Performance report – SupportPac MP1H
WebSphere MQ for z/OS version 7.1.0

Version 1.2

WebSphere 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.

Third edition, May 2013.

This edition applies to WebSphere MQ for z/OS version 7.1.0 (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 **WebSphere MQ for z/OS V7.1.0**. The information is not intended as the specification of any programming interfaces that are provided by WebSphere MQ. Full descriptions of the WebSphere MQ facilities are available in the product publications. It is assumed that the reader is familiar with the concepts and operation of WebSphere 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

EXPORT REGULATIONS

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

Summary of Amendments

Date	Changes
April 2012	Initial Version
November 2012	Update to backup and recovery of structures.
May 2013	Correction of shared queue costs when using multiple queue managers.

Table of Contents

Performance highlights

This report focuses on performance change since the previous versions (v6.0.0, v7.0.0 and v7.0.1) and on the performance of new function in this release.

SupportPac MP16 “Capacity Planning and Tuning for WebSphere MQ for z/OS” will continue to be the repository for ongoing advice and guidance learned as systems increase in power and experience is gained.

Existing function

General statement of regression

CPU costs and throughput are not significantly different in version 7.1.0 for typical messaging workloads.

A user can see how that statement has been determined by reviewing details of the regression test cases in Appendix A.

Some areas where scalability was improved have led to a single queue manager being able to process in excess of 1 million non-persistent messages per second on a 30-way LPAR of a zEnterprise 196 (2817). For more details of this measurement, see Appendix A “Regression – Private Queue”.

Storage usage

Virtual storage constraint relief has not been a primary focus of this release, however new function in version 7.1.0 such as CFLEVEL(5) Shared Message Data Set offload capability and CHLAUTH cache uses 64-bit storage.

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 maximum 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 is used by the queue manager for buffer pools.

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

Initial CSA usage

CSA usage increased by less than 0.2MB between V6.0.0 and V7.1.0 when similarly configured queue managers are started.

CSA usage per connection

In Websphere MQ for z/OS version 6, the CSA usage was typically 2.4KB per connection.

In Websphere MQ for z/OS version 7.1 the CSA usage has increased slightly:

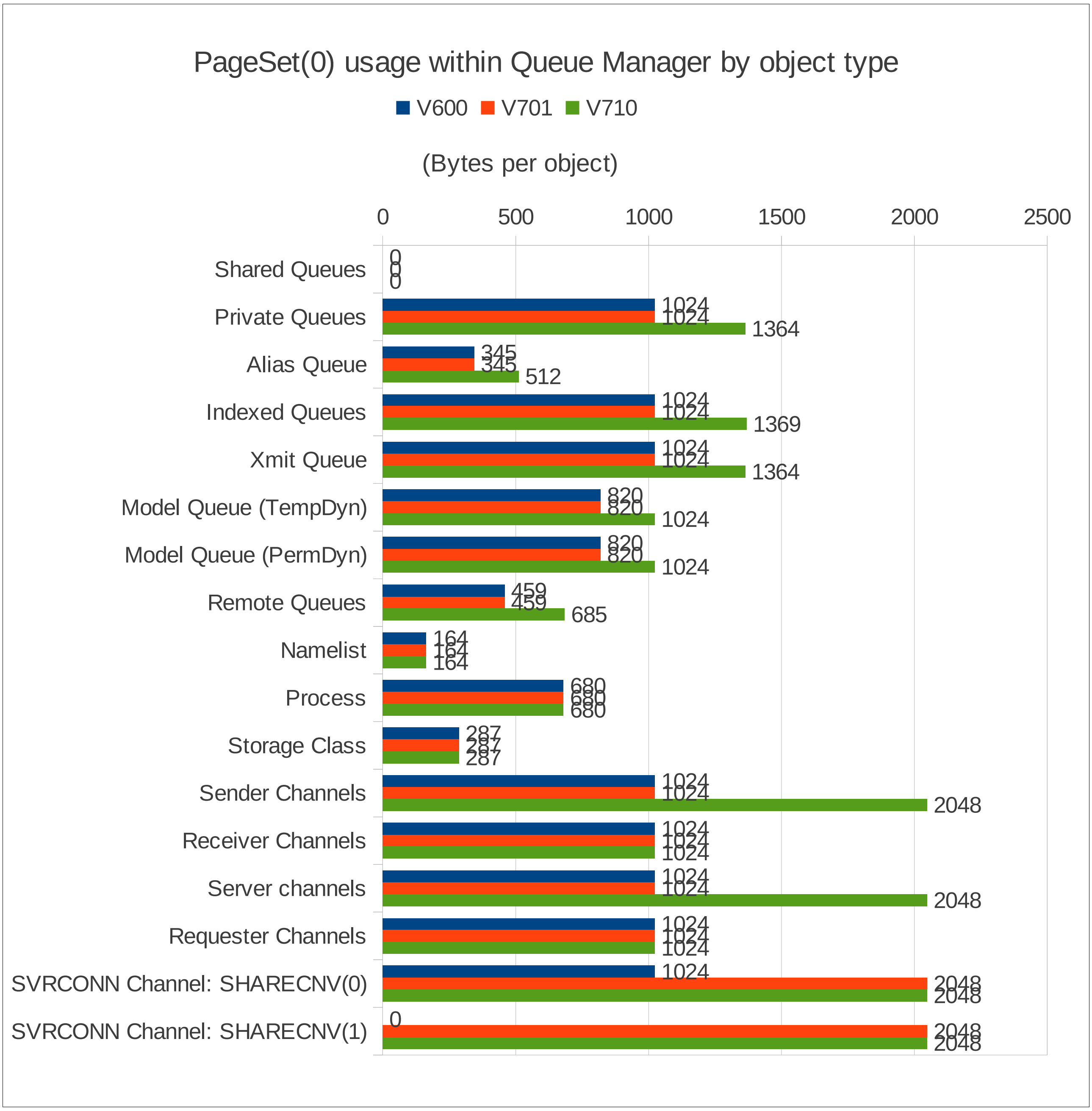
- For local connections, MCA channels and SVRCONN channels with SHARECNV(0), CSA usage is 2.43KB per connection.
- For SVRCONN channels with SHARECNV(1), CSA usage is approximately 4.85KB per connection
- For SVRCONN channels with SHARECNV(10), CSA usage is approximately 2.7KB per connection (based on 10 clients sharing a 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.

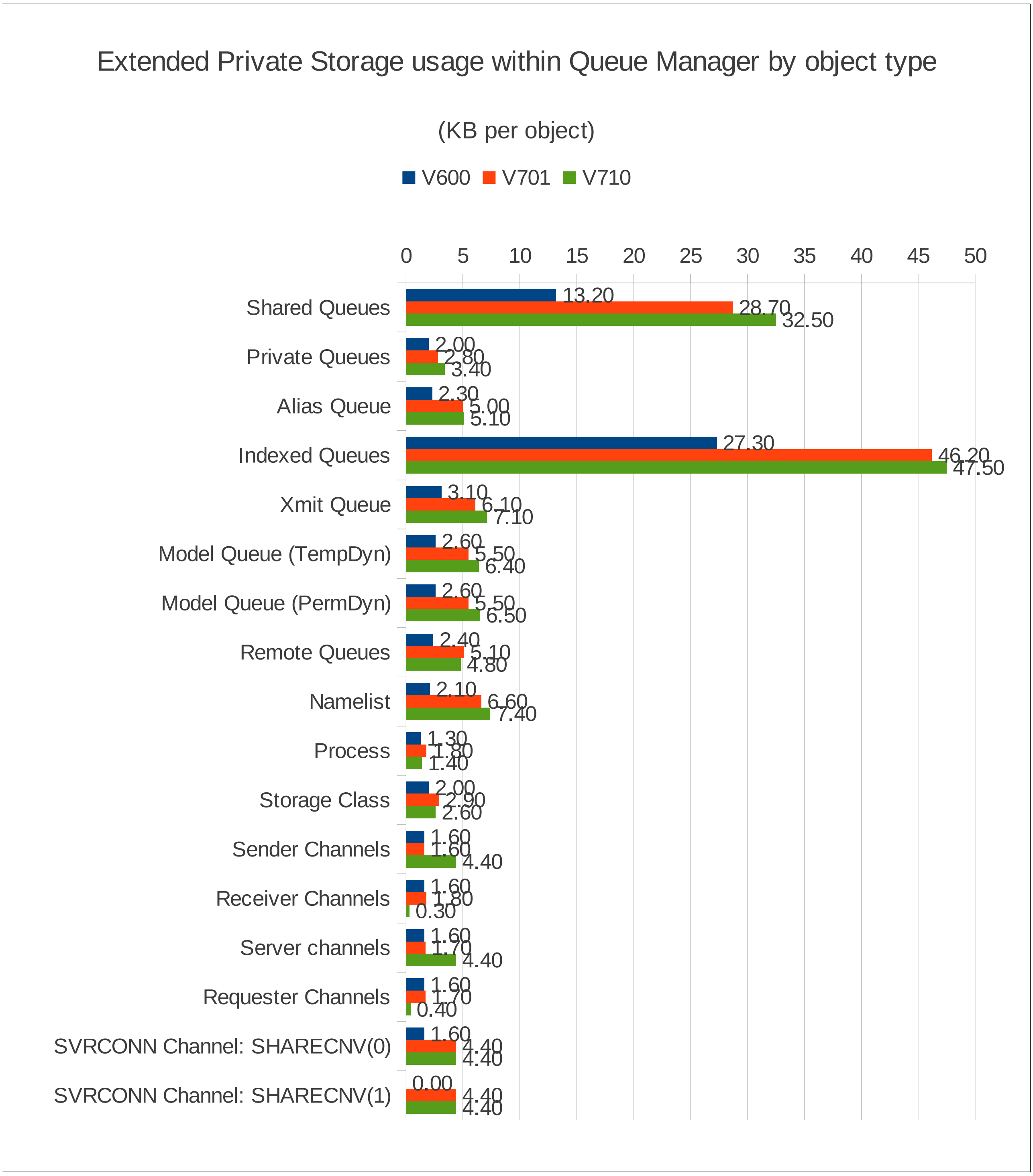
PAGESET(0) Usage

Chart 1: Pageset usage by object type



Virtual Storage Usage

Chart 2: Virtual Storage Usage by object type



NOTE: CHLAUTH objects are cached in 64-bit storage.

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 stored on storage taken from that available below the 2GB bar. In the following examples, only allocations from below the 2GB bar are reported.

Typical storage usage is around 22KB 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 36.5KB
- Where connection is over a SHARECNV(10) channel and has a CURSHCNV of 10, the usage is 23KB per connection (228KB per channel)
- When connection is over a SHARECNV(1) channel to shared queues - either CFLEVEL(4) or CFLEVEL(5) backed by SMDS, the storage is 43.5KB, 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 used 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).

The table below shows the typical 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 size of the footprint and consideration as to the setting should be taken. Guidance on the SHARECNV attribute can be found in SupportPacs MP16 "Capacity Planning and Tuning Guide" and MP1F "WebSphere MQ for z/OS V7.0 Performance Report".

		Channel initiator footprint (KB / SVRCONN channel) **Message size (KB)**			
WebSphere MQ release	**SHARECNV**	**1**	**10**	**32**	**64**
V600	N/A	155	155	210	-
V701	0	98	113	176	207
V710	0	88	102	168	199
V701	1	176	198	290	353
V710	1	173	194	285	349
V701	10	264	284	708	1069
V710	10	253	276	698	1055

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

How many 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**	**1MB**
Footprint per channel (channel initiator)	103	110	121	1134
Overhead of message size increase on 1KB messages		+7K	+18K	+1029K

New function

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

1. CFLEVEL(5) - Shared Message Data Sets (SMDS)
2. V6COMPAT mode
3. CHLAUTH overhead
4. 64-bit global trace
5. Topic scavenger (queue manager “TREELIFE” attribute)

CFLEVEL(5) – Shared message data sets (SMDS)

What is the problem being addressed?

WebSphere MQ for z/OS allows messages to be stored on shared queues. These messages are typically stored in a z/OS Coupling Facility (CF). This allows all the queue managers in the Queue Sharing Group (QSG) to access the messages.

Using queue sharing, WebSphere MQ applications are scalable and highly available.

Prior to WebSphere MQ for z/OS version 7.1.0, all messages put to shared queues that were larger than 63KB were stored on DB2 tables, with entries associated to those messages stored in the CF.

Messages that exceed 63KB see a significant drop in performance when accessing the messages involves DB2.

Coupling Facility storage is expensive, often meaning that capacity is restricted by budgetary constraints.

WebSphere MQ for z/OS version 7.1.0 introduces a new level of CF structure – CFLEVEL(5) to store large messages in Shared Message Data Sets (SMDS), instead of DB2 for messages larger than 63KB.

The term “offload” is used to mean the MQPUT-time process which stores the message payload in either SMDS or DB2 and a reference to this payload in the CF. The CF reference is used for locating the message, message selection and locking.

In addition, a 3-tiered message size offload threshold is introduced to increase the capacity of the CF.

WebSphere MQ for z/OS version 7.1.0 still requires DB2 to be available to hold definitions for shared and group objects and status.

Why use CFLEVEL(5)?

There are 2 primary reasons to use CFLEVEL(5) structures:

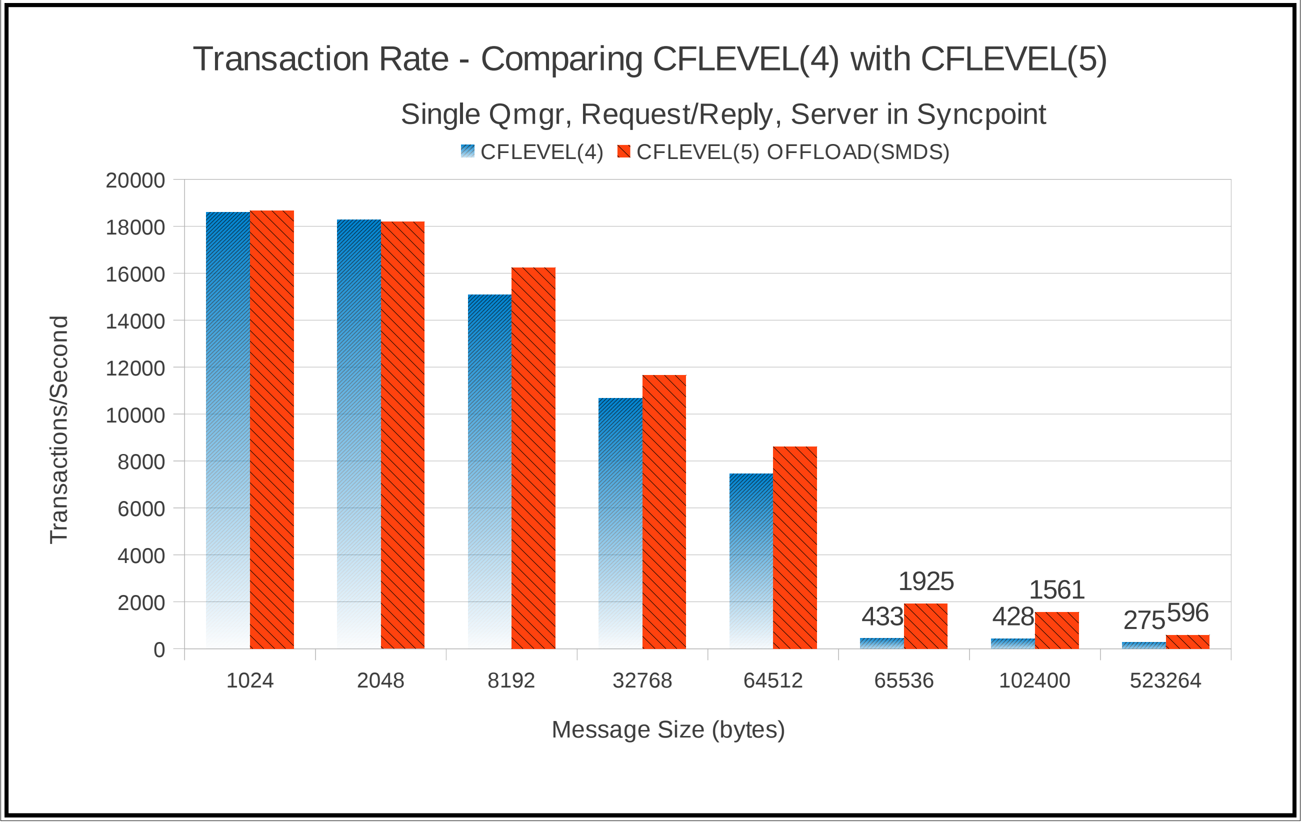
1) Performance of storing and accessing large messages on shared queues
2) Increasing the capacity of the Coupling Facility.

Performance of storing and accessing large messages on shared queues

Queues defined to CF structures at CFLEVEL(5) have large message data offloaded to shared message data sets (SMDS) instead of DB2, and the offload threshold and size parameters are respected.

Prior to CFLEVEL(5), messages put onto shared queues that were larger than 63KB were offloaded onto DB2. This offloading process was both relatively slow and expensive as can be seen in the following chart.

Chart 3: Achieved transaction rate for 1 queue manager in QSG



There is still a marked step in performance when the message size increases from 63KB to 64KB, but whilst the achieved transaction rate for messages less than 64KB is comparable between CF levels 4 and 5, once the messages are 64KB or larger, there is a 3-times increase in transaction rate in this simple configuration.

Increasing the capacity of the Coupling Facility

CFLEVEL(5) provides the ability to increase the capacity of the CF by implementing a 3-tiered offload procedure.

Implementing tiered thresholds allows higher capacity whilst not penalising performance until the CF resource becomes constrained.

By default, the CFLEVEL(5) structure will offload messages greater than 63KB to the shared message data set.

In addition, there are 3 default thresholds:

1. Offload all messages larger than 32KB (including headers) when the structure is 70% full.
2. Offload all messages larger than 4KB (including headers) when the structure is 80% full
3. Offload all messages when the structure is 90% full.

Example: Consider a 0.5GB structure that has only 16KB messages.

CFLEVEL(4)
A 16KB message would require 1 entry and 66 elements (applying algorithm detailed in the "Shared Queue Setup Considerations - Application Structures of SupportPac MP16).

A 0.5GB structure would support approximately 24,000 messages (with ALLOWAUTOALT=YES)

CFLEVEL(5)
16KB messages stored in the CF would still require 1 entry and 66 elements, whereas all offloaded messages use 1 entry and 2 elements.

For example, 16KB messages would not be offloaded until the structure reaches 80% full.
This means that **17,101** messages are stored in their entirety in the CF. Upon reaching the 80% threshold, remaining messages are offloaded to the SMDS datasets. Provided the SMDS datasets are large enough, the CF would then be able to store a total of **158,166** messages.

Is there benefit in offloading all messages

(Or how will performance be affected if the messages are being offloaded?)

Use a CFLEVEL(5) structure to set thresholds at which messages are offloaded to shared message datasets rather than storing the message payload in the CF.

