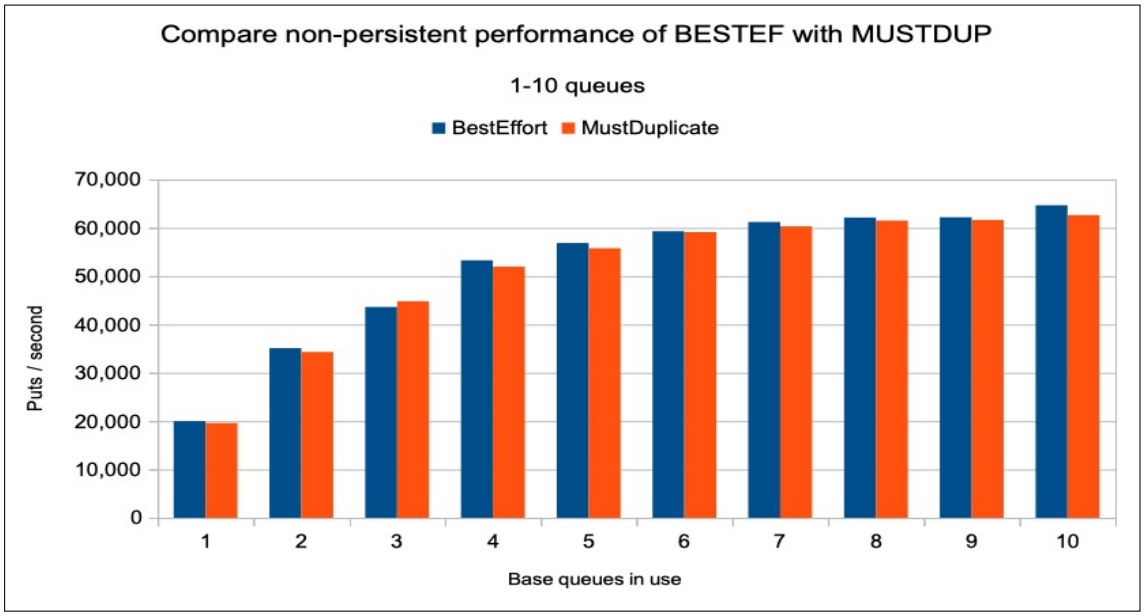
The test shown uses a set of applications - one putting task and two getting tasks per queue. One of the getting tasks will additionally reply to the putting task, whereas the second task will perform gets-only.

As time progresses, the test adds further sets of applications until there are 10 queues in use, with a total of 10 putting tasks and 20 getting tasks.

The data is reported as the average transaction rate over multiple 15 second SMF intervals per queue count.

This chart compares the performance of streaming queues using BESTEF against MUSTDUP and shows a small impact to the performance due to the additional processing required to ensure that puts to both the base and streaming queue are completed with the same outcome to both queues.

Chart: Non-Persistent message comparison of Streaming Queues BESTEF against MUSTDUP



Notes on chart:

- The BESTEF streaming queues configuration achieved a 16% improvement in transaction rate over the equivalent MUSTDUP streaming queue configuration.
- MQ queue manager costs were comparable when using streaming queues with MUSTDUP and BESTEF.
- Put application costs increased by 1-2 CPU microseconds when using MUSTDUP rather than BESTEF.
- Get application costs were similar in both configurations.

Streaming Queues and Message Expiry

One use for streaming queues is to create duplicate messages which will be stored for a short period of time as a contingency measure. In this case, the `CAPEXPRY` attributed is set on the streaming queue to set a time limit on messages, for example to set the messages to expire after 20 minutes:

```
ALTER QL(STRMQ000) CAPEXPRY(12000)
```

Note: In IBM MQ for z/OS 9.4 the CAPEXPRY attribute has been updated to be a "first-class" attribute, rather than an attribute needing to be set as a custom property. Using custom properties to set CAPEXPRY is still supported but using the first-class attribute takes priority.

IBM MQ for z/OS schedules an expiry task run at an interval specified by the queue manager `EXPIRYINT` attribute, for example:

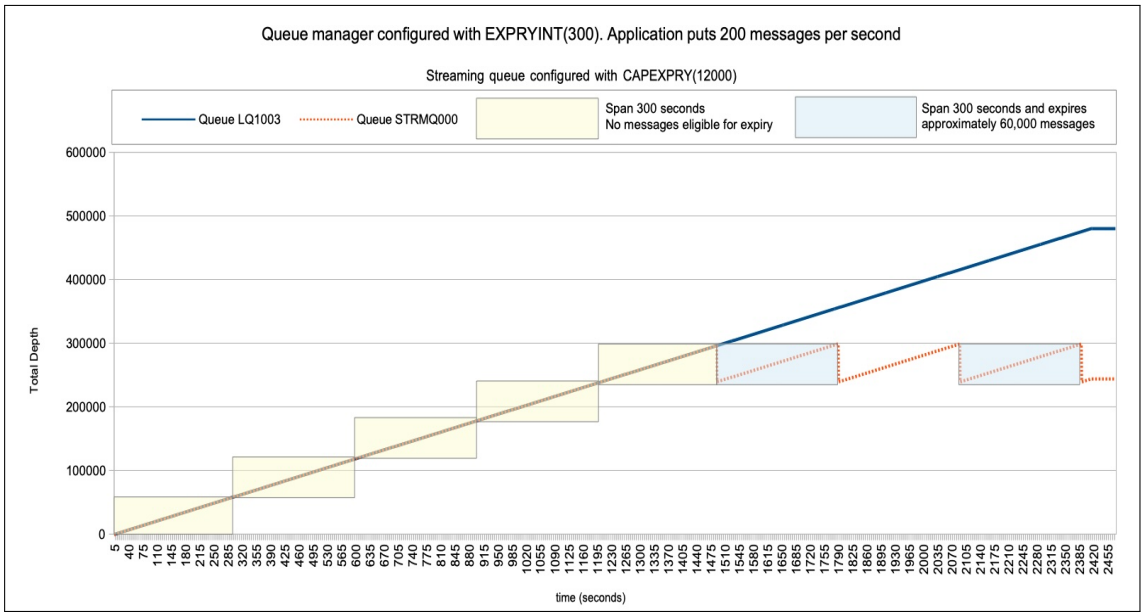
`ALTER QMGR EXPRYINT(300)` will set the expiry timer to run every 300 seconds.

This expiry task deletes any messages that are older than the `CAPEXPRY` value set on it. `CAPEXPRY` has queue scope, and only affects messages arriving after it has been set, similarly amending the `CAPEXPRY` value will only affect subsequent messages.

If the messages on a streaming queue with `CAPEXPRY` set are not being consumed, then the queue will grow until the expiry task is scheduled and finds messages that are old enough to be deleted.

The following diagram shows the depth of the base queue (LQ1003) and the streaming queue (STRMQ000) as MQ's expiry task is run every 300 seconds and messages are expired from the streaming queue after 20 minutes.

Diagram: EXPRYINT and CAPEXPRY example:



Notes:

In this example the messages are arriving at 200/second on a single queue. The plot shows the depth of the base and streaming queue, where the streaming queue has `CAPEXPRY` is set to 20 minutes and the queue manager has `EXPRYINT` set to the value of 300 seconds. Both queues grow in depth until the expiry task encounters messages that are 20 minutes or more old. At that point, each time the

expiry task is run, it will find another 5 minutes worth of messages that can be deleted, creating the saw-tooth effect above.

When using message expiry with streaming queues, you should consider the depth that the queues will grow to and ensure that MQ page sets are configured such that streaming queues with `STRMQOS(MUSTDUP)` do not cause failures in the primary applications.

EXPRYINT - why is the default value "OFF"?

The expiry task will only expire messages that are eligible for expiry based on their `CAPEXPRY` value.

Typically the cost of the MQ task identifying and subsequently deleting eligible messages for expiry is relatively small but is not zero.

For **private queues**, we have measured the costs as approximately 4 CPU microseconds per message expired when the messages are exclusively held in buffers. This cost increases to 7 microseconds when the task needs to access the data on the MQ page set, however MQ does not need to read the full message back into buffer pools, which does reduce the amount of page set I/O.

As the process to identify eligible messages is de-coupled from the process to delete the messages, the expiry task is able to identify and delete up to 250,000 messages per CPU second for expiring.

For **shared** queues, the impact of expiry processing is more significant.

MP16 "Capacity Planning and Tuning Guide" states:

- Each defined shared queue must be processed.
- A single queue manager, of those with non-zero `EXPRYINT` in the QSG, will take responsibility.
- ***All*** queues, including those where expiry is not set, will be scanned.

What this means is that if `EXPRYINT` is set on a queue manager that is part of a QSG, even if the intention is to expire messages solely on private queues, at the expiry interval the queue manager will also scan all shared queues, regardless of whether `CAPEXPRY` has been set.

In our system where we have a local CF which is non-duplexed with the fastest available links and sufficient dedicated processors, scanning shared queues with combined depths of 225,000 messages where there were no queues configured to use message expiry, resulted in 0.877 CPU seconds of CF. In this instance there were 225,000 synchronous requests to the CF with an average time of 3.9 microseconds.

When message expiry was set on the shared queues holding 225,000 messages, the impact to the CF increased due to 450,000 synchronous requests with an average time of 3.1 microseconds. This meant that to expire 225,000 messages, the CF used 1.4 CPU seconds.

If the `EXPRYINT` is set to a small interval on systems with less responsive CF's or more messages on queues, there is a risk that the expiry task will be overwhelmed (*unable to complete before the next interval begins*).

Due to the risk of increased cost of scanning all shared queues for messages eligible for expiry, even on queues where `CAPEXPRY` is not set, the default setting for `EXPRYINT` is `OFF`.

Performance of Streaming Queues vs Pub/Sub on private queues

In this section, the performance of streaming queues is compared directly with a pub/sub configuration where there are 2 subscribers. This gives the best comparison point when two copies of a message are required.

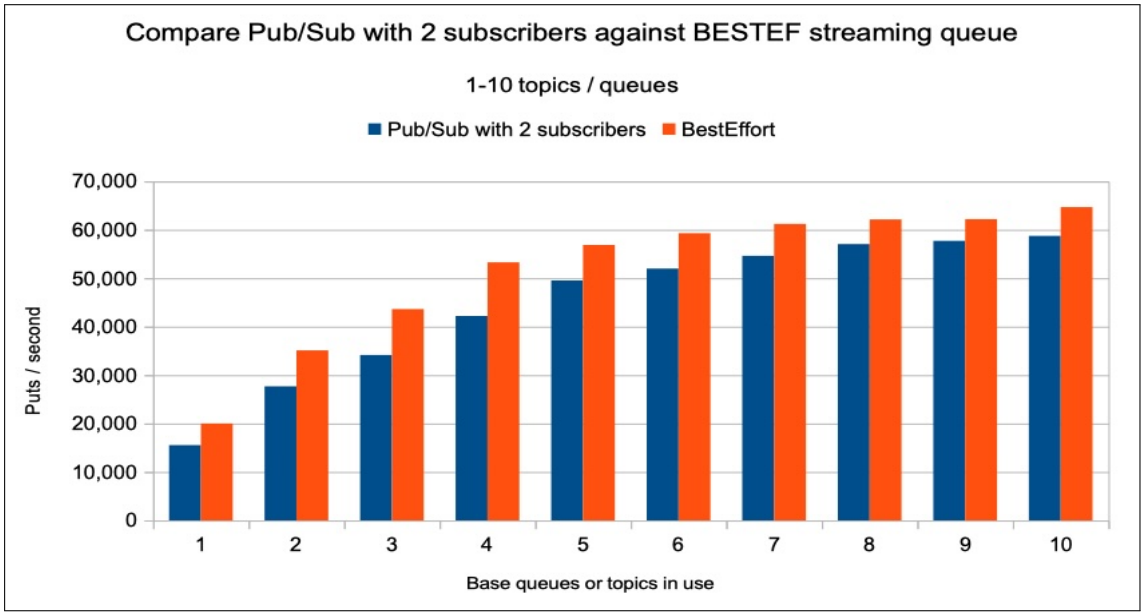
The test shown uses a set of applications - one putting task and two getting tasks per queue / topic alias. One of the getting tasks will additionally reply to the putting task, whereas the second task will perform gets-only.

As time progresses, the test adds further sets of applications until there are 10 queues (or topic aliases) in use, with a total of 10 putting tasks and 20 getting tasks.

The data is reported as the average transaction rate over multiple 15 second SMF intervals per queue count.

Performance of Non-Persistent messages

Chart: Non-Persistent message comparison of Pub/Sub with Streaming Queues (BESTEF)



Notes on chart:

- The streaming queues configuration is achieving an 18% improvement in transaction rate over the equivalent pub/sub configuration.
- MQ queue manager costs were comparable per transaction when using streaming queues compared to the equivalent pub/sub configuration.
- Put application costs were 30% lower per transaction when using streaming queues compared to the equivalent pub/sub configuration.
- Get application costs were similar in both configurations.

Comparing the performance of MUSTDUP against pub/sub with 2 subscribers can be summarised as:

- The MUSTDUP streaming queues configuration is achieving up to 16% improvement in transaction rate over the equivalent pub/sub configuration.

- MQ queue manager costs per transaction were comparable when using MUSTDUP streaming queues compared to the equivalent pub/sub configuration.
- Put application costs were 27% lower per transaction when using MUSTDUP streaming queues compared to the equivalent pub/sub configuration.
- Get application costs were similar in both configurations.

Performance of Persistent messaging

When enabling streaming queues on workloads that use persistent messages, it is important to recognise that there will be additional load on your MQ logging capacity.

Putting persistent messages to queues configured with streaming queues will effectively result in **twice** the amount of data being logged for those transactions.

This additional load may impact your system in a number of ways.

If the queue manager is already running at the maximum capacity of the MQ logger task, the additional load may result in a decrease in transaction rate of the putting tasks - or impact other workloads that also rely on persistent messaging.

If the queue manager is not running at capacity, or else there is additional capacity in the MQ logger task but not for your existing workload(s), enabling streaming queues with persistent messages will increase the API time of the applications opening, putting and closing the queues with streaming queues configured as well as increasing the time to write the data to the MQ log data sets(s).

Note, to clarify the statement "there is existing capacity in the MQ logger task but not for your existing workload(s)", is where the MQ SMF log manager records indicate that the logger task is fully utilised for the entire SMF interval but the pages per I/O is less than 128. This can be determined from MP1B's MQSMF program using the `LOG` report, as below:

```
...
Pages written per I/O:   12
...
Log write rate:          227MB/s per copy
Logger I/O busy:         94.82%
Logger task busy:        100%
```

In this instance, the `LOG` report shows that the logger task was 100% busy throughout the interval, therefore can be regarded as fully utilised and the time spent waiting for I/O accounts for 94.82% of the interval. The simplest, but not necessarily the least costly, way to increase the capacity of this queue manager whilst writing 12 pages per I/O would be to have more responsive DASD and faster links to the DASD. Alternatively in IBM MQ for z/OS 9.4, there is the option to use zHyperlink™ which is discussed in "zHyperlink support for MQ log task".

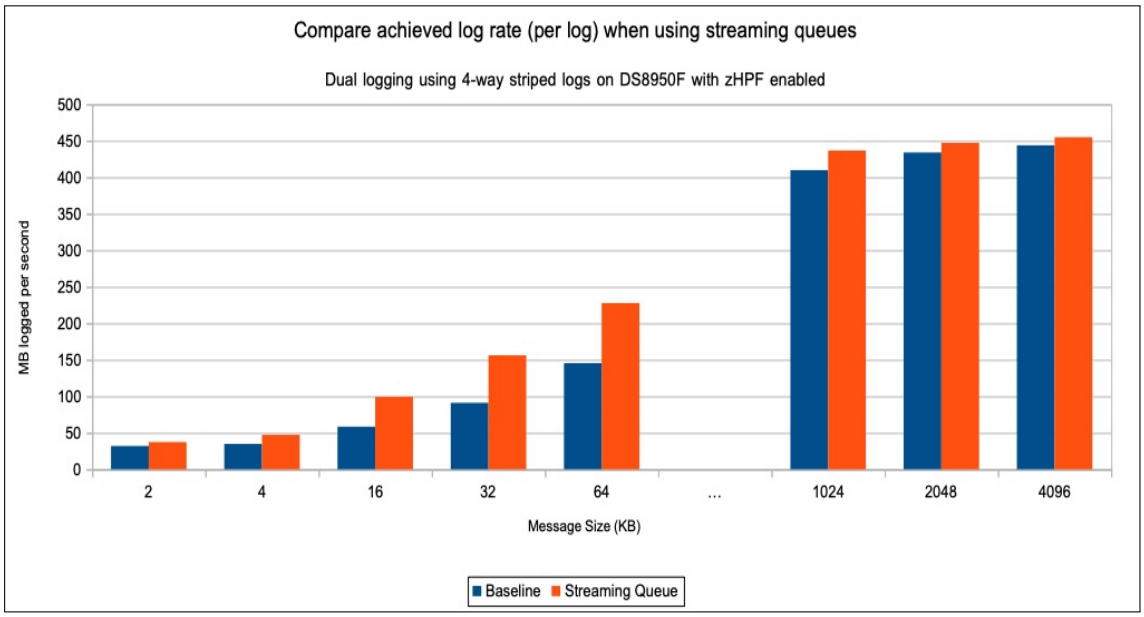
Additionally there were 12 pages written per I/O (i.e. 48KB per I/O) and the log task was able to sustain 227MB per second per log copy.

In this example we can regard the log task as fully busy *for the current workload*, but this does not mean that the queue manager does not have capacity for more persistent work. Potentially the log task could write 128 pages per I/O request, which is a significant step up from the 12 pages it is currently writing. It is worth noting that writing 128 pages in a single I/O is likely to take longer than writing 11 pages per I/O - and therefore the *existing* workloads may see a drop in performance.

To illustrate the impact of streaming queues on persistent messaging, we can compare the performance with the upper bounds of persistent logging rate tests in the regression section of this report.

This particular workload uses just 3 batch tasks to drive the MQ log task to 100% busy. One of the tasks uses a 1KB persistent message that is put and got repeatedly from a queue. The remaining two tasks vary the size of their message as per the x-axis but also repeatedly perform MQPUT and MQGETs.

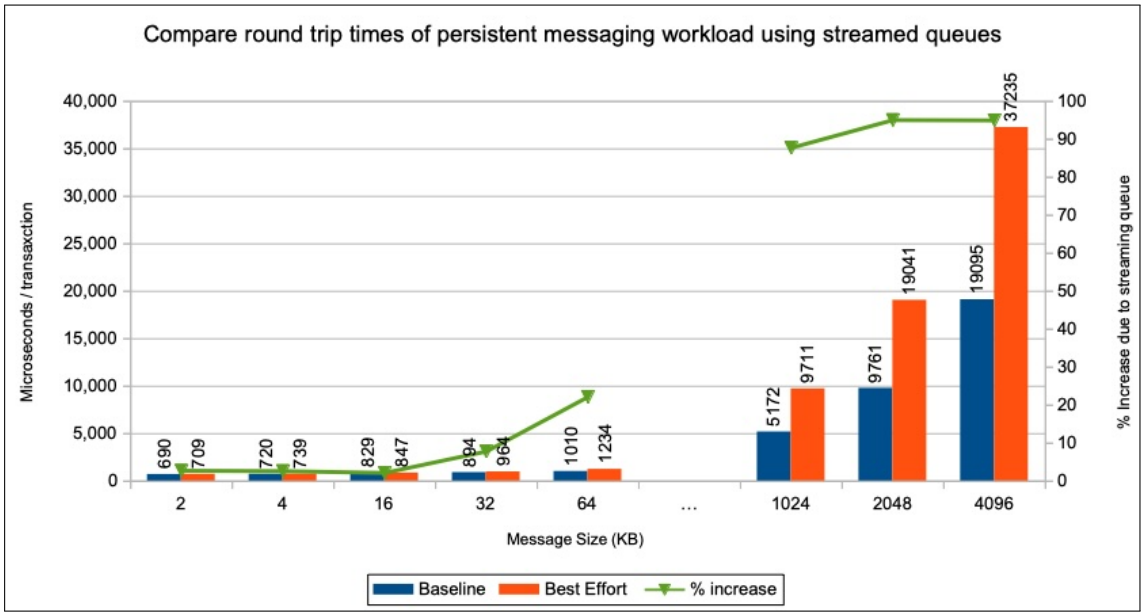
Chart: Achieved log rate when using streaming queues.



For each message size used, the baseline queue tests reported the log task as being fully busy throughout the SMF intervals. Despite this, for the message sizes up to 64KB, adding streaming queues resulted in the log write rate increasing by up to 70%. For messages of 1MB and larger, where the log task was already writing 100+ pages per I/O, the log task was already close to capacity, and adding streaming queues only increased the maximum log rate by 2-6%.

The following chart shows the impact that applying a streaming queue to the workload, so that all messages put by the 3 batch tasks are duplicated. The chart shows the round-trip times, i.e. the time to complete a transaction consisting of the MQPUT and the MQGET for the tasks using the message size as per the x-axis.

Chart: Impact to transaction round-trip times when enable streaming queue on persistent workload.



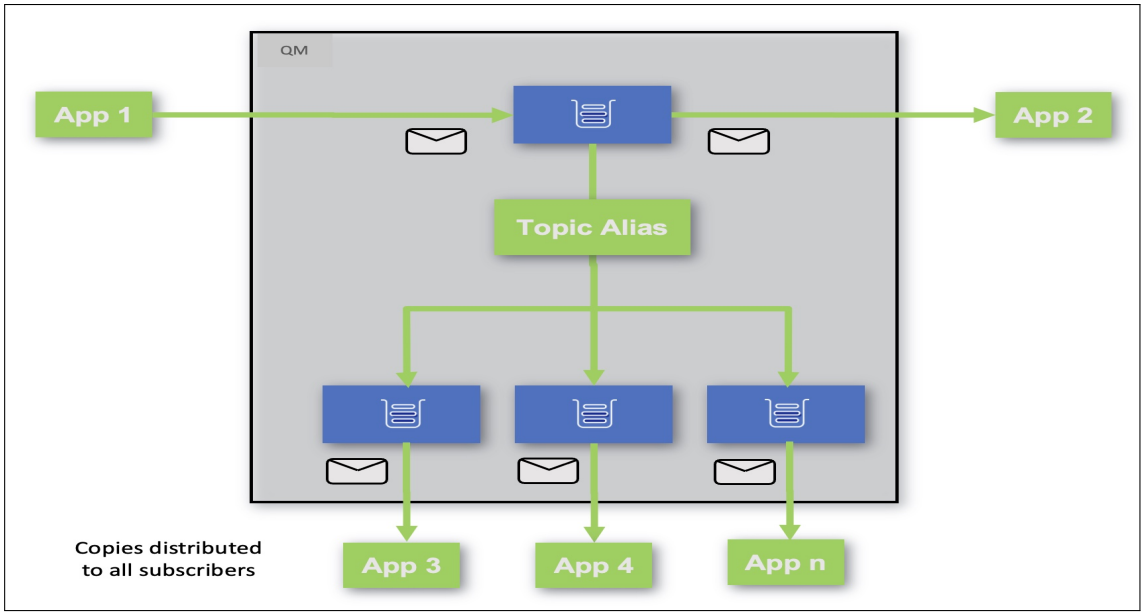
Notes on chart:

- For messages up to 32KB, the round-trip times increase by less than 10% as despite the log task reported 100% utilisation, there were low numbers of pages per I/O.
- For 64KB messages, the round-trip times have increased by 22% even though the overall log rate has increased by 56%
- For 1-4MB messages the round-trip times have nearly doubled due to the log task being fully utilised even without log streaming being enabled.

In this example, we see the overall log rate has increased as a result of enabling streaming queues, but the transaction rate has decreased due to log limits.

Streaming queue with multiple copies

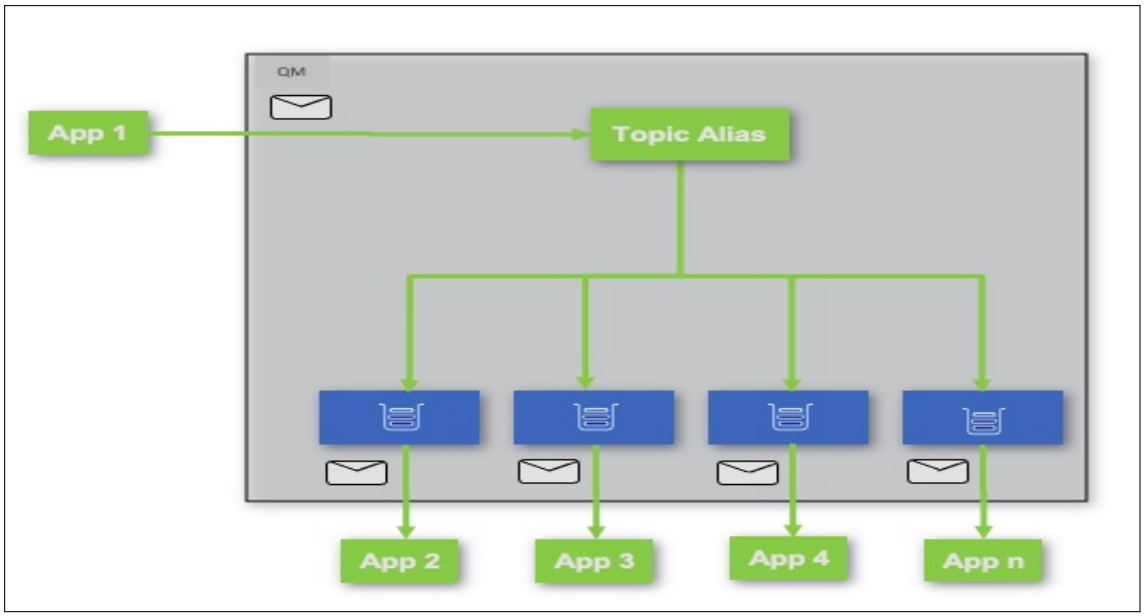
Diagram: Topology for 1 to n duplicate messages using streaming queues with Pub/Sub.



The above diagram shows how multiple duplicates of messages being put onto a queue can be generated by setting the `STREAMQ` attribute to a topic alias and using Pub/Sub to distribute messages to multiple subscribers. The advantages of this solution is that App 2 does not need to be switched to a new subscriber queue and the existing applications do not need to be stopped to make the necessary queue manager changes.

The following diagram shows the topology of using Pub/Sub to generate multiple messages. If this approach is used to create duplicate messages for an existing application, the App 2 needs to be changed to consume from a subscription, rather than getting from the original queue.

Diagram: Topology for 1 to n duplicate messages using Pub/Sub.



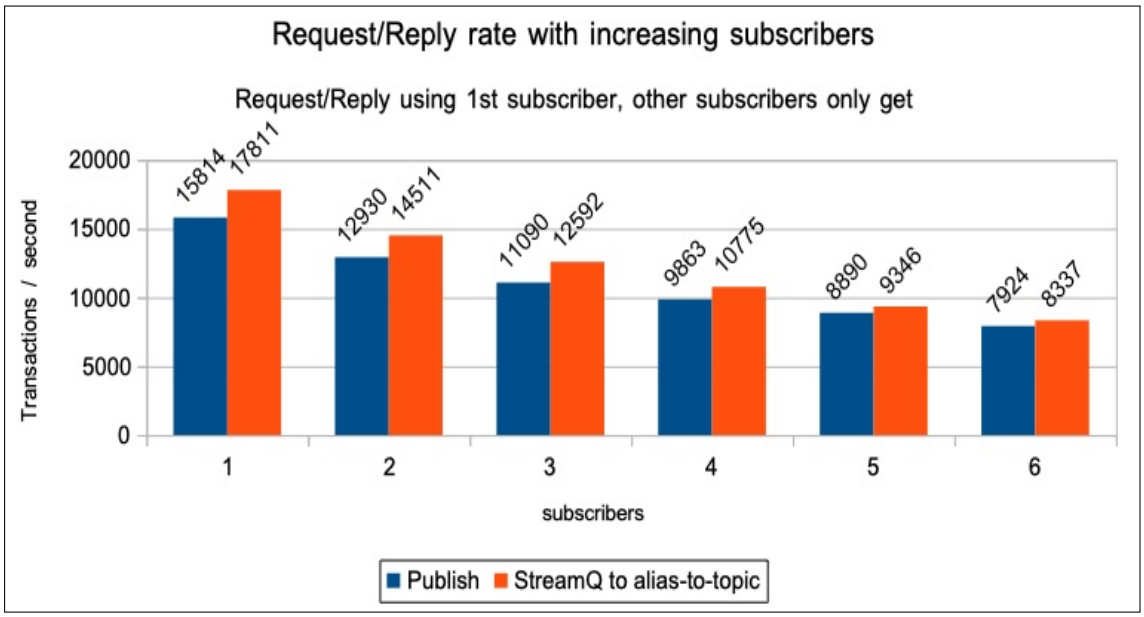
In this section, the performance of using streaming queues configured with the `STREAMQ` attribute referencing a topic alias where there are increasing numbers of subscribers is compared to using Pub/Sub to generate multiple messages.

The Pub/Sub tests are configured with 1 to 6 subscribers, whereas the streaming queue tests are configured with 0 to 5 subscribers in addition to the original getting application. In both configurations the test ends with 6 applications getting messages from a single putting application.

In each case, App 1 puts a message and waits for a reply from App 2. The remaining applications are configured to only get the published messages and ensure that queue depths remain low.

Workload uses 1KB non-persistent messages.

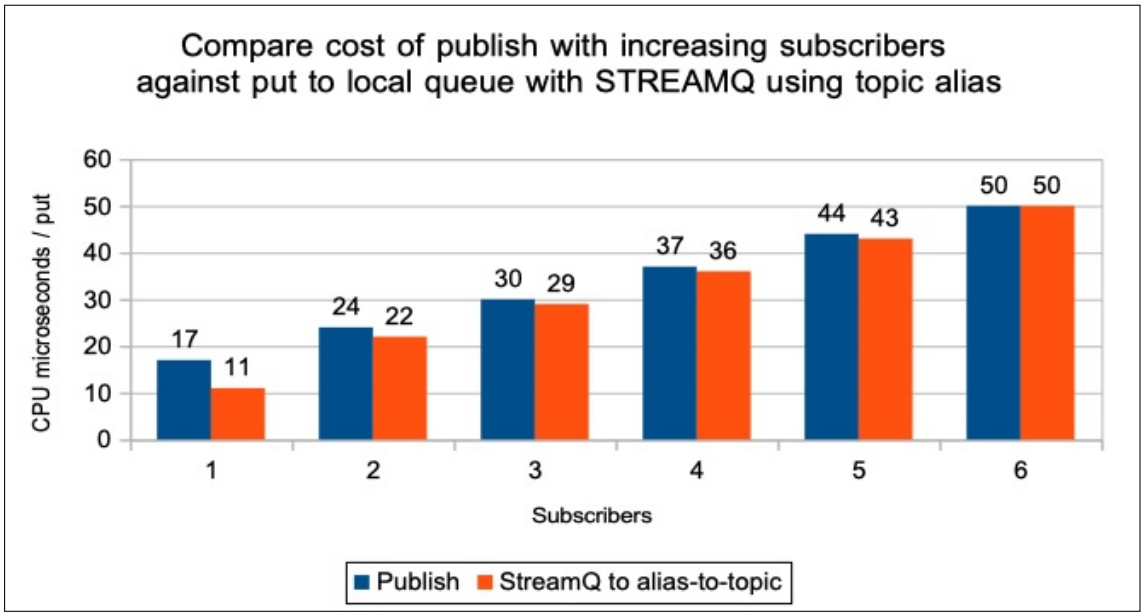
Chart: Compare performance of STREAMQ to topic alias with increasing subscribers against performance of Pub/Sub to topic alias.



Notes on chart:

- The achieved transaction rate for the request/reply workload using the `STREAMQ` configuration is between 5 to 13% higher than the equivalent Pub/Sub configuration.
- The throughput is limited by round-trip times, rather than CPU.
- For applications that were only getting messages, queue depths remained low.

Chart: Compare cost of STREAMQ to topic alias with increasing subscribers against performance of Pub/Sub to topic alias.



Notes on chart:

- Costs shown the MQPUT costs of the driving application. Costs of the server applications, whether just getting messages or getting and putting messages are comparable across configurations.
- Costs shown are as reported by Accounting Class(3) and include CPU time and Additional SRB time, which is used by the Pub/Sub processing.
- For streaming queue to topic alias with 1 subscriber measurement, the topic has 0 subscribers and as a result the cost os the Pub/Sub processing is much less than when there is at least one subscriber.
- Costs are relatively similar between the two configurations except when there are 0 subscribers in the `STREAMQ` configuration.

Pub/Sub and Accounting Class(3) data

When using `STREAMQ` to reference a topic alias whether there are **0, 1 or many subscribers**, there is additional cost of Pub/Sub that may affect the cost of the MQOPEN, MQPUT, MQPUT1 and MQCLOSE.

This cost is reported in Accounting Class(3)'s WQSTAT fields `TOPICOPENSRB`, `TOPICPutSRB`, `TOPICPut1SRB` and `TOPICCLOSESRB`.