By lowering these thresholds, it is possible to store all messages in the shared message datasets, significantly increasing the capacity of the CF, but this does affect the performance.

When putting a message to a shared queue, the queue manager must decide where to store it.
If the queue manager determines that the message is to be stored in the shared message dataset, it will drive one or more I/O requests to write the data to the shared message dataset and then, upon success, will update the CF.

What to consider:

- Type of links to Coupling Facility
- Location of Coupling Facility

In the case of a CF on lower speed hardware or with slower links there may be benefit in storing the message payload in SMDS and only using the CF for the minimum storage.

As an example, consider the following configuration:
MVS LPAR running on z10 EC64, connected to an external CF on a z9 (2094-S38) via CFP link.

By comparing transaction rates when using a CFLEVEL(5) structure where the messages are stored in the CF against a CFLEVEL(5) structure where all messages are offloaded, we can determine whether it is more efficient to offload messages.

In this particular example, the CF is physically located within 100 metres and the majority of calls to the CF are asynchronous.

Using messages of 32KB and larger, you see comparable performance between storing the entire message in the CF and storing the message data in the SMDS.

Your configuration may be different and if CF capacity/performance is an issue with larger messages you may consider offloading all messages of 32KB or larger when a remote CF is involved.

The comparable costs between SMDS offloaded shared and private queues, coupled with increased number of messages for a given size of CF suggests that using shared queues is now possible for new classes of application.

Chart 4: Achieved transaction rate with remote Coupling Facility

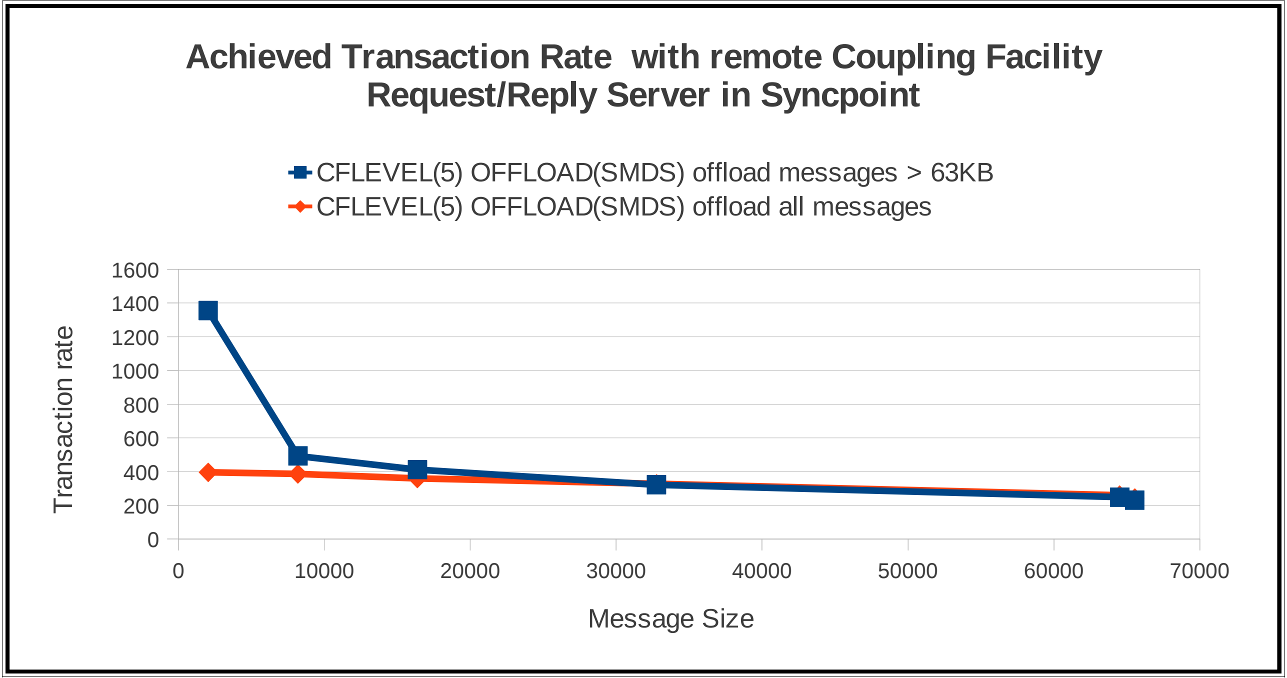
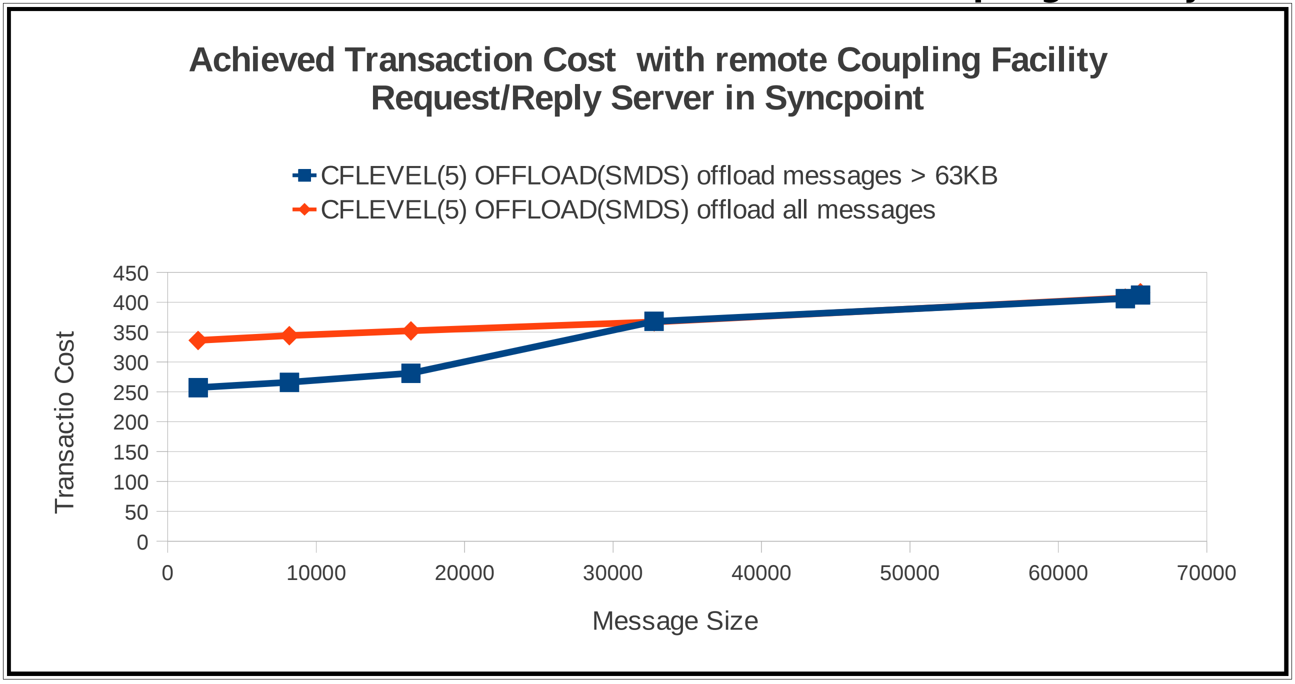


Chart 5: Achieved transaction cost with remote Coupling Facility



Initial Configuration

What impact does CFLEVEL(5) have on queue manager storage?

The queue manager is primarily running with 31-bit addressability. This means that the queue managers address space is limited to storage below a 2GB limit.

In reality, the queue manager will only have a subset of this 2GB to use - in our systems the queue manager has 1.4GB available. Of this 1.4GB storage, the queue manager uses some for buffer pools, trace tables etc. The CSQY220I message gives an indication of how much storage is available e.g.

```
CSQY220I @VKW1 Queue manager storage usage
: local storage : used 576MB,free 884MB : above bar : used 215MB,free >10GB
```

For each structure that the queue manager connects to, some 64-bit “above bar” storage space is reserved. This is calculated by multiplying the value of the DSBLOCK and DSBUFS attributes, which may be specified in the DEFINE CFSTRUCT command.

For example, if the default values are used, i.e. DSBLOCK(256K) and DSBUFS(100), a total of 25MB of above bar queue manager storage will be reserved for use for each structure.

This means that if there are 63 application structures available and each has the default values, 1575MB of above bar storage will be used by each queue manager that is connected to the 63 application structures and this storage is reserved solely for SMDS usage.

How large to make my SMDS datasets

It is possible to define shared message data sets up to a maximum of 16TB. The following table offers a guide as to the capacity that this offers:

Message Size (including headers)	Messages in 16TB	Size of CF (GiBytes)
100MB	167,772	0.12
4MB	4,194,304	3.00
1MB	16,777,216	12.00
100KB	171,798,692	122.88
64KB	268,435,457	192.00
10KB	1,717,986,918	1,228.80
=<4KB	4,319,967,296	3,072.00

NOTE: Each message offloaded requires 0.75KB of CF storage – 1 entry and 2 elements.
These sizes are approximate – some space is required for data management and each message is stored on a 4KB boundary and this will reduce the capacity of the dataset – and also explains why a 1KB message uses as much storage as a 3KB message.

Each queue manager owns 1 SMDS per structure. It can write to its own and read from others, so message capacity is increased by adding queue managers to a queue sharing group.
Since a queue sharing group may have 31 queue managers and each queue manager in a queue sharing group may connect to 63 application structures the absolute limit on capacity is:

```
Maximum Capacity: 16TB x 31 x 63 = 31,248TB (30.5PB)
```

Given sufficient DASD and Coupling Facility storage, SMDS is able to store 30.5 petabytes worth of data or 327,658,716 messages of 100MB.

Where to allocate SMDS?

It is advisable to locate the shared message data sets on different volumes.
This allows multiple I/O operations to be performed concurrently and allows more I/O channel paths to be utilised more efficiently.
By implementing multiple CFLEVEL(5) structures backed by shared message data sets on separate volumes, the workload is able to drive the I/O subsystem more efficiently, rather than being constrained on a single volume.
Examples of the benefit of this can be seen in the Scaling Performance measurements in the performance data section.

What is the maximum number of messages I can store?

This depends on size of data set, size of CF and size of message since there is a limit on the number of list entries in a list structure.

Should I pre-format the dataset?

Before a shared message data set can be used, it must be initialised.

There are 2 points at which this initialisation can occur –

- Prior to the dataset being accessed by an application connected to the queue manager. This may be achieved by using the CSQJUFMT program to format the dataset, similar to the way that LOGCOPY datasets are formatted.
- When the first application to open a queue that is hosted on the structure with the associated data set.

Due to the large size of these data sets, it is recommended to use CSQJUFMT to initialise the data set before the queue manager is started.
For example in our systems, using CSQJUFMT to format a shared message dataset of 900,000 records (3.43GB) took:

- 66 elapsed seconds (13,636 records per second)
- 2.49 CPU seconds (361,445 records per CPU second).

If the data set is not initialised prior to use, the first application to attempt to open any queue on that structure may take a significant time to respond.

Should I allow SMDS to expand and if so, by how much?

The DSEXPAND option specifies whether the queue manager is able to expand a shared message data set when it becomes nearly full.
If expansion is supported but no secondary allocation was specified at allocation time, a secondary allocation amount of approximately 10% of the existing size is used.
This means that for a shared message data set of 1 million records, the first expansion will be 100,000 records and then second will be 110,000 records, as this is 10% of 1.1 million records.
There should be sufficient capacity within the primary allocation of the shared message data set to handle peak message loads without expanding.

Should the shared message data set need to expand, the expanded space needs to be formatted. Although expansion and formatting starts at 90% full threshold, any further MQPUTs are slowed down to allow the expansion and formatting of the SMDS.

In addition, the following message may be observed on the queue manager's log:

```
CSQE239I @VKW8 CSQEDSS2 SMDS(VKW8)  CFSTRUCT(APPLICATION1) data set
MQMDATA.VKW8.SMDS.APPL1 has become full so new large messages can no longer be
stored in it
```

Once the expansion and formatting have completed, MQPUTs may continue.

The slowed MQPUTs whilst the SMDS expands and formats can have a significant effect. For example when running 10 applications putting 64KB messages to a single shared queue, the average put rate was 1980 messages per second which dropped to 780 messages per second whilst expanding and formatting.

Who pays for messages stored on Shared Message Data Sets?

Large shared queue messages offloaded to shared message data sets show different cost characteristic compared to large shared queue messages that are offloaded to DB2.
Consider an application that performs an MQPUT and MQGET of a 100KB message to a shared queue, where the message payload is stored in DB2.

- The costs are accumulated to the applications' address space until the queue manager needs to insert the binary large object or blob to the DB2 table. At this point the queue manager looks at the DB2 blob threads (as specified in the CSQ6SYSP macro under parameter QSGDATA) to choose and / or wait for an available thread. The queue manager then performs a task switch to that chosen thread and waits for the thread to complete before resuming the applications' thread.
- The select blob thread performs the SQL INSERT to the DB2 table and the cost of work performed by this thread is attributed to the queue managers' address space.
- With regards to the MQGET, the application is again charged for the work until the queue manager performs a task switch to one of the blob threads to perform the SQL SELECT of the data from the table. If the data is available in the DB2 buffer, the cost of this task may be relatively small.

Compare this to an application that performs an MQPUT and MQGET of a 100KB message to a shared queue, where the message is stored in shared message data sets:

- When a message is put to a queue defined in an CFLEVEL(5) OFFLOAD(SMDS) structure, the queue manager does not need to perform a task switch when writing the message to the shared message data set.
- Similarly when getting a message from a queue defined on a shared message data set, no task switch is required.
- This lack of task switching performed when accessing a message stored on shared message data sets means that:
  - **The cost of the put and get is attributed to the application.**
  - **There is no need to wait for a DB2 blob thread in the queue managers address space.**

As a result, the application may see an increased cost but this should be more than offset by a decreasing cost in the queue manager.

There is still be some queue manager cost associated with messaging that is not affected by using CFLEVEL(5) OFFLOAD(SMDS), in particular (but not limited to):

- Issuing MQCMIT – This causes the queue manager to task switch to an SRB to ensure the commit is completed.
- Persistent Messages – The logging of persistent messages by a single queue manager task.

Recovery

How long will it take to backup my structure?

Only persistent messages are backed up using the `BACKUP CFSTRUCT(applicationName)` command.

When the `BACKUP CFSTRUCT(applicationName)` command is used, all messages held on queues in that structure, whether offloaded to shared message data sets or not, are written to the log of the issuing queue manager. This includes messages stored in other queue managers shared message data sets connected to this structure.

For example:
If 16000 persistent messages each of 64KB were put to a shared queue on queue manager A, a backup would log around 1GB.
If 16000 persistent messages each of 64KB were put to a shared queue on queue manager A and also on queue manager B, for a total of 32000 messages, the BACKUP CFSTRUCT issued on queue manager A would result in 2GB of data being written to the log data sets.

Consider 3 configurations:

1) 1K persistent messages on queue - structure offloads all messages
   Achieved backup rate of 71MB/second
2) 5K persistent messages on queue - structure offloads all messages
   Achieved backup rate of 74MB/second
3) 32K persistent messages on queue – offload rules have caused some messages be offloaded to shared message data sets.
   - 80,000 messages on queue:
   - 7,597 are held in CF only – requires 250.4MB to backup data
   - 72,403 are offloaded to shared message data set, requires 2,386MB to backup data.

   Achieved backup rate of 73MB/second.

Do multiple structures affect recovery?

The space map used by SMDS is rebuilt in the following circumstances:

- At connect time if data set is marked as ACTIVE (or RECOVERED) but there is no saved space map from the previous disconnect, probably because of queue manager abnormal termination or problem accessing data set.
- Immediately following the recovery of the structures.

When the queue manager is restarted following a failure, the structure is connected to and the space map is rebuilt.

When multiple structures are in use, each structure is connected to and the space map is rebuilt in series.

For a single structure we saw a rebuild rate of approximately 77,000 messages per second.
When 20 structures were in use, each space map was able to process at 77,000 messages per seconds but because there were 20 structure connects, the elapsed time is greater than expected.

How long will it take to recover my structure?

Example 1:
1,000,000 messages on a single structure:

1,000,000 at 77,000 per second	= 12.98 seconds
+ connect time	= 0.5 seconds
TOTAL	= 13.48 seconds.

Example 2:
50,000 messages on each of 20 structures (i.e. a total of 1,000,000 messages)

50,000 at 77,000 per second	= 0.659 seconds
+ connect time	= 0.5 seconds
TOTAL per structure	= 1.159 seconds
TOTAL	= 23.18 seconds

NOTE: The backup and recovery was run on a dedicated system with dedicated links to dedicated disks.

With the increased throughput capability, additional workload will mean extra load on your I/O subsystem and the period of time between structure backups should, therefore, be reviewed.

How do I know what is going on?

When using CFLEVEL(5) in a queue manager, a number of messages will be displayed on the queue manager's log.

Messages

Messages on start up

- connect to shared message data set

```
CSQE241I @VKW8 SMDS(VKW8) CFSTRUCT(APPLICATION1) now has STATUS(NEW)
CSQE241I @VKW8 SMDS(VKW8) CFSTRUCT(APPLICATION1) now has STATUS(ACTIVE)
```

- rebuild space map

```
CSQE252I @VKW8 SMDS(VKW8) CFSTRUCT(APPLICATION1) data set
MQMDATA.VKW8.SMDS.APPL1 space map will be rebuilt by scanning the structure

CSQE255I @VKW8 SMDS(VKW8) CFSTRUCT(APPLICATION1) data set
MQMDATA.VKW8.SMDS.APPL1 space map has been rebuilt, message count 141065
```

NOTE: There are 158,166 messages on the shared queue of which 141,065 have been offloaded to shared message dataset and the remaining 17,101 are held on the CF in their entirety.