Accounting class(3) data reports the number of messages published using the `WQSTATS PublishedN` field.

In a Pub/Sub environment, this value can be divided by the number of puts to give an indication of the number of subscribers over the interval.

For example, where Pub/Sub is configured with 2 subscribers, the put count would equal 1 and the publish count would equal 2 - and a total of 2 messages would be put to queues.

In a streaming queue environment where the `STREAMQ` references a topic alias, the published count reports the number of published messages and does not include the original put. In this environment where there are 2 subscribers, the put count would equal 1 and the publish count would equal 2, but in this instance **3** messages would have been put to queues.

Chapter 7

Increased channel initiator capacity for SVRCONN channels

64-bit private storage for SVRCONN channels

The channel initiator has traditionally been limited to 31-bit storage and this was used for many purposes including task storage (adapters, dispatchers etc), cluster cache and buffer pools. The exception to this was the storage used when class(4) accounting and statistics trace were enabled, but this is of the order of 256MB.

Prior to IBM MQ for z/OS 9.4, the number of SVRCONNs that a channel initiator could concurrently support was limited by 2 factors - the value of the channel initiator attribute MAXCHLS and the amount of storage used by each channel.

The amount of storage used by each SVRCONN channel will depend on the message size, but in previous releases of MQ, this storage was allocated from what remains of the 2GB's of 31-bit storage that the channel initiator is able to access.

Whilst the MQ channel initiator has had the concept of buffer pools for many releases, and this is discussed in MP16 "Capacity Planning and Tuning Guide", the capacity of prior releases of MQ were significantly limited when larger messages flowed over SVRCONN channels.

In IBM MQ for z/OS 9.4, SVRCONN connections use 64-bit private storage for message buffers when transferring messages larger than 32KB in size, which means that the size of the data flowing over the channel has a much smaller impact on the channel initiators' limited 31-bit capacity.

It is important to note:

1. Some 31-bit storage is still required for SVRCONN channels
2. MAXCHLS still is limited to 9,999.
3. MCA channels, for example types sender or receiver, are not affected by these 64-bit changes.

Limiting memory usage

The IBM Documentation Channel Initiator storage configuration from IBM MQ 9.4.0 suggests that the MEMLIMIT parameter may be used in the channel initiator JCL to define the maximum amount of 64-bit storage available, where 2 GB of storage, i.e. `MEMLIMIT=2G` is the minimum value you should use. Depending on your configuration, significantly more may be required.

There are alternatives for limiting the amount of 64-bit storage is available to the channel initiator address space and these include:

- IEFUSI - Step initiation exit, can be used to set default limits for region and 64-bit storage.
- SMFPRMxx parameters - System Management Facilities (SMF) parameters, allows the setting of MEMLIMIT system-wide.
- SMFLIMxx (specifying region and memlimit values) - System Management Facilities (SMF) specifying region and MEMLIMIT values, allowing these values to be set for a specific job or set of jobs, steps or program names.

Monitoring storage usage

The channel initiator provides a number of messages to the console to inform the user of how much storage is being used.

The **CSQX004I** message has been updated in IBM MQ for z/OS 9.4 to provide additional information on 64-bit private storage usage, for example:

```
CSQX004I CSQXSPRM Channel initiator storage usage:  local storage:  used 56MB, free
1445MB: above bar:  used 1MB, free 1023MB
```

In the CSQX004I example, the channel initiator has 64-bit storage restricted to 1GB using `MEMLIMIT=1G`, and at the point the message is written has only used 1MB of that storage.

Periodically and when 64-bit storage usage approaches the limit of what is available, the channel initiator will drive an internal process to free some of the used storage. The results of this process are logged using the `CSQX068I` message, and may take the form of:

```
CSQX068I CSQXBP64 Channel initiator has scavenged 429 MB of transmission buffers
```

If there is insufficient 64-bit storage available, despite the channel initiators' attempts to free unused storage, the **CSQX027E** message may be written to the console. This takes the form of:

```
CSQX027E CSQXRESP Unable to get storage, RC=00000000
```

In this instance, channels attempting to use the storage that resulted in the **CSQX027E** message being logged may be terminated with a `CSQX599E channel ended abnormally` message.

MQ Class(4) Statistics

The channel initiator statistics records do offer an indication of the storage usage at the end of SMF interval.

For example, formatting the SMF data using MP1B "Interpreting accounting and statistics data" and reviewing the CHINIT report we can see the following data:

```
QSG............PRF3
Peak number used of current channels...........  13
Peak number used of active channels ...........  13
MAXCHL. Max allowed current channels...........  9999
ACTCHL. Max allowed active channels............  9999
TCPCHL. Max allowed TCP/IP channels............  9999
LU62CHL. Max allowed LU62 channels.............  40
31-bit storage used............................  56 MB
64-bit storage limit...........................  1024 MB
64-bit storage used............................  486 MB
64-bit storage free............................  538 MB
```

In the above data, the channel initiator has been restricted to 1GB of 64-bit private storage and at the end of this interval has used 486MB of the 1024MB available.

Note: The units reported e.g. MB, will change depending on how much space is used/free.

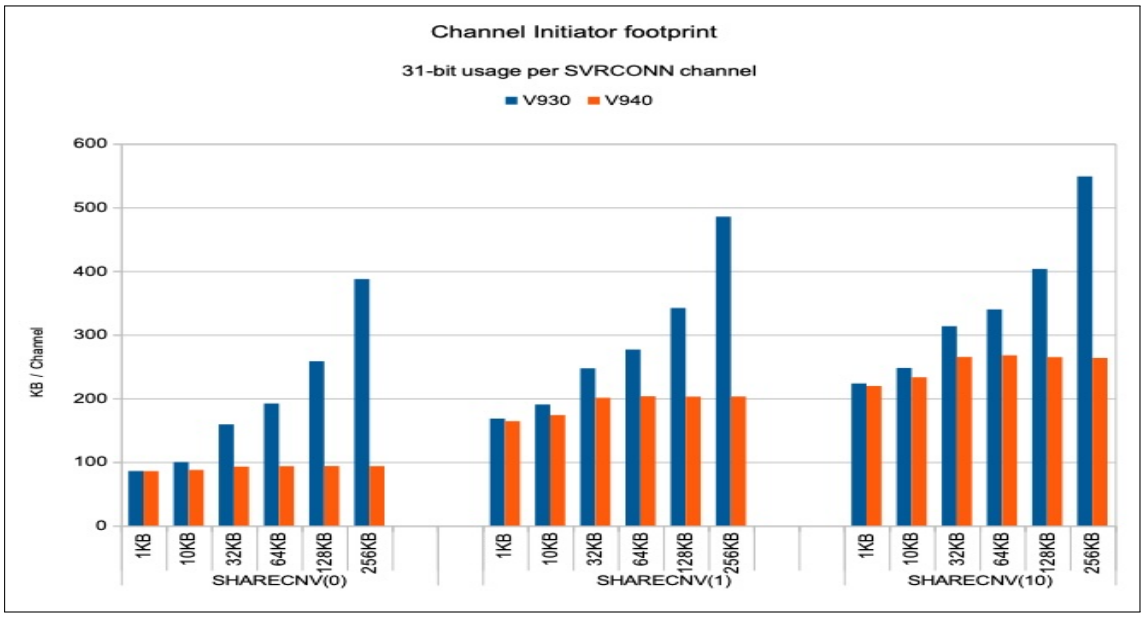
How much storage is used by a SVRCONN channel?

The table in the "How many clients can I connect to my queue manager?" section discussed the 31-bit storage usage per SVRCONN channel and showed that for messages less than 32KB, the storage used was similar between MQ releases 9.2, 9.3 and 9.4. For messages of 32KB or larger, the MQ 9.4 31-bit storage usage was less than earlier MQ releases.

Since the storage usage for SVRCONN channels is similar in MQ 9.2 and 9.3, we will limit the comparison of storage requirements to MQ 9.3 and MQ 9.4.

The following chart shows the 31-bit storage used for a range of message sizes for SHARECNV values of 0, 1 and 10, such that when using MQSC command `DISPLAY CHSTATUS(*) CURSHCNV`, the CURSHCNV output shows a value that matches the SHARECNV attribute.

Chart: 31-bit storage usage for SVRCONN channels

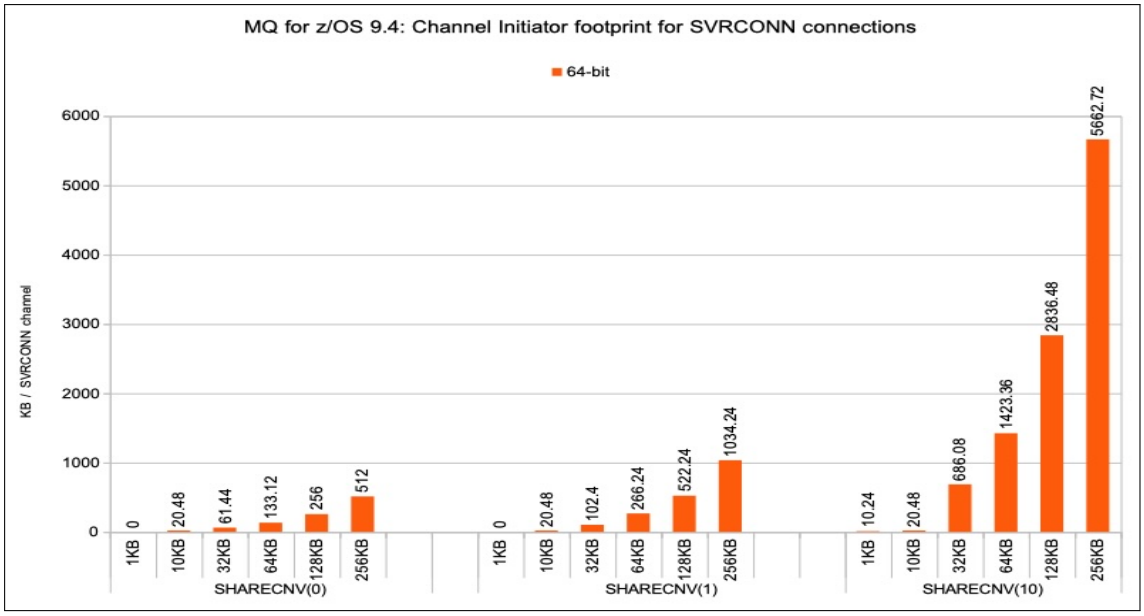


Notes on chart:

- For IBM MQ for z/OS 9.3, the 31-bit storage usage requirements increase with both message size and the value of SHARECNV.
- For IBM MQ for z/OS 9.4, the 31-bit storage usage increases for all for message sizes up to 32KB and then levels off, meaning that the same amount of 31-bit storage is used in the channel initiator regardless of whether the message is 32KB or 100MB.
- Maximum 31-bit storage usage for a SVRCONN channel
  - SHARECNV(0): Approximately 94KB
  - SHARECNV(1): Approximately 204KB
  - SHARECNV(10): Approximately 264KB
  - These calculations are based on simple SVRCONN channels using the same size message across all conversations in the channel.

The following chart provides an indication of how much 64-bit private storage might be required for SHARECNV values of 0, 1 and 10.

Chart: 64-bit storage usage for MQ 9.4 SVRCONN channels



Notes on chart:

- The 64-bit private storage is assigned in a range of pool sizes from 4KB doubling in size up to 1MB. For messages larger than 1MB, storage is allocated on a per MB requirement.
- In the above chart, we see relatively high 64-bit storage requirements. For example
  - SHARECNV(0): Approximately 2 times the message size.
  - SHARECNV(1): Approximately 4 times the message size.
  - SHARECNV(10): Approximately 22 times the message size per channel, or 2.2 times the message size per conversation.
- It is worth noting that the message sizes used in the chart are not optimised for the 64-bit pool sizes used in the channel initiator, such that for each of the messages used of 32KB or more, storage was used from the next sized-buffer pool, i.e. a 32KB message required a 64KB buffer. This is simply because the storage requirements includes message headers, which takes the message size over the lower pool-size threshold.
  - With a SHARECNV(0) channel using a 31KB message, we would use a single 32KB buffer, reducing the 64-bit private storage usage by half.
- Additional storage is required for channels with non-zero SHARECNV values, and the storage requirements for channels with SHARECNV(1) are relatively high compared to SVRCONN channels with SHARECNV of greater than 1 and matching CURSHCNV.

Rule of Thumb

For all SVRCONN channel conversations, allow space for the message header(s), approximately 500 bytes, and round-up to the nearest buffer pool size (using a power of 2) e.g. 4KB, 8KB, 16KB, etc.

- For SVRCONN channels with SHARECNV(0), this gives an approximate 64-bit storage requirement per channel.
- For SVRCONN channels with a non-zero SHARECNV value but only a single conversation i.e. CURSHCNV(1), double the calculated space to determine an approximate 64-bit storage requirement.
- For SVRCONN channels where multiple conversations are sharing a single SVRCONN channel, i.e. SHARECNV of greater than 1, and CURSHCNV of greater than 1, multiply the size by the number of conversations and then again by 1.1 to determine an approximate 64-bit storage requirement.

How many SVRCONN channels can be run?

In IBM MQ for z/OS 9.4, the factors affecting the number of SVRCONN channels that can be run concurrently now include the amount of 64-bit private storage allowed by the channel initiator, i.e. the value of MEMLIMIT.

Provided a system has sufficient 64-bit storage available, the limit to the number of SVRCONN channels may still be limited by the factors listed in the introduction to this section.

The following example, that was measured in our controlled system, uses `MEMLIMIT=NOLIMIT` but this is generally not recommended as there is the potential to use up all available z/OS virtual storage, leading to paging in your system.

In this environment, the maximum number of clients supported primarily depends on the amount of 31-bit storage available.

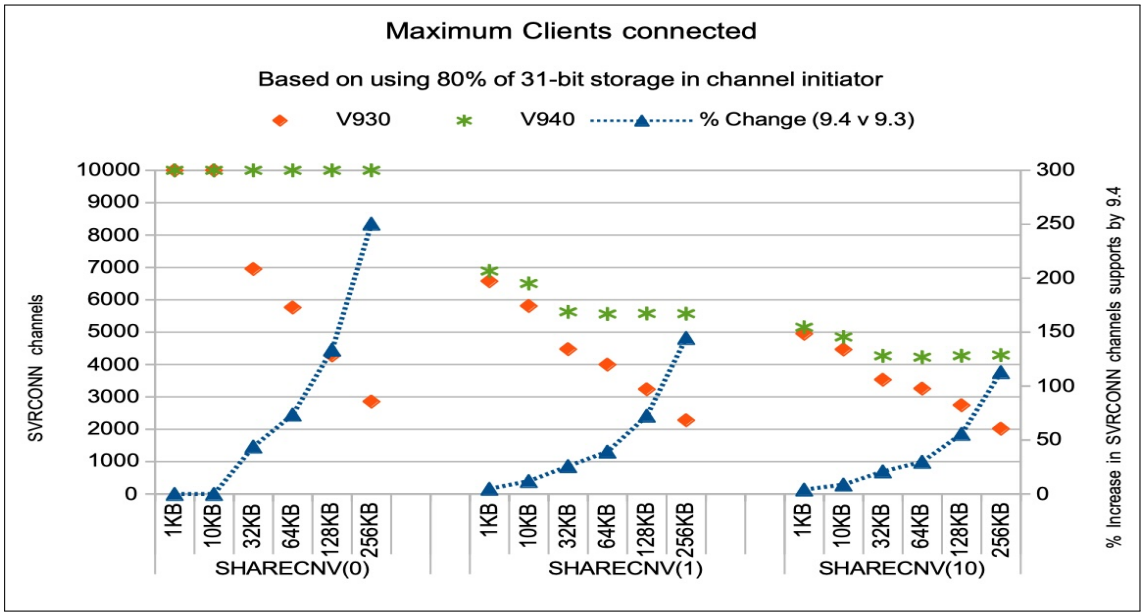
The `CSQX004I` message reports how much storage is available, for example:

```
CSQX004I CSQXSPRM Channel initiator storage usage:  local storage:  used 56MB, free
1445MB: above bar:  used 1MB, free 1023MB
```

Typically we would advise that allowing no more than 80% of the available local storage.

In the above example this equates to 80% of 1501MB, or 1200MB.

Chart: Maximum clients connected without exceeding 80% of 31-bit storage



Notes on chart:

- SHARECNV(0):
  - Both MQ 9.3 and MQ 9.4 were able to support the maximum number of channels (9,999) for messages up to 10KB.
  - For 32KB messages, MQ 9.3 could support 7,000 SVRCONN channels but MQ 9.4 now can support the maximum number of channels.
  - For 256KB messages, MQ 9.3 could support less than 3,000 SVRCONN channels but MQ 9.4 can support the maximum number of channels. This equates to a 250% increase in capacity in IBM MQ for z/OS 9.4's channel initiator capacity.

    - As can be seen in the 31-bit storage usage for SVRCONN channels chart, MQ 9.4's 31-bit usage does not significantly increase with message size, and as a result a channel initiator with 1375MB or more local storage available, with sufficient 64-bit storage, could support 9,999 SVRCONN channels with 100MB messages.

- SHARECNV(1):
    - For messages less than 32KB, there is little difference in the capacity of MQ 9.3 and MQ 9.4.
    - MQ 9.3: For messages of 32KB or larger, the number of channels that can be run decreases with message size, such that for a 32KB message 4,500 SVRCONNs could be running but with a 256KB message the limit drops to 2,275 SVRCONN channels.
    - MQ 9.4: For messages of 32KB or larger, the number of channels that can be run remains relatively flat such that for a 32KB message 5,600 SVRCONNs could be running and with a 256KB message the limit is 5,570 SVRCONN channels.
    - For a 256KB message, this equates to MQ 9.4 having 145% more capacity than the equivalent MQ 9.3 channel initiator.
- SHARECNV(10):
    - For messages less than 32KB, there is little difference in the capacity of MQ 9.3 and MQ 9.4.
    - MQ 9.3: For messages of 32KB or larger, the number of channels that can be run decreases with message size, such that for a 32KB message 3,530 SVRCONNs could be running but with a 256KB message the limit drops to 2,010 SVRCONN channels.
    - MQ 9.4: For messages of 32KB or larger, the number of channels that can be run remains relatively flat such that regardless of message size 4,270 SVRCONNs could be running concurrently.
    - For a 256KB message, this equates to MQ 9.4 having 113% more capacity than the equivalent MQ 9.3 channel initiator.

It is also worth noting that these workloads were running with 1 message per conversation per second. With a slower messaging rate, more buffer pool reuse may occur, and more clients may be connectable. Monitor your channel initiator usage patterns.

Performance

This section looks at any impact to MQ connect, as well as MQPUT and MQGET as a result of using 64-bit private storage for SVRCONN channels.

MQ Connect performance in MQ 9.4 shows no significant difference in cost or response time when compared against earlier MQ releases.

MQ Put performance in MQ 9.4 shows some reduction in cost for message sizes 32KB to 100KB. For 32KB non-persistent messages, we observed cost reductions of up to 31% on MQ 9.4.

MQ Get performance in MQ 9.4 shows no significant difference in cost or elapsed time when compared against previous MQ releases.

Chapter 8

Kafka Connectors with exactly-once delivery

IBM MQ Advanced for z/OS 9.4 only

Introduction

IBM Event Streams is an event streaming platform built on open source Apache Kafka®.

Kafka connectors allow data to be passed between MQ subsystems and Apache cluster(s).

The IBM MQ source connector enables the copying of data from IBM MQ into the Apache cluster, i.e. MQ is the source of the data.

To copy data from Kafka into IBM MQ, use the IBM MQ sink connector, i.e. Kafka is the source and IBM MQ is the target or the "sink".

IBM MQ Advanced for z/OS 9.4 introduces support for exactly-once delivery in the form of v2 connectors. These v2 connectors are only supported with the IBM Event Streams platform.

The IBM MQ v1 connectors provide at-least-once message delivery by default. For the IBM MQ sink connector, this means that each Kafka message is delivered to IBM MQ, but in failure scenarios it is possible to have duplicated messages delivered to IBM MQ.

The IBM MQ v2 connectors offer exactly-once message delivery semantics. An additional IBM MQ queue is used to store the state of message deliveries. When exactly-once delivery is enabled, Kafka messages are delivered to IBM MQ with no duplicated messages.

What is Kafka?

Apache Kafka is an event streaming platform, that combines three key capabilities, namely to publish (write) and subscribe (read) streams of events, to store streams of events durably and reliably and to process streams of events as they occur or retrospectively.

Some of the differences between Kafka and a typical MQ messaging system are:

- Producers of the messages publish to topics.
- Consumers subscribe to the topics.
- Messages are arrays of bytes and can be JSON objects, Strings etc.
- Topics are logs of messages.
- Kafka is run as a cluster of servers, each of which is called a broker.

There are many overlaps between IBM MQ and Apache Kafka with each having their own specific benefits and the Kafka connectors provide a bridge between the two products.

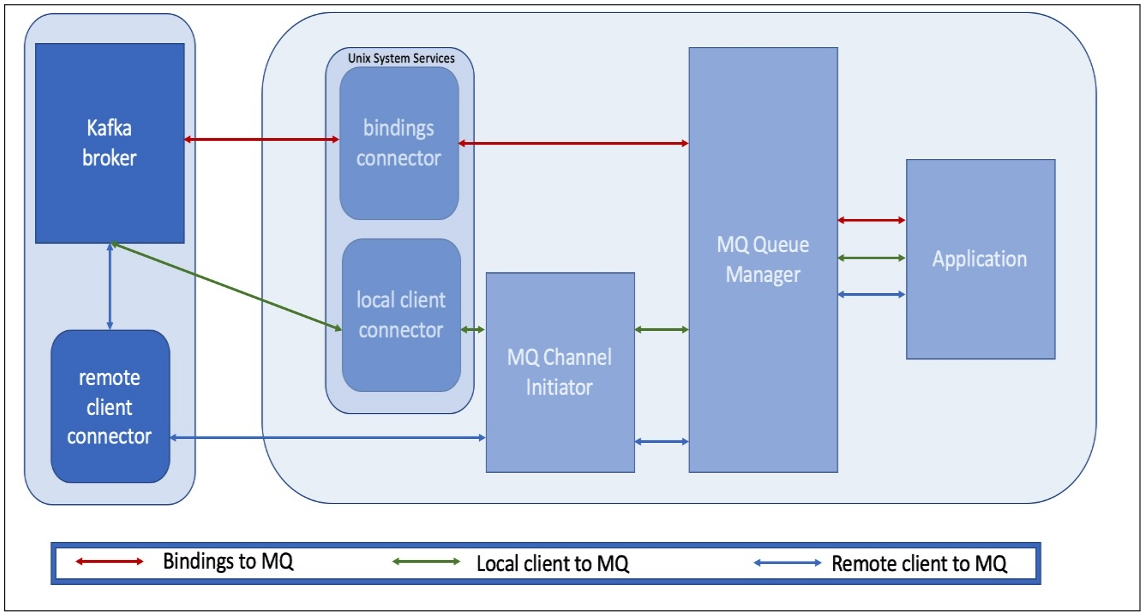
What configurations can be used with Kafka connectors?

The Kafka connectors may be run in a number of configurations, and do not have to be run locally to the Kafka broker, so there are three configurations that may be used with IBM MQ for z/OS:

1. Connectors local to Kafka but remote from z/OS, connecting as a client via a SVRCONN channel to the z/OS queue queue manager.
2. Connectors local to z/OS, running in the Unix System Services (USS) environment, connecting as a client via a SVRCONN channel to the z/OS queue queue manager.
3. Connectors local to z/OS, located in the USS environment, connecting via bindings to the z/OS queue manager.

These configurations can be represented thus:

Chart: Kafka connector configurations

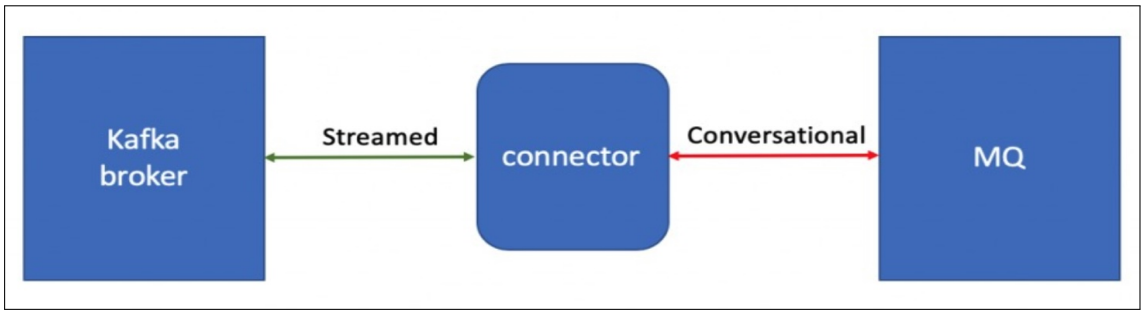


In earlier measurements, for example as discussed in the blog Kafka Connectors for IBM MQ: an IBM MQ for z/OS perspective, we determined that the configuration offering the best performance in our environment was the bindings mode configuration as this allowed us to achieve the best results in terms of lowest cost on z/OS and potentially the highest throughput.

As the Kafka connectors are written in Java™, when running in the USS environment, the connectors are largely eligible for offload to zIIP processors.

Does the location of the connector matter?

The simple answer is yes, but to explain why that is the case, consider the following diagram:



Data flows from Kafka Connectors to MQ and Kafka.

This consists of two sets of flows:

1. Kafka broker to/from connector - this flow is largely a uni-directional stream.
2. Connector to/from IBM MQ - this is a pseudo-conversational flow.

For the connector to/from IBM MQ running in a conversational mode, there is more impact from network latency as the connector must wait for a response from MQ before performing the next action.

Also consider that TCP/IP is optimised for streaming-type workloads, as TCP is able to use features such as dynamic right sizing to increase the size of the send windows, to send data more efficiently.

Finally, the data flowing between MQ and the connector is typically larger as it contains MQ's implementation headers - so a 1KB message payload may result in 1.5KB being transferred, i.e. more data is transferred between the MQ and connector.

As a result, we would typically suggest the following order in terms of preference to be able to achieve the best throughput from your Kafka connector:

1. Connector in Unix System Services using binding-type connections to MQ, which has the added advantage of bypassing the MQ channel initiator.
2. Connector in Unix System Services using client-type connections to MQ.
3. Connector remote from MQ and using client-type connections.

When does Kafka commit?

Unlike MQ which uses explicit commit processing, Kafka uses an auto-commit approach by default. For example a consumer might commit based a period of time having passed.

This can lead to different commit points when transferring data between IBM MQ and Kafka using connectors and this is part of the problem that the v2 connectors are fixing.

In the blog Kafka Connectors for IBM MQ: an IBM MQ for z/OS perspective, I referred to this as Kafka having "lazy" commits, such that when sending or receiving messages from IBM MQ, the connector does not issue the commit for each message, rather it issues the commit periodically. In some instances with workloads that had bursts of activity interspersed with idle periods resulted in messages not being committed for a period of minutes, or until the net burst of work occurred.

Performance of Kafka connectors

For the purposes of the Kafka connector performance, this section will concentrate on the preferred bindings-type configuration.

In this section, we will look at the performance of the Kafka MQ connectors, both the v1 and v2 (exactly-once) connector configuration.

For the connector performance, there will be data showing the peak rate the connector is able to process data on our systems and a sustained rate, where our environment is able to run for extended periods without queue depths building up from un-processed messages.

Additionally the section will consider the cost of the connectors running in a Unix System Service environment when there are no messages flowing.

Kafka / MQ performance environment

For the purposes of the measurements in this section, there is a single Kafka broker defined on an IBM z15 running RedHat Enterprise Linux 8.9, with 128 GB of memory and 6 CPUs configured with 2 threads per core.

The IBM z16 z/OS LPAR is configured with 3 dedicated general purpose CPUs and 2 zIIP configured with `SMT=1`, i.e. single-threaded mode.

The Kafka broker has two topics, each with 5 partitions defined:

- TO_MQ
- FROM_MQ

These topics have corresponding queues on the MQ queue manager defined as following:

```
DEFINE STGCLASS(KAFKA) PSID(3)

DEF QL(KAFKA_MQ) SHARE DEFSOPT(SHARED) STGCLASS(KAFKA) +
INDXTYPE(CORRELID) MAXMSGL(100000001) REPLACE

DEF QL(MQ_KAFKA) SHARE DEFSOPT(SHARED) STGCLASS(KAFKA) +
INDXTYPE(CORRELID) MAXMSGL(100000001) REPLACE
```

The Kafka and MQ definitions are then connected using the Kafka Connector properties, for example:

Kafka MQ sink connector

```
topics=TO_MQ
mq.queue=KAFKA_MQ
mq.queue.manager=VTS1
mq.connection.mode=bindings
```

Does message persistence affect throughput?

The performance of the Kafka MQ connectors with persistent messages can be significantly affected by many factors including:

- The number of messages put or gotten per MQ commit.
- What other workloads are being logged by the MQ queue manager.
- The I/O response times.

Impact of message size on throughput

The primary impact of message size for non-persistent messages is the network performance (capacity and latency.

In our measurements, the **IBM MQ sink connector** was able to sustain the following rates using 1 connector task:

Message Size	Messages / Second	Approximate MB / Second
1KB	**28,298**	27.6
10KB	**8,000**	78
32KB	**4,580**	143

The **IBM MQ source connector** was able to sustain the following rates using 1 connector task:

Message Size	Messages / Second	Approximate MB / Second
1KB	**28,754**	28.1
10KB	**5,581**	54.5
32KB	**3,751**	117

Performance of v1 connector

This section shows the performance (throughput) of the Kafka MQ connectors running without the exactly-once delivery options enabled.

For this configuration, the use of non-persistent messaging can help to improve the throughput rates achieved as this can help remove any impact from MQ log response times.

IBM MQ sink connector - Sustained rate

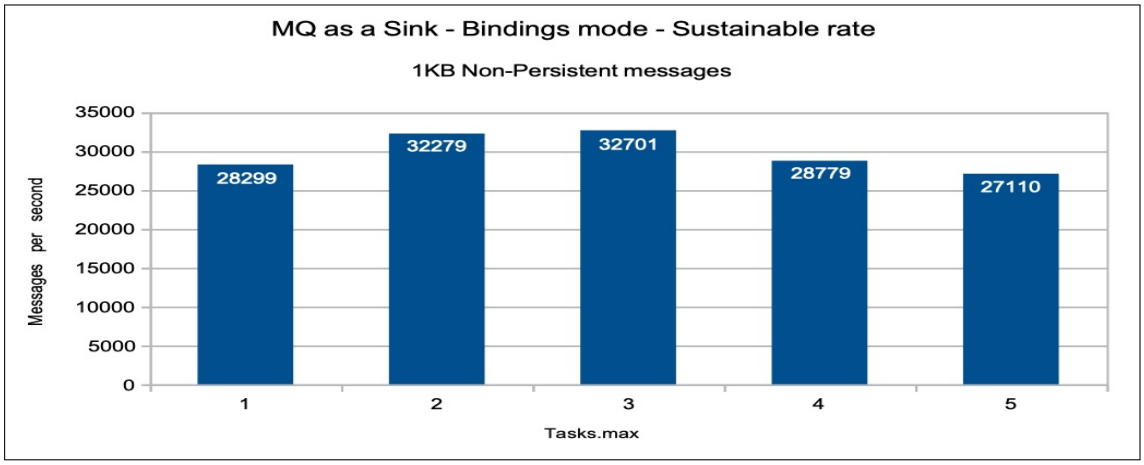
Kafka sink connectors only support 1 connector task per partition. Typically you may have multiple partitions, so it seems sensible to have *at least* the same number of connector tasks for IBM MQ sink connector as the number of partitions. It may be beneficial to have larger numbers of connector tasks in the event of more partitions being configured.

When using the IBM MQ sink connector, the messages are generated by the Kafka "kafka-producer-perf-test.sh" shell script. The messages are consumed on z/OS by a small number of batch applications that get the messages and discard them.

In our environment, the Kafka connector can get 28,300 non-persistent 1K B messages per second from the Kafka topic and put to the MQ queue when configured with a single connector task, i.e. `tasks.max=1`.

Adding a second connector task, using `tasks.max=2`, increases the sustainable workload to 32,280 messages per second, which is a 14% increase. In our environment, further connector tasks did not provide further benefit.

Chart: MQ as a sink connector - sustainable rate



With regard to the cost of running the IBM MQ source connector in the Unix System Services (USS) environment, a large proportion of the cost can be offloaded by using zIIPs.

For example, typically we observed a cost per message of 35 to 44 CPU microseconds in the Kafka connector for 1KB non-persistent messages, of which up to **98%** was zIIP-eligible.