Messages whilst queue manager is running

- expand shared message data set

```
CSQE213I @VKW8 CSQEDSS2 SMDS(VKW8) CFSTRUCT(APPLICATION1) data set
MQMDATA.VKW8.SMDS.APPL1 is now 90% full

CSQE211I @VKW8 CSQEDSI1 Formatting is in progress for 10080 pages in SMDS(VKW8)
CFSTRUCT(APPLICATION1) data set MQMDATA.VKW8.SMDS.APPL1

CSQE213I @VKW8 CSQEDSS2 SMDS(VKW8)  CFSTRUCT(APPLICATION1) data set
MQMDATA.VKW8.SMDS.APPL1 is now 92% full

CSQE213I @VKW8 CSQEDSS2 SMDS(VKW8)  CFSTRUCT(APPLICATION1) data set
MQMDATA.VKW8.SMDS.APPL1 is now 94% full

CSQE213I @VKW8 CSQEDSS2 SMDS(VKW8)  CFSTRUCT(APPLICATION1) data set
MQMDATA.VKW8.SMDS.APPL1 is now 96% full

CSQE213I @VKW8 CSQEDSS2 SMDS(VKW8)  CFSTRUCT(APPLICATION1) data set
MQMDATA.VKW8.SMDS.APPL1 is now 98% full

CSQE212I @VKW8 CSQEDSI1 Formatting is complete for SMDS(VKW8)
CFSTRUCT(APPLICATION1) data set MQMDATA.VKW8.SMDS.APPL1

CSQE217I @VKW8 CSQEDSI1 Expansion of SMDS(VKW8) CFSTRUCT(APPLICATION1) data set
MQMDATA.VKW8.SMDS.APPL1 was successful, 10080 pages added, total pages 60120
```

Messages as a result of commands

- **DIS CFSTATUS**

```
CFSTATUS(APPLICATION1)
TYPE(SUMMARY)
CFTYPE(APPL)
STATUS(ACTIVE)
OFFLDUSE(SMDS)
SIZEMAX(512000)
SIZEUSED(1)
ENTSMAX(230014)        ← Absolute maximum number of messages in structure
ENTSUSED(33)
FAILTIME( )
FAILDATE( )
 END CFSTATUS DETAILS
```

- **DIS CFSTRUCT(APPLICATION1)**

```
CFSTRUCT(APPLICATION1)
DESCR( )
CFLEVEL(5)
RECOVER(YES)
OFFLOAD(SMDS)
OFFLD1TH(0)            ← Chosen to offload all messages to SMDS
OFFLD1SZ(0K)
OFFLD2TH(0)
OFFLD2SZ(0K)
OFFLD3TH(0)
OFFLD3SZ(0K)
DSGROUP(MQMDATA.*.SMDS.APPL1)
DSBLOCK(256K)
DSBUFS(100)
DSEXPAND(YES)
RECAUTO(YES)
CFCONLOS(ASQMGR)
ALTDATE(2011-12-20)
ALTTIME(11.14.47)
```

- **DIS CFSTATUS(APPLICATION1) TYPE(SMDS)**

```
CFSTATUS(APPLICATION
TYPE(SMDS)
SMDS(VKW8)
STATUS(ACTIVE)
ACCESS(ENABLED)
RCVTIME(11.14.47)
RCVDATE(2011-12-20)
FAILTIME( )
FAILDATE( )
```

- **DIS SMDS(VKW8) CFSTRUCT(APPLICATION1) ALL**

```
SMDS(VKW8)
CFSTRUCT(APPLICATION1)
DSBUFS(DEFAULT)
DSEXPAND(DEFAULT)
```

- **DIS SMDSCONN(VKW8) CFSTRUCT(APPLICATION1)**

```
SMDSCONN(VKW8)
CFSTRUCT(APPLICATION1)
OPENMODE(UPDATE)        <--- Owned by this queue manager and open for update
STATUS(OPEN)            <--- Has been opened successfully is is available
AVAIL(NORMAL)
EXPANDST(NORMAL)        <--- No errors detected, so OK to expand.
```

- **DIS USAGE TYPE(SMDS)**

```
CSQE280I @VKW8 SMDS usage ...
 Application    Offloaded       Total  Total data   Used data  Used
 structure       messages      blocks      blocks      blocks  part
_APPLICATION1         601       14062       14061         260    1%
 End of SMDS report
CSQE285I @VKW8 SMDS buffer usage ...
 Application  Block   --------- Buffers ---------  Reads  Lowest  Wait
 structure     size   Total  In use  Saved  Empty  saved    free  rate
_APPLICATION1  256K     100       2      0     98   100%      97    0%
 End of SMDS buffer report
```

The output from the “DISPLAY USAGE TYPE(SMDS)” command is for the current SMF interval and reports the number of messages that have been offloaded to the shared message data set for the structures defined appropriately.

In this above example, 601 messages were offloaded using 260 data blocks, where each data block is 256K. This means that a maximum 66,560KB of data has been offloaded. Assuming that each message is of the same size, we can calculate the message size to be 110.7KB.

Are the default options the correct ones for me?

(Or when do the following fields affect me, and by how much?)

DSBUFS
DSBLOCK
OFFLDnSZ / OFFLDnTH

When running messaging workload using shared message datasets, there are 2 levels of optimizations that can be achieved by adjusting the DSBUFS and DSBLOCK attributes.

The amount of above bar queue manager storage used by the buffer is DSBUFS x DSBLOCK. This means that by default, 100 x 256KB (**25MB**) is used for each CFLEVEL(5) structure in the queue manager.

DSBUFS

The **DSBUFS** attribute specifies the number of buffers, taken from above bar storage, that is used to hold a cached copy of the messages. This enables faster access when reading the message from the queue, when performed by the putting (local) queue manager.

Level 1 Optimization: Avoid Put Time I/O waits

If there are insufficient buffers to handle the maximum concurrent number of I/O requests, then requests will have to wait for buffers, causing a significant performance impact. This can be seen in the CSQE285I message (issued as a response to the “DISPLAY USAGE TYPE(SMDS)” command) when the “lowest free” is zero or negative and the “wait rate” is non-zero. If the “lowest free” is negative, increasing the DSBUFS parameter by that number of buffers should avoid waits in similar situations.

When the message data exceeds one SMDS block, a request to start overlapping I/O operations to transfer multiple blocks concurrently is initiated but this is limited to a maximum of 10 active buffers per request.

If the message is 100KB and DSBLOCK(8K) is set, each message would require 13 buffers but would only be able to use 10 buffers. In this example, it would be more appropriate to use a larger DSBLOCK size to ensure that the I/O operations were completed in an optimum manner, for example if the message were 100KB and DSBLOCK(64K) is set, each message would use only 2 buffers.

Level 2 Optimization: Cache of recently put messages

Once there are sufficient buffers to avoid waits, the next level of optimization occurs when enough data can be buffered so that when recently written data is read back from the same queue manager, it is possible to find the data in a buffer rather than having to read it from disk. This can save significant elapsed time in the reading transaction.

The following example shows the benefits of tuning the DSBUFS attribute:

- A single queue manager in a QSG with a single CFLEVEL(5) application structure, with DSBLOCK(64K)
- All messages are offloaded to the SMDS dataset.
- A request/reply workload is run.
- There are 12 requester tasks which each put a 63KB message to a single shared queue.
- These messages are got by 1 of 4 server tasks that get and put a reply message in-syncpoint to a second shared queue which is indexed by CORRELID.
- These reply messages are got by the requester tasks using CORRELID.
- This is repeated multiple times until the requester tasks are stopped.

- Command DIS USAGE TYPE(SMDS) is used to determine the status of the SMDS buffer.
- The DSBUFS value is altered using ALT SMDS(*) CFSTRUCT(APPLICATION1) DSBUFS(new_value).
- The test is repeated using DSBUFS values ranging from 4 to 32.

The following 2 charts show the two levels of optimization. There is a distinct increase in transaction rate when there are sufficient buffers such that the tasks are not waiting for a buffer, i.e. when “wait rate” is 0%. In these examples, a 65% increase in throughput is seen (from 1500 to 2500 transactions per second)

The second increase occurs when the queue manager is able to get the message data from the buffer rather than having to perform disk I/O operations. In this example, an additional 50% increase in transaction rate is seen (from 2750 to 4100 transactions per second)

Chart 6: Effect of DSBUFS on transaction rate

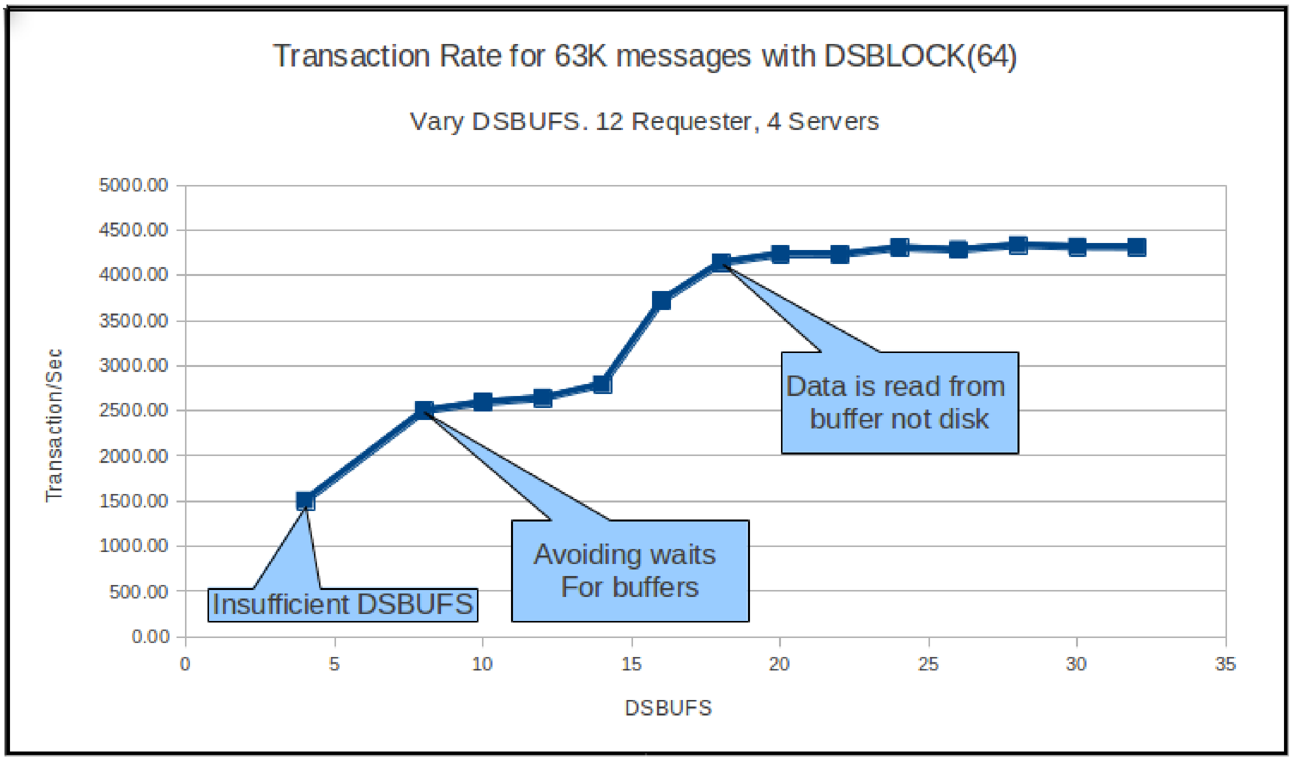
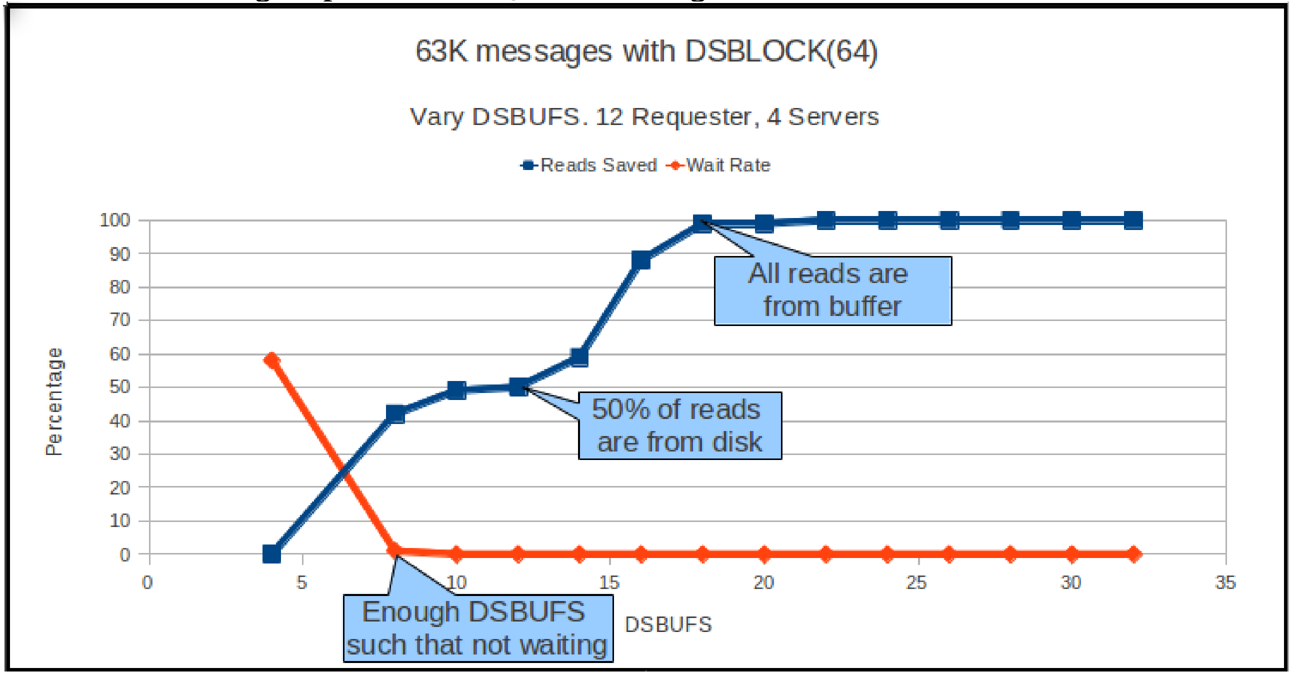
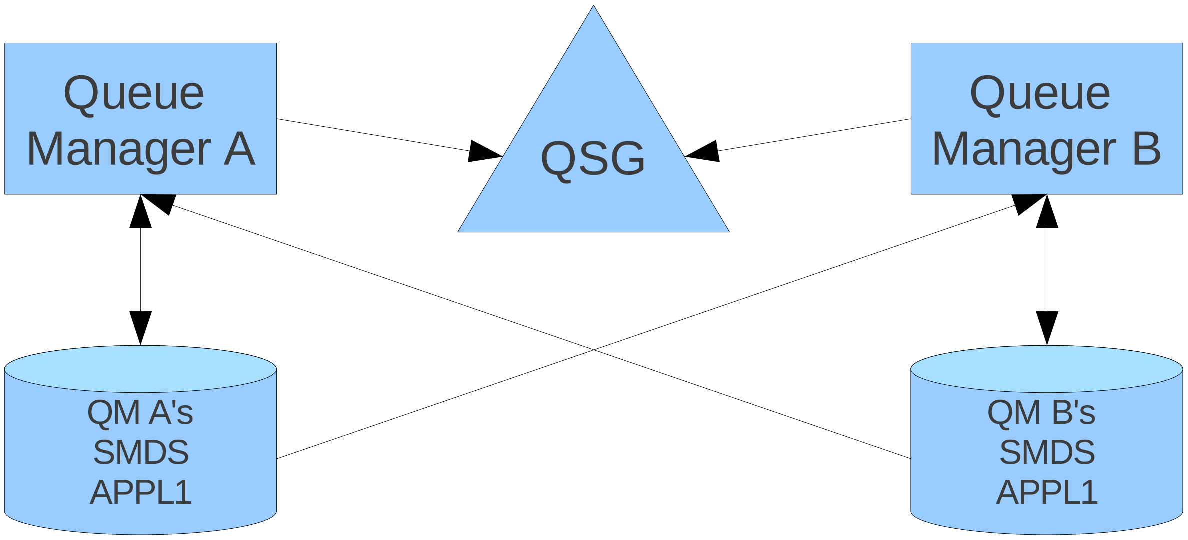


Chart 7: Using output from CSQE285I message to determine best DSBUFS value



The value of DSBUFS is evident in the preceding charts. However this benefit is only realised when the message is being got from the same queue manager that the message was put on, i.e. the get is from the local queue manager.

Consider the following configuration:



When messages are put to a shared queue by an application connected to queue manager A, the queue manager updates the CF with key information and writes the message to its SMDS. The message will be kept in queue manager A's DSBUFS buffers, as well as the CF and SMDS, until more messages are put to queues in that structure on that queue manager.

Once the buffers are full, they become re-used, so older messages are only stored in the SMDS and CF.

Once the entire message is lost from the buffers and held only in the SMDS, the rate at which the message can be retrieved is comparable, whether the get is from queue manager A (a local get) or from queue manager B (a remote get)

NOTE: Transaction cost is comparable too, however when the get is from a remote queue manager, the queue manager that held the message in its SMDS will perform the delete of the message from the SMDS. This incurs a minor cost to that local queue manager.

DSBLOCK

DSBLOCK is the logical block size in kilobytes in which the SMDS space is allocated for individual queues.

The **DSBLOCK** attribute divides the SMDS and buffer into blocks for holding the WebSphere MQ message.

- By selecting the DSBLOCK size that is larger than the message, storage usage in the buffer is less efficient.
- By selecting a DSBLOCK size that is smaller than the message means that multiple I/O requests are required to store the message on SMDS dataset.

To aid performance, SMDS attempts to overlap I/O requests, but each task is limited to :

10 buffers
OR where the total size of the buffer is less than or equal to 4MB

So, maximum overlapped I/O requests is currently 10 but if DSBLOCK(512K) is set, the maximum overlapped I/O requests is 8 (4MB/512K).

Understanding how messages are stored in SMDS

Each message is written, starting at the next page within the current block and is allocated further blocks as needed.

Using the default setting of DSBLOCK(256K), each logical block will be 256K (64 pages).

Consider messages that use 100KB:

- Message 1 will be written to block 1 (page 0, using 25 pages)
- Message 2 will be written to block 1 (page 25, using 25 pages)
- Message 3 will cause 2 I/O requests because part of the message will be in block 1 and the remainder will be in block 2.

A larger DSBLOCK decreases space management overheads and reduces I/O for very large messages but increases buffer space requirements and disk space requirements for small messages.

How should I size my DSBLOCK?

The combination of DSBLOCK and DSBUFS affects how much above bar storage is used by SMDS.

The following lists the order in preference for performance for sizing the DSBLOCK attribute:

1. Message fits into a single DSBLOCK (i.e. a single I/O request)
2. Message fits into maximum overlapped I/O requests, so that all writes can be requested concurrently.
3. Message is too large and is split into separate I/O requests which cannot be overlapped.

An example of how these configurations affect transaction rate and cost is shown below.

Example:

- A single queue manager in a QSG with 1 structure at CFLEVEL(5).
- The SMDS under test has DSBUFS(200) – which has been determined to be sufficient buffers for this work.
- Request/Reply workload is run against a pair of queues using 100KB non-persistent messages
- The 12 requesters put a message to a request queue and wait for a reply message with a known CORRELID on the reply queue. All requester work is performed out of syncpoint.
- The 4 server applications get-with-wait on the request queue and put a corresponding reply message. These gets and puts are performed in syncpoint.
- Measurements are performed on a single z/OS v1r12 LPAR with 3 dedicated processors on zEnterprise 196 (2817-779). CF is internal, with 3 shared processors.
- DSBLOCK is set to the 3 following values:
    - 8K – Message does not fit into 10 buffers, so will require multiple overlapped I/O requests.
    - 16K – Message fits into 7 buffers, so will be written to disk in 1 set of overlapped I/O requests.
    - 128K – Message fits into a single buffer. One I/O request would typically be required to write to disk, unless message spills over from a previously part-used DSBLOCK.