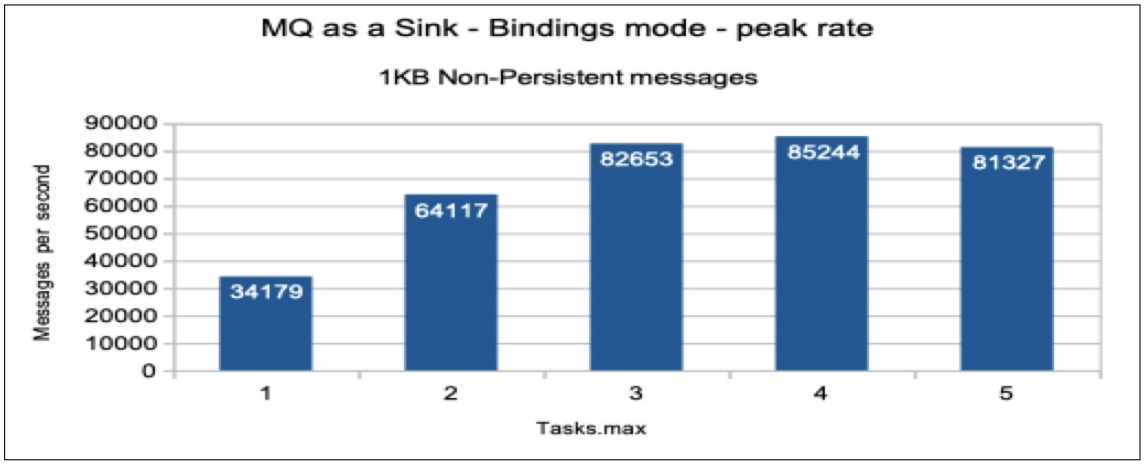
Additionally, the MQ class(3) accounting data reports the cost of the MQPUT1 as 13 to 22 CPU microseconds.

IBM MQ sink connector - Peak rate

The peak rate achieved when using the IBM MQ sink connector uses a different model to the sustained rate measurements. In this configuration, the messages are put to the MQ queue but no applications are getting the message. This avoids any contention on the queue that would occur when applications were to get the messages, but it does mean that the workloads are limited to the amount of space in the MQ buffer pool and page set.

In our measurements, the IBM MQ sink connector configuration can benefit from delaying the message consumption on IBM MQ, such that the peak rate with a single task increased by 21% and a peak rate 2.6 times higher than the sustained rate.

Chart: MQ as a sink connector - peak rate



IBM MQ source connector - Sustained rate

When using the IBM MQ source connector, the messages are put by a small set of simple applications running in batch.

The connector task(s) will then get the messages from the queue and stream them to the Kafka cluster.

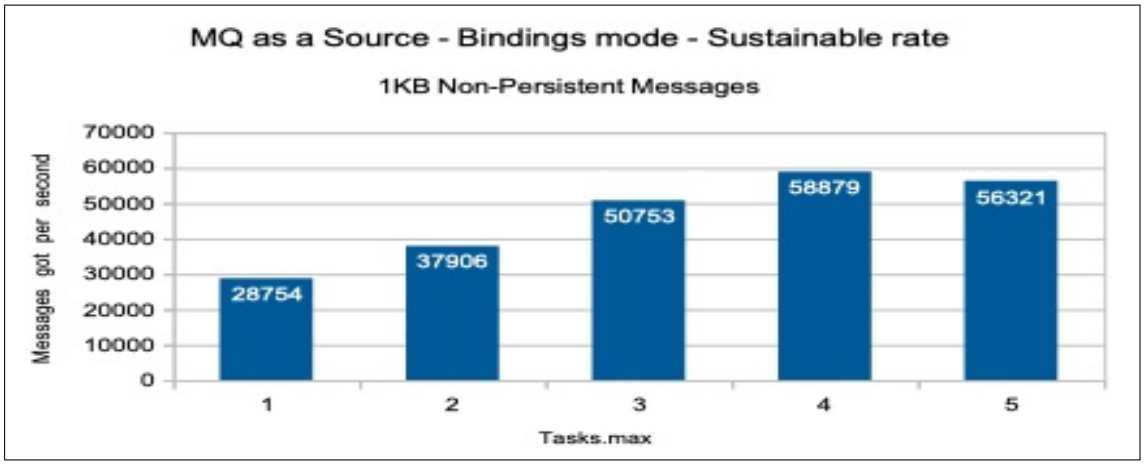
For the purposes of these sustainable tests, the connector can consume messages at a rate such that the queue depth does not increase significantly.

The test uses multiple instances of Kafka's kafka-consumer-perf-test.sh shell script to consume the messages when they arrive on the Kafka topic.

In our environment, the Kafka connector can get 28,750 non-persistent 1KB messages per second from the MQ queue when configured with a single connector task, i.e. `tasks.max=1`.

Adding a second connector task, using `tasks.max=2`, increases the sustainable workload to 37,900 messages per second, which is a 32% increase. Further connector tasks did continue to improve the sustainable MQGET rate up to a peak rate of 58,880 messages per second.

Chart: MQ as a source connector - sustainable rate



As with the IBM MQ sink connector in the USS environment, a large proportion of the cost can be offloaded by using zIIPs. In our environment, we observed a cost per message of 25 to 37 CPU microseconds of which up to 96% was zIIP-eligible.

The MQ class(3) accounting data reports the cost of the MQGET as 4 to 6 CPU microseconds.

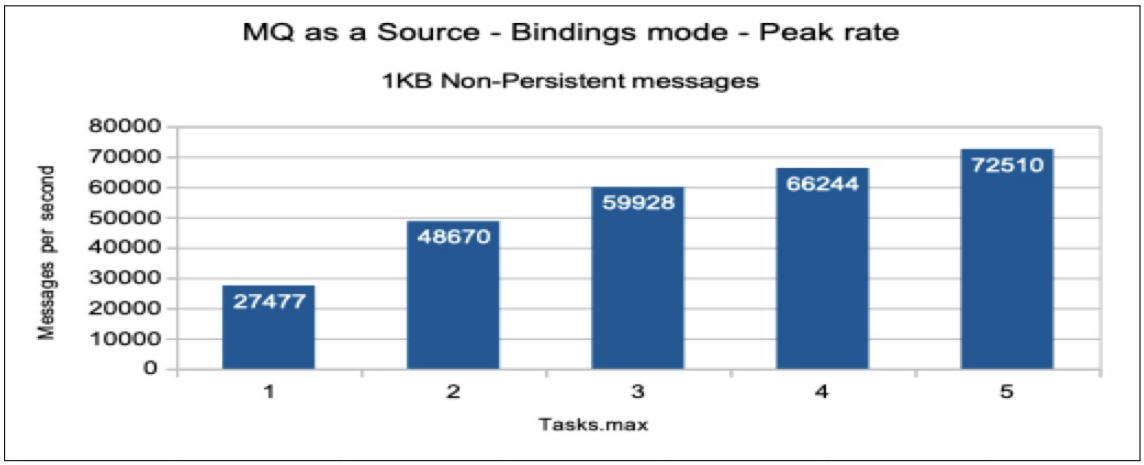
IBM MQ source connector - Peak rate

For absolute peak rate throughput measurements using the IBM MQ source connector, the tests run such that the queue is pre-loaded prior to the connector being started, and no further messages are put to the queue whilst the connector is active.

By pre-loading the queue, the workloads are designed to remove as much contention as possible and allow the connector to run at peak capacity. This does mean that the workloads are limited in size due to the amount of space in the MQ buffer pool and may mean that the Kafka connector is not fully warmed up for the entire measurement.

In our measurements, the peak rate achieved by the IBM MQ source connector increased by 23% from 58,800 to 72,500 messages per second by removing any contention due to MQPUTs running concurrently with the MQGETs requested by the connector.

Chart: MQ as a source connector - peak rate



Performance of v2 connector

Whilst the "performance of v1 connector" section reports on both the peak rate and the sustainable rates, there is a small impact to performance when using the v2 connectors, which is most notable in the peak rate configuration.

Therefore when looking at the performance of the v2 connectors, only peak message rates will be discussed.

The v2 connectors support exactly-once message delivery, both for the source and the sink connectors. This is achieved by storing state messages on a separate MQ queue.

It is important to note that only a single connector task can run in the Kafka Connect instance. As a consequence the property `tasks.max=1` must be set to ensure that failure scenarios do not cause duplicated messages to be delivered.

There are a number of pre-requisites when configuring the v2 connectors which are detailed in IBM MQ source connector and IBM MQ sink connector in the "Exactly-once message delivery semantics" documentation. Rather than repeating all of those requirements, please refer to the latest documentation referenced in the above links.

In the earlier MQ definitions, the MQ resources were defined on page set 3 to a storage group specifically for Kafka use. When using the exactly-once configuration, it is necessary to define the state queue. The following example allocates a state queue for the IBM MQ source connector.

```
DEF QL(SYSTEM.KAFKA.SOURCE.STATE) DEFSOPT(EXCL) +
  STGCLASS(KAFKA) INDXTYPE(MSGID) MAXMSGL(100000001)
```

Message persistence matters with v2 connectors

The v2 connectors are designed to ensure that messages are delivered exactly one time. This must occur whether the queue manager, Kafka connector or indeed the Kafka broker fail.

In MQ, using non-persistent messages and restarting the queue manager, will result in those non-persistent messages being deleted from the queue.

The persistence of the messages put to the `STATE` queue will mirror the persistence of the messages flowing over the Kafka connectors,

In order to ensure the state is maintained across restart and/or failure, the state messages should be persistent. Indeed all messages sent through the MQ v2 connectors should be persistent and have no expiry set.

Since the true benefit of the exactly-once delivery mechanism is to be able to recover from failure or outage, the performance data in the following section will use only persistent messages.

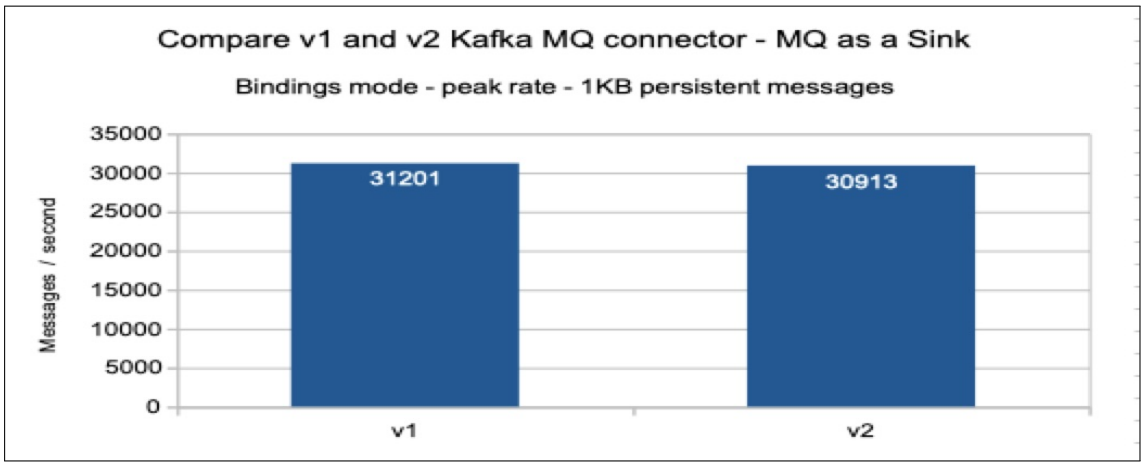
Performance

When comparing the performance of the v2 connectors with the v1 connector, there is a small impact in terms to the throughput rate. For the peak rate persistent measurements, this reduced the throughput by less than 6%.

IBM MQ sink connector

As the following chart shows, the impact on throughput is minimal for the IBM MQ sink connector peak throughput performance, where the rate achieved decreased by 1% for the v2 connector compared to the v1 connector.

Chart: Compare performance of v1 and v2 connectors with MQ as a sink connector



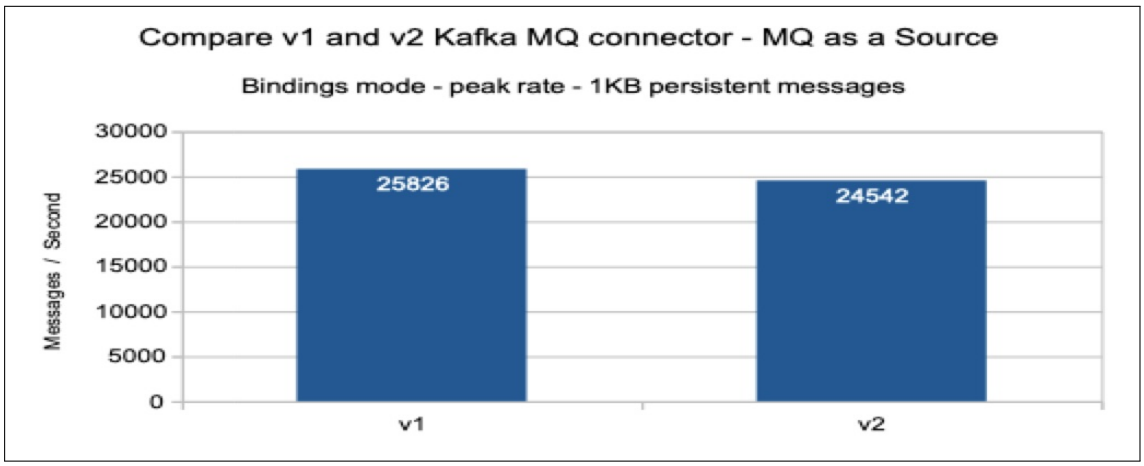
Reviewing the class(3) accounting data for the IBM MQ sink connectors, there are the following observations:

- Both v1 and v2 connectors uses MQPUT1 to put the messages to the MQ queue.
- Both v1 and v2 connectors additionally issue inquire periodically against the MQ queue. For the 1.7 million MQPUT1s issued, there were 241 inquires, which is approximately 1 inquire for every 6960 MQPUT1s.
- In these measurements, the class(3) accounting data cost for the v1-processing is 7 CPU microseconds per message put to the MQ queue.
- The v2 IBM MQ sink connector also maintains state data periodically which uses the form of MQGET and MQPUT1 access to the named state queue.
    - The impact of the state queue processing is relatively small in a high throughput rate workload as the updates occur approximately once every 440 messages put to the MQ queue.
    - The update of the state queue costs, based on class(3) accounting data, is 23 CPU microseconds cost of the MQGET and MQPUT1.
    - In our measurements, the message on the state queue is less than 500 bytes.
    - The average cost per message processed by the IBM MQ sink connector using the exactly-once processing equates to 7.05 CPU microseconds, which is the equivalent of a 0.75% increase over the v1 connector cost.
    - For these persistent workloads, the IBM MQ sink connector was 96% zIIP-eligible.

IBM MQ source connector

For the IBM MQ source connector peak throughput measurements, the rate achieved decreased by 5% for the v2 connector compared to the v1 connector performance as indicated in the following chart.

Chart: Compare performance of v1 and v2 connectors with MQ as a source connector



Reviewing the class(3) accounting data for the IBM MQ source connectors, there are the following observations:

- Both v1 and v2 connectors uses MQGET to get the messages from the MQ queue.
- Both v1 and v2 connectors additionally issue inquires periodically against the MQ queue. For the 1.7 million MQGETs issued, there were 120 inquires.
- In these measurements, the class(3) accounting data cost for the v1-processing is 5 CPU microseconds per message got from the MQ queue.
- The v2 IBM MQ source connector also maintains state data periodically which uses the form of MQGET and MQPUT1 access to the named state queue.
    - The impact of the state queue processing is relatively small in a high throughput rate workload as the updates occur approximately once every 250 messages put to the MQ queue.
    - The update of the state queue costs, based on class(3) accounting data, is 30 CPU microseconds cost of the MQGET and MQPUT1.
    - In our measurements, the message on the state queue is larger for the source connector and was up to 13.5KB
    - The average cost per message processed by the IBM MQ sink connector using the exactly-once processing equates to 5.12 CPU microseconds, which is the equivalent of a 2.4% increase over the v1 connector cost.
    - For these persistent workloads, the IBM MQ source connector was 91% zIIP-eligible.

How much does an idle connector cost?

There may be periods where the IBM MQ connectors are not processing any messages and it may be helpful to understand what costs might be incurred in those periods.

When idle, the IBM MQ sink connector does not interact with the MQ queue manager, so the cost is entirely in the connector. In terms of connector cost, we measured the cost as approximately 6.2 CPU seconds per hour when `tasks.max=1` and 6.9 CPU seconds when `tasks.max=5`. Using the v2 connector did not affect the idle cost.

The IBM MQ source connector periodically issues MQGETs on the MQ queue and these occur every 2 seconds, so in a one hour period, there would be 300 MQGETs issued by the idle source connector.

This has a small cost to both the connector and the MQ queue manager.

In terms of connector cost, the impact was approximately 2.6 CPU microseconds per hour when `tasks.max=1` and 12 CPU seconds when `tasks.max=5`. Using the v2 connector did not affect the idle cost.

The cost of processing the MQGETs from the source connector on the MQ queue manager was less than 0.3 CPU microseconds per hour for up to 5 connector tasks.

Kafka and MQ - anything else?

This section contains some observations on MQ and Kafka Connectors that have not previously been addressed in this document.

- When using client connections, i.e. those using SVRCONN channels with Kafka MQ connectors, it is necessary to ensure the SVRCONN channel is configured with a SHARECNV value greater than 0, in order to support JMS2.0 APIs.
- When using shared queues with either the IBM MQ sink connector (v1 or v2) or the IBM MQ source connector (v2) which use MQPUT1 to either the MQ queue and/or the state queue, there can be an impact of first-open/last-close effects as discussed in MP16 "Capacity Planning and Tuning Guide"section "Frequent opening of shared queues"
- There can be an impact to the IBM MQ source connector when there is a low rate of messages to transfer from MQ to the Kafka broker. Initially the connector will issue an MQGET-with-wait and when a message is successfully gotten, the connector immediately attempts another MQGET but this time without a wait period specified. When messages are put at a low rate to the queue serviced by the IBM MQ source connector such that the queue depth is consistently low, there may be additional MQ gets attempted that do not result in the successful get of a message. This can lead to class(3) accounting and class(5) queue statistics reporting multiple MQGETS per successful get.
    - When the connector is configured in a bindings environment, the cost of the unsuccessful MQGET will be low (approximately 1-2 CPU microseconds)
    - When the connector uses a client configuration to access the MQ queue, the failed MQGETs will result in additional adaptor and dispatcher calls in the MQ channel initiator address space, with more noticeable cost.

Appendix A

Regression

When performance monitoring both IBM MQ Advanced for z/OS 9.4 and IBM MQ for z/OS 9.4, a number of different categories of tests are run, which include:

- Private queue
- Shared queue
- Moving messages using MCA channels
  - SSL
  - Channel compression (ZLIBFAST / ZLIBHIGH)
  - Streaming messages
- Moving messages using cluster channels
- Client
- Bridges and adaptors
- Trace
- AMS message protection

These tests are run against version V9.2 (V920) and 9.3 (V930) and 9.4 (V940) and the comparison of the results is shown in subsequent pages.

The statement of regression is based upon these results.

All measurements were run on a performance sysplex of an IBM z16 (3931) which was configured as described in "System Configuration".

Given the complexity of the z/OS environment even in our controlled performance environment, a tolerance of +/-6% is regarded as *acceptable* variation between runs.

For the tests in this regression section, the 9.4 performance has been measured such that it is within +/-3% of the equivalent 9.3 performance on IBM z16.

Private Queue

Non-persistent out-of-syncpoint workload

Maximum throughput on a single pair of request/reply queues

The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have gotten the message, they put another message to the request queue. The messages are put and got out of syncpoint.

There are 4 batch server tasks that action MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into the MQGET-with-wait call. The messages are put and got out of syncpoint.

Chart: Transaction rate for non-persistent out-of-syncpoint workload

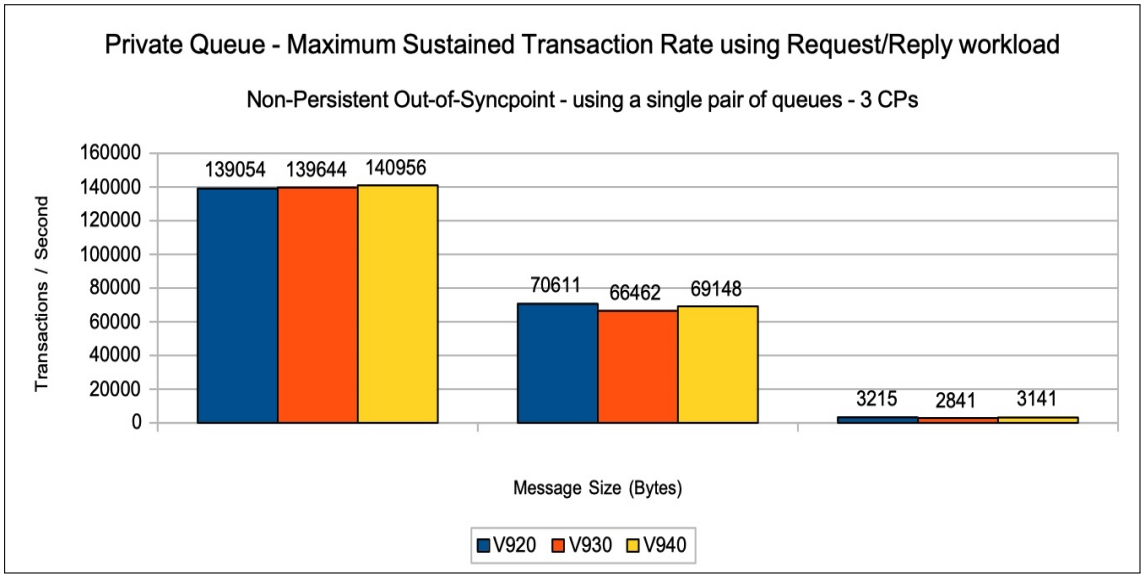
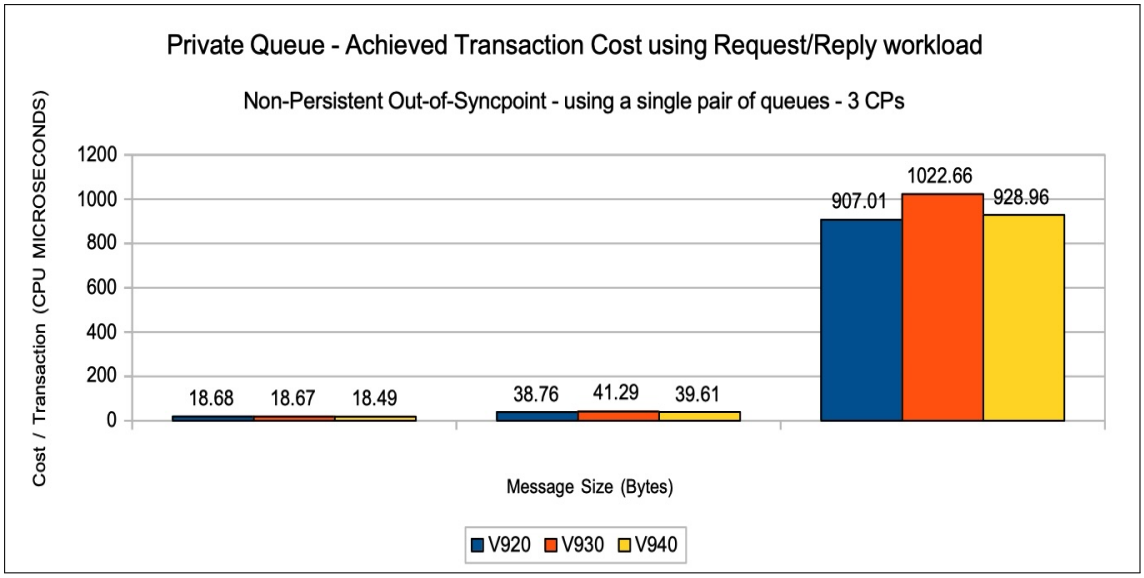


Chart: Transaction cost for non-persistent out-of-syncpoint workload



Scalability of request/reply model across multiple queues

The queue manager is configured with pagesets 0 through 15 and with 1 buffer pool per pageset.

On each of pagesets 1 to 15, a pair of request and reply queues are defined. The test starts up 1 requester and 1 server task accessing the queues on pageset 1 and runs a request/reply workload. At the end, the test starts a second pair of requester and server tasks which access the queues on pageset 2 and so on until there are 15 requester and 15 server tasks using queues on all 15 pagesets with the application queues defined.

The requester and server tasks specify NO_SYNCPOINT for all messages.

The measurements are run on a single LPAR with 16 dedicated processors online on the z16 (3931) used for testing.

Chart: Transaction rate for non-persistent out-of-syncpoint scalability workload

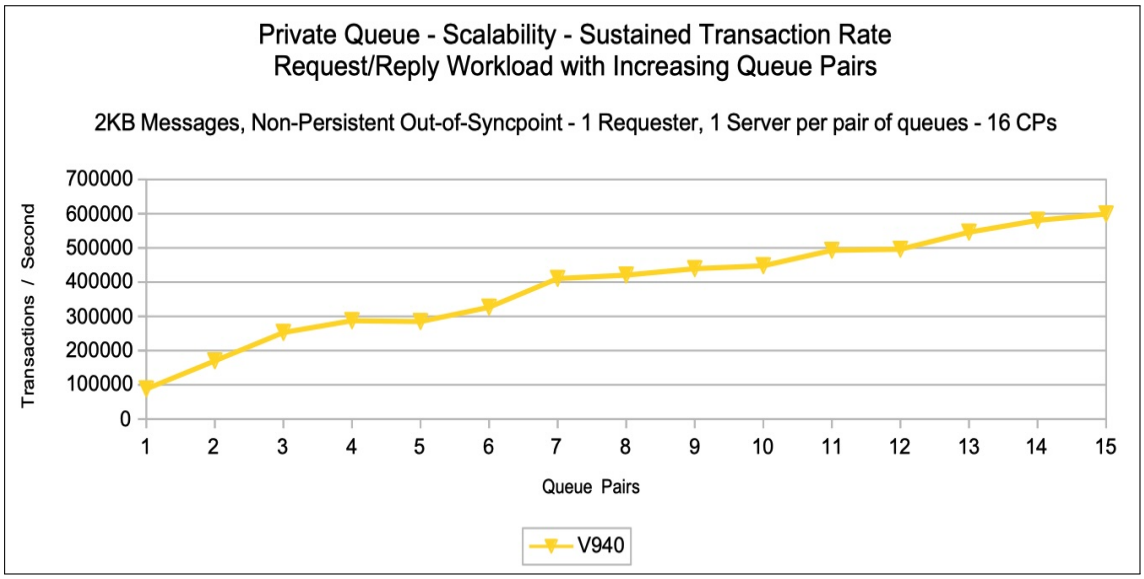
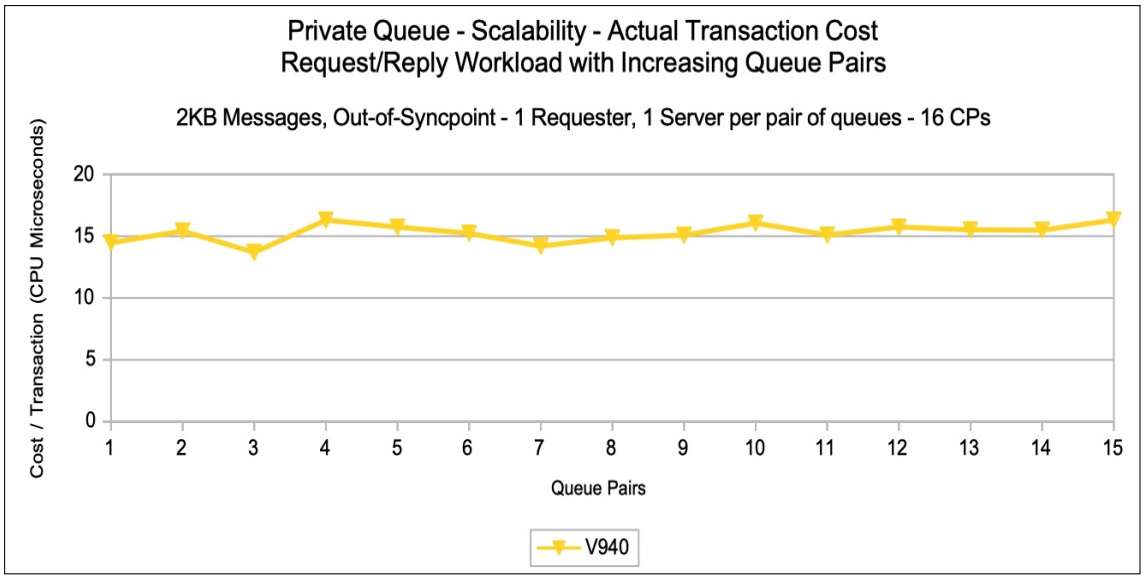


Chart: Transaction cost for non-persistent out-of-syncpoint scalability workload



The preceding 2 charts show that a single queue manager is able to drive in excess of 598,000

transactions per second - or 1,197,000 non-persistent messages per second on a 16-way LPAR.

Note: Since the performance for 9.2, 9.3 and 9.4 is comparable, the charts show only 9.4 data for the purpose of clarity.

Non-persistent server in-syncpoint workload

Maximum throughput on a single pair of request/reply queues

The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have gotten the message, they put another message to the request queue. The messages are put and got out of syncpoint.

There are 4 batch server tasks that action MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into the MQGET-with-wait call. The messages are got and put in syncpoint with 1 MQGET and 1 MQPUT per commit.

Chart: Transaction rate for non-persistent in syncpoint workload

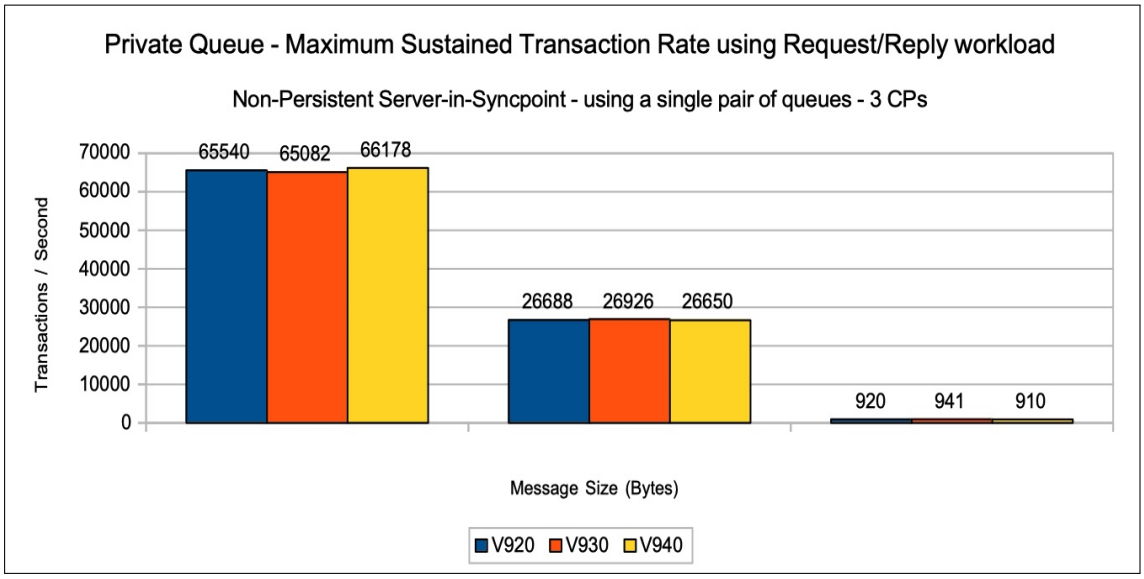
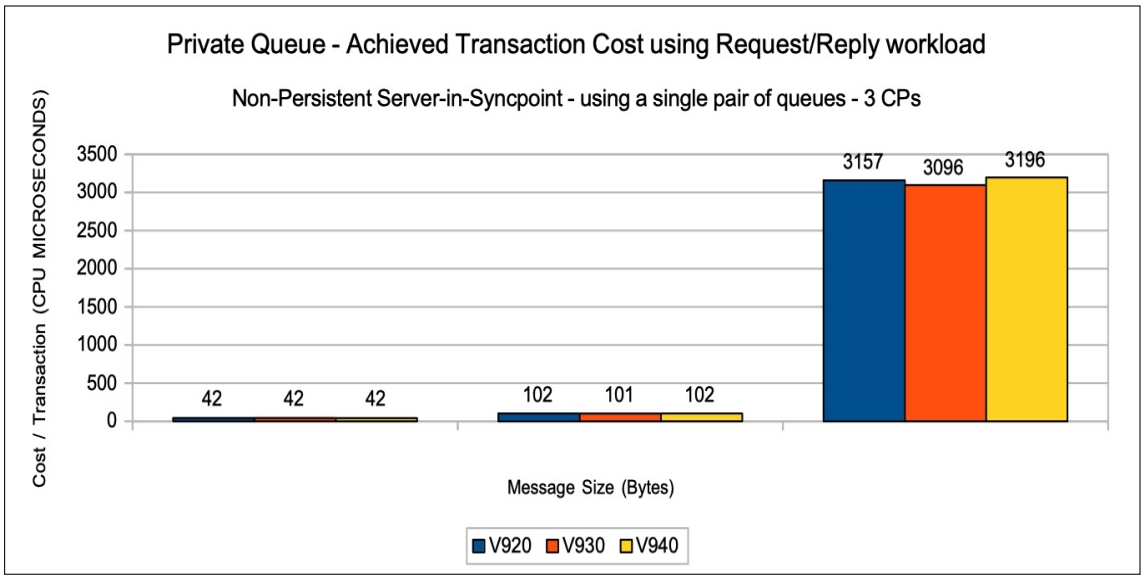


Chart: Transaction cost for non-persistent in syncpoint workload



Scalability of request/reply model across multiple queues

The queue manager is configured with pagesets 0 through 15 and with 1 buffer pool per pageset.

On each of pagesets 1 to 15, a pair of request and reply queues are defined. The test starts up 1 requester and 1 server task accessing the queues on pageset 1 and runs a request/reply workload. At the end, the test starts a second pair of requester and server tasks which access the queues on pageset 2 and so on until there are 15 requester and 15 server tasks using queues on all 15 pagesets with the application queues defined.

The requester tasks specify NO_SYNCPOINT for all messages and the server tasks get and put messages within syncpoint.

The measurements are run on a single LPAR with 16 dedicated processors online on the z16 (3931) used for testing.

The measurements are run using 2KB, 64KB and 4MB messages. The 4MB message measurement uses a maximum of 6 sets of queues and tasks.

Note: Since the performance for 9.2, 9.3 and 9.4 is comparable, the charts show only 9.4 data for the purpose of clarity.

Chart: Transaction rate for non-persistent in syncpoint scalability workload with 2KB messages

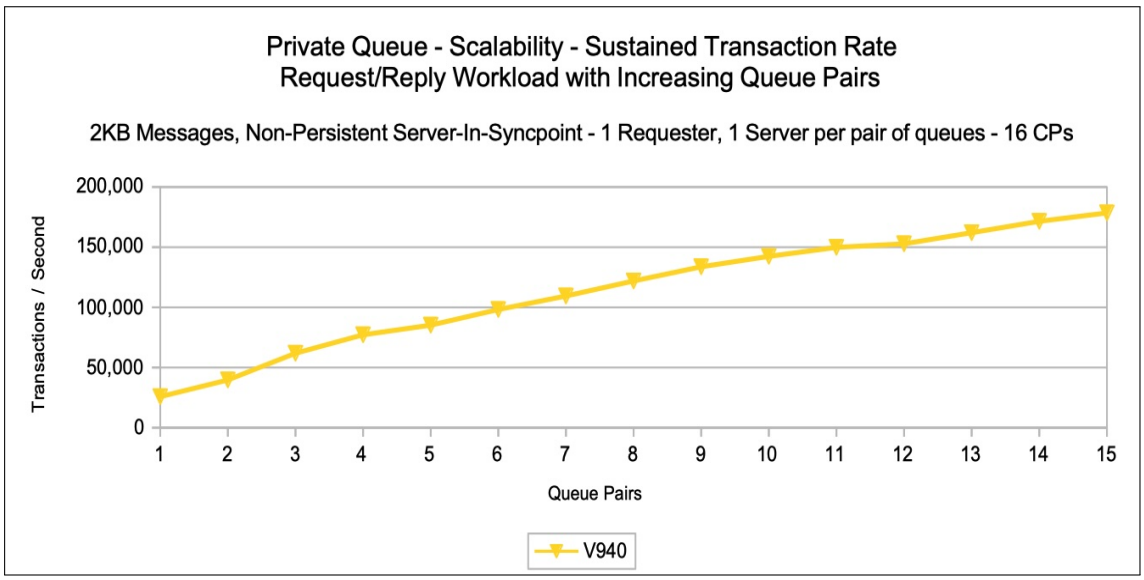


Chart: Transaction cost for non-persistent in syncpoint scalability workload with 2KB messages

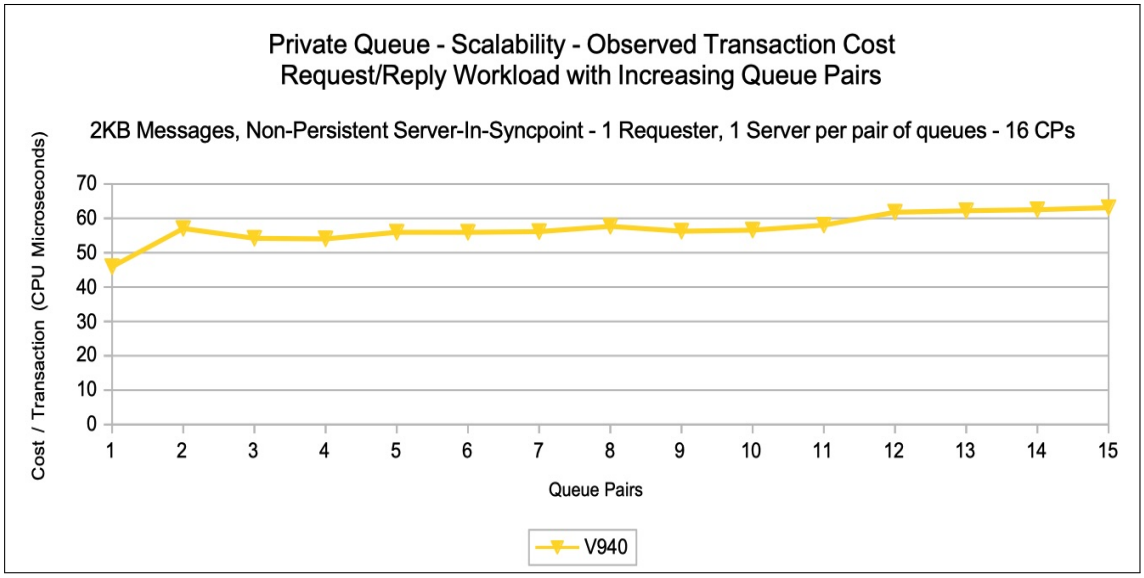


Chart: Transaction rate for non-persistent in syncpoint scalability workload with 64KB messages

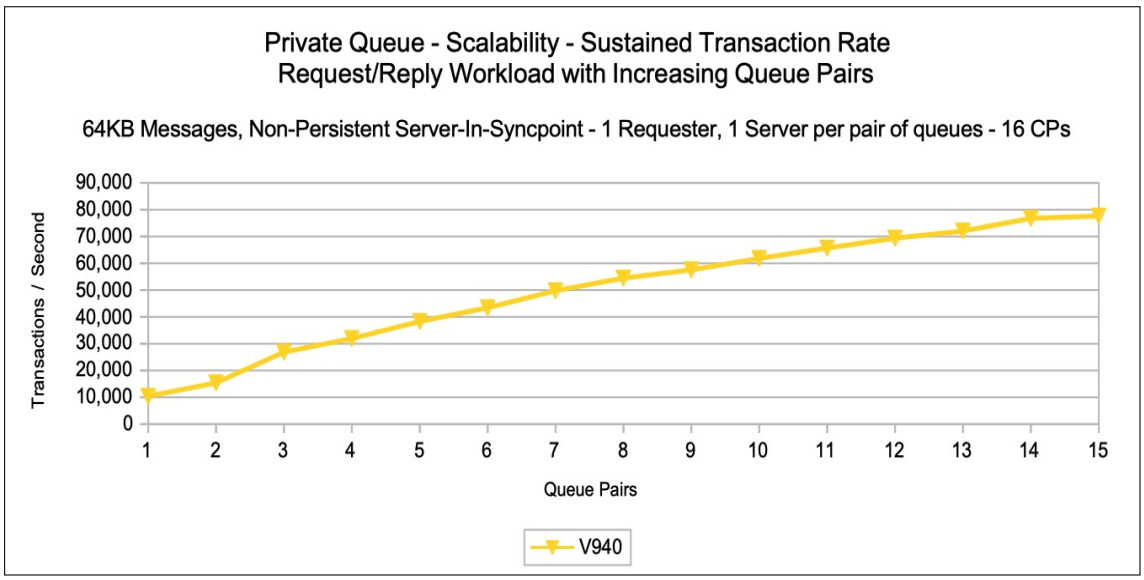


Chart: Transaction cost for non-persistent in syncpoint scalability workload with 64KB messages

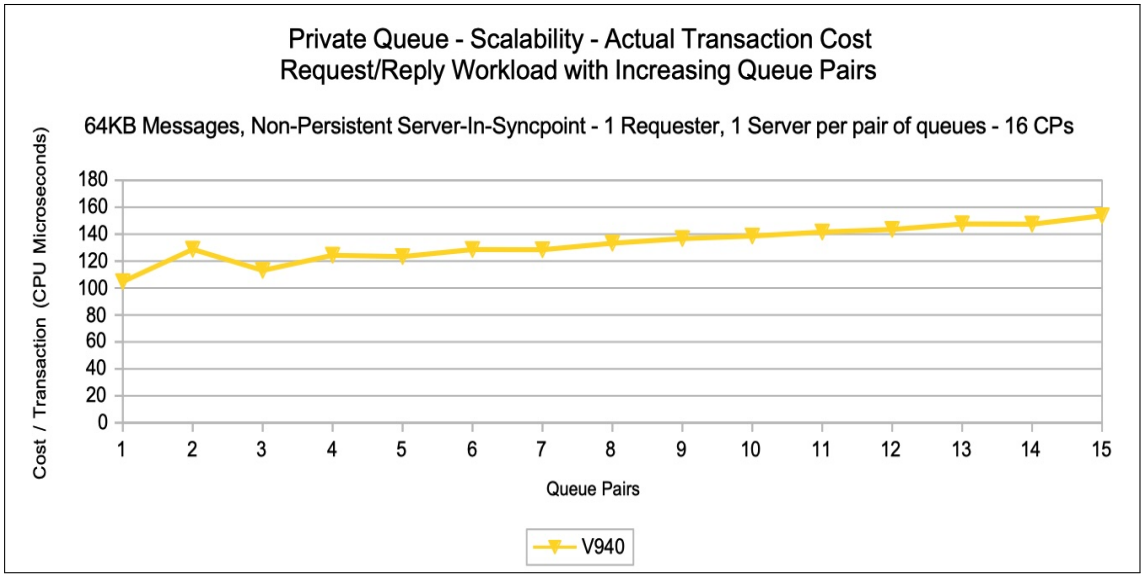


Chart: Transaction rate for non-persistent in syncpoint scalability workload with 4MB messages

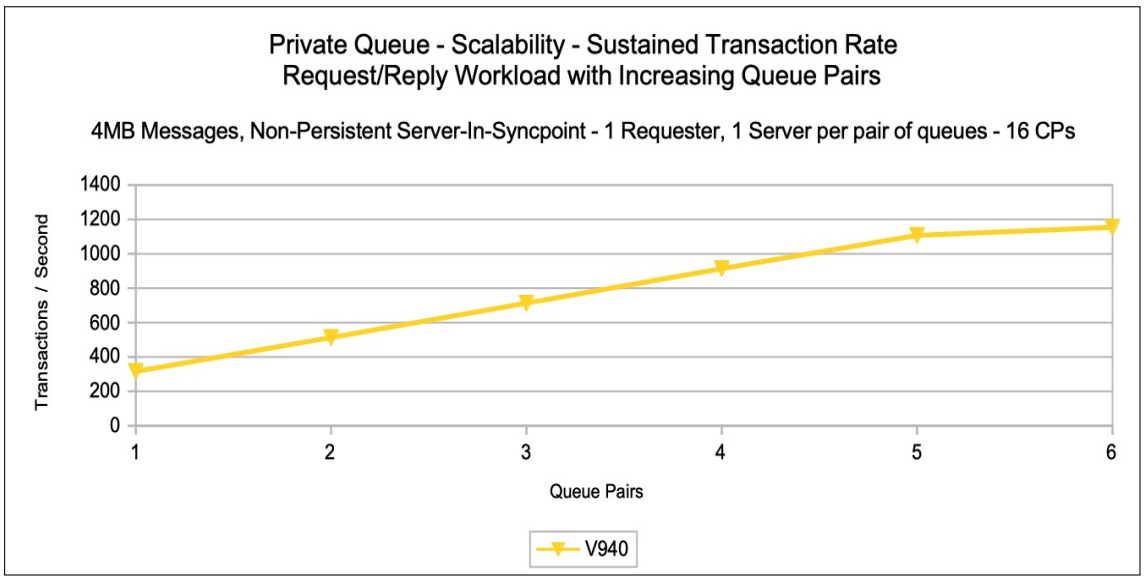
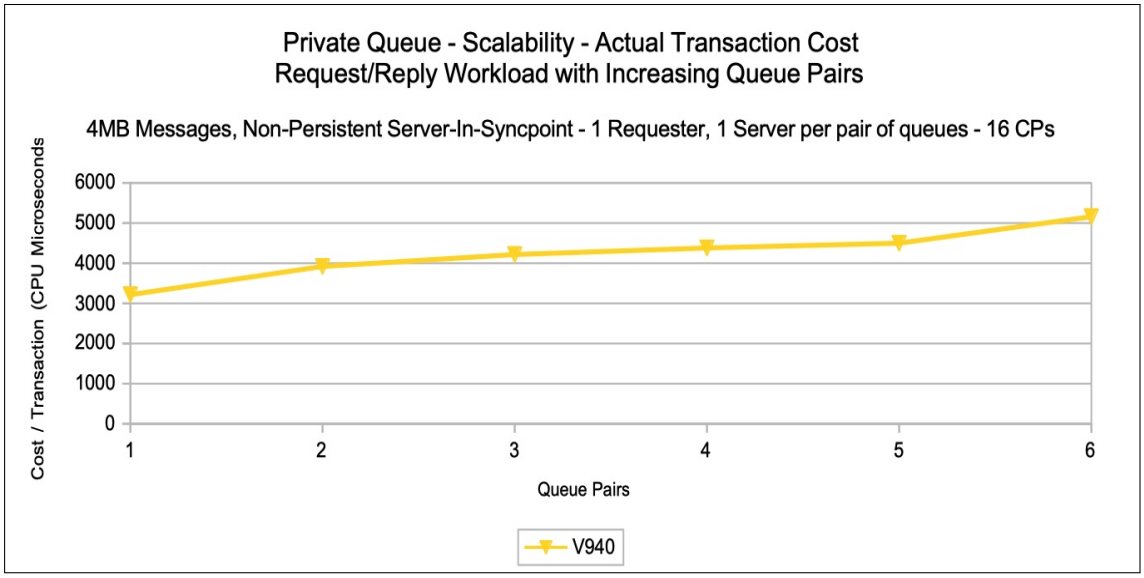


Chart: Transaction cost for non-persistent in syncpoint scalability workload with 4MB messages



Persistent server in-syncpoint workload

Maximum throughput on a single pair of request/reply queues

The test uses 60 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have gotten the message, they put another message to the request queue. The messages are put in-syncpoint and got in-syncpoint.

There are 10 batch server tasks that action MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into the MQGET-with-wait call. The messages are got and put in syncpoint with 1 MQGET and 1 MQPUT per commit.

Chart: Transaction rate for persistent in syncpoint workload

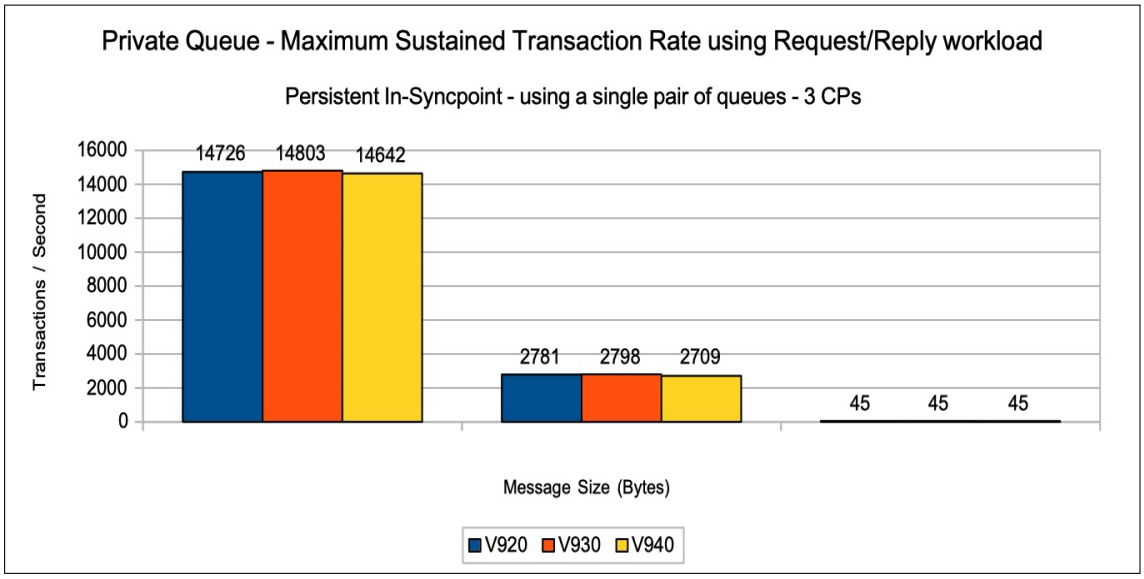
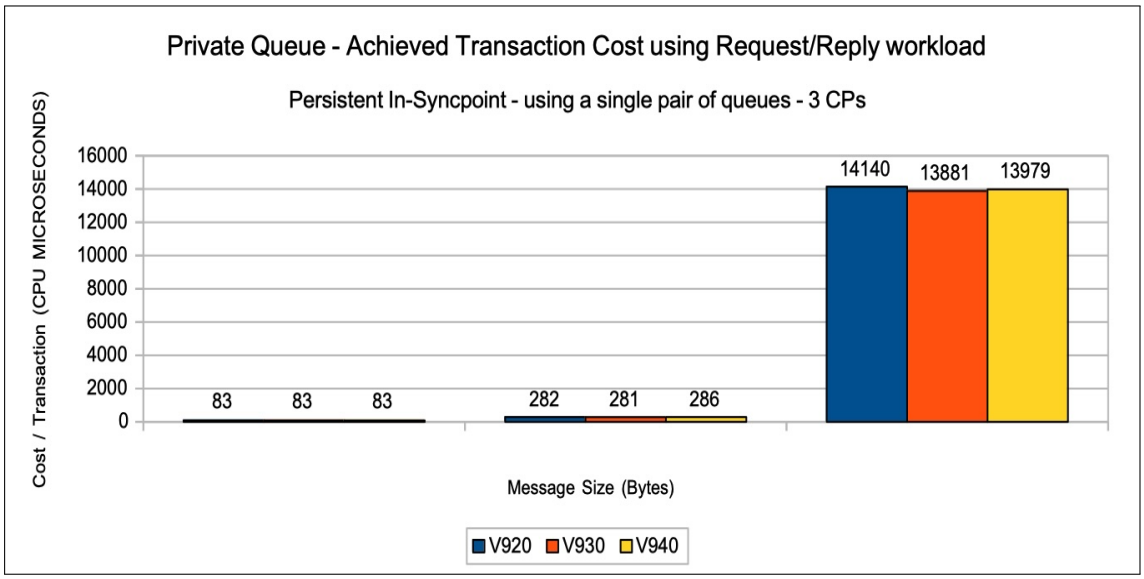


Chart: Transaction cost for persistent in syncpoint workload



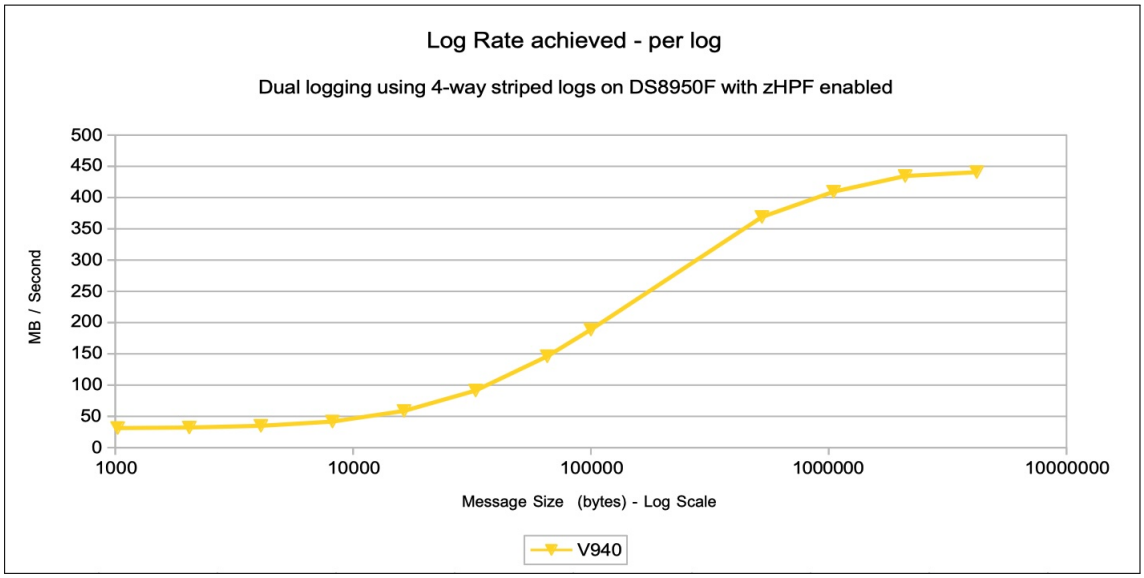
Upper bounds of persistent logging rate

This test uses 3 batch tasks that each put and get messages from a common queue. One of the tasks only uses a 1KB persistent message. The remaining 2 tasks vary the size of their message from 1KB to 4MB.

The following chart shows the achieved log rate on a 3-way LPAR of the z16 (3931).

Since the performance for 9.2, 9.3 and 9.4 is comparable, the chart shows only 9.4 data for the purpose of clarity.

Chart: Peak log rate achieved during persistent message workload



Notes:

The peak log rate in excess of 440MB per second is for each copy of the dual logs, i.e. there is more than 880 MB of data being written to IBM MQ log data sets per second.

These logs are configured with 4 stripes.

CICS Workload

When a CICS transaction makes a call using an MQ API, the resulting processing performed at end of task can have a significant effect on the transaction cost. This cost is most noticeable when adding the first MQ call e.g. MQPUT1 to the application.

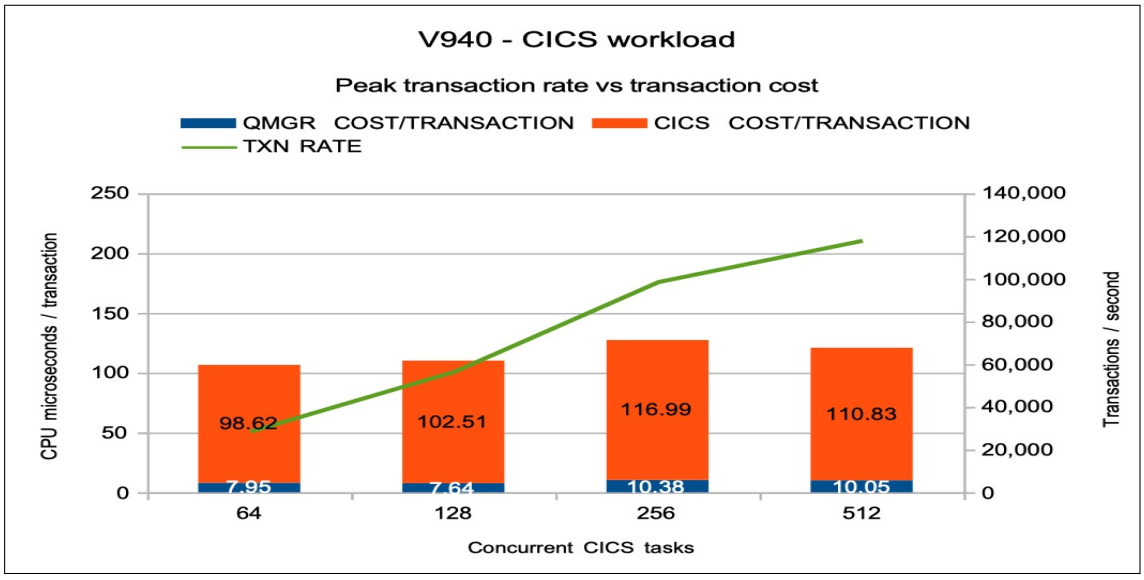
The following chart uses a set of simple CICS transactions that open queues, put a 2KB non-persistent message, get a 2KB non-persistent message, close queues and initiate a new CICS transaction. Running multiple sets of these transactions simulates a number of concurrent transactions across multiple CICS regions.

In these measurements, the system is running with 16 dedicated processors and becomes CPU constrained with 512 concurrent CICS tasks.

Notes on chart:

- As the 9.2, 9.3 and 9.4 data is similar, only 9.4 data is shown.

Chart: CICS workload with MQ 9.4



Shared Queue

Version 7.1.0 introduced CFLEVEL(5) with Shared Message Data Sets (SMDS) as an alternative to Db2 as a way to store large shared messages. This resulted in significant performance benefits with both larger messages (greater than 63KB) as well as smaller messages when the Coupling Facility approached its capacity by using tiered thresholds for offloading data. For more details on the performance of CFLEVEL(5), please refer to performance report MP16 "Capacity Planning and Tuning Guide".

CFLEVEL(5) with SMDS with the default offload thresholds has been used for the shared queue measurements.

Non-persistent out-of-syncpoint workload

Maximum throughput on a single pair of request/reply queues

The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have gotten the message, they put another message to the request queue. The messages are put and got out of syncpoint.

There are 4 batch server tasks that action MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into the MQGET-with-wait call. The messages are put and got out of syncpoint.

An increasing number of queue managers are allocated to process the workload. Each queue manager has 5 requester tasks and 4 server tasks.

Chart: Transaction rate for non-persistent out-of-syncpoint workload - 2KB messages.

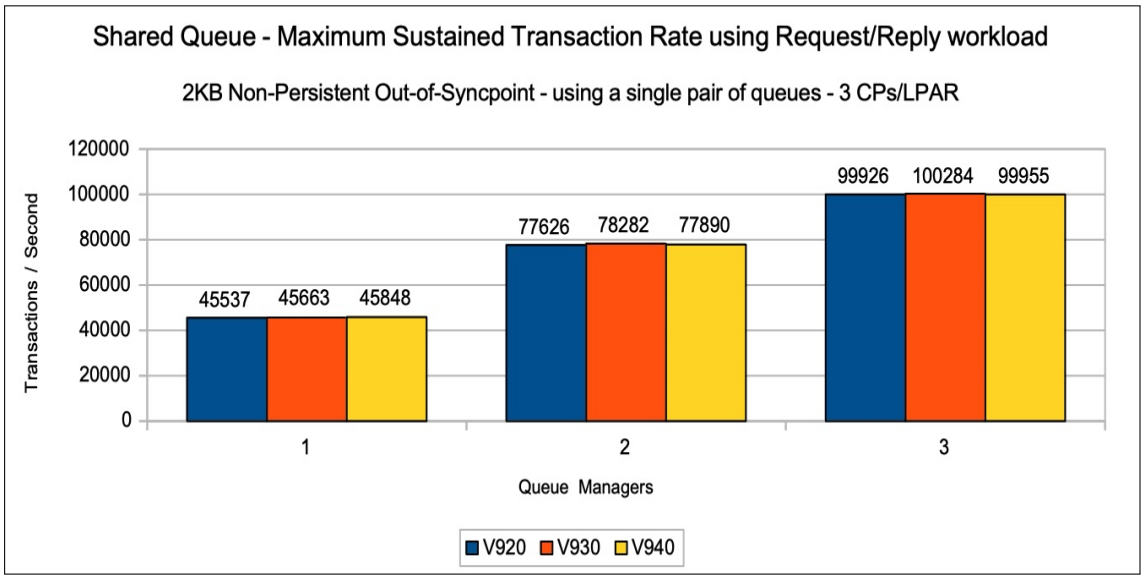


Chart: Transaction cost for non-persistent out-of-syncpoint workload - 2KB messages.

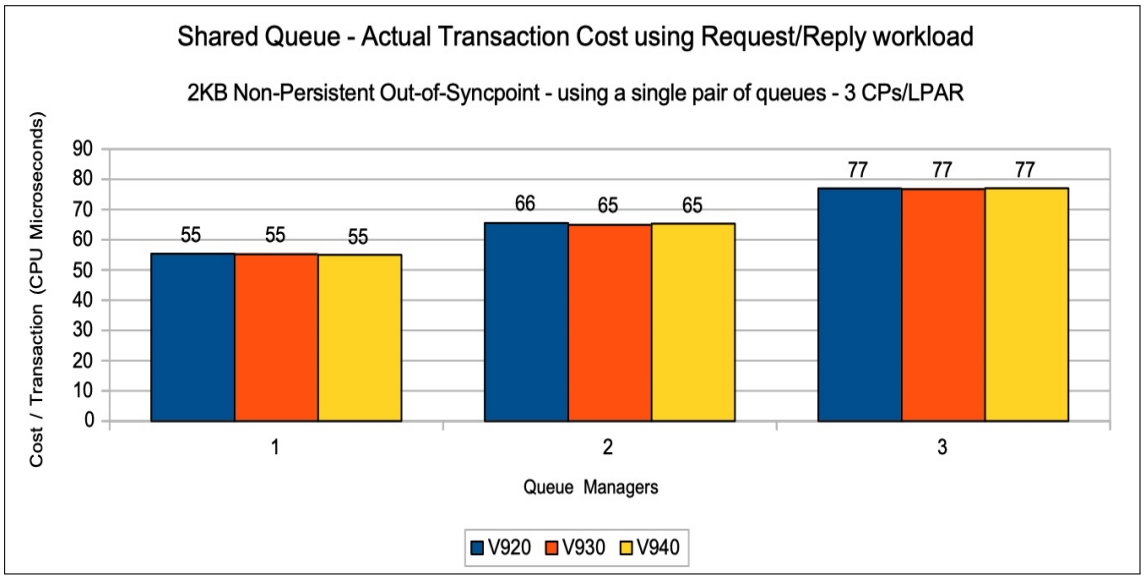


Chart: Transaction rate for non-persistent out-of-syncpoint workload - 64KB messages.

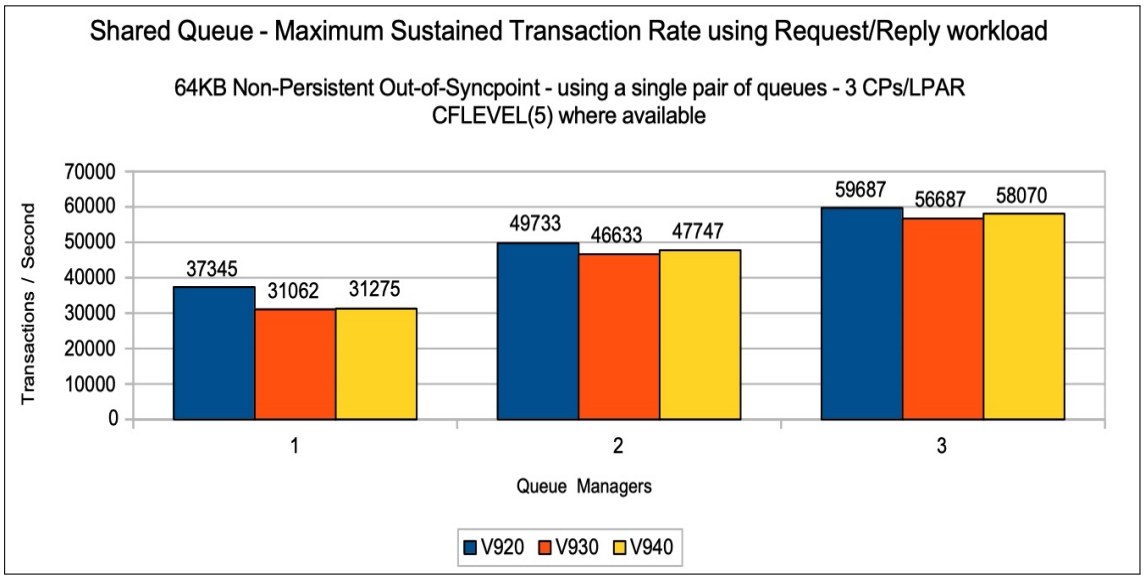


Chart: Transaction cost for non-persistent out-of-syncpoint workload - 64KB messages.