The following table shows the achieved transaction rate and cost per transaction for these tests:

DSBLOCK	**8K**	**16K**	**128K**
Transaction Rate/Second	466	1603	2139
Cost/Transaction (CPU microseconds)	741	573	451
LPAR %Busy	9.7	25.6	24.9
MB written to SMDS / second. Based on: Transaction rate * 100K * 2 messages put per transaction	91	313	417
RMF “Channel Path Report” %Busy (average for 4 channel paths in use)	24.7	39.5	27.6
Used Data Blocks in 4K pages (“used data blocks” from CSQE280I x DSBLOCK size/ 4)	352	384	448

What does this tell us?
It is more space efficient to use 8K buffers to store 128K messages than either 16K or 128K. However the system allows a higher transaction rate with the larger DSBLOCK size and at less strain to the I/O system.

OFFLDnTH / OFFLDnSZ

As can be seen in the earlier section “Is there benefit in offloading all messages”, there can be performance benefits in setting the OFFLDnTH and OFFLDnSZ attributes so that all messages are offloaded.
In addition to the benefits observed when running with a duplexed Coupling Facility (CF) on a separate machine, it may be of benefit to offload messages when the CF's capacity is a limiting factor.

CFLEVEL(5) and small messages

Using the offload thresholds and sizes it is possible to specify that all messages are offloaded, which will increase the capacity of the coupling facility.

Because of the way the message header is stored in the coupling facility, there is space in the required elements for a message of less than 122 bytes to be stored. WebSphere MQ is optimized to store these small messages in the CF structure as it does not impact the CF. This means that messages of less than 122 bytes are not offloaded into the shared message data sets.

Can I switch between offloading SMDS and DB2 mid-workload?

Yes you can. The following 2 charts show the transaction rate and cost observed when the offload option is changed from SMDS to DB2 and then back to SMDS.

Chart 8: Achieved transaction rate when switching offload destination

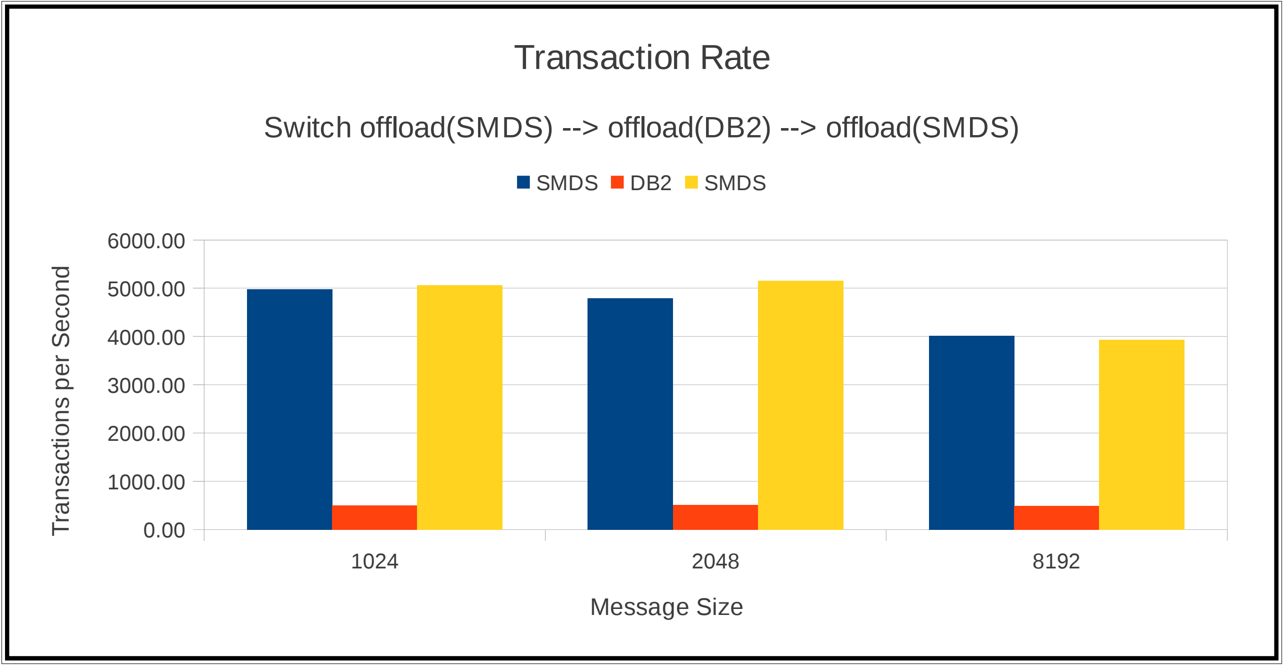
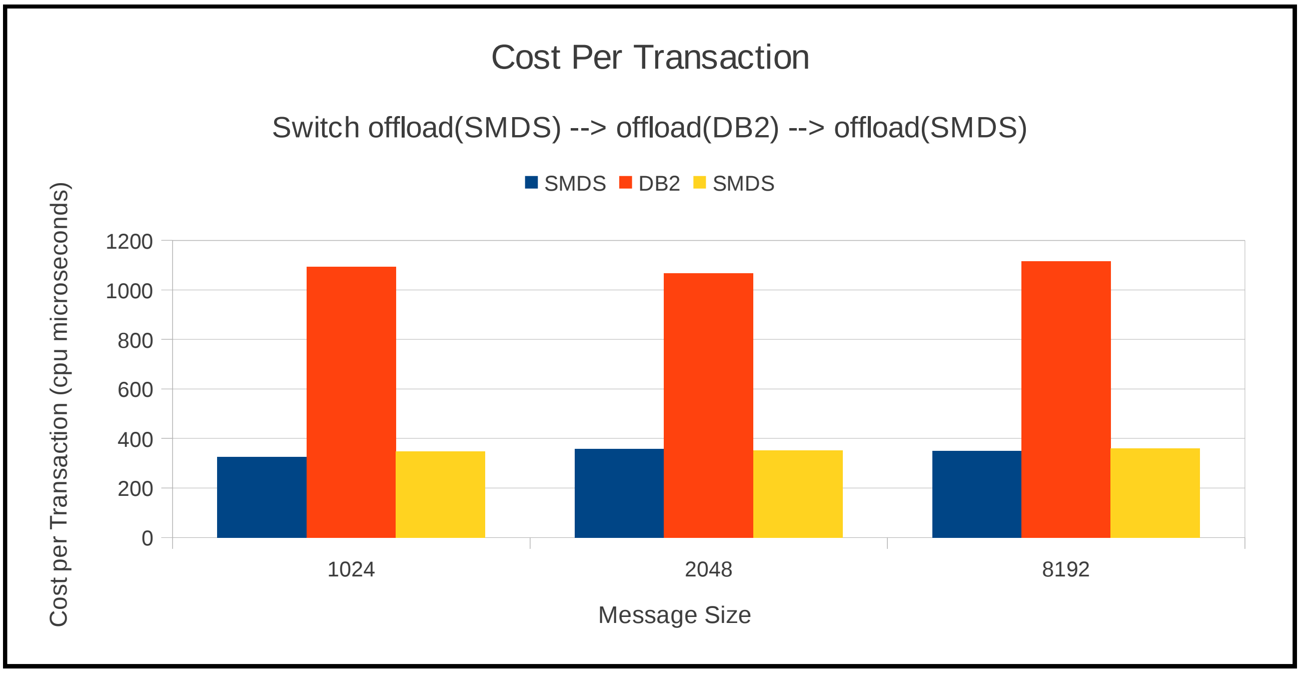


Chart 9: Actual transaction cost observed when switching offload destination



Predicting limits

This prediction is based on workload running on a single queue manager in a QSG.
This means that all reads will be local and from the SMDS buffers.

- Workload is request/reply, 100KB, non-persistent, server gets and puts in-syncpoint.
- Workload is spread across multiple structures – each backed by a shared message dataset.
- Each SMDS is on a separate volume where possible, although there are some data sets sharing volumes.

Previous measurements have shown our CF processors and links are capable of processing 22,000 transactions per second in this particular request/reply model with 2KB messages. Since the 100KB message size means that the CF access is only 0.75KB we predict that the CF is not constrained.

Structures in Use	Transaction Rate (per structure)	Write MB/Sec per SMDS	Total Transaction Rate	Total Writes MB/second
8	469	93	3752	744
32	469	93	15008	3976

Our system has 4 FICON channel paths defined to the DASD subsystem. Using the “Channel Path Activity” RMF report from the 8 structure measurement, we can predict that each channel path is capable of supporting writes to 32 shared message datasets whilst maintaining the 70MB/second write rate.

V6 Compatibility Mode

The parameter “V6COMPAT” can be applied to the queue attribute “PROPCTL” and applies for local, alias and model queues.

V6COMPAT is the default value when migrating from WebSphere MQ V6.0.0, whereas when migrating from either V7.0.0 or V7.0.1 the default value is COMPAT.

The V6COMPAT queue option causes the application to receive the MQRFH2 structure as it was sent, subject to character set conversion and numeric encoding changes.

The V6COMPAT queue option reduces the amount of parsing that the queue manager has to complete when storing the message and can reduce the cost and improve the performance.

In our measurements, V6COMPAT has not provided any benefit in JMS Publish/Subscribe tests.

However, in put/get measurements where message properties are present, the cost of the MQPUT has decreased in some cases by up to 60% and the MQGET has decreased by up to 40%.

If you are not running JMS pub/sub on a specific queue, the V6COMPAT queue option may offer some reduction in CPU cost over the equivalent costs in WebSphere MQ V7 when message properties are included in the message.

Channel authentication rules

Channel authentication rules introduced in version 7.1.0, mean that the queue manager can deny connections from unauthorised sources.

Queue managers migrated from previous versions have the queue manager attribute CHLAUTH set to DISABLED.

Queue managers created against version 7.1.0 have basic channel authentication rules defined.

Additional channel authenication rules to permit specific connections or a range of connections using wildcards can be defined.

Further documentation on setting up channel authentication rules is available in the WebSphere MQ V7.1.0 infocenter.

The channel authentication checks are performed at channel start; in the case of server connections, this occurs at WebSphere MQ connect time.

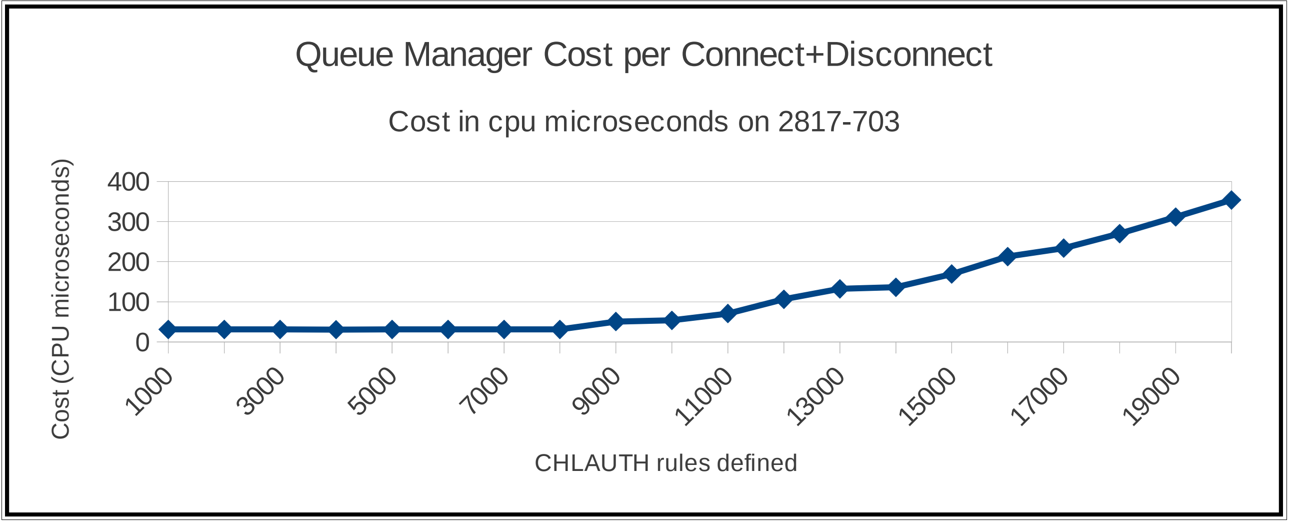
There is a small increase in the cost of the MQCONN when running with channel authentication enabled. **Typically we have observed a 5-10% increase in the cost of the MQCONN call but will increase with larger numbers of rules defined.**

The queue manager checks the information passed in the connection attempt against the rules defined in the queue manager. Any explicitly defined rules are checked first, followed by wildcard entries. This means that if many rules are defined, for example one rule for each user in a medium to large organisation, additional cost may be observed in the queue managers address space for each connect.

The cost attributed to the queue manager is relatively small when compared to the cost seen in the channel initiator but with many rules to check, the proportion of cost can rise noticeably.

The following chart shows the increase in cost in the queue manager address space as more channel authentication rules have to be checked before the user is accepted. The chart shows that with up to 8000 channel authentication records defined, the cost in the queue manager is relatively flat (at around 30 microseconds on a 2817-703).

Chart 10: Queue manager cost per connect / disconnect with increasing CHLAUTH rules



Once the number of rules to check increases to 20,000, the queue manager cost increases to 350 microseconds. This cost can seem considerable but should be taken into context of the cost of the connect in both the queue manager and channel initiator address space.

The following 2 charts show the costs of an MQCONN and MQDISC with both few channel authentication rules defined and with many rules defined.

Chart 11: MQCONN / MQDISC cost with minimal CHLAUTH rules defined

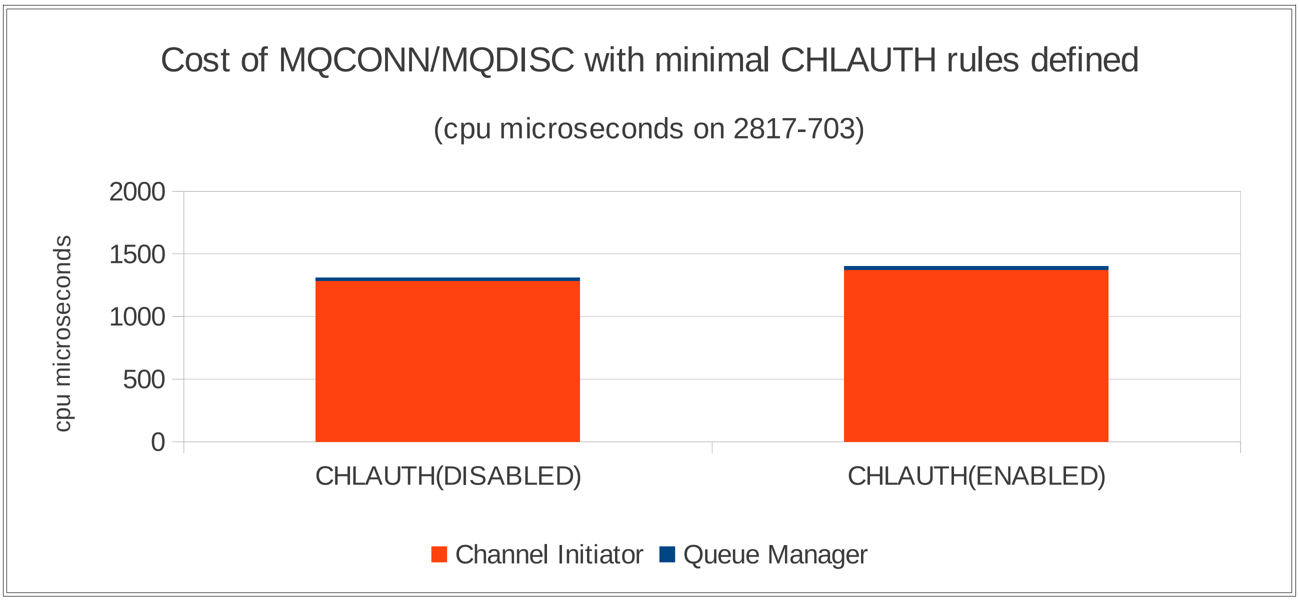
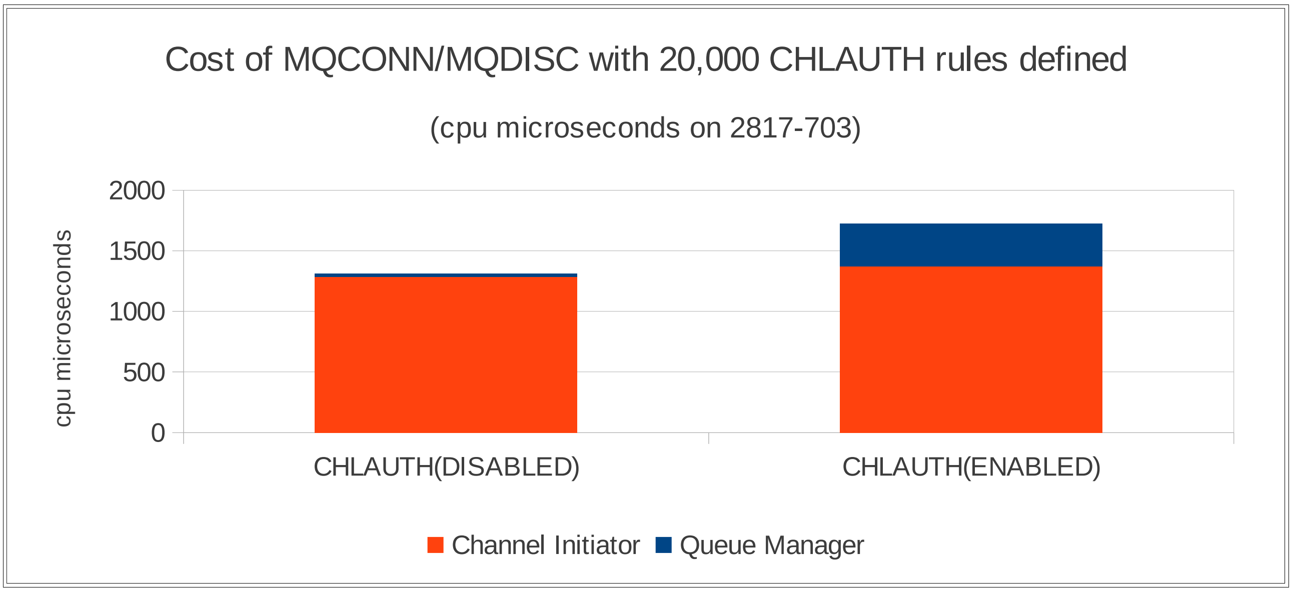


Chart 12: MQCONN / MQDISC cost with 20,000 CHLAUTH rules defined



Channel authentication and the effect on queue manager restart

Defining many rules can affect the performance of queue manager restart.