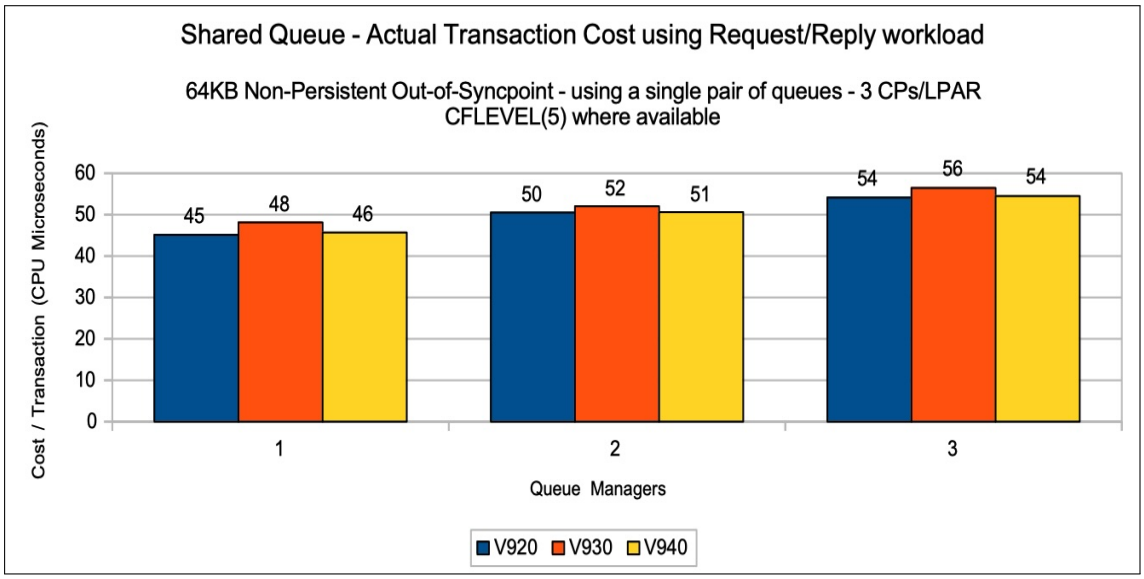
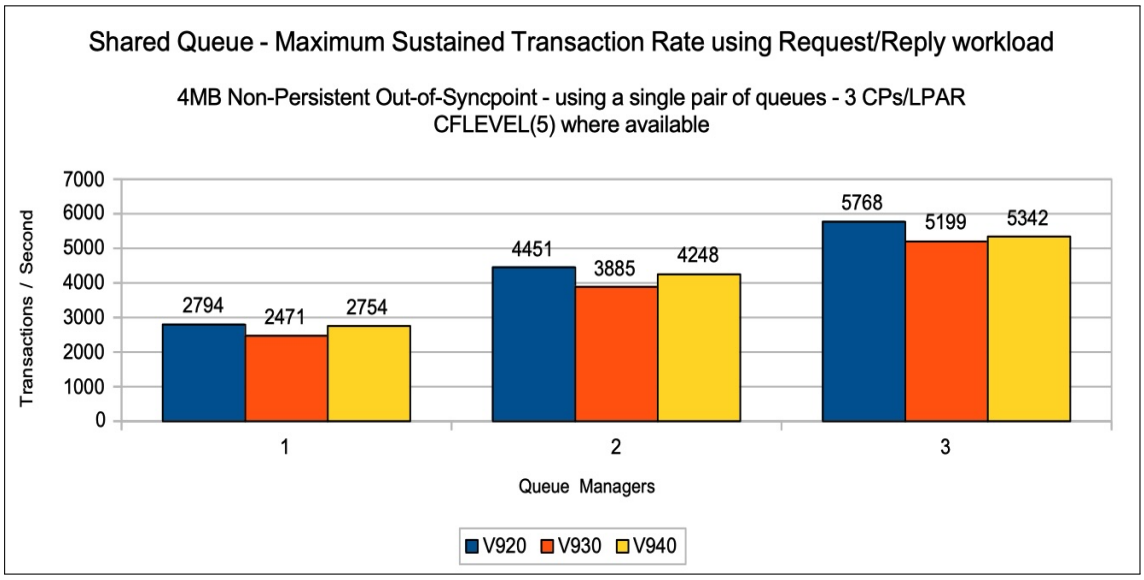
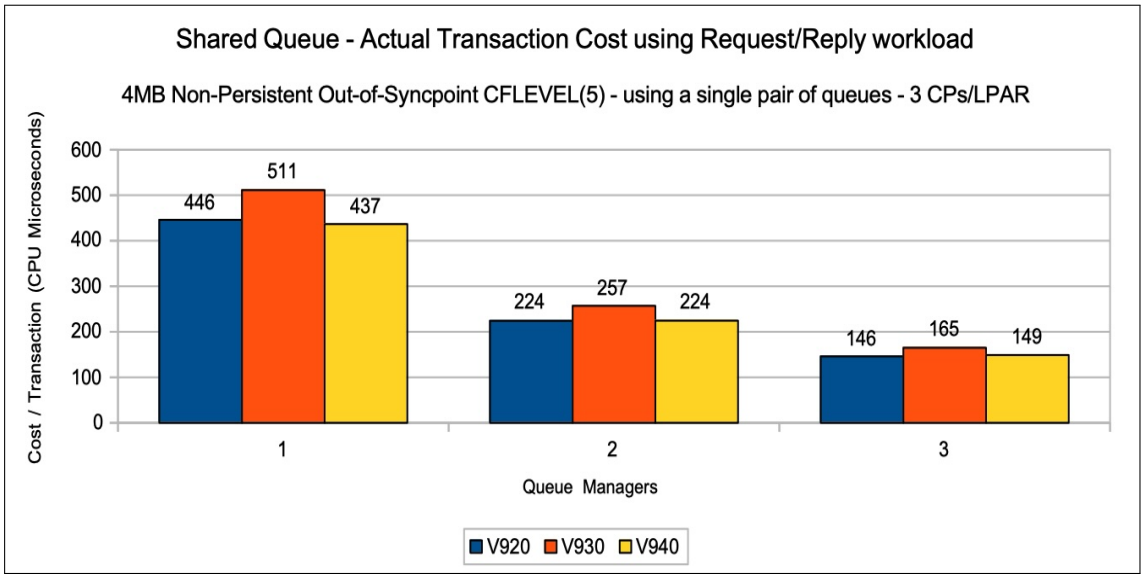


Chart: Transaction rate for non-persistent out-of-syncpoint workload - 4MB messages.



The transaction rates achieved with MQ for z/OS 9.2 when using 4MB messages appear to outperform subsequent releases. Investigation of the test shows that the success rate of put to waiting getting was marginally higher, which for larger messages can make a significant difference to the transaction rate.

Chart: Transaction cost for non-persistent out-of-syncpoint workload - 4MB messages.



Non-persistent server in-syncpoint workload

Maximum throughput on a single pair of request/reply queues

The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have gotten the message, they put another message to the request queue. The messages are put and got out of syncpoint.

There are 4 batch server tasks that action MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into the MQGET-with-wait call. The messages are got and put in syncpoint with 1 MQGET and 1 MQPUT per commit.

An increasing number of queue managers are allocated to process the workload. Each queue manager has 5 requester and 4 server tasks.

Chart: Transaction rate for non-persistent in syncpoint workload - 2KB

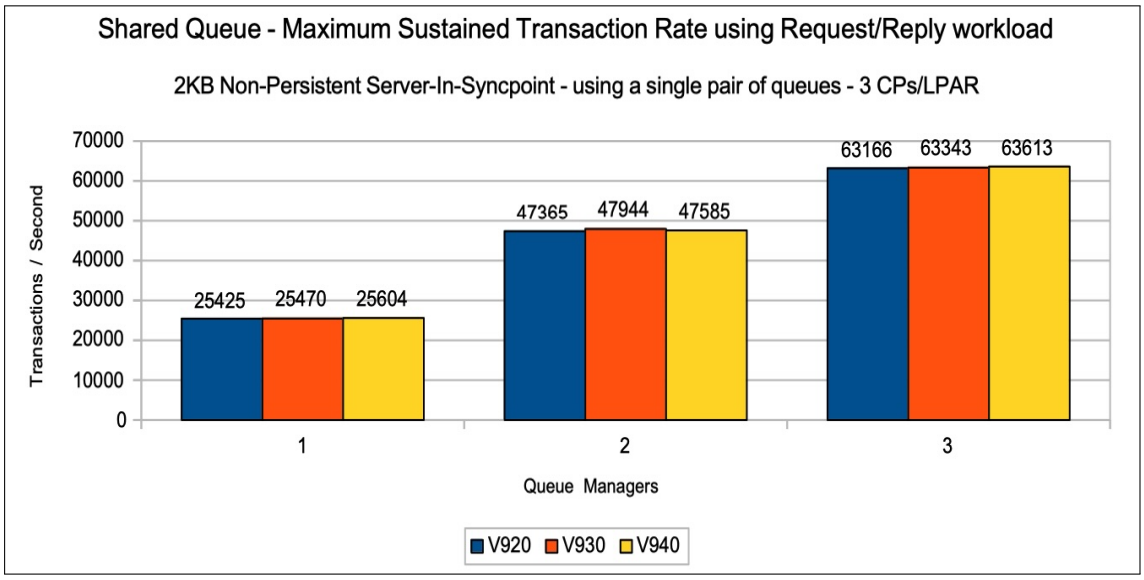


Chart: Transaction cost for non-persistent in syncpoint workload - 2KB

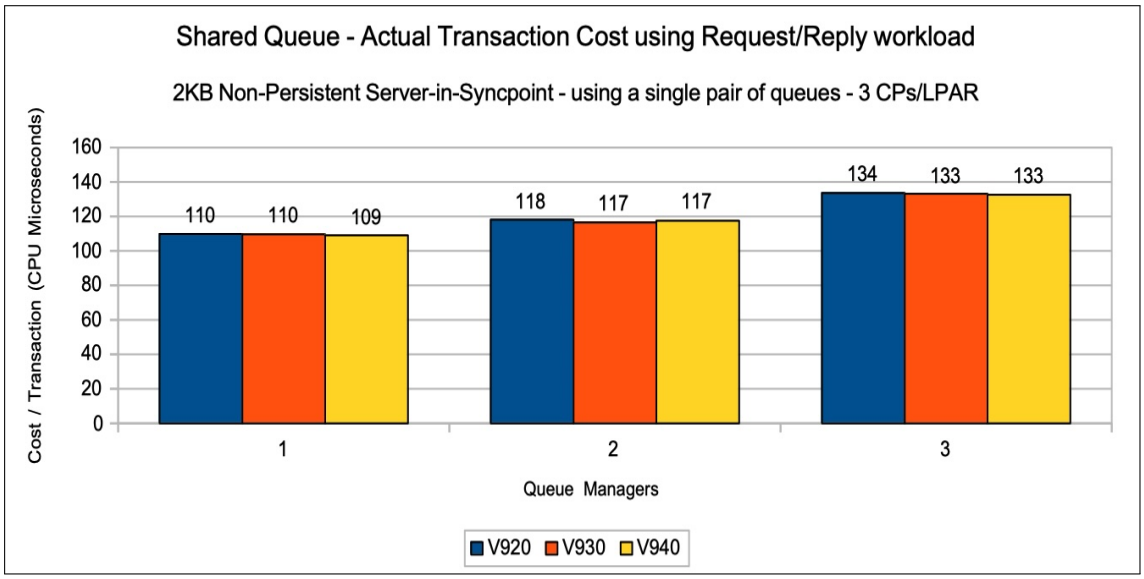


Chart: Transaction rate for non-persistent in syncpoint workload - 64KB

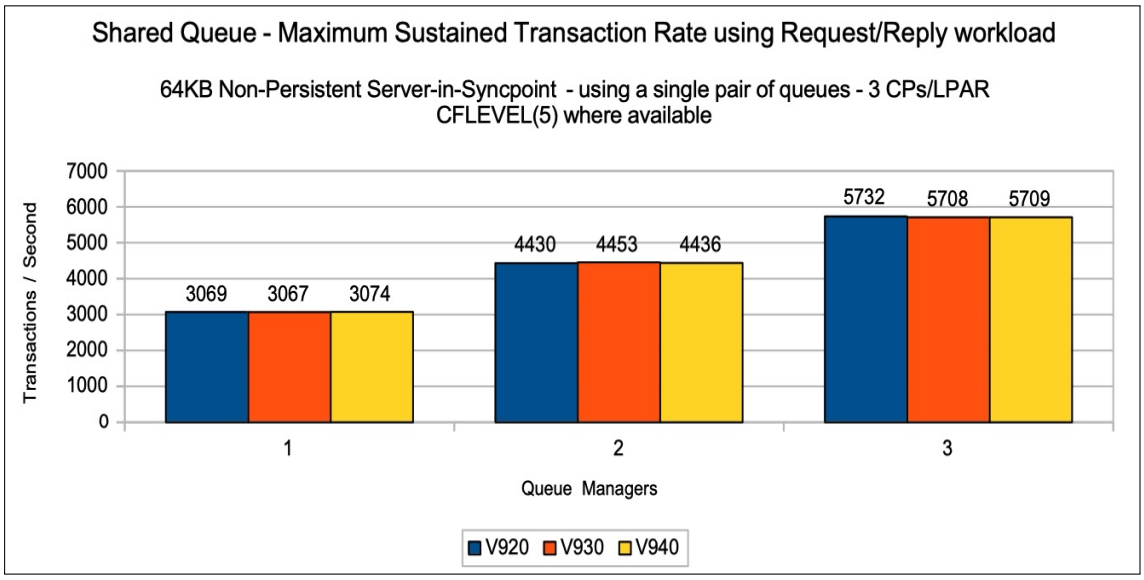


Chart: Transaction cost for non-persistent in syncpoint workload - 64KB

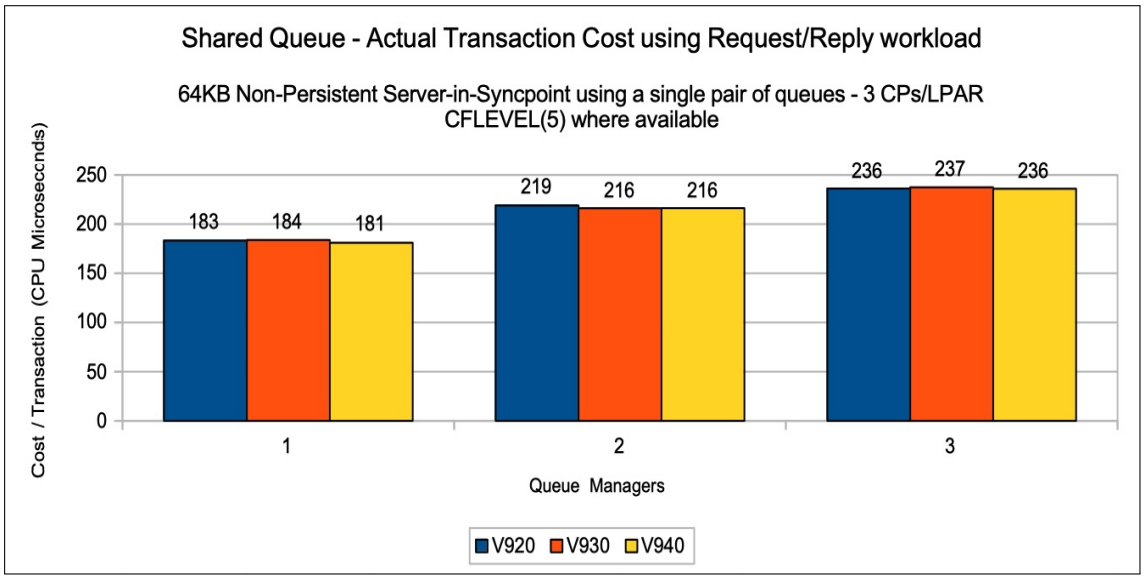


Chart: Transaction rate for non-persistent in syncpoint workload - 4MB

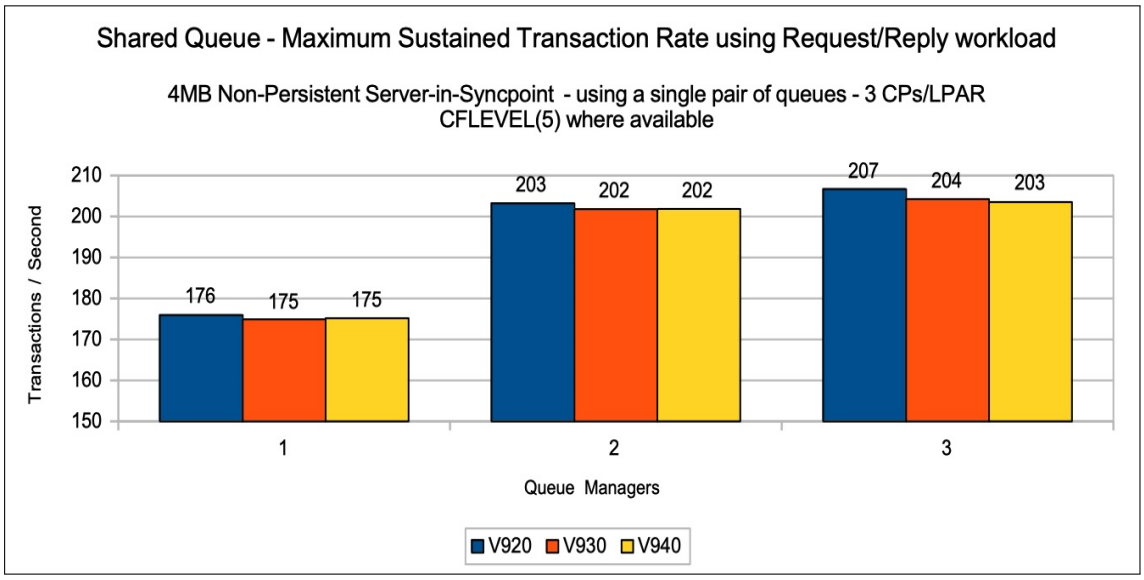
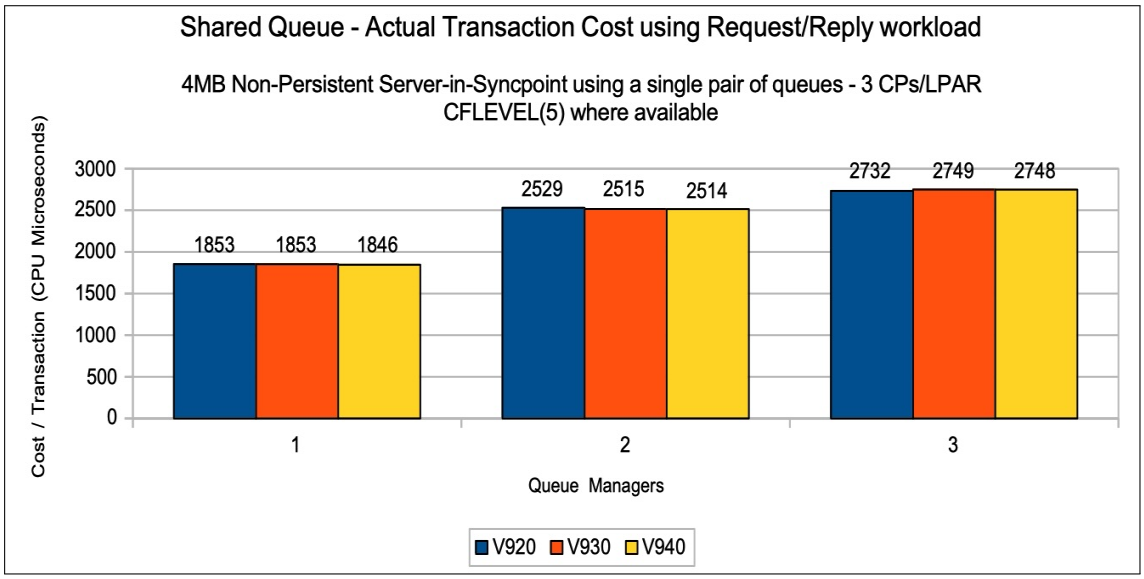


Chart: Transaction cost for non-persistent in syncpoint workload - 4MB



Data sharing non-persistent server in-syncpoint workload

The previous shared queue tests are configured such that any queue manager within the queue sharing group (QSG) can process messages put by any particular requester application. This means that the message may be processed by a server application on any of the available LPARs. Typically the message is processed by a server application on the same LPAR as the requester.

In the following tests, the message can only be processed by a server application on a separate LPAR. This is achieved by the use of multiple pairs of request/reply queues.

The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have gotten the message, they put another message to the request queue. The messages are put and got out of syncpoint.

There are 4 batch server tasks that action MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into the MQGET-with-wait call. The messages are got and put in syncpoint with 1 MQGET and 1 MQPUT per commit.

Chart: Transaction rate for data sharing non-persistent in syncpoint workload - 2KB

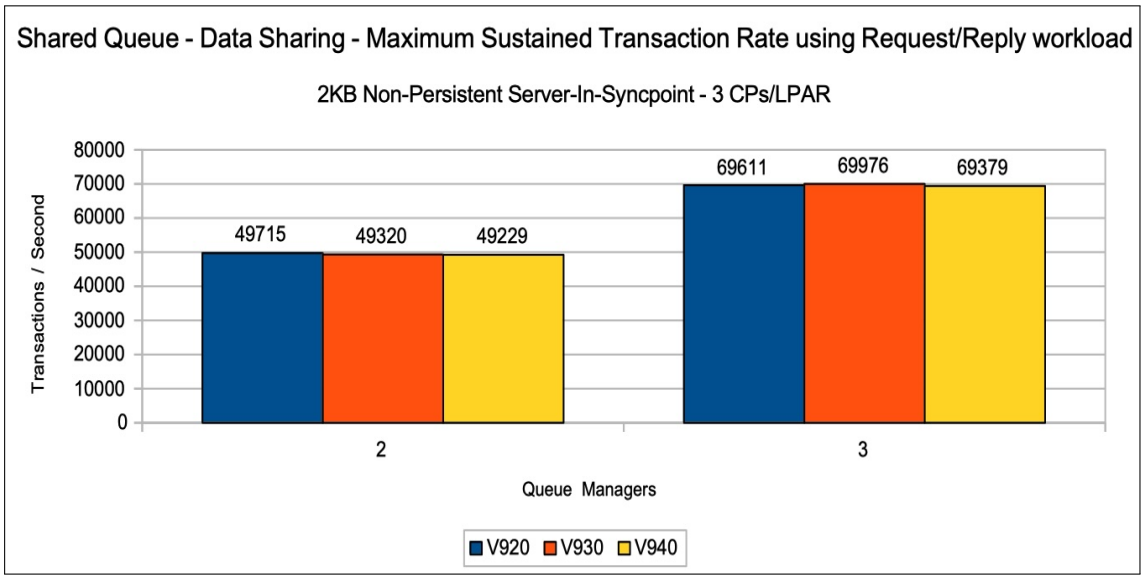


Chart: Transaction cost for data sharing non-persistent in syncpoint workload - 2KB

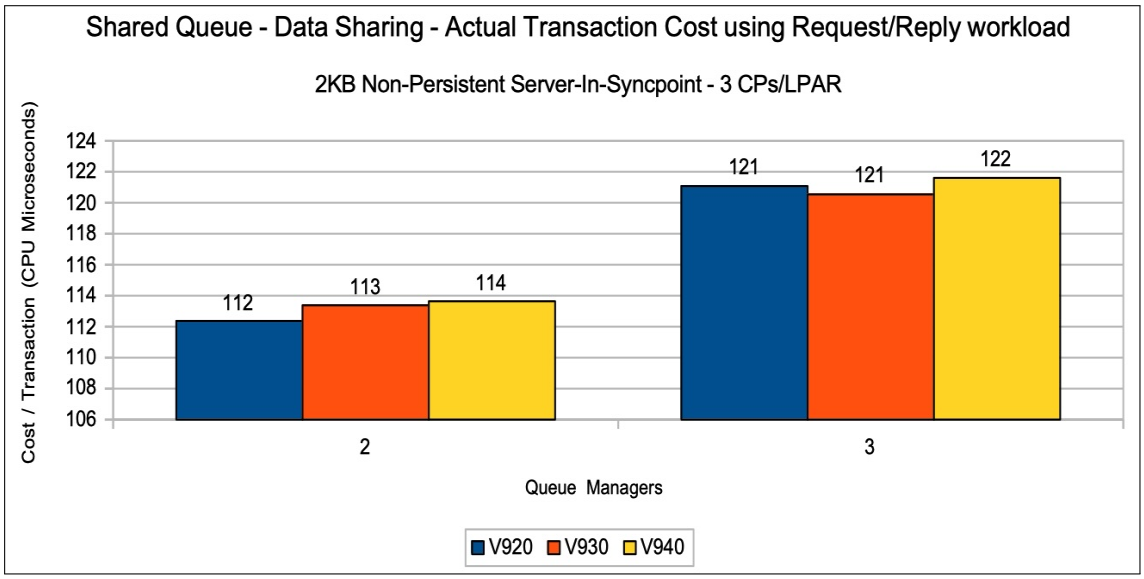


Chart: Transaction rate for data sharing non-persistent in syncpoint workload - 64KB

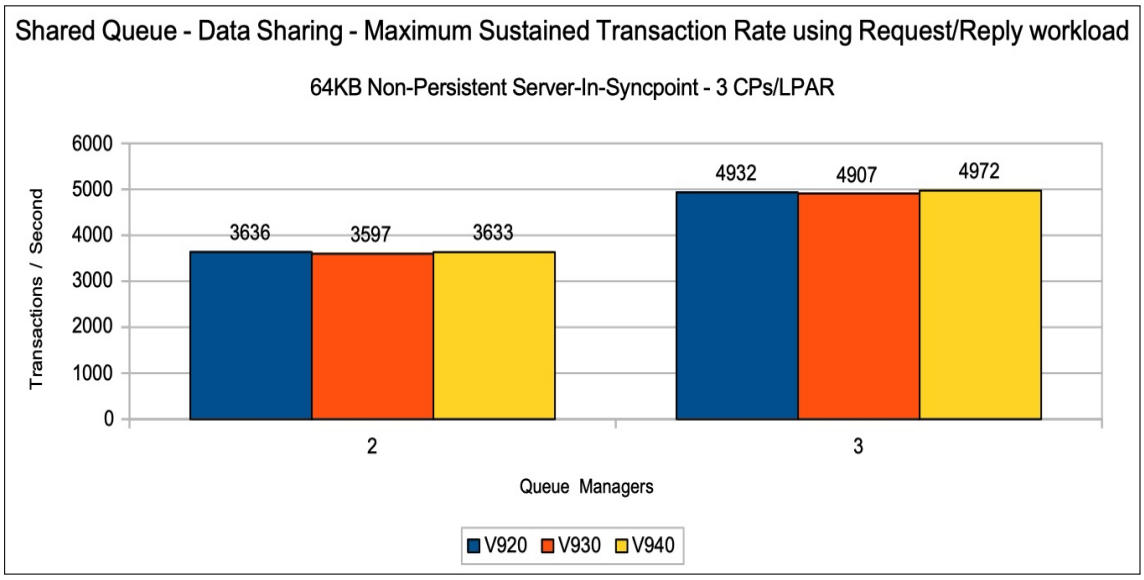
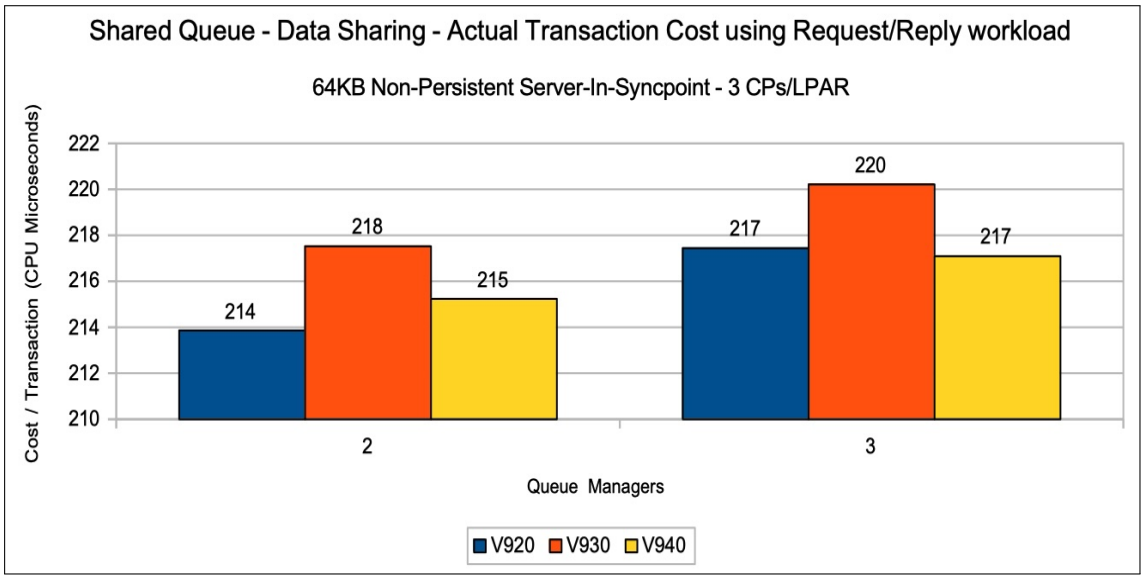


Chart: Transaction cost for data sharing non-persistent in syncpoint workload - 64KB



Moving messages across channels

The regression tests for moving message across channels e.g. sender-receiver channels, are designed to use drive the channel initiator such that system limits rather than MQ are the reason for any constraints, and typically it is CPU that is the constraining factor. Therefore the tests use non-persistent messages in-syncpoint, so that they are not constrained by logging etc.

Within the channel tests there are measurements with small numbers of channels (1 to 5 inbound plus the same number outbound), which was suitable for driving the LPARs to capacity.

Note: Higher throughput can be achieved with additional CPUs but this can increase the transaction cost.

Initially measurements are run both with and without the TLS/SSL encryption enabled.

The cipher spec "ECDHE_RSA_AES_256_CBC_SHA384" was used for all TLS/SSL tests.

The SSLRKEYC attribute was set as follows:

- 0 such that the secret key is only negotiated as channel start. This allows us to determine the impact of encryption on the data.
- 1MB of data can flow across the channel before renegotiating the keys.

The TLS/SSL tests all used a certificate key-size of 2048 bits, which has been the default size for RSA since z/OS 2.2. The impact of certificate key-size is discussed in more detail in blog Impact of certificate key-size on TLS-protected MQ channels.

Channel compression

Since IBM MQ version 8.0.0, the ZLIBFAST compression option is able to exploit zEDC hardware compression, which is discussed in detail in MQ for z/OS - Channel Compression. The compression measurements shown in this document use zEDC hardware compression where possible.

For further guidance on channel tuning and usage, please refer to performance report MP16 "Capacity Planning and Tuning Guide".

Additionally, the white paper "MQ for z/OS - Channel Compression" discusses the performance of channel compression on z/OS including where compression may help improve the performance, either by reducing MQ costs or improving channel throughput.

Measurements:

- The measurements using 1 to 5 channels use batch applications to drive the workload at each end of the channel.
- The compression tests use 32KB message of varying compressibility, so there is something to compress, e.g. a message of 32KB that is 80% compressible would reduce to approximately 6.5KB. These tests are run using 1 outbound and 1 inbound channel so include the compression and inflation costs for the request and reply message.

Streaming messages across channels

Many of the performance tests in the regression section use a request/reply model to demonstrate the performance of the MQ channels.

One common use of MQ is to provide the transport mechanism when data is moved between data centres, such as in an IBM InfoSphere Data Replication queue replication (QREP) scenario. This model sends data in a single direction and can show different characteristics to a typical request/reply model. For example the queue depths may naturally vary significantly, and in some cases may drive MQ page set I/O.

Typically a QREP-scenario would use persistent messages and may have different message sizes for online and batch processing. In this section we use 10KB messages for online processing and 1MB messages for batch processing.

In the streaming configuration, the transmit queue is pre-loaded with messages to ensure the channel is able to send full batches (of 200 messages), thus driving the channel to capacity.

Non-persistent in-syncpoint - 1 to 5 sender receiver channels

Chart: Transaction rate with 32KB messages

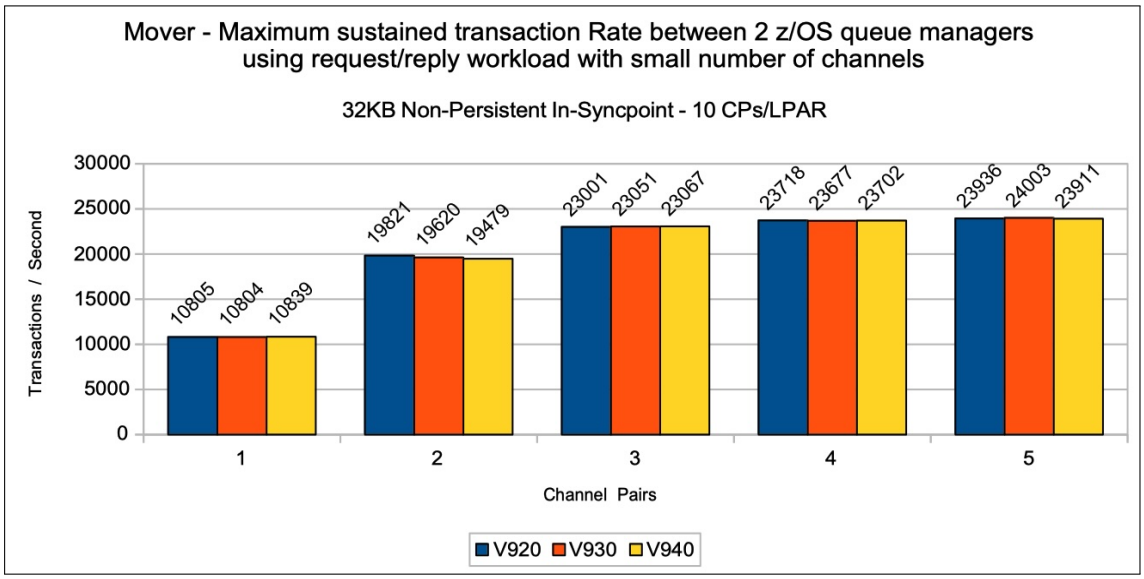


Chart: Transaction cost for 32KB messages

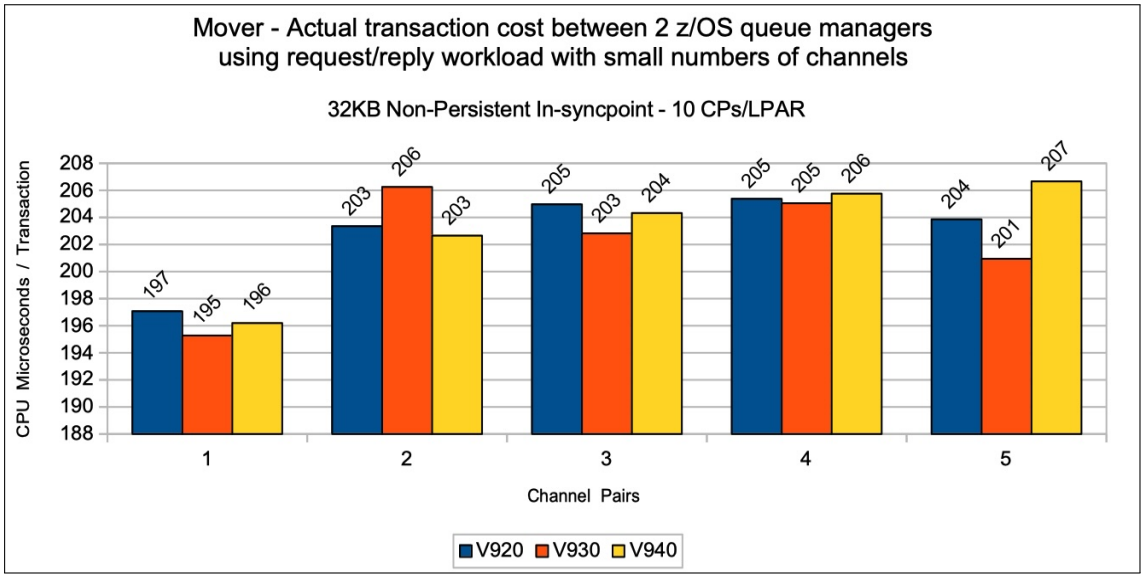


Chart: Transaction rate with 32KB messages over SSL channels with no secret key negotiation

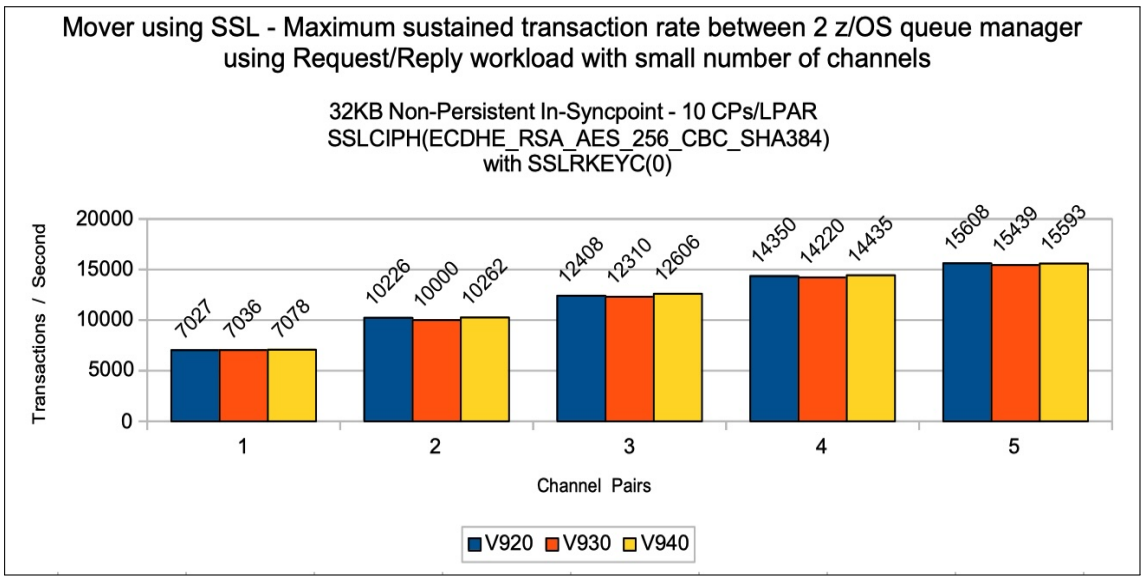


Chart: Transaction cost for 32KB messages over SSL channels with no secret key negotiation

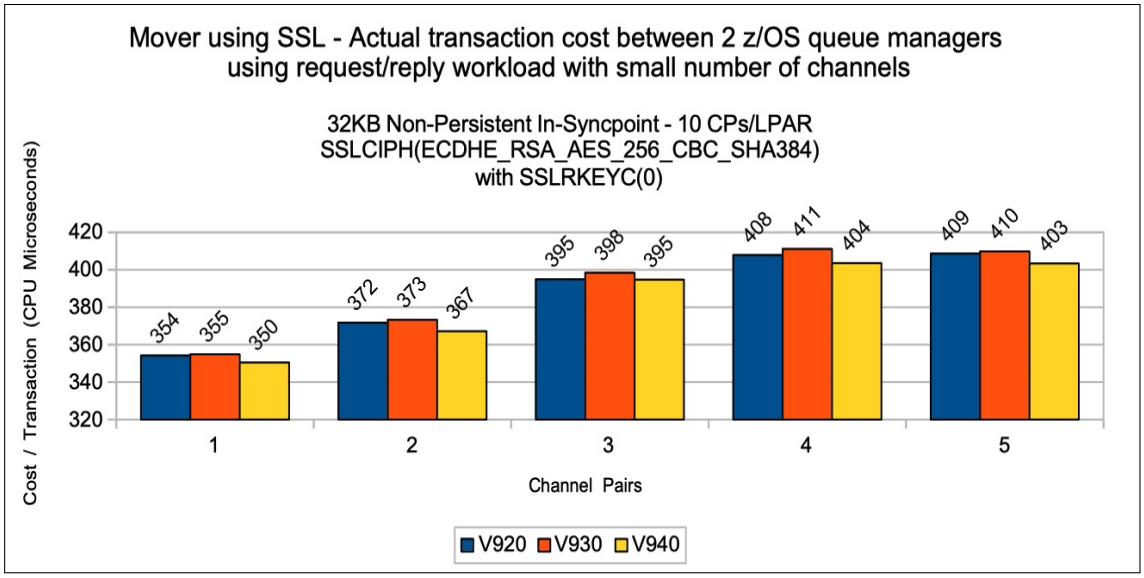


Chart: Transaction rate with 32KB messages over SSL channels with secret key negotiation every 1MB

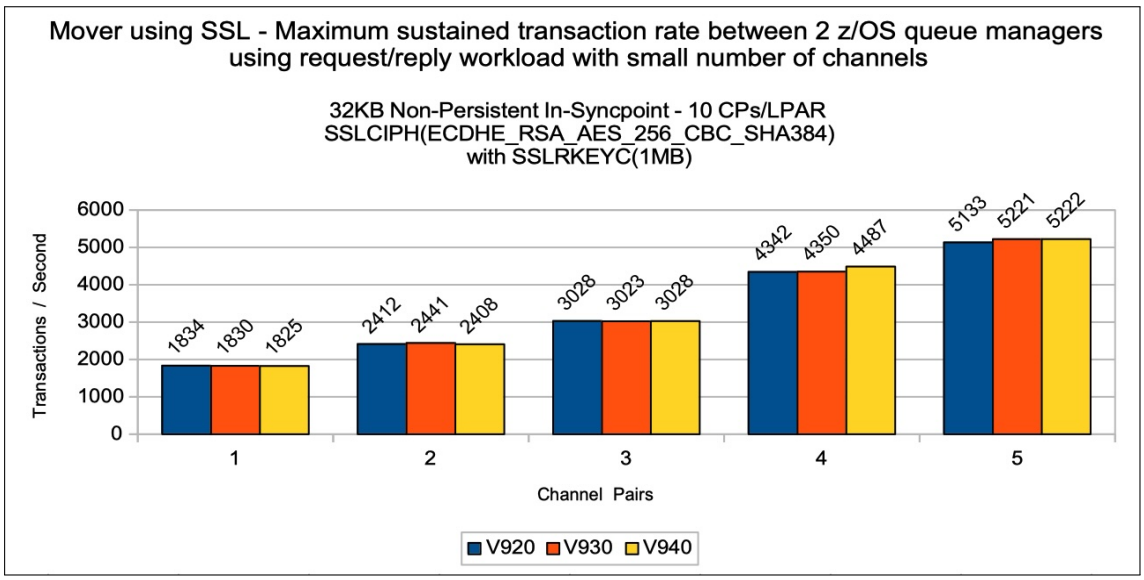
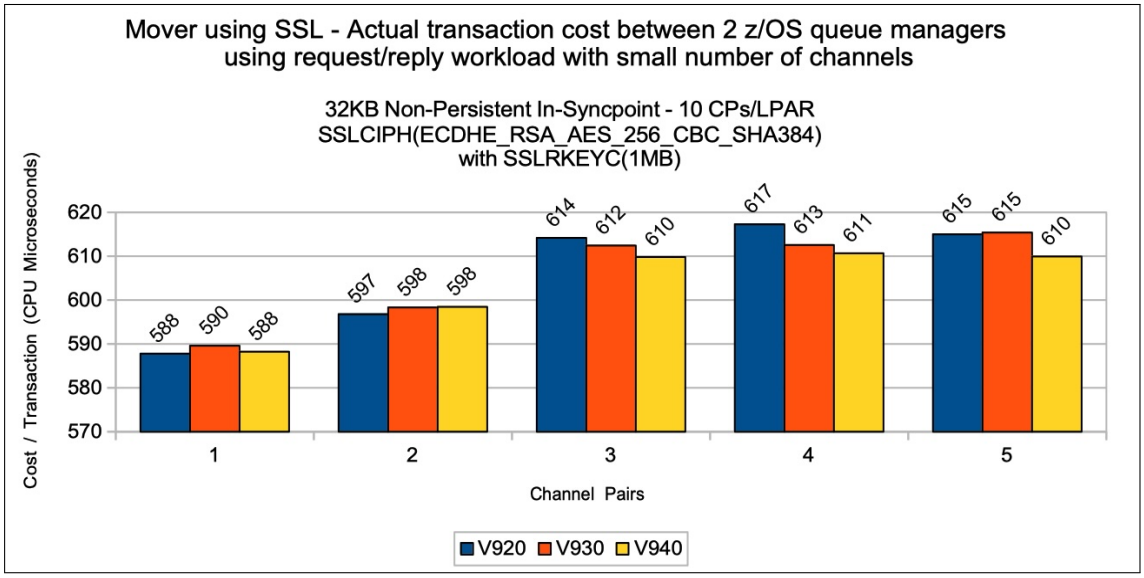


Chart: Transaction cost for 32KB messages over SSL channels with secret key negotiation every 1MB



Channel compression using ZLIBFAST

Version 8.0.0 onwards are able to exploit compression using the available zEDC hardware features.

Chart: Transaction rate with ZLIBFAST on 32KB messages

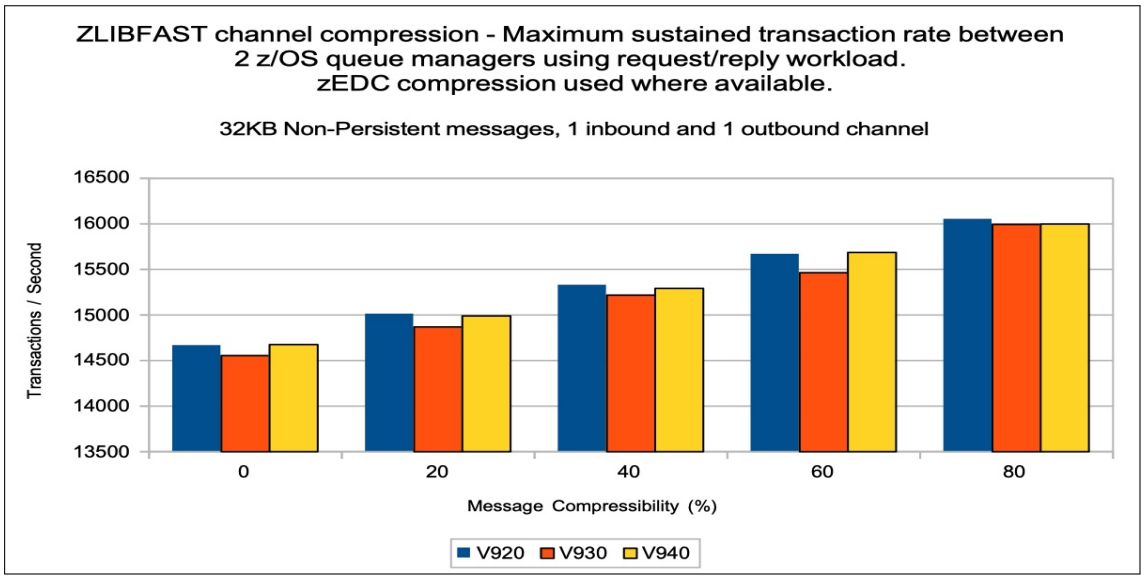
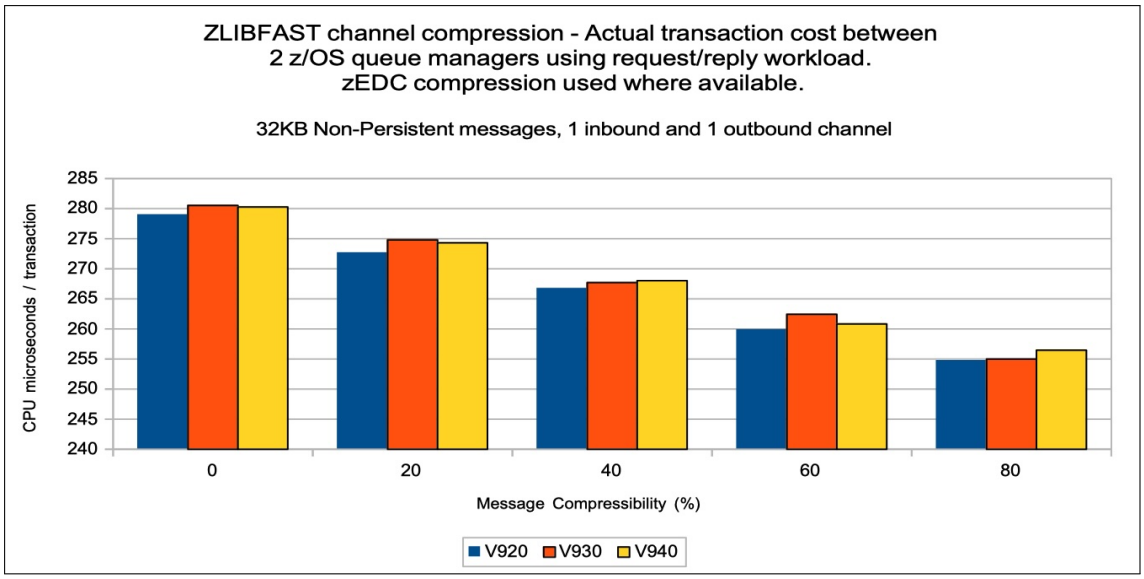


Chart: Transaction cost for ZLIBFAST on 32KB messages



Channel compression using ZLIBHIGH

Chart: Transaction rate with 32KB messages with ZLIBHIGH

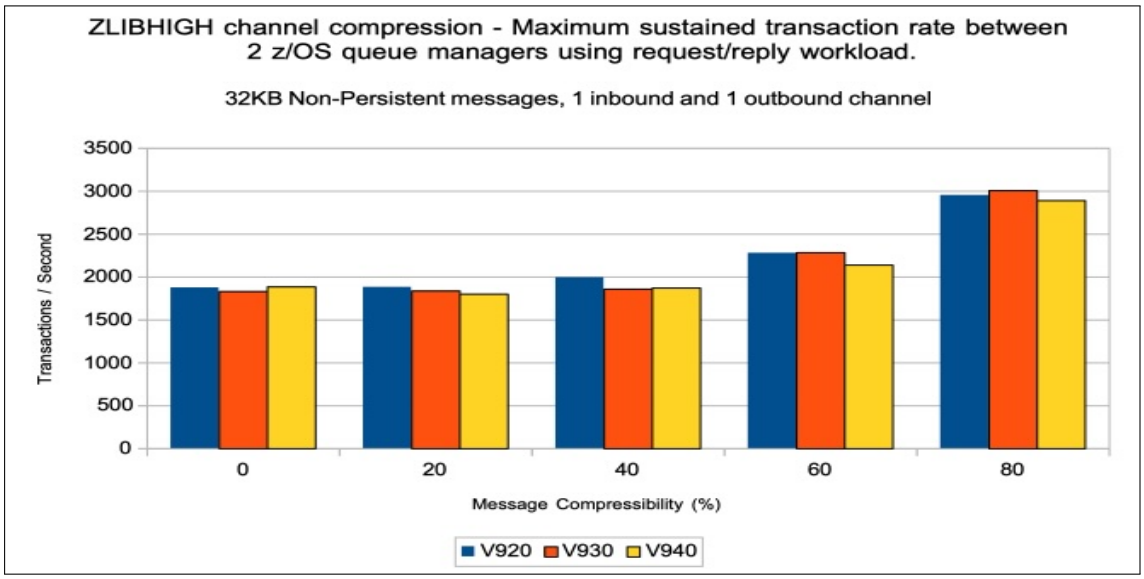
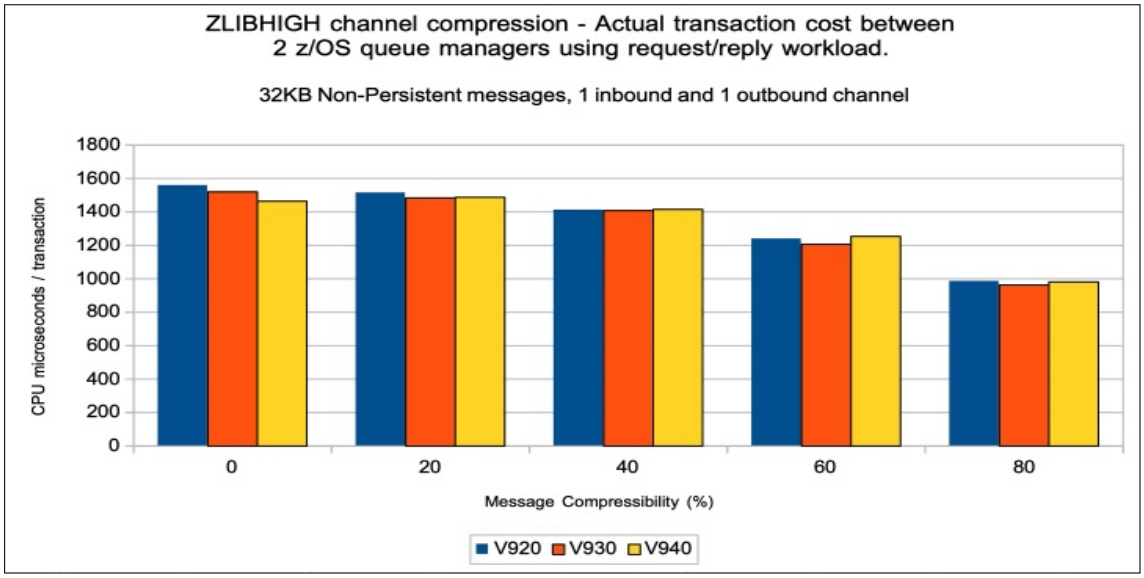


Chart: Transaction cost for 32KB messages with ZLIBHIGH



Channel compression using ZLIBFAST on SSL channels

Version 8.0.0 onwards are able to exploit compression using the available zEDC hardware features.

Chart: Transaction rate with 32KB messages with ZLIBFAST on SSL channels

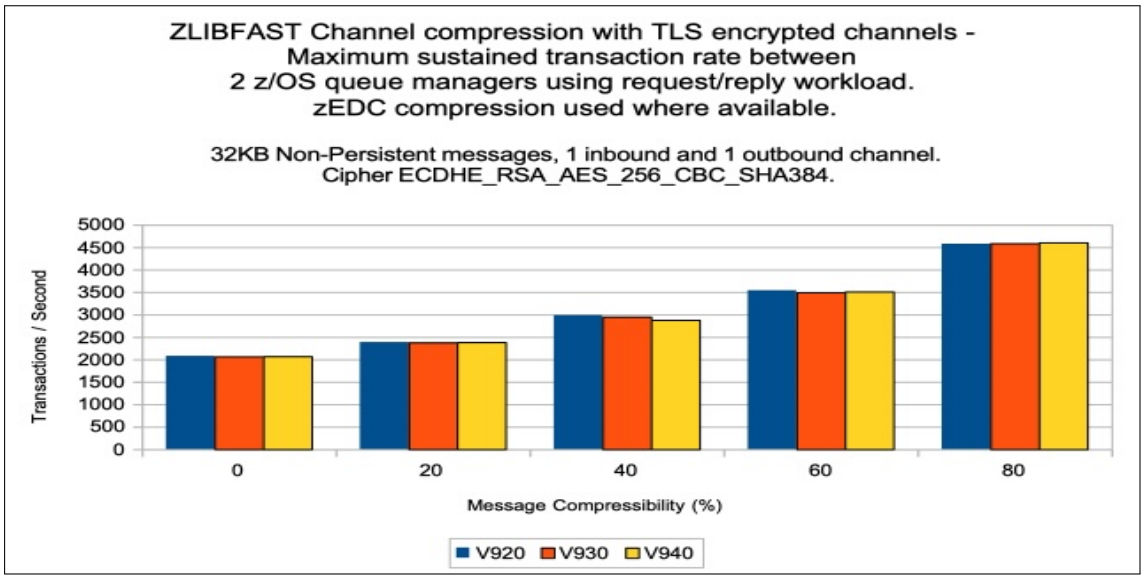
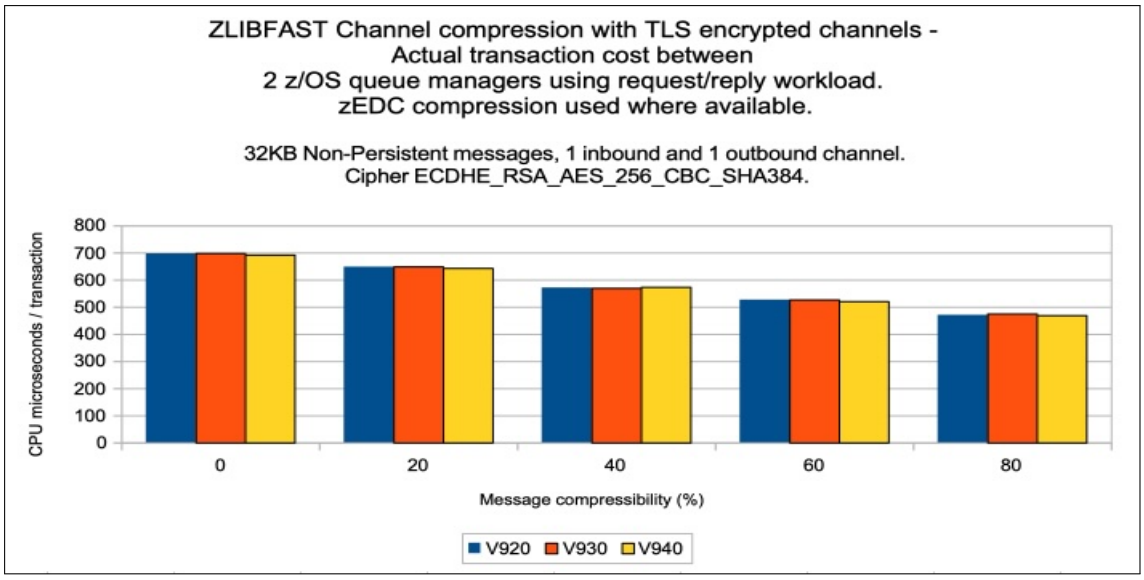


Chart: Transaction cost with 32KB messages with ZLIBFAST on SSL channels



Channel compression using ZLIBHIGH on SSL channels

Chart: Transaction rate with 32KB messages with ZLIBHIGH on SSL channels

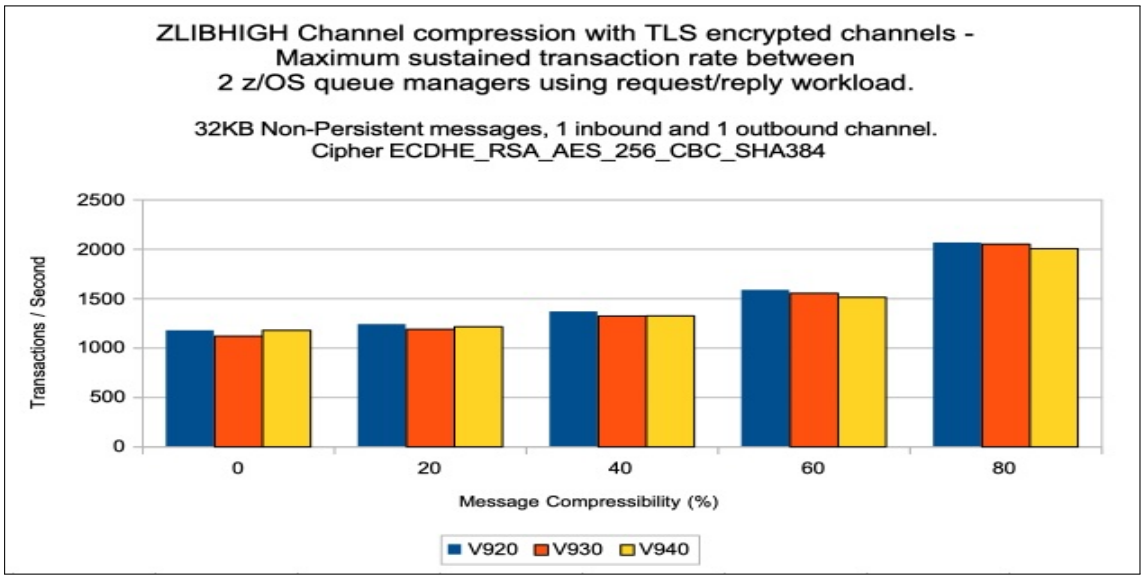
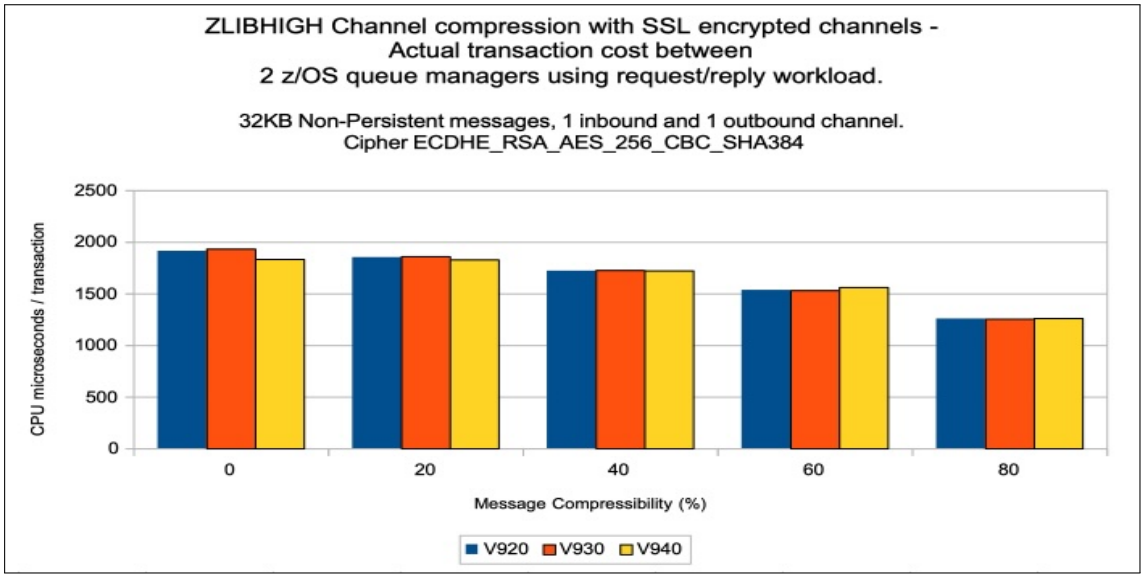


Chart: Transaction cost for 32KB messages with ZLIBHIGH on SSL channels



Streaming workload between 2 z/OS queue managers

Chart: Rate (MB/Second) to stream persistent messages between 2 z/OS queue managers

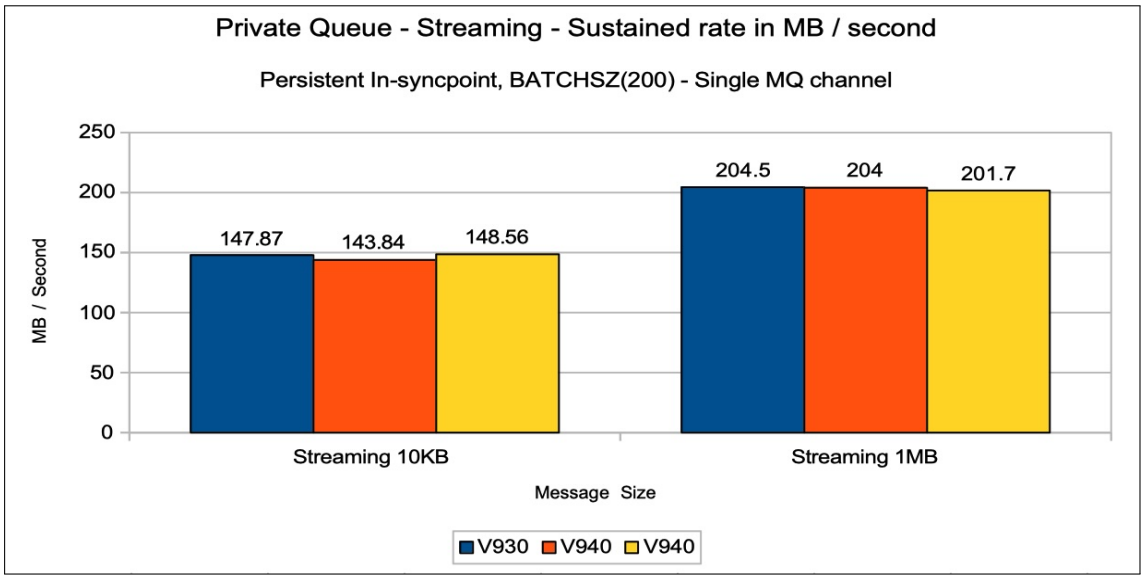
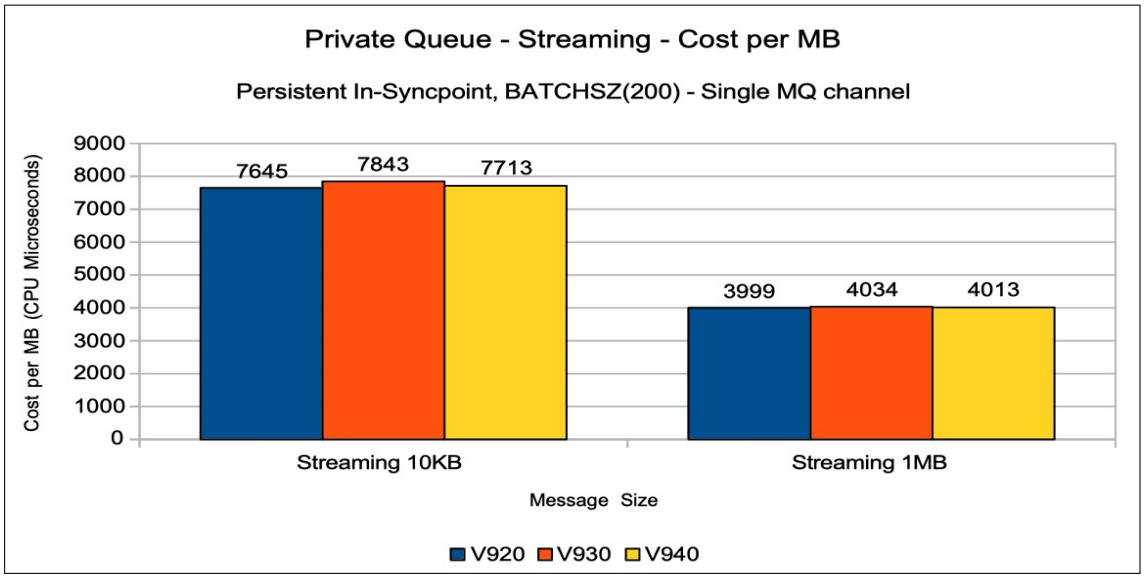


Chart: Cost to stream persistent messages between 2 z/OS queue managers



Moving messages across cluster channels

The regression tests for moving messages across cluster channels are relatively simple, providing an increasing number of destinations for each message put.