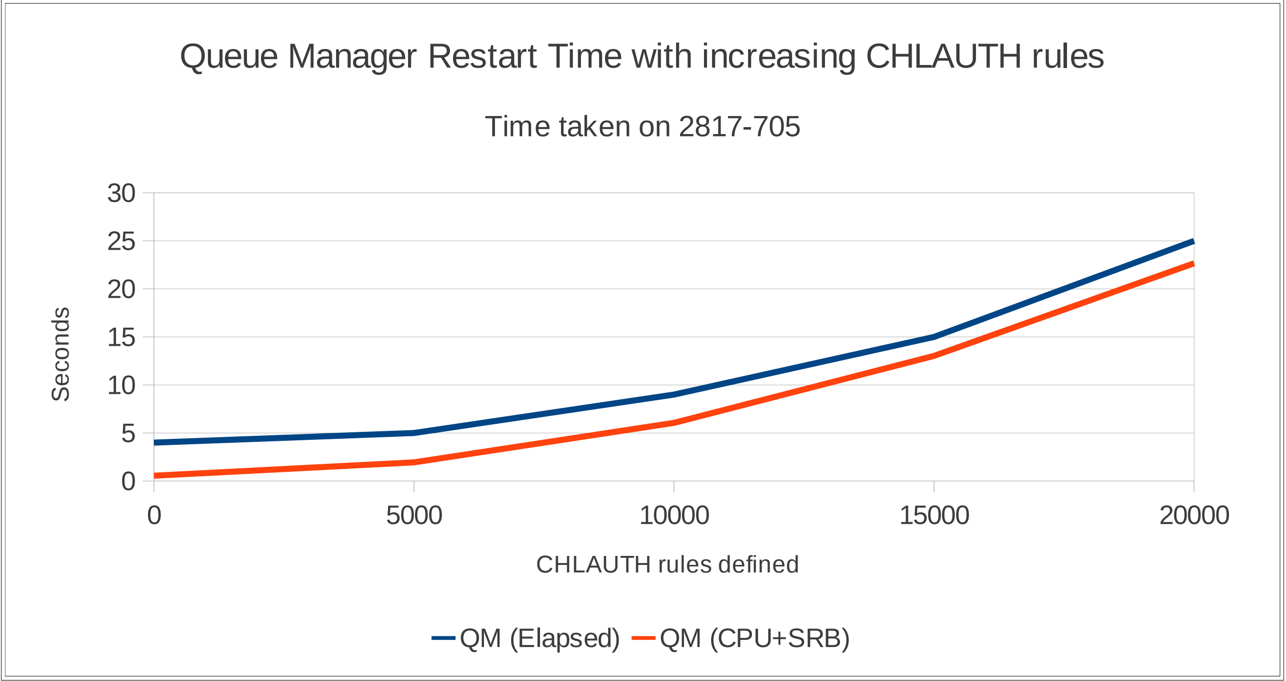
Each rule defined is stored in a message on the SYSTEM.CHLAUTH.DATA.QUEUE, and depending on the number of specific channels, the depth of the queue may increase.

Rules are grouped by channel name; there is one message per channel name. Each message may contain many rules. For example, if a set of channel authentication rules are defined for each user on your system for explicit access using SYSTEM.DEF.SVRCONN, only 1 message on SYSTEM.CHLAUTH.DATA.QUEUE is created for all of these rules.

If a significant number of unique channel names have rules defined, the depth of the SYSTEM.CHLAUTH.DATA.QUEUE may be high and may benefit from having an index type of MSGID defined, e.g. ALT QL(SYSTEM.CHLAUTH.DATA.QUEUE) INDXTYPE(MSGID), but it is not expected that large numbers of channels will be explicitly referenced so the queue depth will be low.

When there are a significant number of rules defined, either for many channels or on a few channels, the restart time of the queue manager can be affected. The chart below shows the impact on restart time with increasing numbers of users permitted to access the channels matching the wildcard “SYSTEM.*”. In these tests, there were only 3 channels affected by the defined rules on the SYSTEM.CHLAUTH.DATA.QUEUE, but with an increasing number of users defined.

**Chart 13:	Queue manager restart time with increasing CHLAUTH rules**



Queue manager trace (64-bit global trace)

WebSphere MQ for z/OS version 7.1.0 changes how the queue manager global trace is gathered.

In previous releases, the global trace data was stored in a single storage area which on a busy system with multiple processors could result in a high degree of contention when writing the trace data. Version 7.1.0 exploits 64-bit storage to allocate an area of storage for each thread, which reduces the contention issues seen previously.

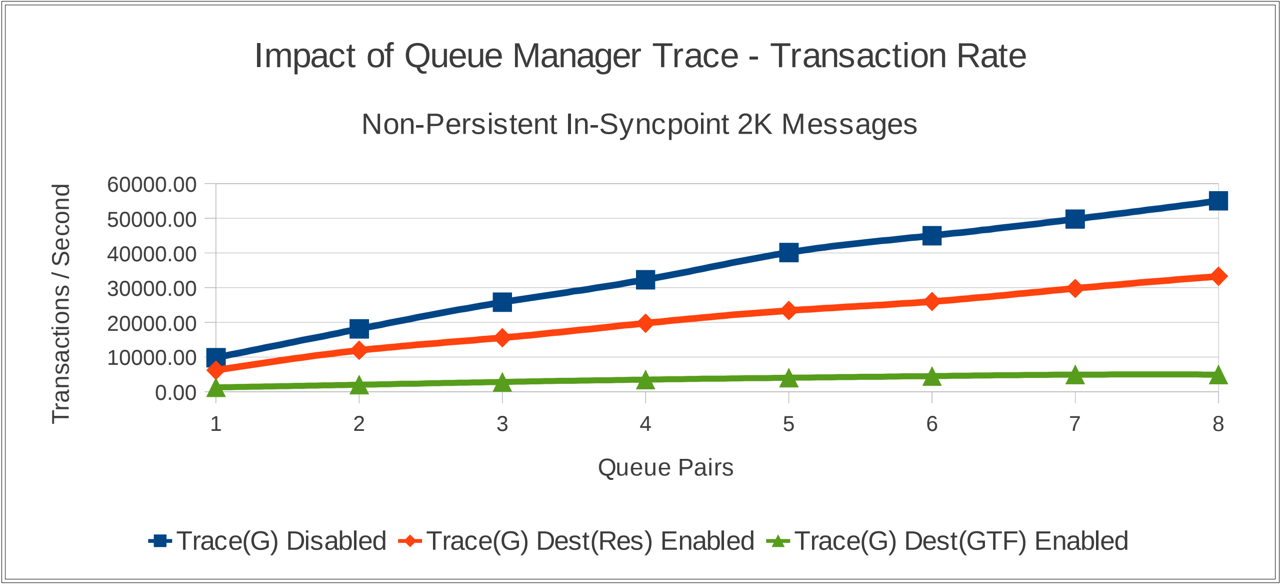
The relative performance benefits of 64-bit global trace can be seen in "Appendix A – Regression – Queue Manager Global Trace". The V6 trace costs are similar to the V7.1 trace costs as the measurement was unable to drive the V6 workload at such a high rate without filling the pagesets with unscavenged messages, so was artificially delayed, which in turn reduces the contention impact discussed previously.

Performance observations

Where the trace data is captured can affect the performance achieved, for example specifying a destination of GTF can impact the performance more than specifying RES as the destination.

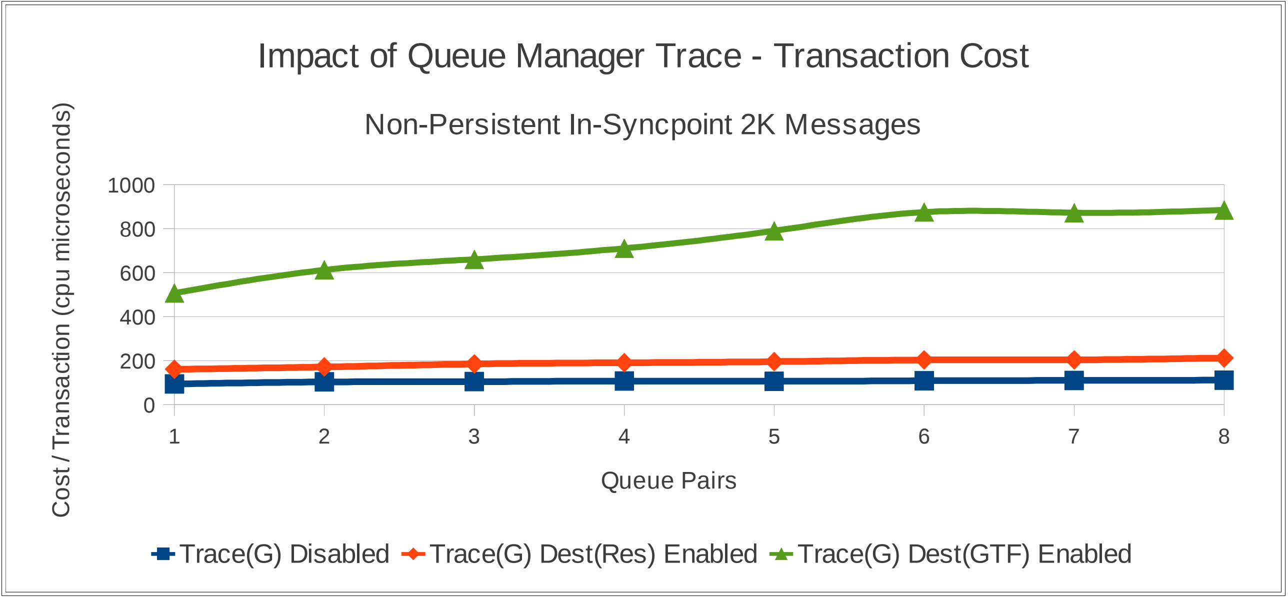
The charts following show the effect of enabling global trace and varying the destination on both transaction rate and transaction cost when running a non-persistent workload on a 16 processor LPAR.

Chart 14: Impact of enabling queue manager global trace on transaction rate



The impact of specifying a destination of GTF is significant compared to using the RES destination. This can also be seen in the chart below detailing the transaction costs.
NOTE: The applications do no work other than messaging and it is not expected that enabling trace in the queue manager would double the cost of a typical business transaction.

Chart 15: Impact of enabling queue manager global trace on transaction cost



Recommendations

In a system with a high number of connected tasks and queue manager trace enabled, an increase in page/swap activity may be seen.

For performance reasons, queue manager global trace should not run by default, but in V7.1.0, the impact of running with global trace is reduced.

Topic scavenger / queue manager attribute TREELIFE

What is it?

See WebSphere MQ V710 Infocenter "Reducing the number of unwanted topics in the topic tree" for a detailed description of the role of the topic scavenger task.

Can you see it running?

No - There are no messages logged at start or end.

How do you configure it?

Use the queue manager attribute TREELIFE to set the frequency. This is the time in seconds which non-administrative topics are allowed to remain in the system until they are removed. Valid values range from 0 through 604,000 – approximately 7 days, where the default is 1800 seconds. A value of 0 means that non-administrative topics are not removed.

What is the impact of not running frequently enough?

If you are using the pub/sub function and use the remove subscription or de-register subscription options, the topic node remains in the topic tree until the topic scavenger runs.

In the majority of cases the topic tree will be fairly static. It is only applications which use dynamic topic strings (for example containing a time or date stamp) where topic nodes become redundant over time and require removing.

If there is a large amount of de-registering occurring between scavenger runs, the topic tree can grow significantly which can affect the performance of any new subscriptions and increase the size of the queue manager 64-bit storage used. The performance impact can be particularly significant :

- if lots of wildcard subscriptions are being used.
- If the topic tree is dynamic and changing significantly.

What is the impact of running too frequently?

The topic scavenger will use a small amount of CPU to complete its scan of the topic tree – the more frequently it is run, the more CPU will be used. However there is a trade-off between the topic scavenger costs and the increased cost of subscription against a topic tree with many unused topics.

In addition, the topic scavenger runs as a TCB within the queue manager address space. This means that it is a high priority task and running it frequently on a system that is CPU constrained may affect the response times of other tasks.

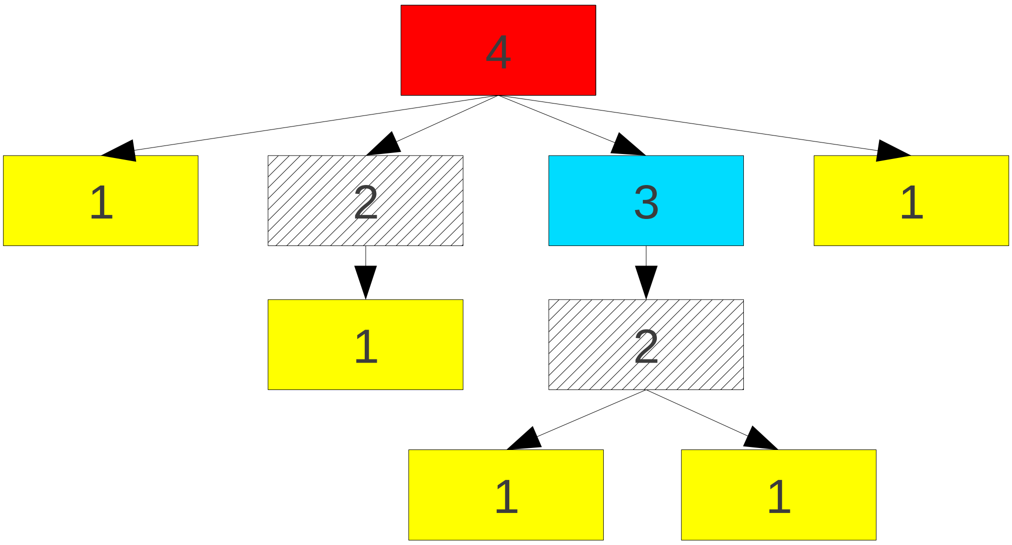
How do I know what is the right frequency?

This depends on the type of work the queue manager is processing.
Running the topic scavenger incurs a small cost but when your business model results in a build up of redundant topics, this scavenger cost is offset by the cost reduction at subscribe time.
If there is no pub/sub, then alter the queue manager and set TREELIFE(0).
If there is pub/sub then it depends on whether there is a high turn-over of subscriptions eligible for the topic scavenger.
We saw benefits of running the topic scavenger frequently enough, such that there were no more than 200,000 unwanted topics known to the queue manager.
The actual setting of TREELIFE depends on the rate of unsubscribing and the number of leaf nodes available for scavenging.

Why are my topics still visible after the topic scavenger has run?

The topic scavenger can only remove topics that have no child topics (also known as leaf nodes) in each pass, so a large topic tree with many levels may take multiple passes of the topic tree to remove all unused topics.
The diagram following shows a simple tree with just 4 levels – this tree would take 4 passes of the scavenger to remove all of the unwanted topics – so TREELIFE(120) would take 8 minutes.

Scavenging a simple topic tree takes multiple runs of the topic scavenger



The number in each node is the iteration number that the topic scavenger will remove the node

Topic scavenger measurements

Topic scavenger on an idle queue manager – with no topics

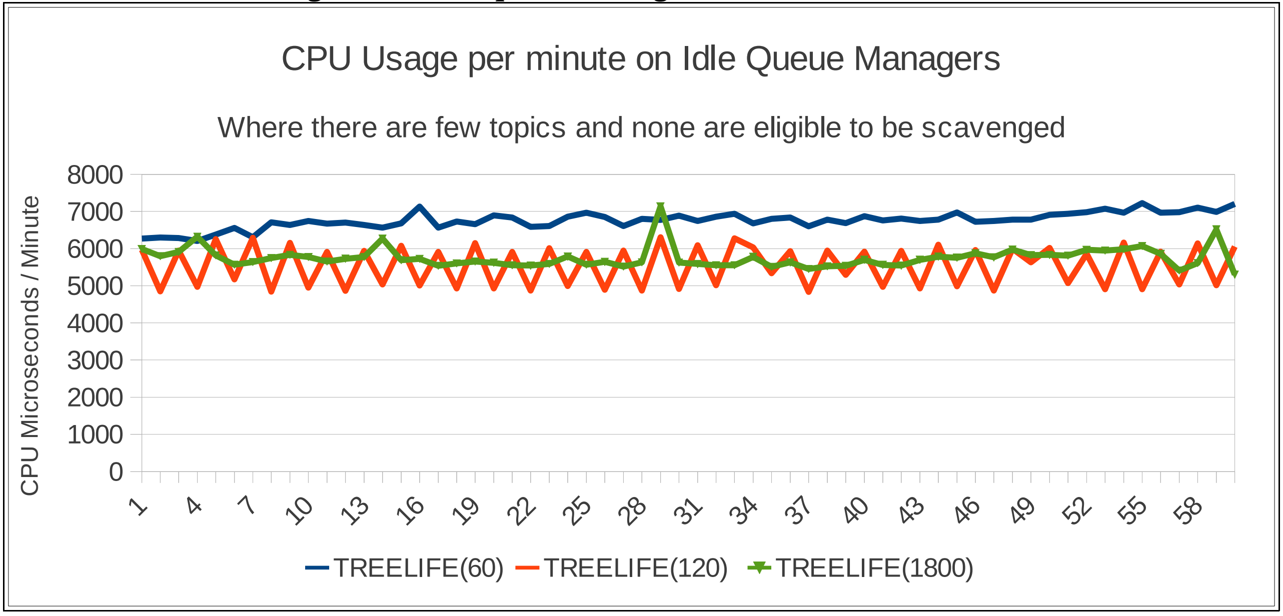
Running the topic scavenger has an impact on the costs attributed to the queue manager address space.

The following chart shows the queue manager address space CPU usage per minute. Costs are CPU microseconds on a 3 processor LPAR of a 2817-779

Between measurements, the TREELIFE attribute of the queue manager is altered.

The queue manager is not being used for any activities so only the additional cost of the TREELIFE setting is observed.

Chart 16: CPU usage on an idle queue manager



As can be seen from the measurements for TREELIFE(1800), when the topic scavenger runs there is a noticeable peak after 30 minutes.

The measurement for TREELIFE(120) shows peaks for the intervals where the scavenger is running.

The measurement for TREELIFE(60) is consistently higher and suggests that with no topics in the queue manager and no pub/sub occurring, a value of 60 is too frequent.

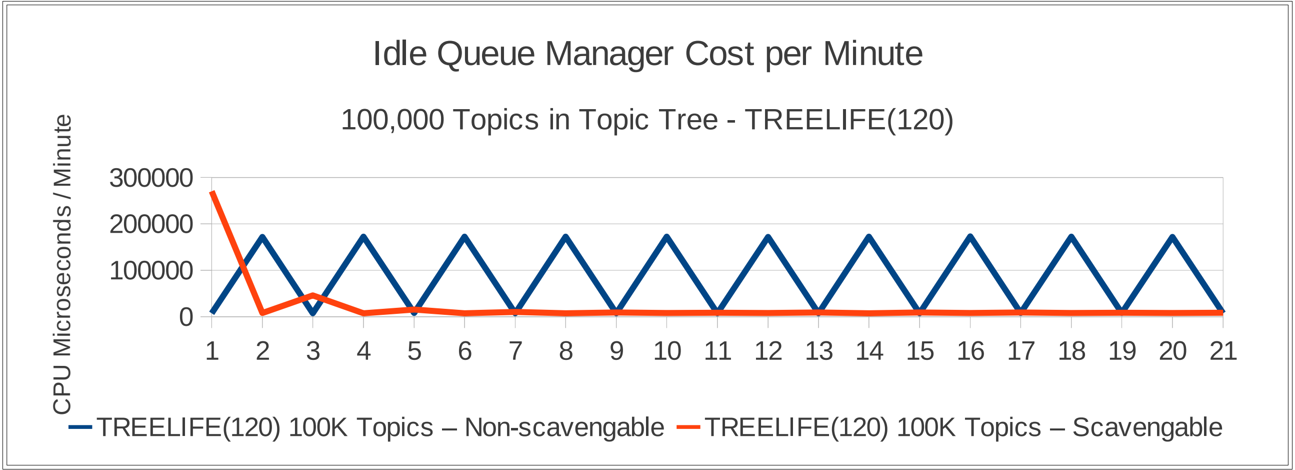
Topic scavenger on idle queue manager – with 100,000 Topics

In a system which involves a number of topic nodes, their eligibility for removal can affect your decision to set the value of TREELIFE.

The following chart shows the cost of an idle queue manager when the TREELIFE is set to 120 seconds. In each case, the queue manager costs are the total for 60 second periods.
The costs are in CPU microseconds on a 3 processor LPAR of a 2817-779.
The 2 measurements shown on the chart are for:

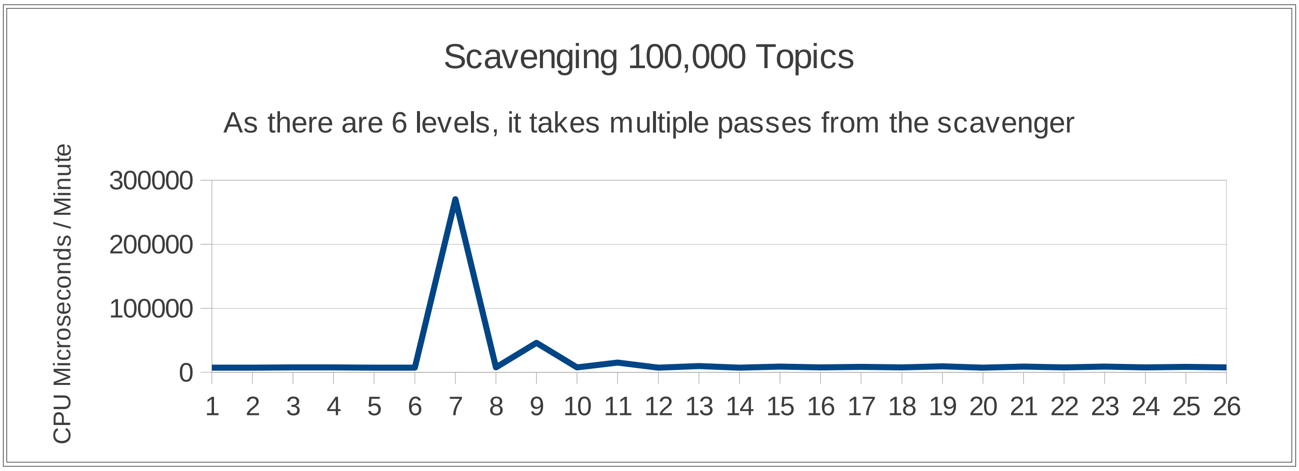
1. 100,000 topics nodes are known to the queue manager but are not eligible for removal
2. 100,000 topics nodes are known to the queue manager and are eligible for removal

Chart 17: Cost of topic scavenger on an idle queue manager



The measurement for the topics that are ineligible for removal shows the peaks when the topic scavenger runs, scanning the entire tree for eligible leaf nodes to remove, and finding none.
Once all topics have been removed in the scavengable case, the costs go back to those shown for no topics on the previous page.
The measurement for topics that are eligible for removal shows several peaks where the topic scavenger is able to remove leaf nodes over several runs. This is shown in more detail in the following chart.

Chart 18: Cost of scavenging 100,000 topics

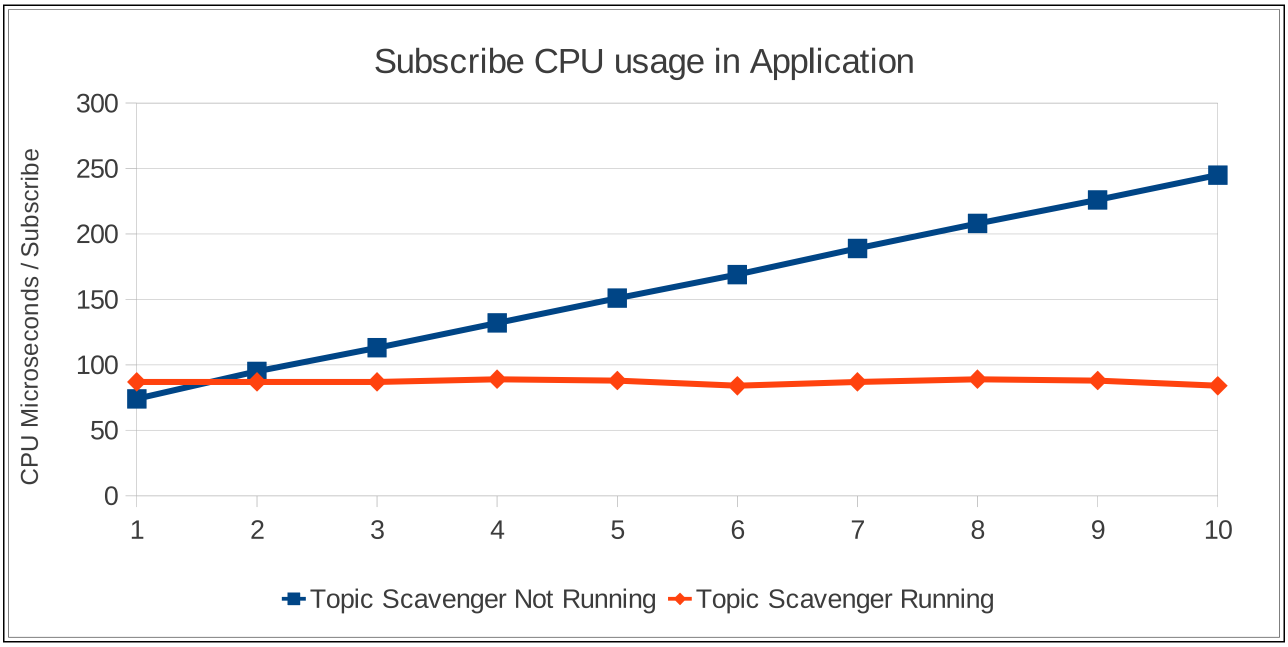


The 2 previous charts show that if there are a significant number of topics known to the queue manager that are ineligible for removal there will be an increased cost associated with running the scavenger too often.

Effect of topic scavenger on subscribe

When a topic tree contains many unwanted topics, the cost of adding a new subscription can be higher than subscribing to an empty tree. This can be particularly significant where wildcard subscriptions are being made and the wildcard topic string needs to be evaluated against all of the topic nodes that exist below the point in the wildcard topic string where the first wildcard character is located or where the topic tree is dynamic.
Each point on the chart is the average cost (in CPU microseconds) of the MQSUB based on an average of 100,000 subscribes.
When the topic scavenger is not running (TREELIFE(3600)) the average cost of the MQSUB when there are 1,000,000 unwanted topics is 3.5 times higher than when there are fewer than 100,000 subscriptions in the queue manager.
By running the topic scavenger sufficiently frequently, the cost of the subscribe remains consistent as the number of unwanted topics does not increase.

Chart 19: Cost of subscriber application with increasing subscriptions in the queue manager



The total CPU usage for the queue manager when the scavenger was running often was 3% higher than when running with the scavenger running too infrequently.
However this 3% increase in queue manager cost was more than offset by the 45% reduction in application cost gained from running the topic scavenger.
In total, the CPU costs were 10% less when running the subscribes with topic scavenger running every 120 seconds.
This show that in an environment with a high turnover of topic subscriptions, there is benefit to be had running the topic scavenger more frequently than the default.

DISPLAY TPSTATUS with many topics in your system

The command “DISPLAY TPSTATUS('#') TYPE(TOPIC)” can be used to display the status of all of the topics in the topic tree.

WARNING: When there are many topics in the topic tree, this command should be used with caution.

If the command is issued to the console, only a small subset of the responses is displayed in the queue manager log, however the queue managers command server task builds an internal list of all of the matching topics which will have 2 effects:

1. A significant amount of queue manager private storage is used, potentially leaving the queue manager short on storage for a period of time. This can be seen with the appearance of CSQY220I messages.
2. A significant amount of CPU is used to scan all of the topics.

In addition, if the command is issued using CSQUTIL, the RESPTIME attribute passed should be increased from the default of 30 seconds otherwise the job will time-out. In the event of a time-out, the outstanding responses are delivered to the dead letter queue, potentially resulting in thousands of messages appearing on that queue.

As a guide, when the command was issued against a queue manager with 1 million topics defined in the system, the queue manager used 500MB of private storage to hold the response and 1 processor on a 3-processor LPAR of a zEnterprise 196 (2817-779) was running at 100% for 55 minutes processing the command.

Performance data

Shared message data set scenarios

- Single Queue Manager – non-persistent out-of-syncpoint
- Single Queue Manager – non-persistent server-in-syncpoint
- Single Queue Manager – persistent in-syncpoint

- Two Queue Manager – data sharing non-persistent out-of-syncpoint
- Two Queue Manager – non-data sharing non-persistent out-of-syncpoint
- Two Queue Manager – data sharing non-persistent server-in-syncpoint
- Two Queue Manager – non-data sharing non-persistent server-in-syncpoint

- Three Queue Manager - data sharing non-persistent out-of-syncpoint
- Three Queue Manager - data sharing non-persistent server-in-syncpoint
- Three Queue Manager - data sharing persistent in-syncpoint

- Six Queue Manager – data sharing non-persistent (server-in-syncpoint)
- Six Queue Manager – data sharing persistent

- Scaling – Non-Persistent Large Messages
  - Non-Persistent 10KB Messages
  - Non-Persistent 100KB Messages
  - Non-Persistent 1MB Messages
  - Non-Persistent 1MB Messages – multiple applications per queue pair

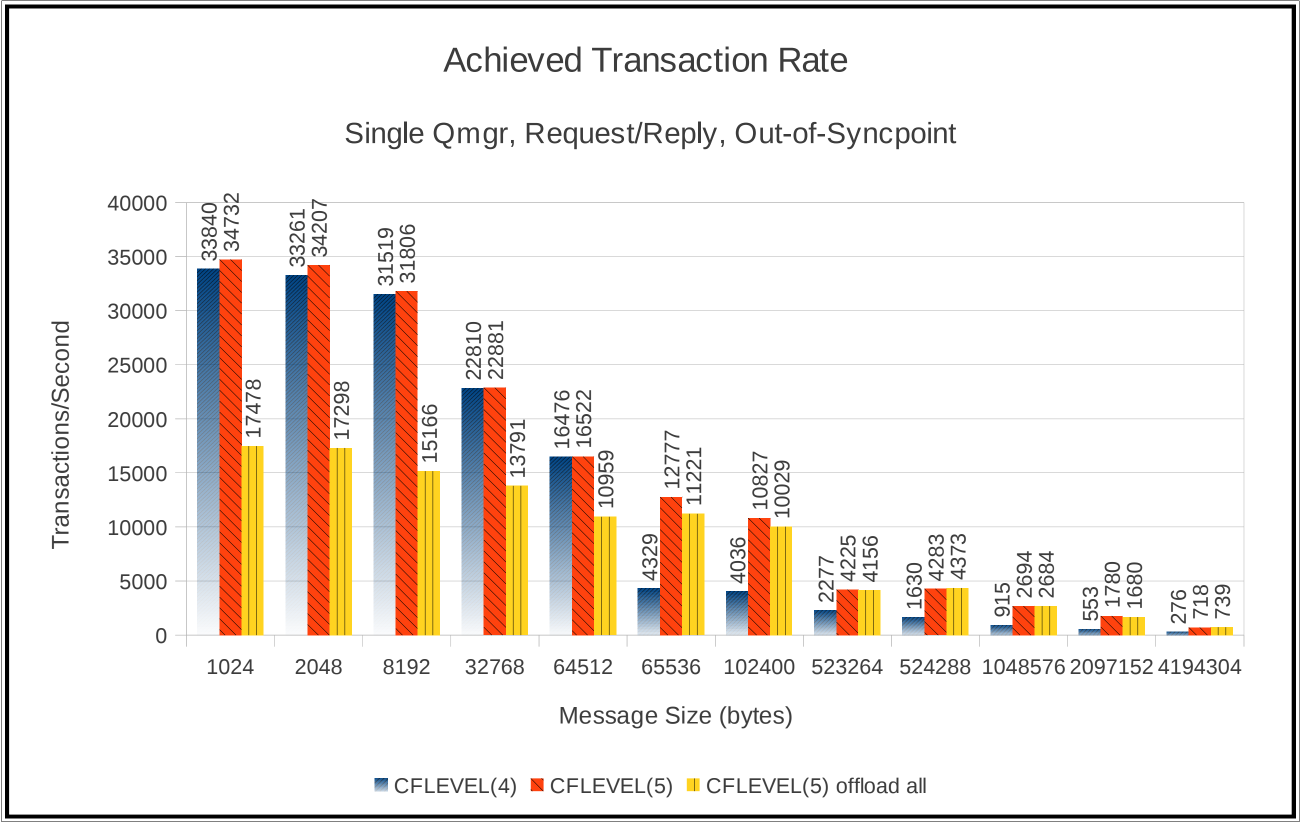
Notes:

1. Costs are in CPU microseconds unless otherwise stated.
2. For simplicity, the two, three and six queue manager measurements show comparisons between CFLEVEL(4) and CFLEVEL(5) with OFFLOAD for all messages. For message sizes below 63KB, there is no difference between CFLEVEL(4) and CFLEVEL(5) with no OFFLOAD.

Single queue manager – non-persistent out-of-syncpoint

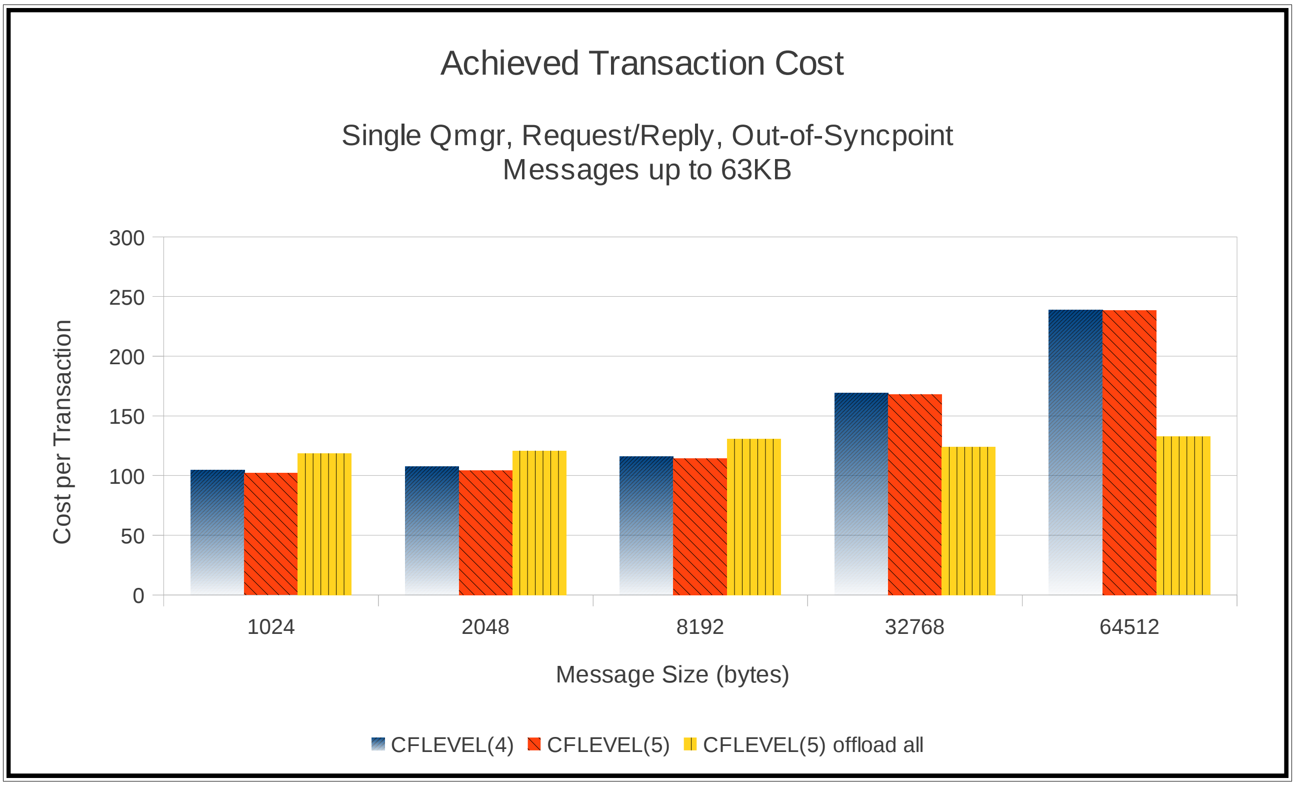
These tests run a request/reply model for a range of message sizes.
The requesters put a request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. These puts and gets are out of syncpoint.
The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are out of syncpoint.