The cluster has 8 queue managers - one for the requester workload and the remaining 7 for the server workload. As the test progresses, valid destination queues are defined on an increasing number of the server queue managers. Only the requester queue manager is a full repository - this is not a recommended configuration!

A set of requester tasks is run against one queue manager and the application chooses either bind-on-open or bind-not-fixed. The requester application is written to use an 'MQOPEN, MQPUT, MQCLOSE' model to put messages to the queue. This ensures that the messages are distributed evenly when using the bind-on-open option.

The messages flow across the cluster channels to one of the server queue managers to be processed by the server applications, at which point they are returned to the originating requesting applications.

The measurements are run with 2KB messages and again with 32KB messages. In the 2KB workloads, the influence of the selection of the destination and the general cluster "overhead" is more prevalent.

Bind-on-open

Chart: Bind-on-open transaction rate with 2KB messages

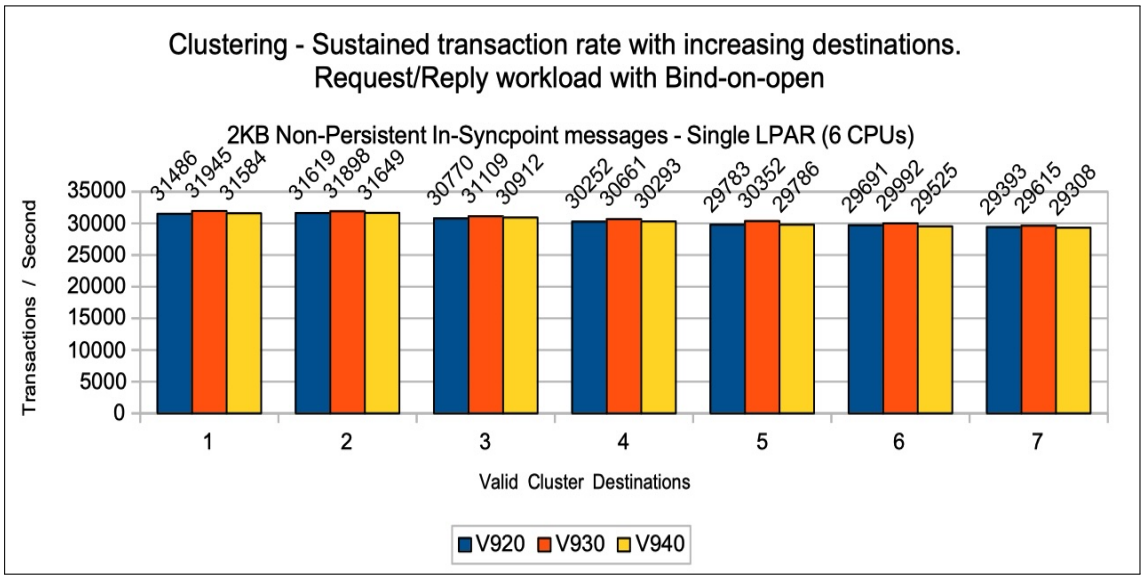


Chart: Bind-on-open transaction cost for 2KB messages

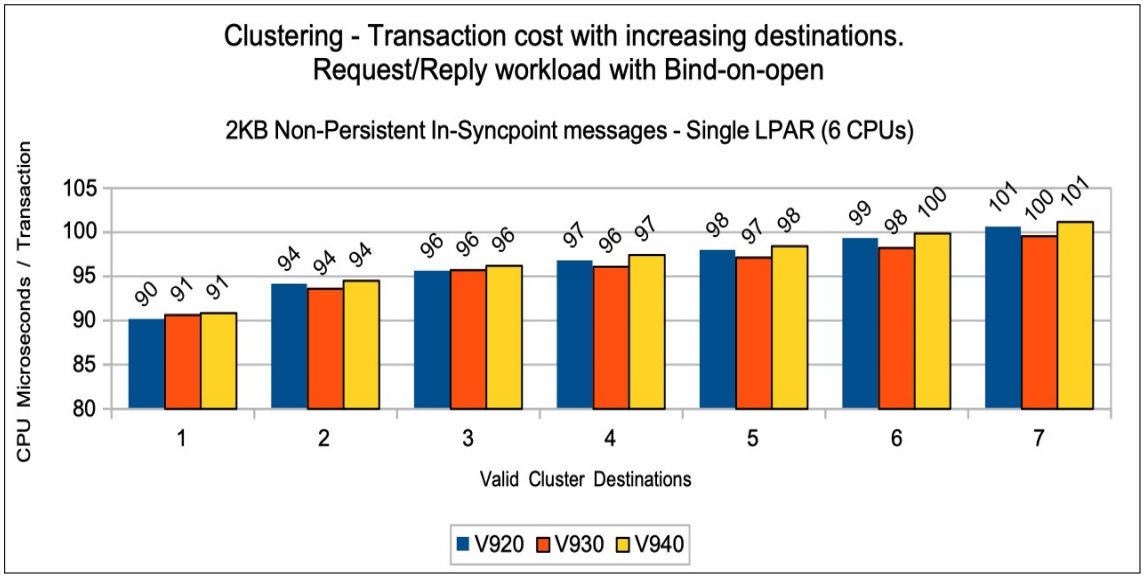


Chart: Bind-on-open transaction rate with 32KB messages

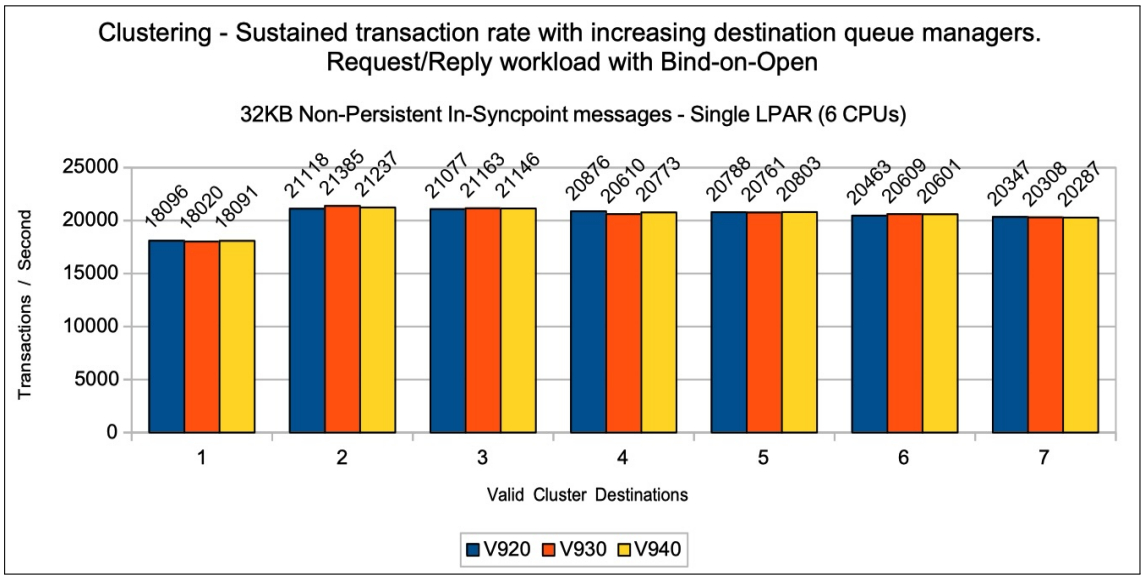
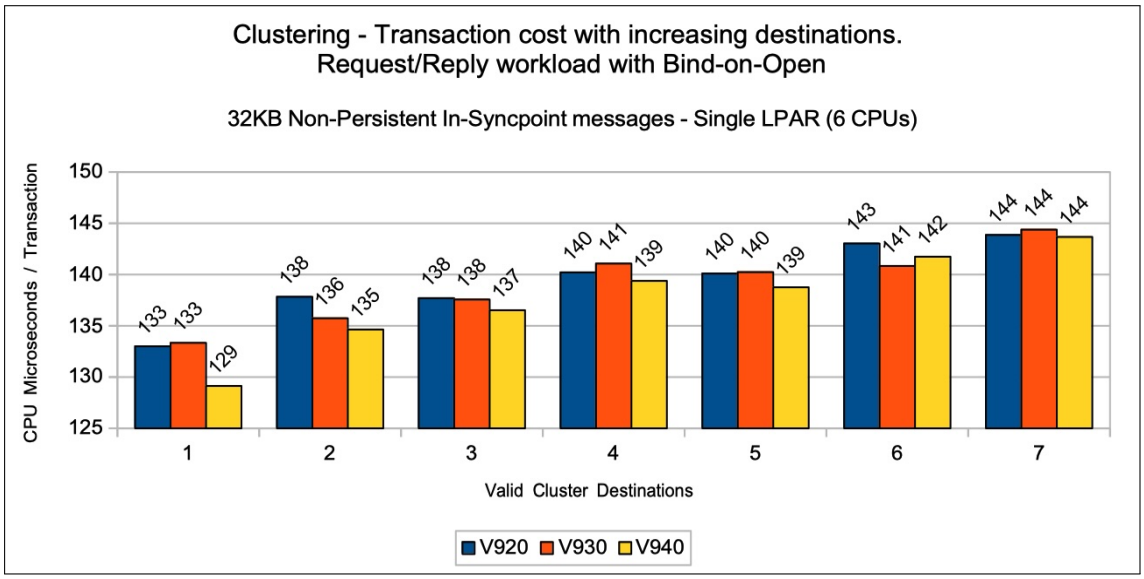


Chart: Bind-on-open transaction cost for 32KB messages



Bind-not-fixed

Chart: Bind-not-fixed transaction rate with 2KB messages

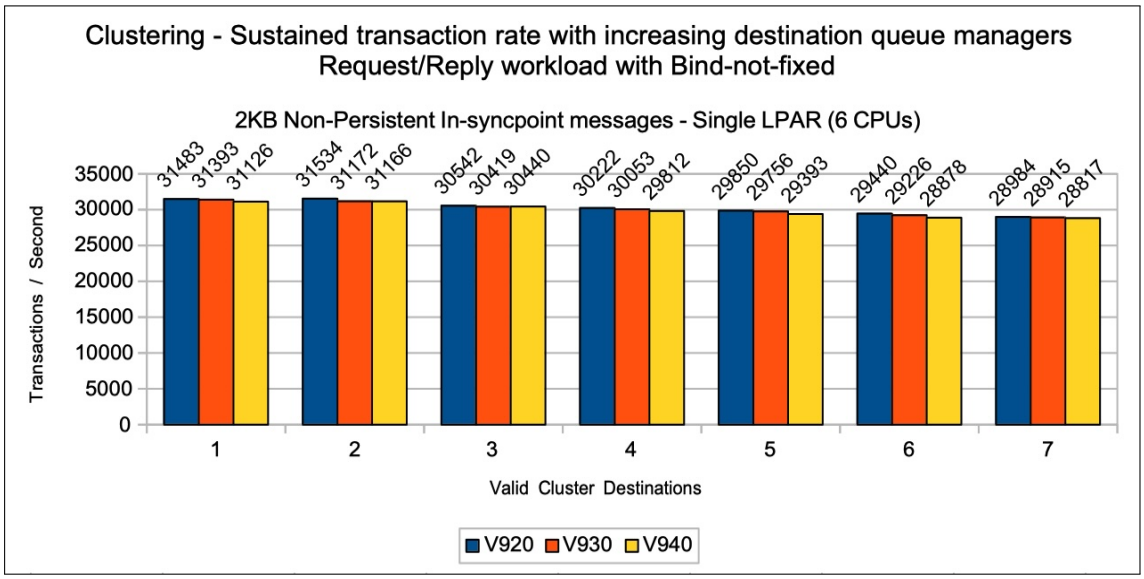


Chart: Bind-not-fixed transaction cost for 2KB messages

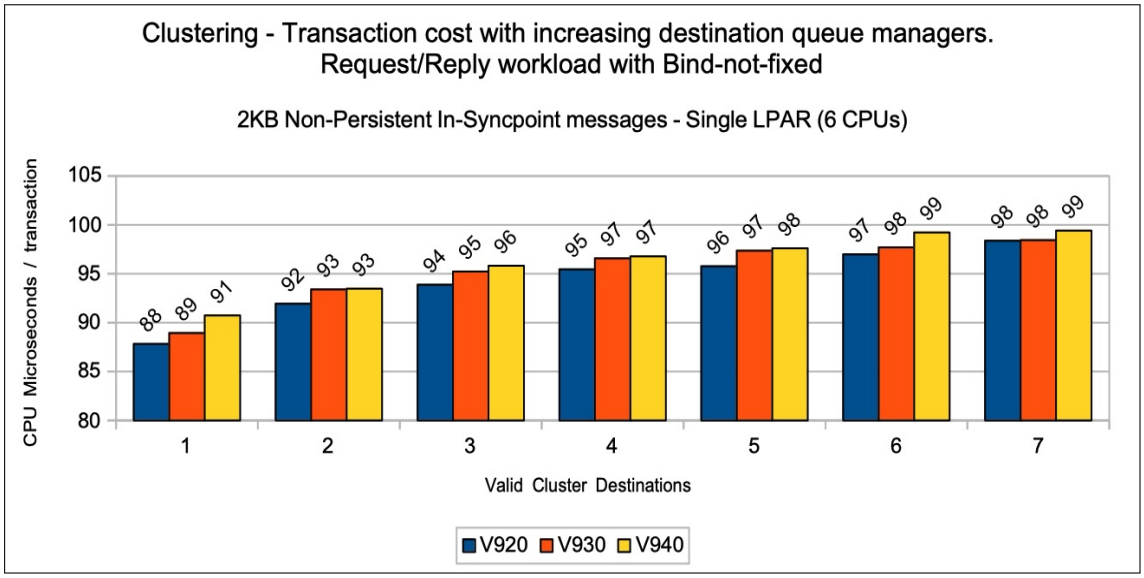


Chart: Bind-not-fixed transaction rate with 32KB messages

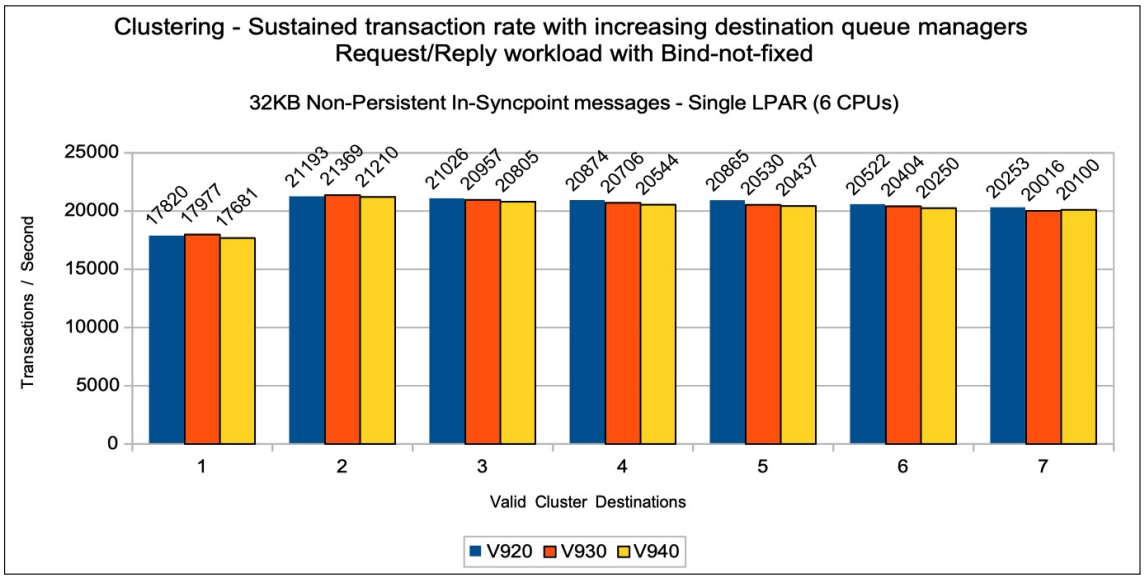
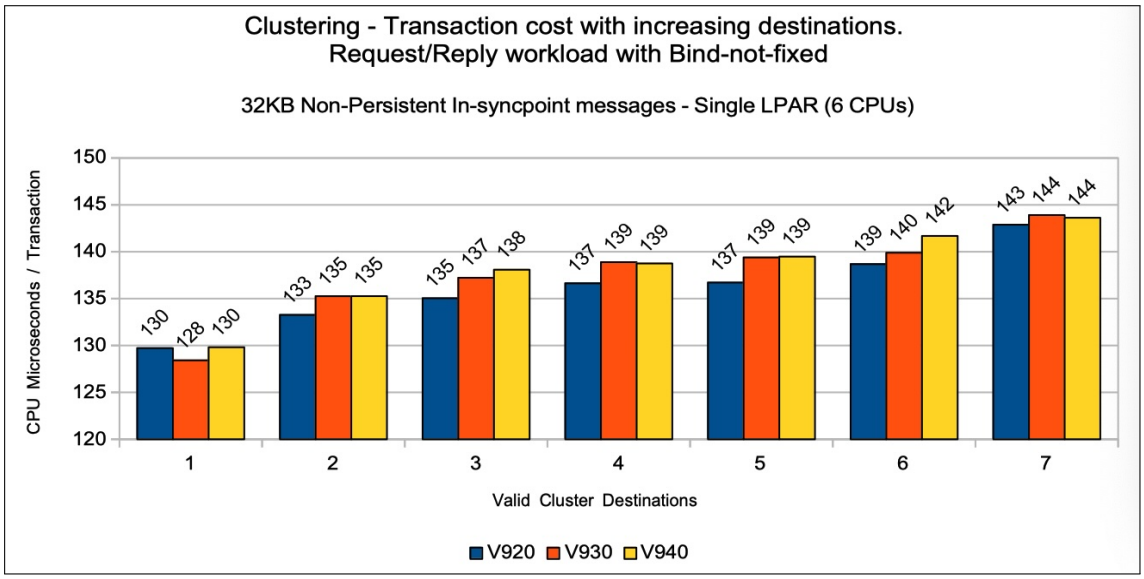


Chart: Bind-not-fixed transaction cost for 32KB messages



Moving messages across SVRCONN channels

The regression tests for moving messages across SVRCONN channels have a pair of client tasks for each queue that is hosted on the z/OS queue manager.

One of the pair of client tasks puts messages to the queue and the other client task gets the messages from the queue. As the test progresses, an increasing number of queues are used, with a corresponding increase in the number of putting and getting client tasks.

Two sets of tests are run, the first uses SHARECNV(0) on the SVRCONN channel to run in a mode comparable to that used pre-version 7.0.0. The second uses SHARECNV(1) so that function such as asynchronous puts and asynchronous gets are used via the DEFPRESP(ASYNC) and DEFREADA(YES) queue options.

Choosing which SHARECNV option is appropriate can make a difference to capacity and the performance which is discussed in more detail in performance report MP16 "Capacity Planning and Tuning Guide" Channel Initiator section.

Note: The rate and costs are based upon the number of MB of data moved per second rather than the number of messages per second.

Client pass through test using SHARECNV(0)

Chart: Throughput rate for client pass through tests with SHARECNV(0)

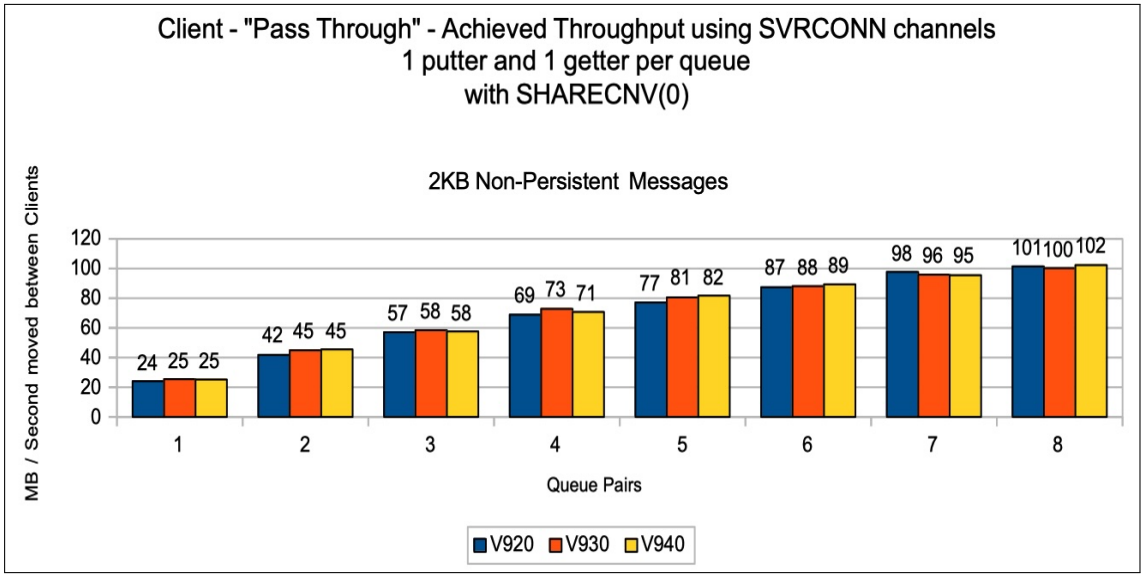
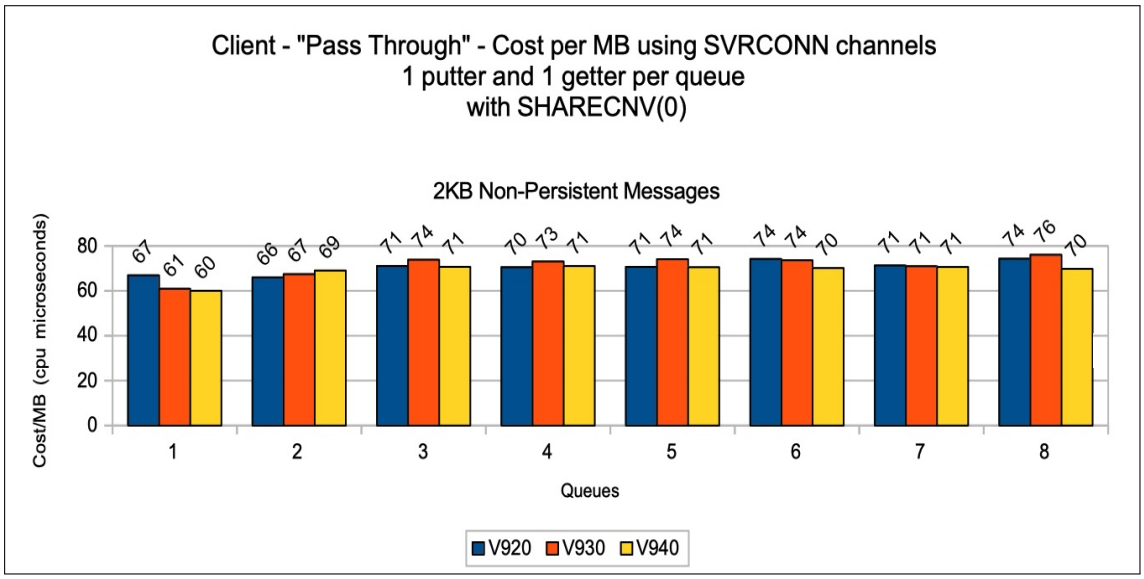


Chart: Cost per MB of client pass through tests with SHARECNV(0)



Client pass through test using SHARECNV(1)

Chart: Throughput rate for client pass through tests with SHARECNV(1)

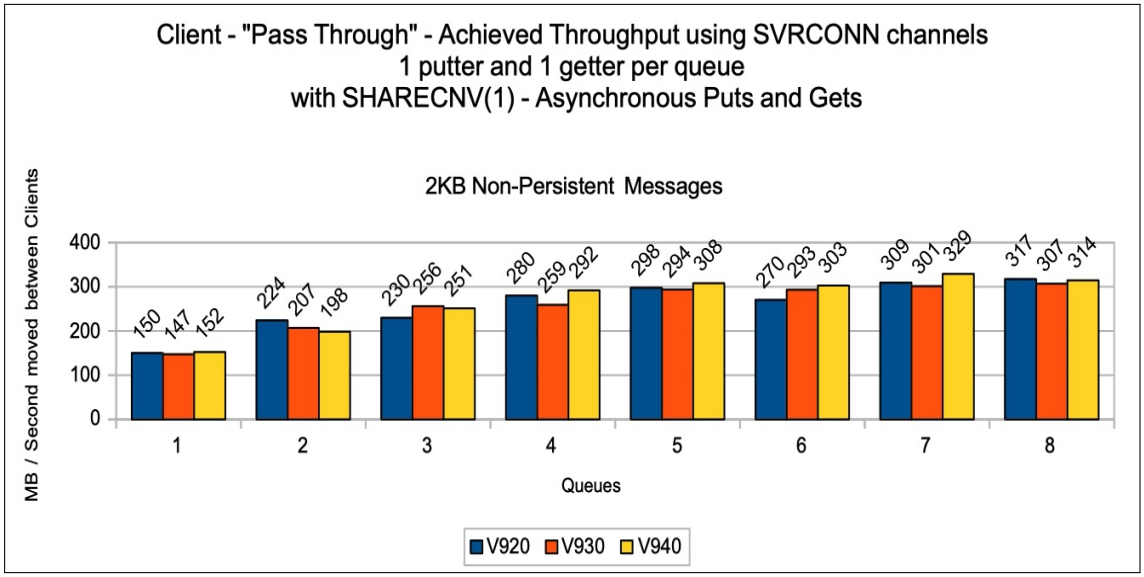
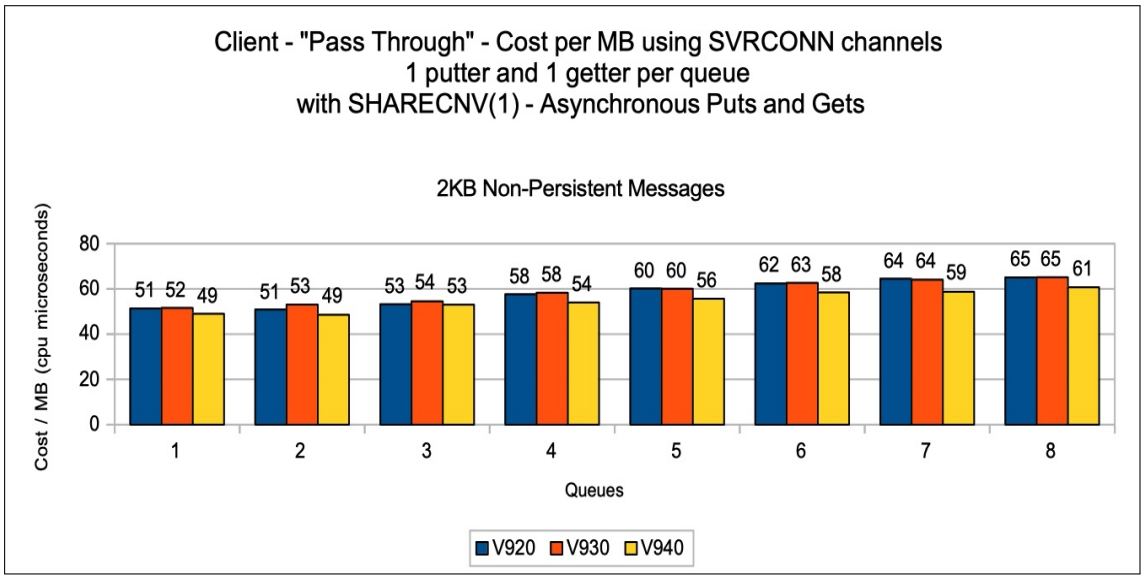


Chart: Cost per MB of client pass through tests with SHARECNV(1)



IMS Bridge

The regression tests used for IMS Bridge use 3 queue managers in a queue sharing group (QSG) each on separate LPARs. A single IMS region is started on 1 LPAR and has 16 Message Processing Regions (MPRs) started to process the MQ workload.

The IMS region has been configured as detailed in performance report MP16 "Capacity Planning and Tuning Guide" using the recommendations in the section "IMS Bridge: Achieving Best Throughput".

Note: 16 MPRs are more than really required for the 1, 2 and 4 TPIPE tests but they are available for consistent configuration across the test suite.

There are 8 queues defined in the QSG that are configured to be used as IMS Bridge queues.

Each queue manager runs a set of batch requester applications that put a 2KB message to one of the bridge queues and waits for a response on a corresponding shared reply queue.

Tests are run using both Commit Mode 0 (Commit-then-Send) and Commit Mode 1 (Send-then-Commit).

Commit mode 0 (commit-then-send)

Chart: IMS Bridge commit mode 0 - Throughput rate

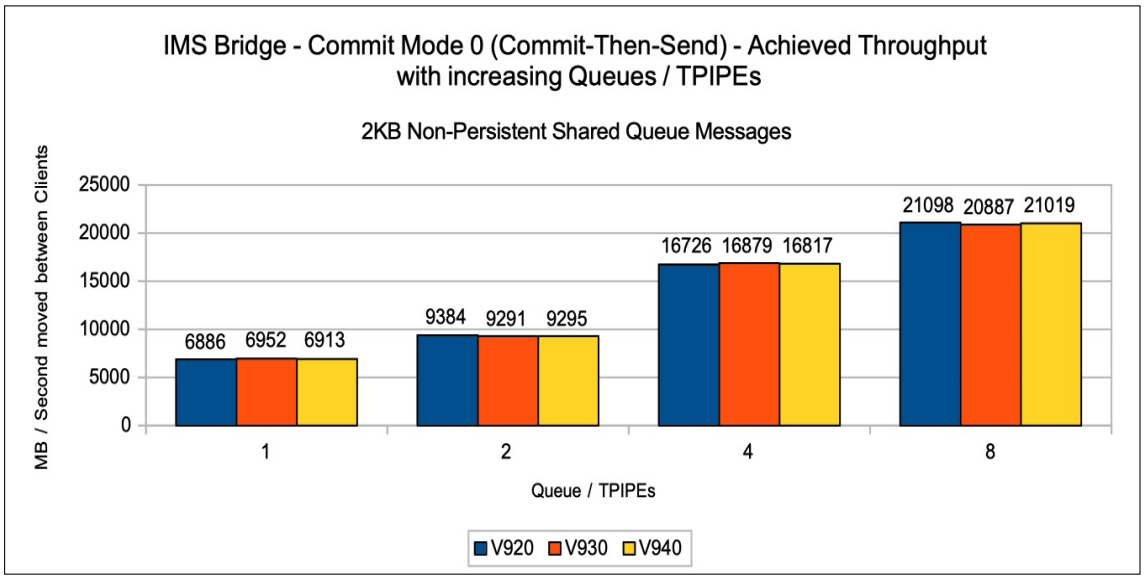
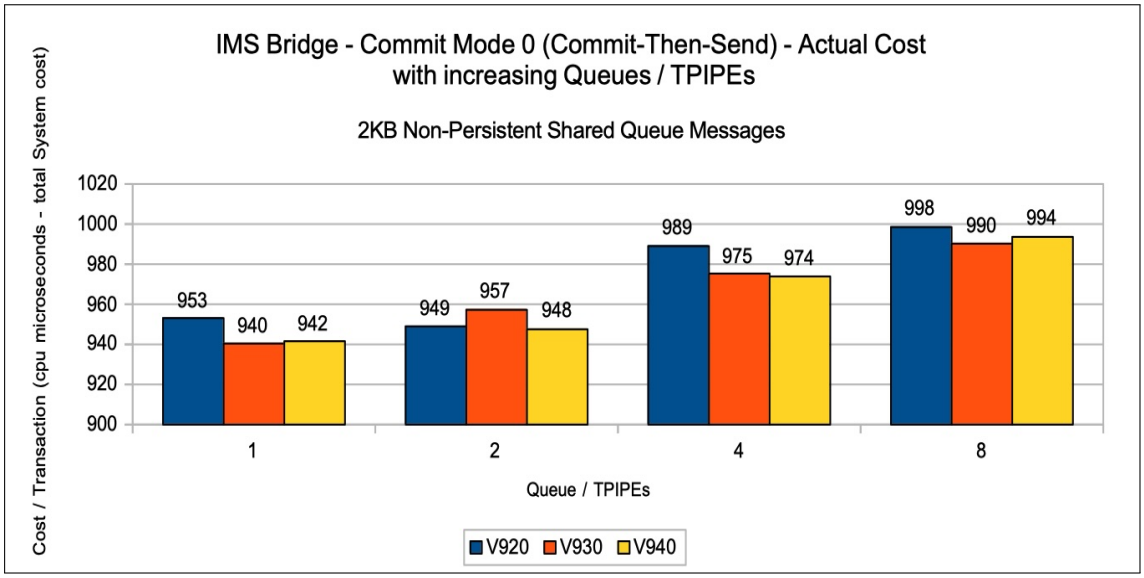


Chart: IMS Bridge commit mode 0 - Transaction cost



Commit mode 1 (send-then-commit)

Chart: IMS Bridge commit mode 1 - Throughput rate

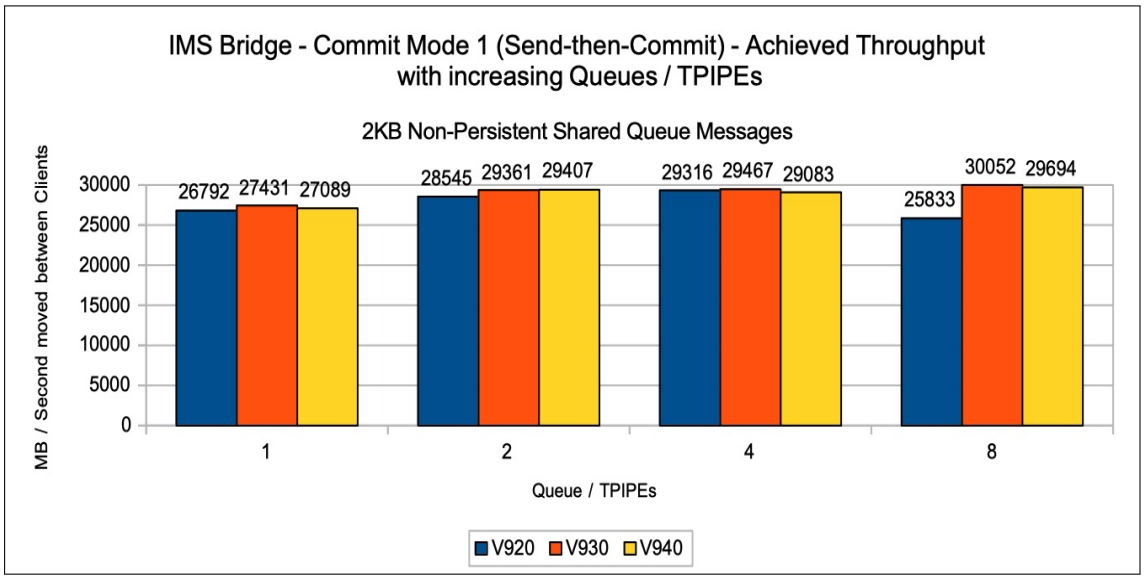
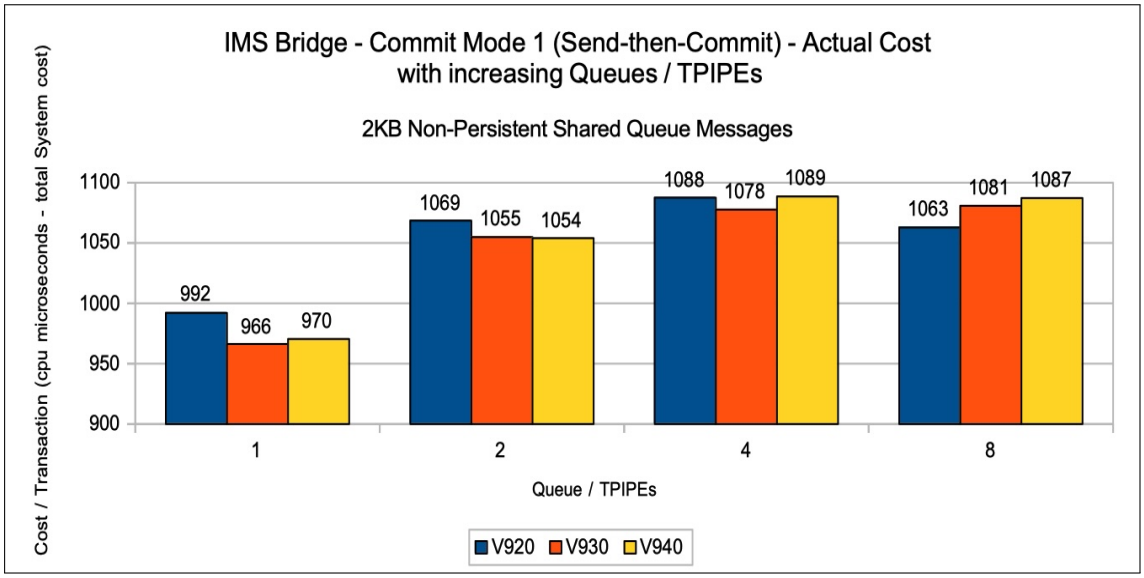


Chart: IMS Bridge commit mode 1 - Transaction cost



Trace

The regression tests for trace cover both queue manager global trace and the channel initiator trace.