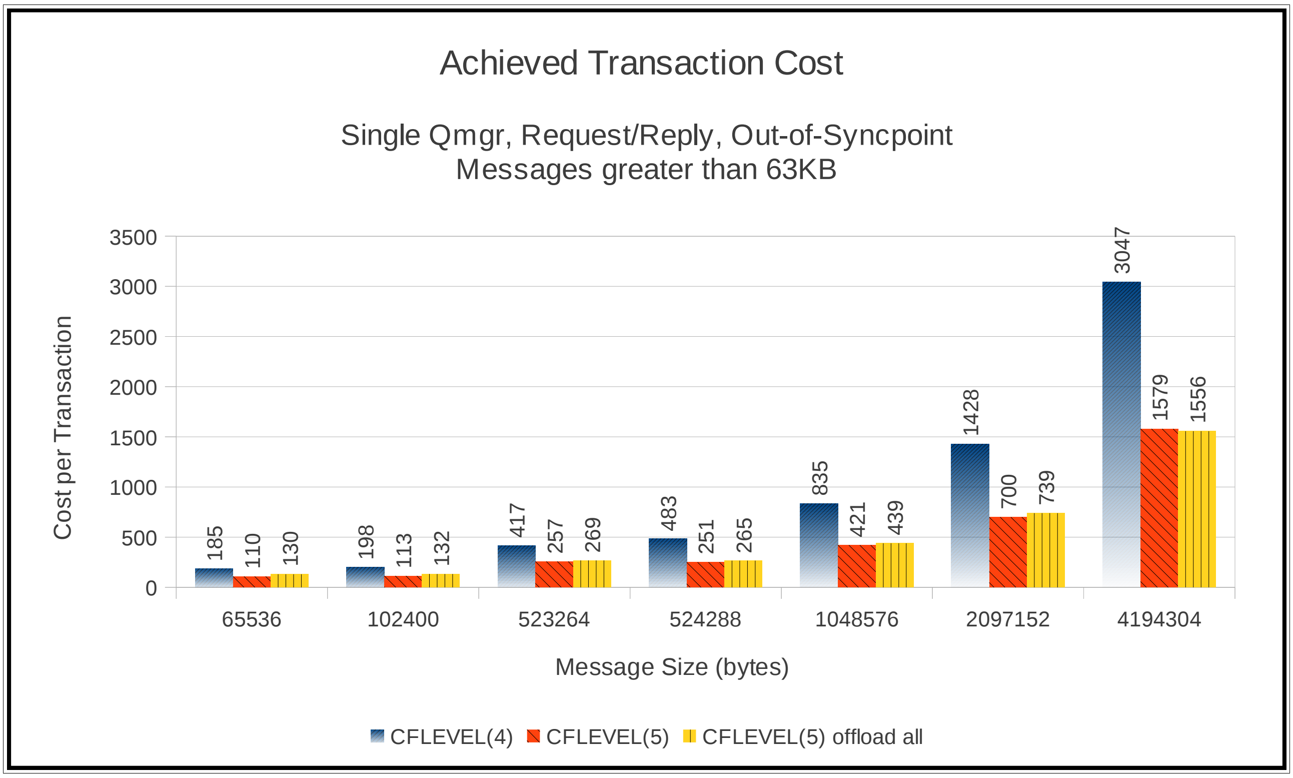
Chart 20: Achieved transaction rate



The above chart shows that CFLEVEL(4) and CFLEVEL(5) are comparable for messages up to 63KB. Once the message size exceeds 63KB, offloading the message to SMDS rather than DB2 allows the transaction rate to decrease more steadily as the message size increases.
The “CFLEVEL(5) offload all” (to SMDS) column shows the effect where the offload thresholds have been exceeded, and messages smaller than 64KB are offloaded.

Chart 21 and 21a: Observed transaction cost



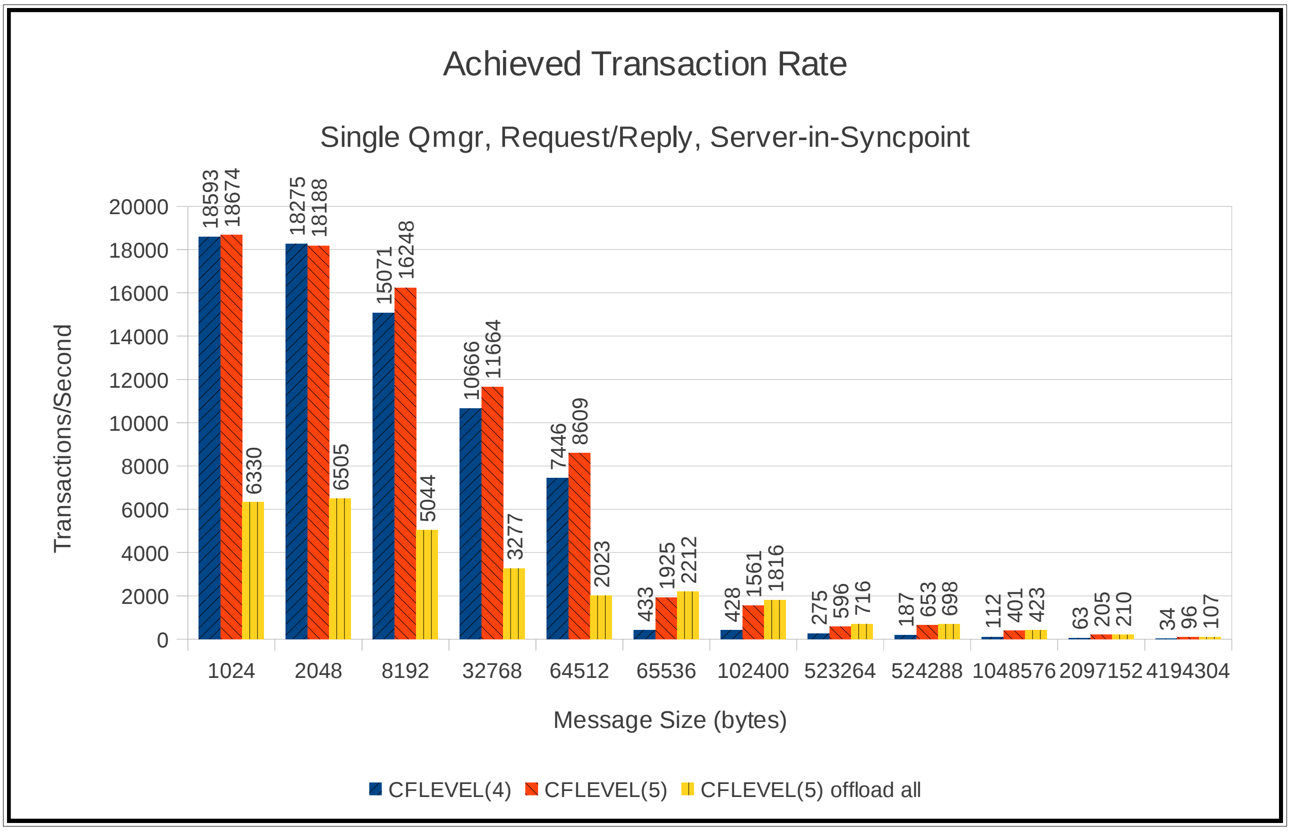


The cost of offloading messages to shared message data sets is significantly less than offloading to DB2.

Single queue manager – non-persistent server-in-syncpoint

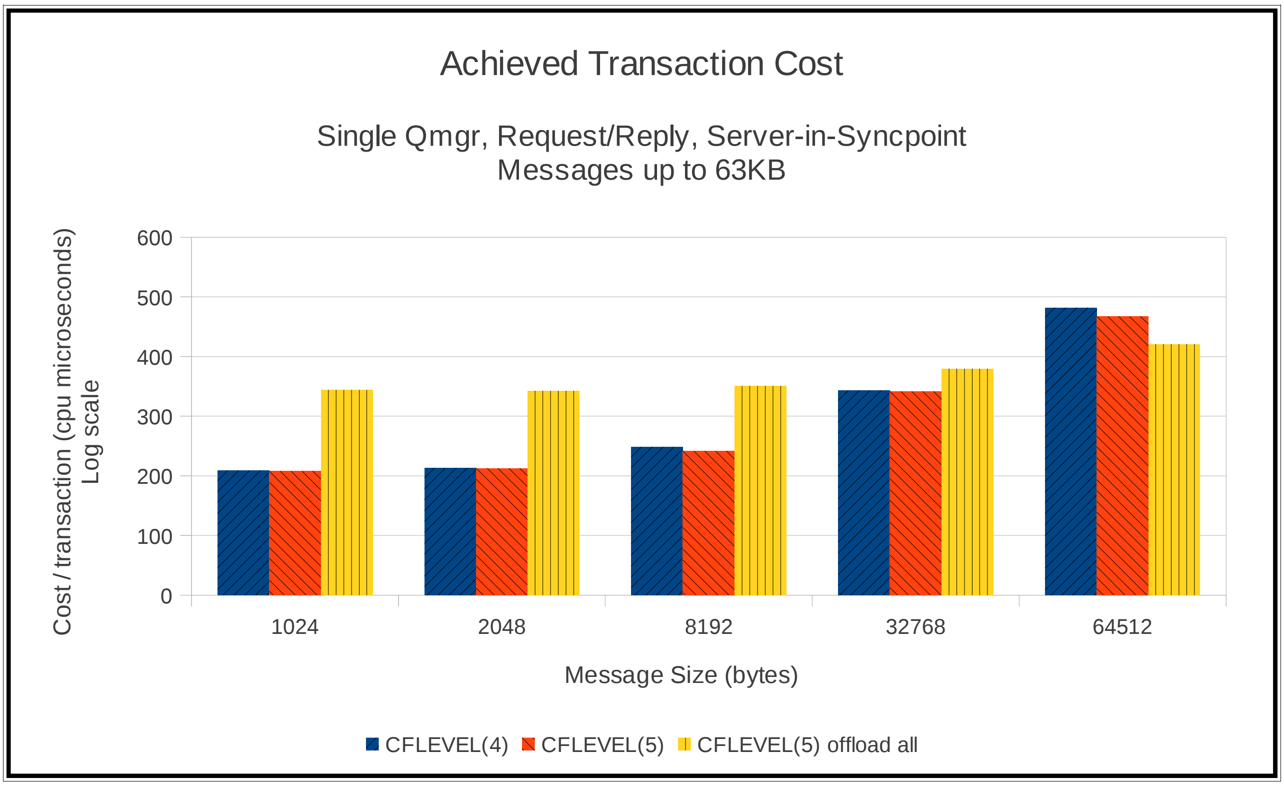
These tests run a request/reply model for a range of message sizes.
The requesters put a request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. These puts and gets are out of syncpoint.
The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are in-syncpoint.

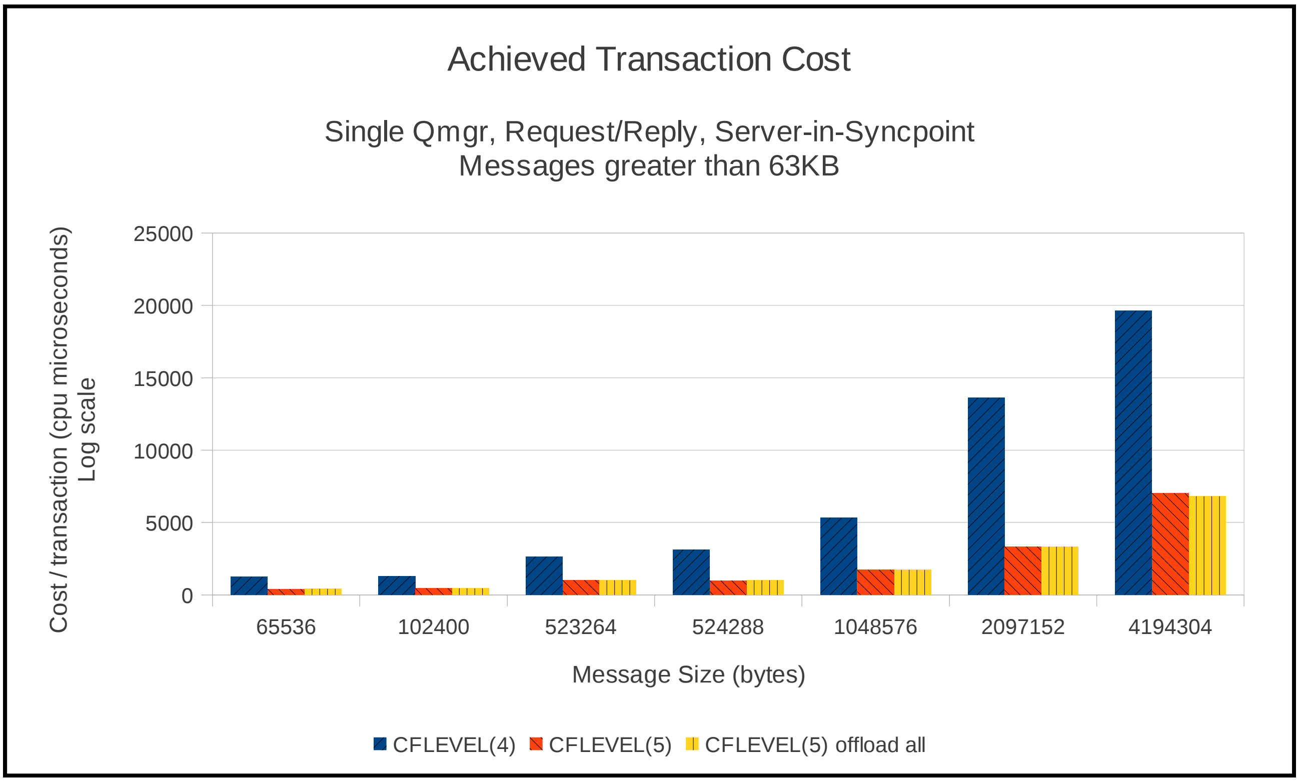
Chart 22: Achieved transaction rate



This chart shows that CFLEVEL(4) and CFLEVEL(5) are comparable for messages up to 63KB. Once the message size exceeds 63KB, offloading the message to SMDS rather than DB2 allows the transaction rate to decrease more steadily as the message size increases.
The “CFLEVEL(5) offload all” column shows the achieved transaction rate when the offload thresholds have been exceeded, and messages smaller than 64KB are offloaded.

Chart 23 + 23a: Observed transaction costs

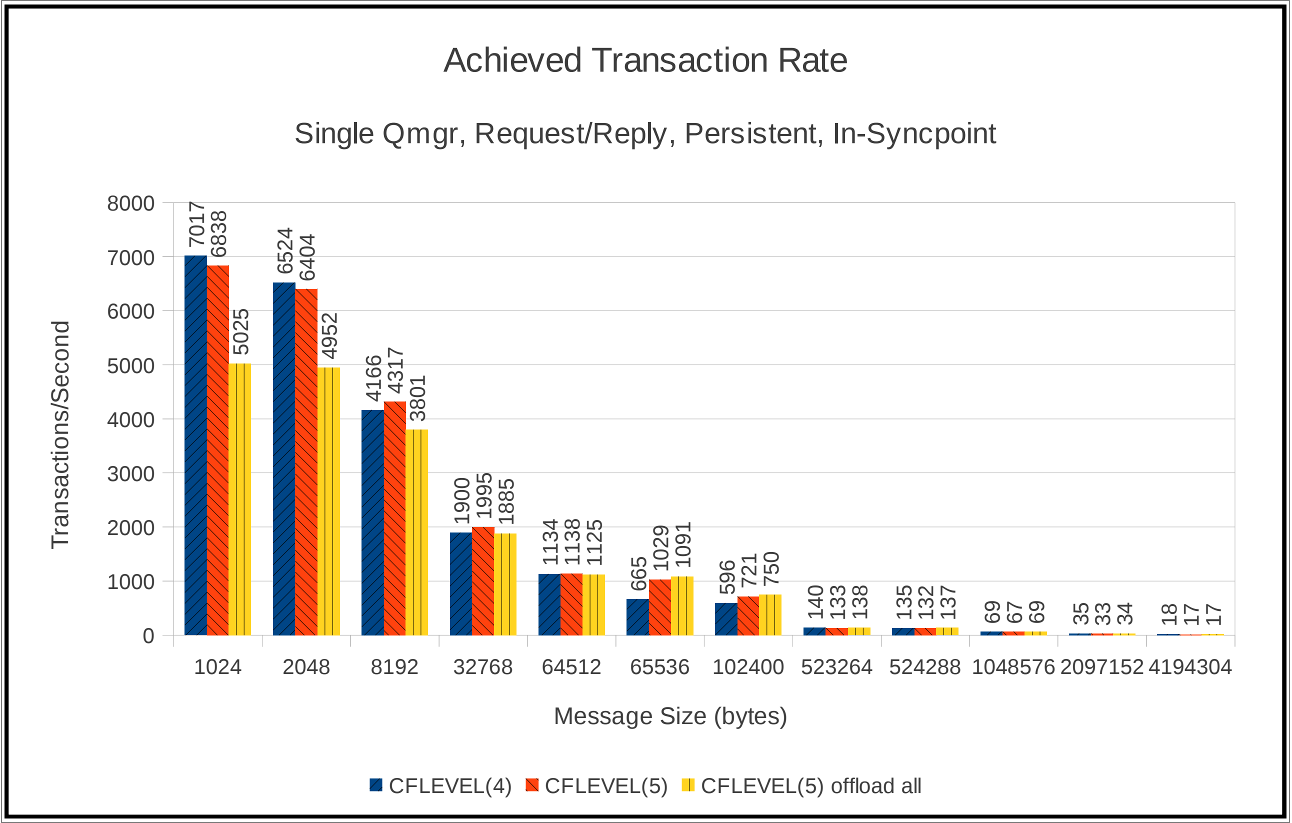




Again the cost of offloading the messages to shared message datasets is considerably less than offloading to DB2.