Since the performance for 9.2, 9.3 and 9.4 is comparable, the charts in this section show only 9.4 data for the purpose of clarity.

Queue manager global trace

The queue manager global trace tests are a variation on the private queue non-persistent 2KB scalability tests with TRACE(G) DEST(RES) enabled.

To enable a comparison of impact from enabling MQ's global trace, the results from the equivalent measurements without trace enabled are included.

Chart: Queue manager TRACE(G) DEST(RES) - transaction rate

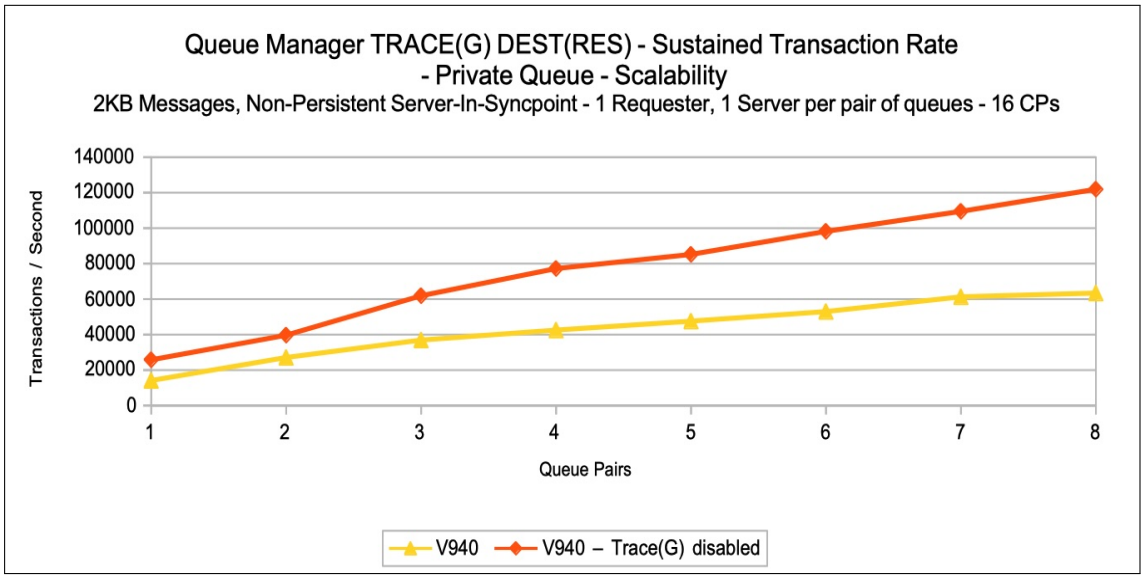
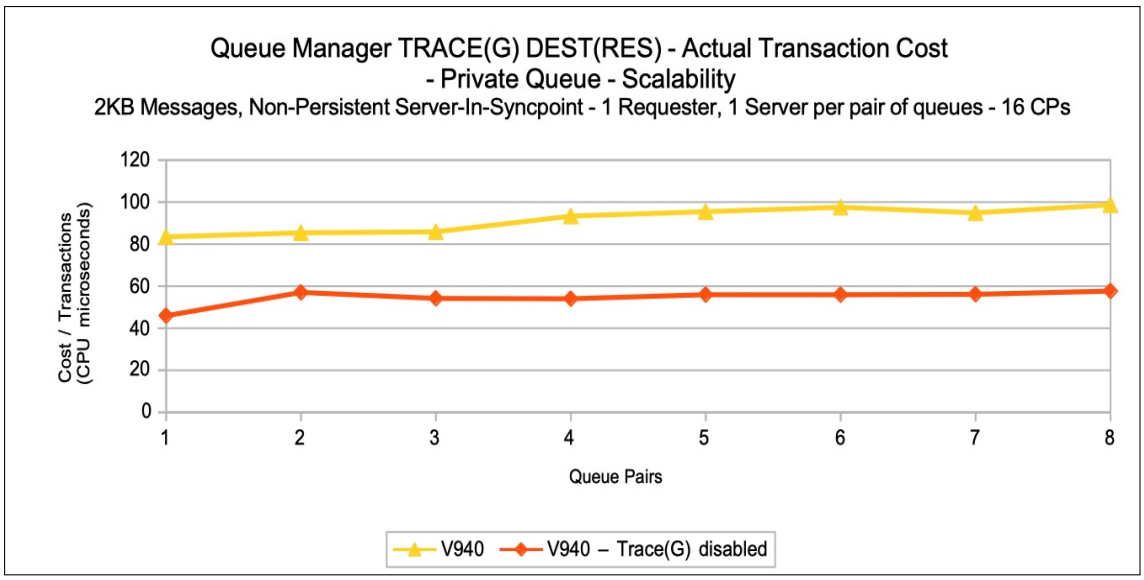


Chart: Queue manager TRACE(G) DEST(RES) - transaction cost



It must be emphasised that whilst the impact from enabling TRACE(G) for these workloads is effectively to double the overall transaction cost, this is primarily due to the high proportion of MQ API work when compared to the overall transaction and the simplistic nature of the application.

Channel initiator trace

The channel initiator trace tests are a variation on the client pass through tests using SHARECNV(0) with TRACE(CHINIT) enabled.

To enable a comparison of impact from enabling MQ's channel initiator trace, the results from the equivalent measurements without trace enabled are included.

Chart: Channel initiator TRACE(CHINIT) - throughput rate

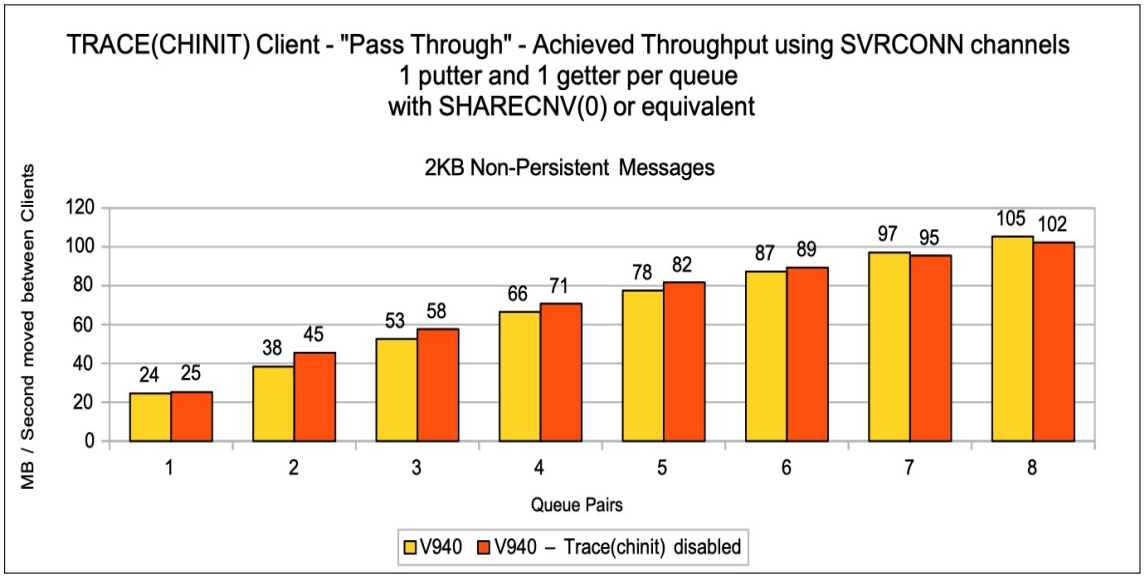
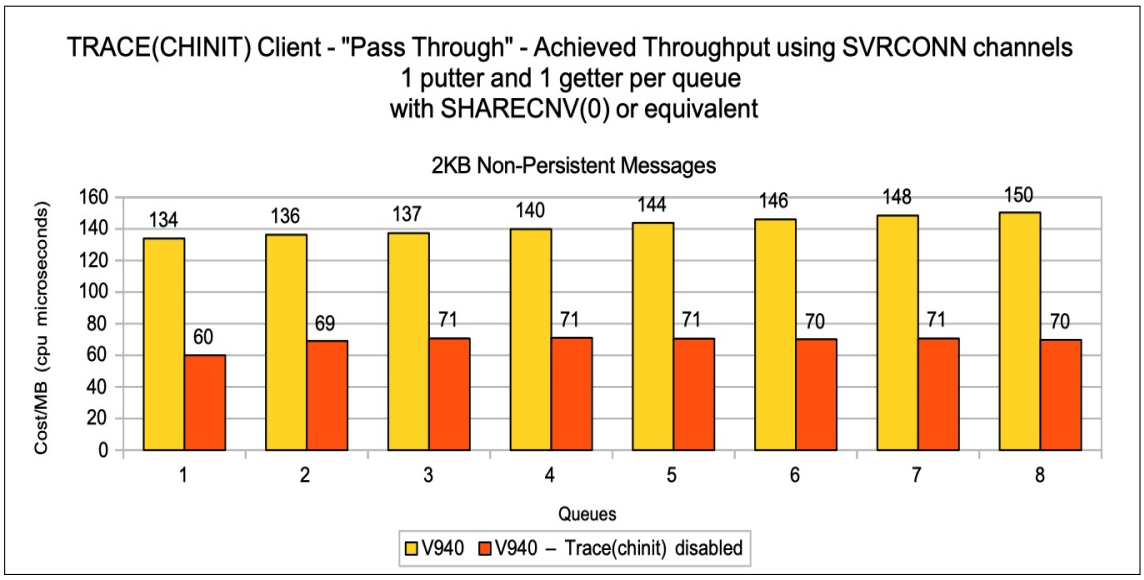


Chart: Channel initiator TRACE(CHINIT) - cost per MB



Note that there was sufficient CPU capacity such that the measurements with TRACE CHINIT enabled were able to achieve similar, or indeed higher, throughput to the measurements where trace was disabled. However, the additional cost would impact the throughput in a CPU constrained system.

Advanced Message Security

IBM MQ Advanced for z/OS 9.4 only

Background

IBM MQ Advanced Message Security (IBM MQ AMS) provides a high level of protection for sensitive data flowing through the MQ network with different levels of protection by using a public key cryptography model.

IBM MQ version 9.0 supplemented the existing two qualities of protection *Integrity* and *Privacy* with a third quality of protection, namely *Confidentiality*.

Integrity protection is provided by digital signing, which provides assurance on who created the message, and that the message has not been altered or tampered with.

Privacy protection is provided by a combination of digital signing and encryption. Encryption ensures that message data is viewable by only the intended recipient, or recipients.

Confidentiality protection is provided by encryption only.

IBM MQ AMS uses a combination of symmetric and asymmetric cryptographic routines to provide digital signing and encryption. Symmetric key operations are very fast in comparison to asymmetric key operations, which are CPU intensive and whilst some of the cost may be offloaded to cryptographic hardware such as Crypto Express8S, this can have a significant impact on the cost of protecting large numbers of messages with IBM MQ AMS.

- **Asymmetric cryptographic routines** as used by Integrity and Privacy

  For example, when putting a signed message the message hash is signed using an asymmetric key operation. When getting a signed message, a further asymmetric key operation is used to verify the signed hash. Therefore, a minimum of two asymmetric key operations are required per message to sign and verify the message data. Some of this asymmetric cryptographic work can be offloaded to cryptographic hardware.

- **Asymmetric and symmetric cryptographic routines** as used by Privacy and Confidentiality

  When putting an encrypted message, a symmetric key is generated and then encrypted using an asymmetric key operation for each intended recipient of the message. The message data is then encrypted with the symmetric key. When getting the encrypted message the intended recipient needs to use an asymmetric key operation to discover the symmetric key in use for the message. The symmetric key work cannot be offloaded to cryptographic hardware but will be performed in part by CPACF processors.

All three qualities of protection, therefore, contain varying elements of CPU intensive asymmetric key operations, which will significantly impact the maximum achievable messaging rate for applications putting and getting messages.

Confidentiality policies do, however, allow for symmetric key reuse over a sequence of messages.

Key reuse can significantly reduce the costs involved in encrypting a number of messages intended for the same recipient or recipients.

For example, when putting 10 encrypted messages to the same set of recipients, a symmetric key is generated. This key is encrypted for the first message using an asymmetric key operation for each of the intended recipients of the message.

Based upon policy controlled limits, the encrypted symmetric key can then be reused by subsequent messages that are intended for the same recipient(s). An application that is getting encrypted

messages can apply the same optimization, in that the application can detect when a symmetric key has not changed and avoid the expense of retrieving the symmetric key.

In this example 90% of the asymmetric key operations can be avoided by both the putting and getting applications reusing the same key.

IBM MQ for z/OS 9.1 further optimised the performance of AMS qualities of protection, for all 3 levels, with the most optimal offering from a performance perspective being AMS Confidentiality.

As was discussed in blog Impact of key-size on AMS performance, the AMS measurements in the MQ performance reports have historically used 1024-bit key sizes in the certificate. This performance report has moved to using 2048-bit keys as these are the system default.

AMS regression test configuration

A simple request/reply workload is configured using a single z/OS queue manager with a pair of request and reply queues protected by an AMS security policy. In this instance there is 1 requester and 1 server task that run as long-running batch tasks.

The AMS policies defined are:

- **Integrity** Messages digitally signed with SHA256.
- **Privacy** Messages digitally signed with SHA256 and encrypted with AES256.
- **Confidentiality** Messages encrypted with AES256 and the key can be reused:
  - For 1 message
  - For 32 messages
  - For 64 messages
  - For an unlimited number of messages whilst the application remains connected.

Impact of AMS policy type on 2KB request/reply workload

Chart: Transaction rate for 2KB non-persistent workload with AMS protection applied to request and reply queues

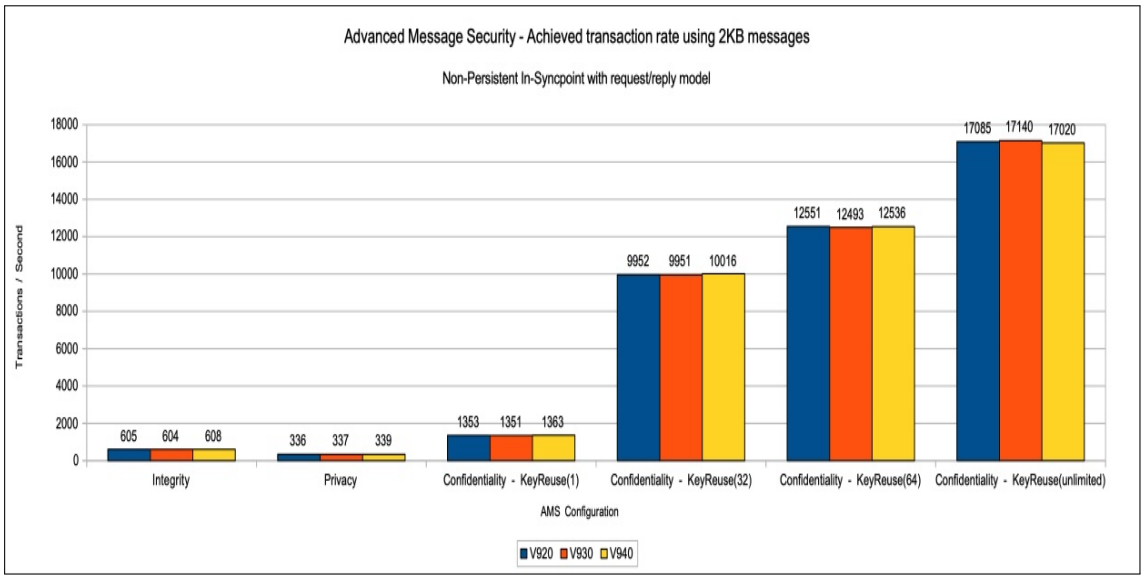
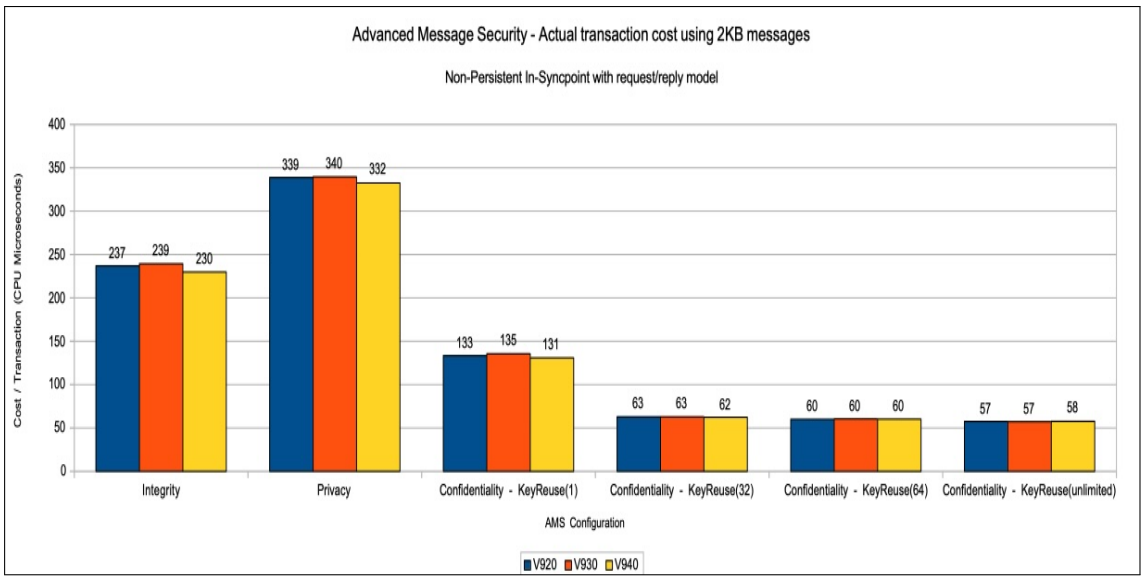


Chart: Transaction cost for 2KB non-persistent workload with AMS protection applied to request and reply queues



Impact of AMS policy type on 64KB request/reply workload

Chart: Transaction rate for 64KB non-persistent workload with AMS protection applied to request and reply queues

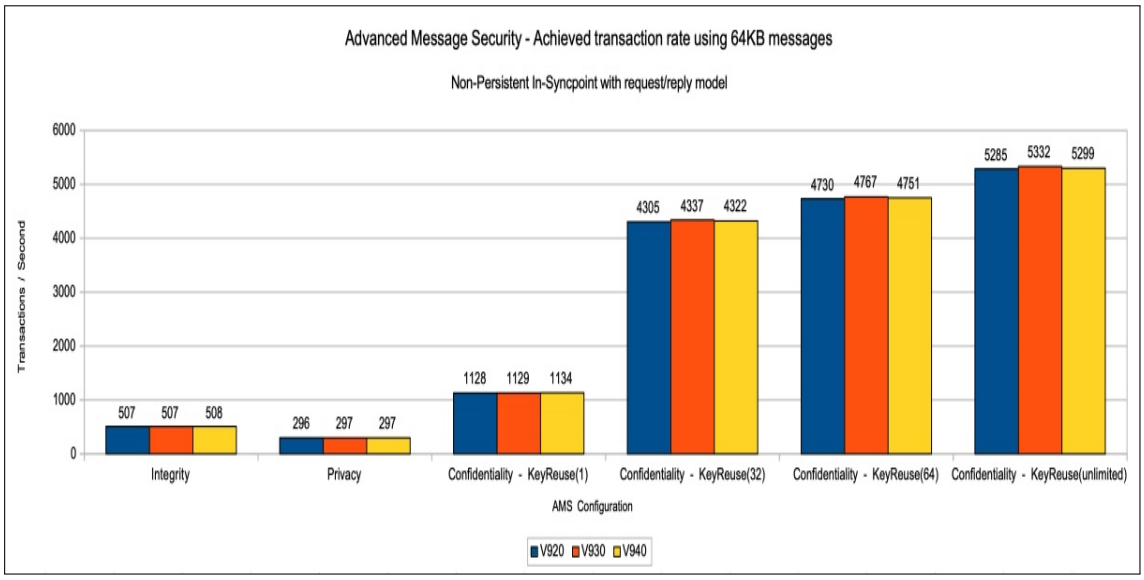
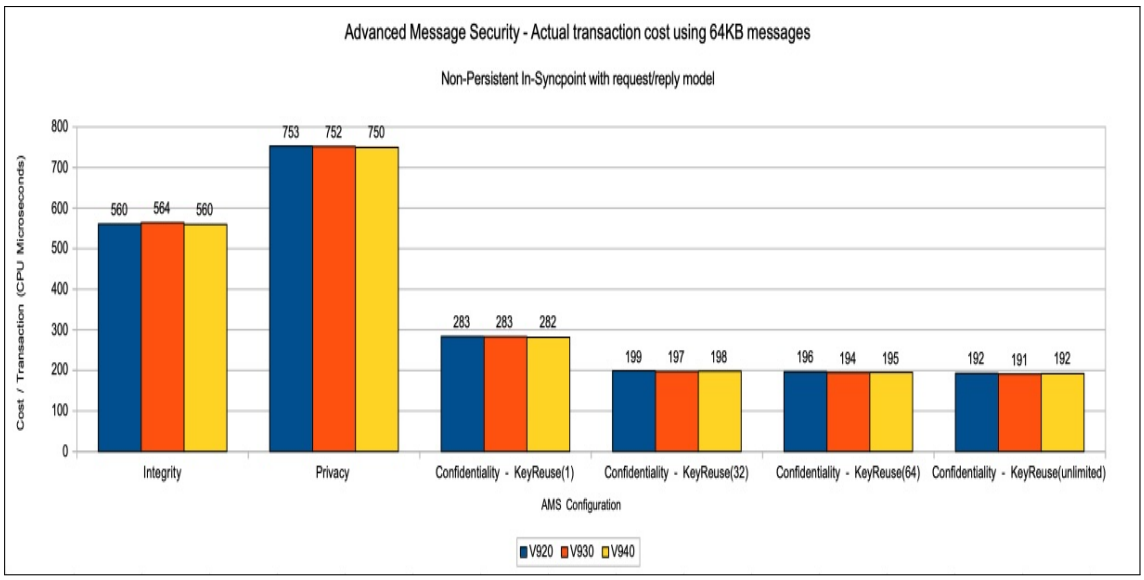


Chart: Transaction cost for 64KB non-persistent workload with AMS protection applied to request and reply queues



Impact of AMS policy type on 4MB request/reply workload

Chart: Transaction rate for 4MB non-persistent workload with AMS protection applied to request and reply queues

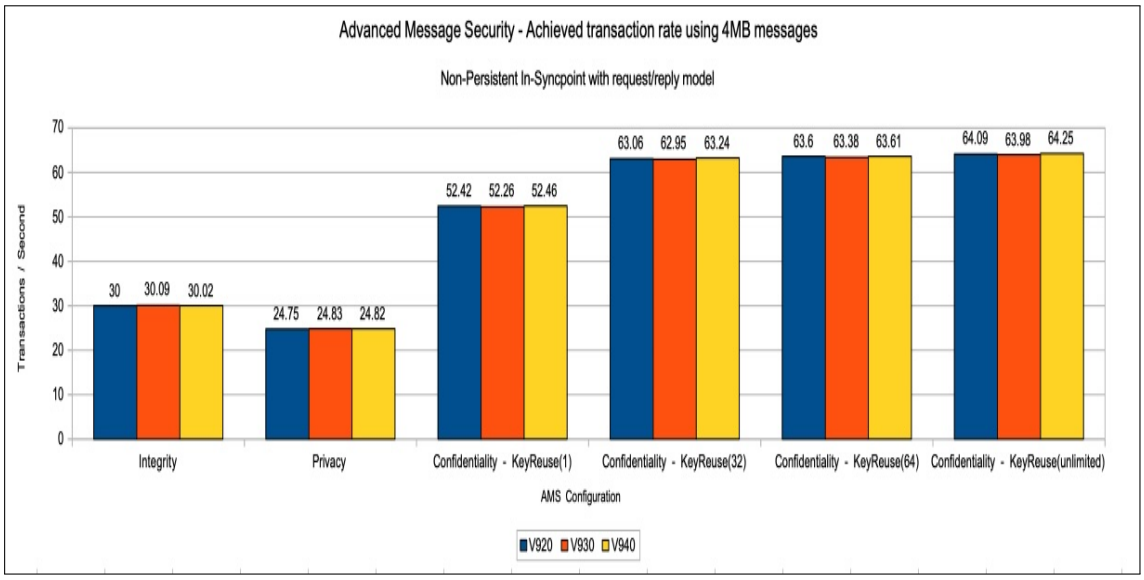
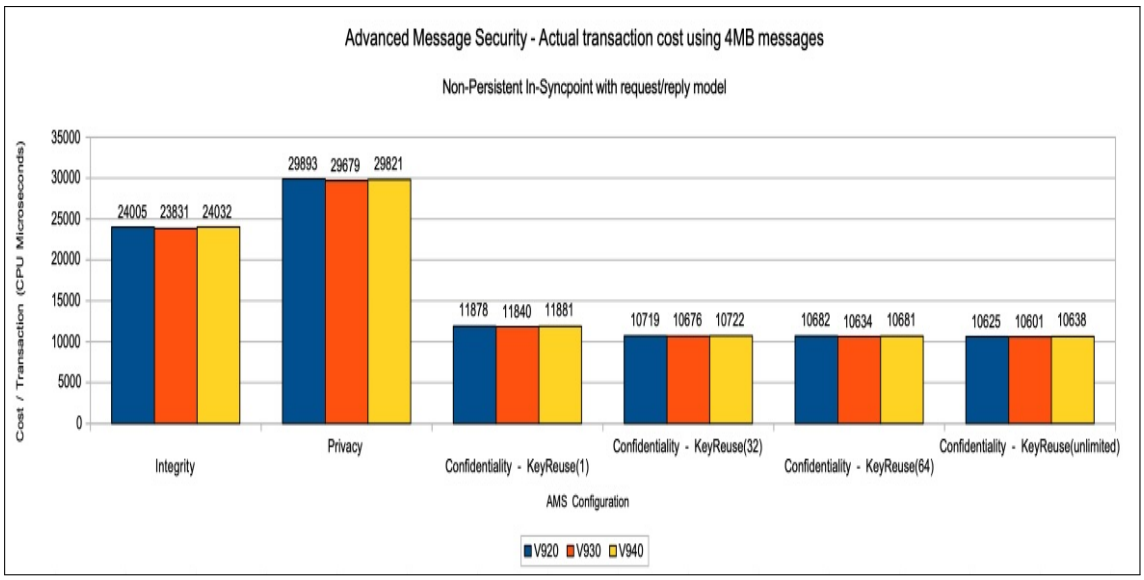


Chart: Transaction cost for 4MB non-persistent workload with AMS protection applied to request and reply queues



Appendix B

System configuration

The majority of the measurements in this report were run on our **IBM MQ Performance Sysplex running on IBM z16 (3931)** configured thus:

LPAR 1: 1-32 dedicated CP processors, plus 2 zIIP, 144GB of real storage.

LPAR 2: 1-3 dedicated CP processors, 48GB of real storage.

LPAR 3: 1-10 dedicated CP processors, plus 2 zIIP, 48GB of real storage.

Default Configuration:

3 dedicated processors on each LPAR, where each LPAR running z/OS 3.1 FMID HBB77E0.

Coupling Facility:

- Internal coupling facility with 4 dedicated processors.
- Coupling Facility running CFCC level 25 service level 02.56.
- Dynamic CF Dispatching off.
- 3 x ICP links between each z/OS LPAR and CF

DASD:

- FICON Express 32S connected DS8950F.
- 4 dedicated channel paths (shared across sysplex)
- HYPERPAV enabled.

System settings:

- zHPF disabled by default.
- HIPERDISPATCH enabled by default.
- LPARs 1 and 3 configured with different subnets such that tests moving messages over channels send data over 10GbE performance network.
    - SMC-R enabled by default between LPARs 1 and 3.
    - SMC-D enabled by default between LPARs 1 and 3.
- zEDC compression available by default - used with MQ channel attribute COMPMSG(ZLIBFAST)..
- Crypto Express8 features configured thus:

- 1 x Accelerator, shared between LPARs 1,2 and 3.
- 2 x Coprocessor on LPAR1.
- 1 x Coprocessor on LPAR2.
- 2 x Coprocessor on LPAR3.

IBM MQ trace status:

- TRACE(GLOBAL) disabled.
- TRACE(CHINIT) disabled.
- TRACE(S) CLASS(1,3,4) enabled.
- TRACE(A) CLASS(3,4) enabled.

General information:

- Client machines:
  - 2 x IBM z 15 (8561) running RHEL 8.9 each with 128GB memory and 6 CPUs configured with 2 threads per core giving 12 CPUs per LPAR.
  - Configured with Kafka 3.7.
- Client tests used a 10GbE performance network.
- Other IBM products used:
  - IBM CICS TS 6.1.
  - Db2 for z/OS version 13.
  - IMS 15.3
  - IBM MQ for z/OS 9.3 with latest service applied as of June 2024.
  - IBM MQ for z/OS 9.2 with latest service applied as of June 2024.