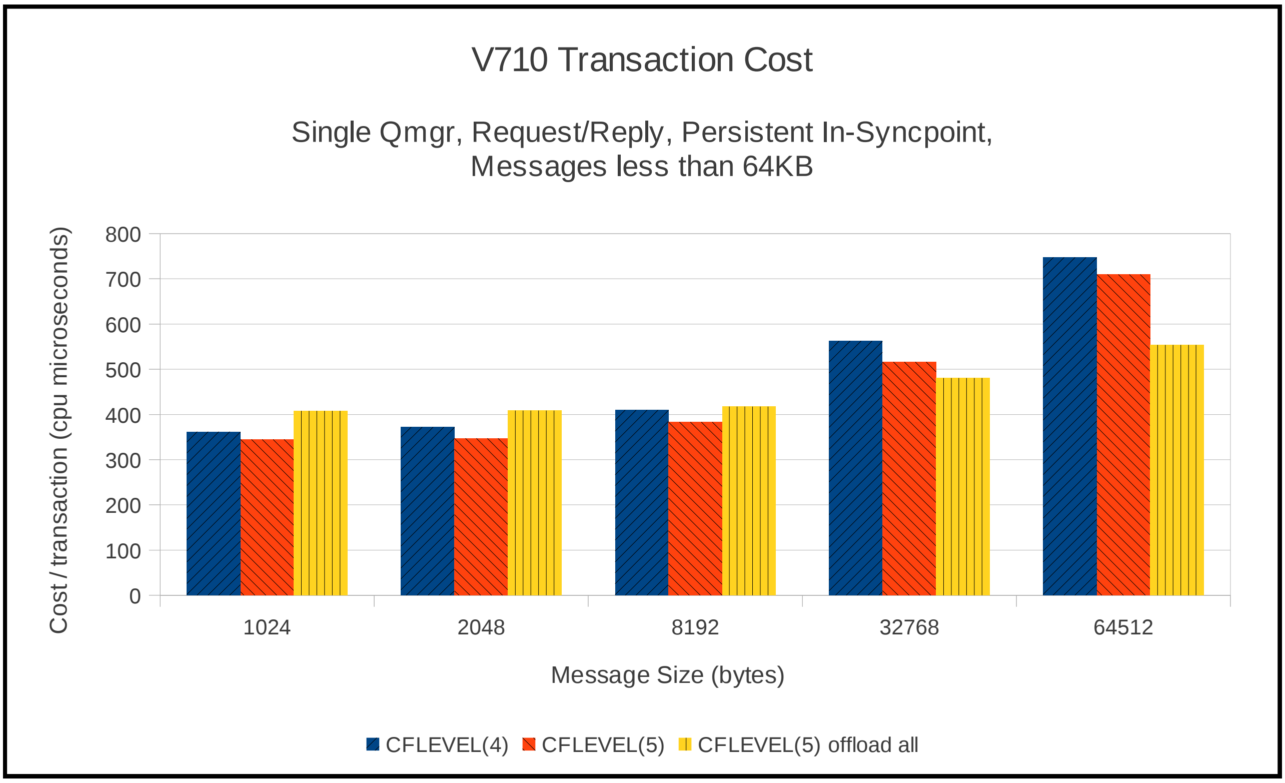
Single queue manager – persistent in-syncpoint

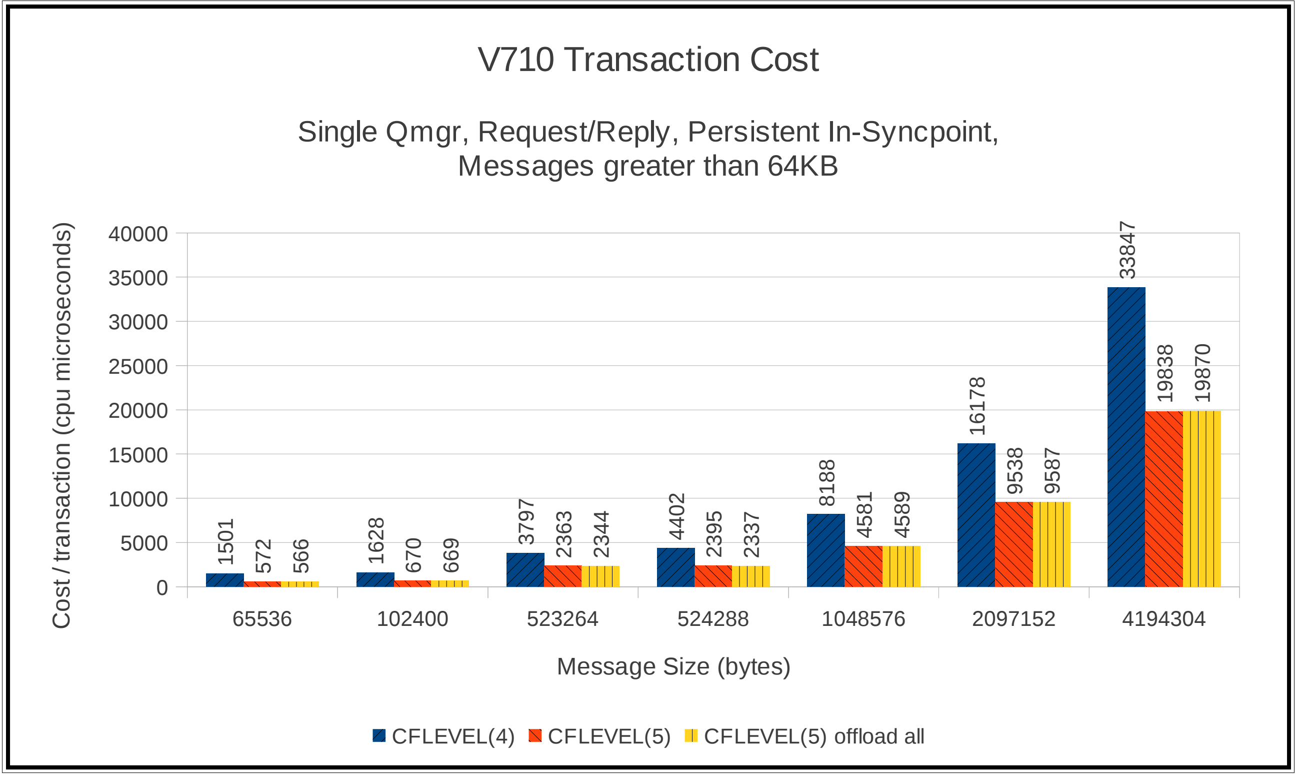
These tests run a request/reply model for a range of message sizes.
The requesters put a persistent request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. The put is in-syncpoint, as is the get.
The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are in-syncpoint.
Chart 24: Achieved transaction rate



This chart shows that CFLEVEL(4) and CFLEVEL(5) are comparable for messages up to 63KB. Once the message size exceeds 63KB, offloading the message to SMDS rather than DB2 allows the transaction rate to decrease more steadily as the message size increases.
The “CFLEVEL(5) offload all” column shows the achieved transaction rate when the offload thresholds have been exceeded, and messages smaller than 64KB are offloaded.

Chart 25 + 25a: Observed transaction cost





Two queue managers – data sharing, non-persistent, out-of-syncpoint

These tests run a request/reply model for a range of message sizes.
The requesters put a non-persistent request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. The put is out of syncpoint as is the get.
The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are also out of syncpoint.
The messages put by applications connected to queue manager 1 will be got by applications connected to queue manager 2 and vice versa. Hence the term “data sharing”.

Chart 26: Achieved transaction rate – small messages

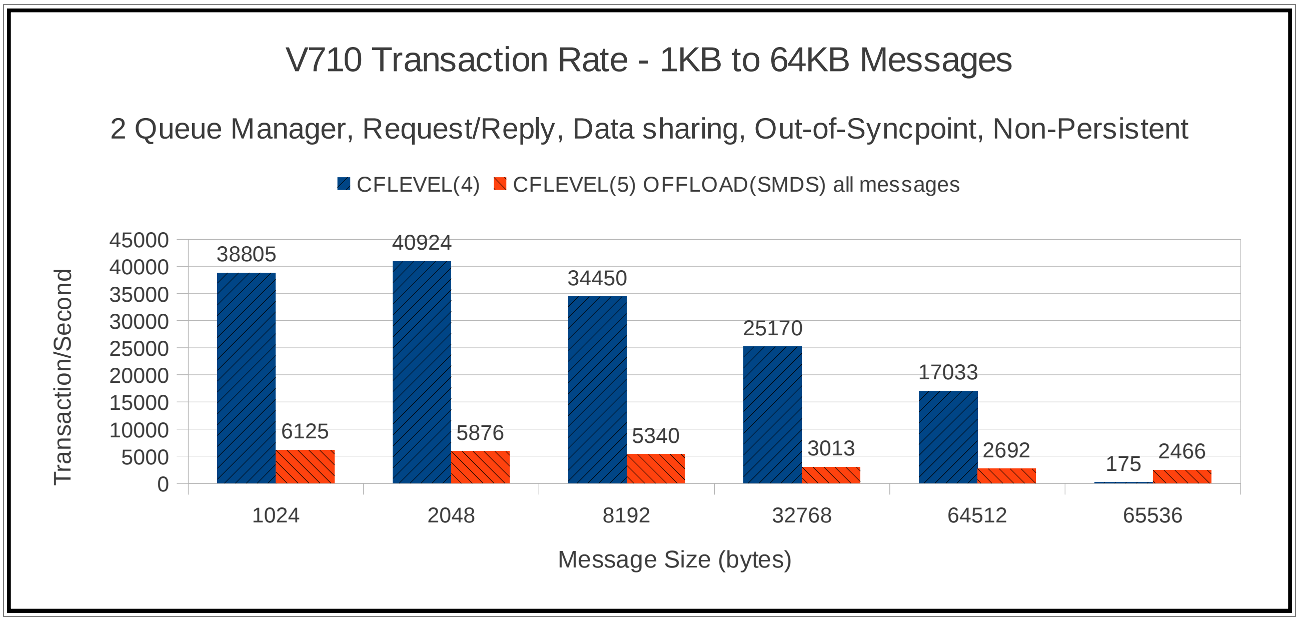


Chart 27: Achieved transaction rate – larger messages

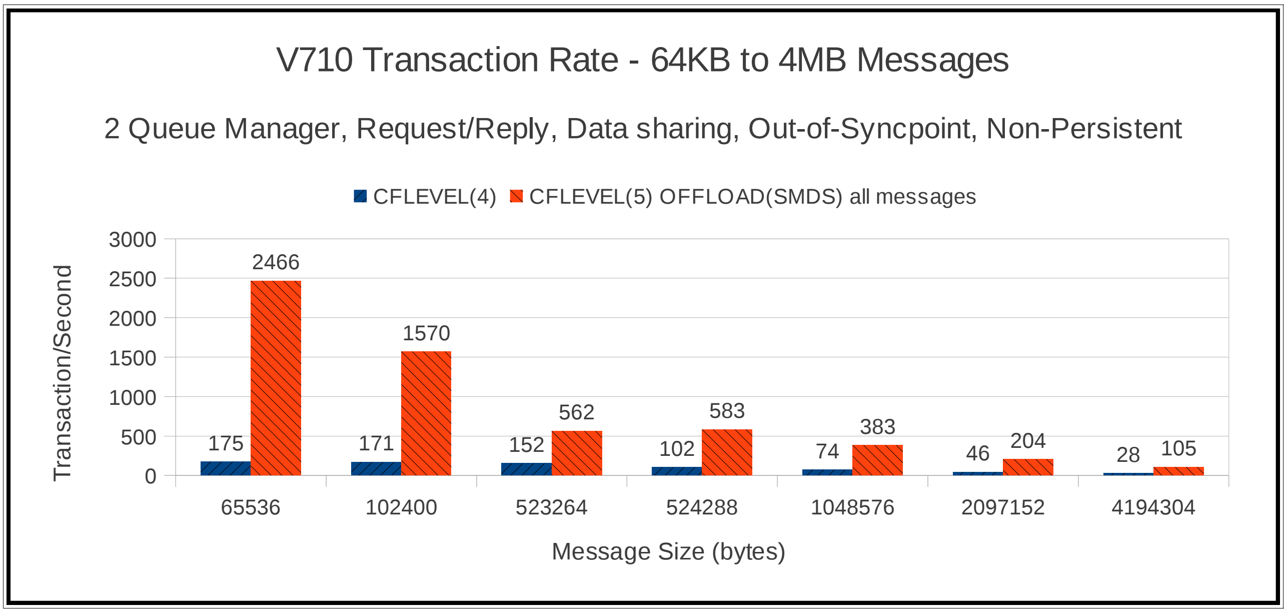


Chart 28: Observed transaction cost – small messages

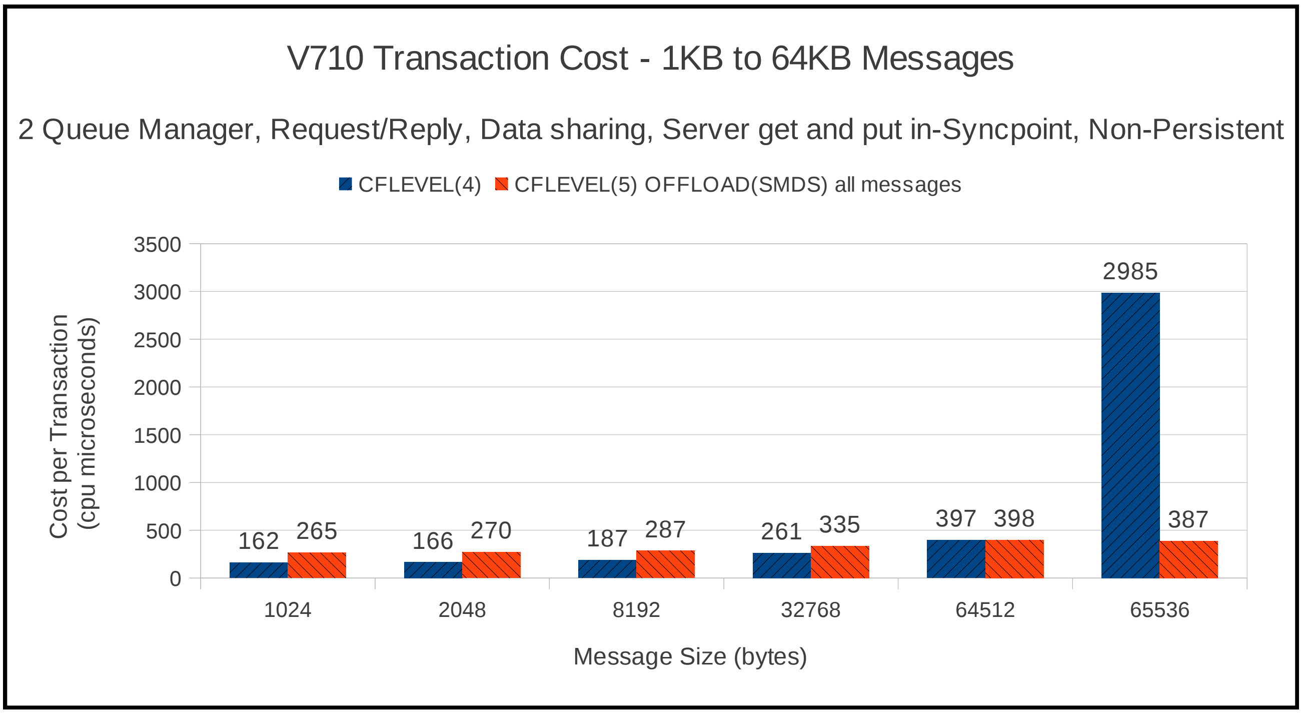
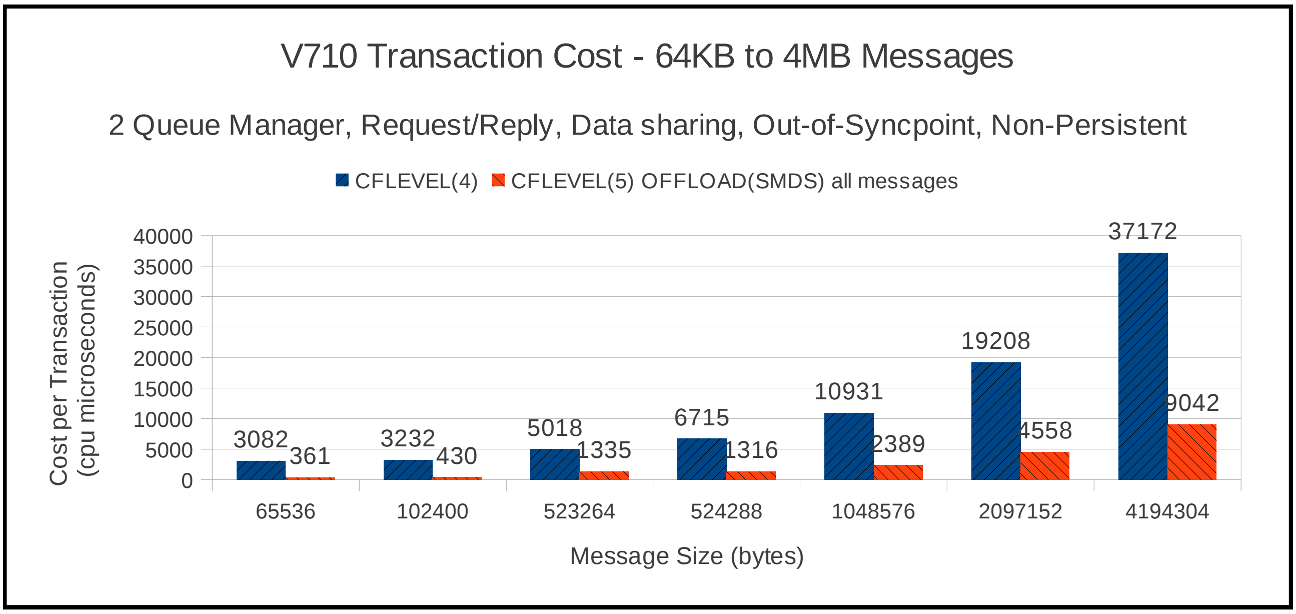


Chart 29: Observed transaction cost – larger messages



As the measurements shown by charts 26 to 29 are data-sharing, this means that the queue manager performing the MQGET always has to read another queue managers' shared message data set. This means that there is no performance optimisation that can be made from shared message buffers.

Two queue managers – non-data sharing, non-persistent, out-of-syncpoint

These tests run a request/reply model for a range of message sizes.
The requesters put a non-persistent request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. The put is out of syncpoint, as is the get.
The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are also out of syncpoint.
In these measurements, a common set of queues are used – all applications putting messages use the same set of request and reply queues. This means that a message put by an application connected to queue manager 1 may be processed by an application connected to either queue manager 1 or queue manager 2.

Chart 30: Achieved transaction rate – small messages

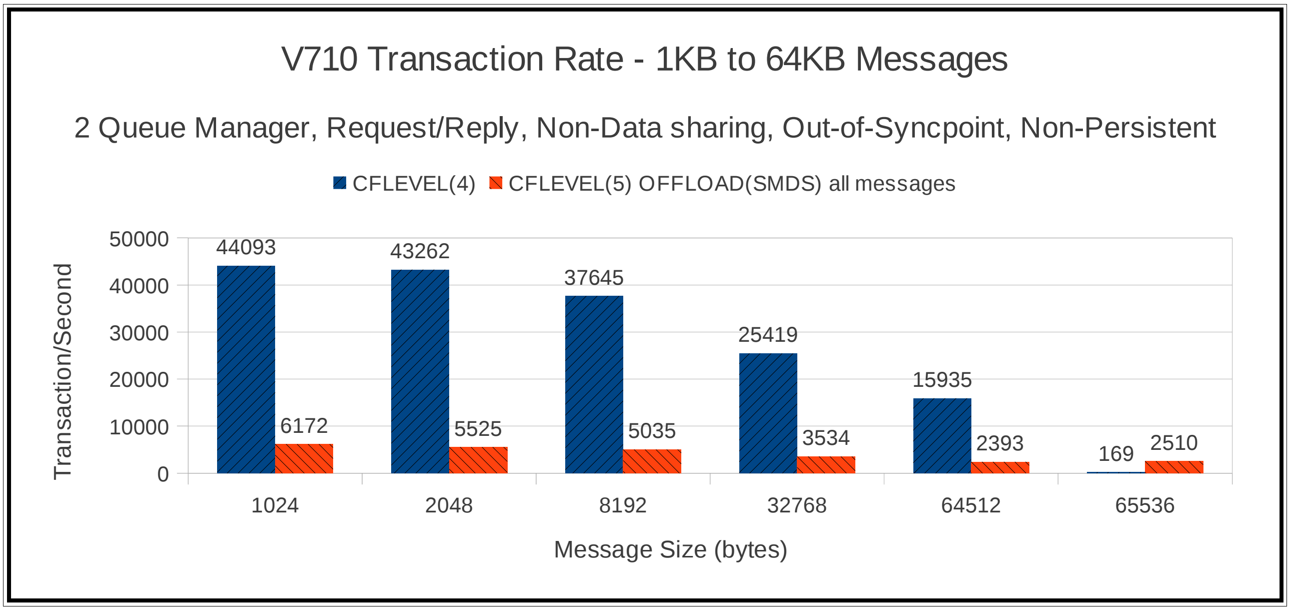


Chart 31: Achieved transaction rate – larger messages

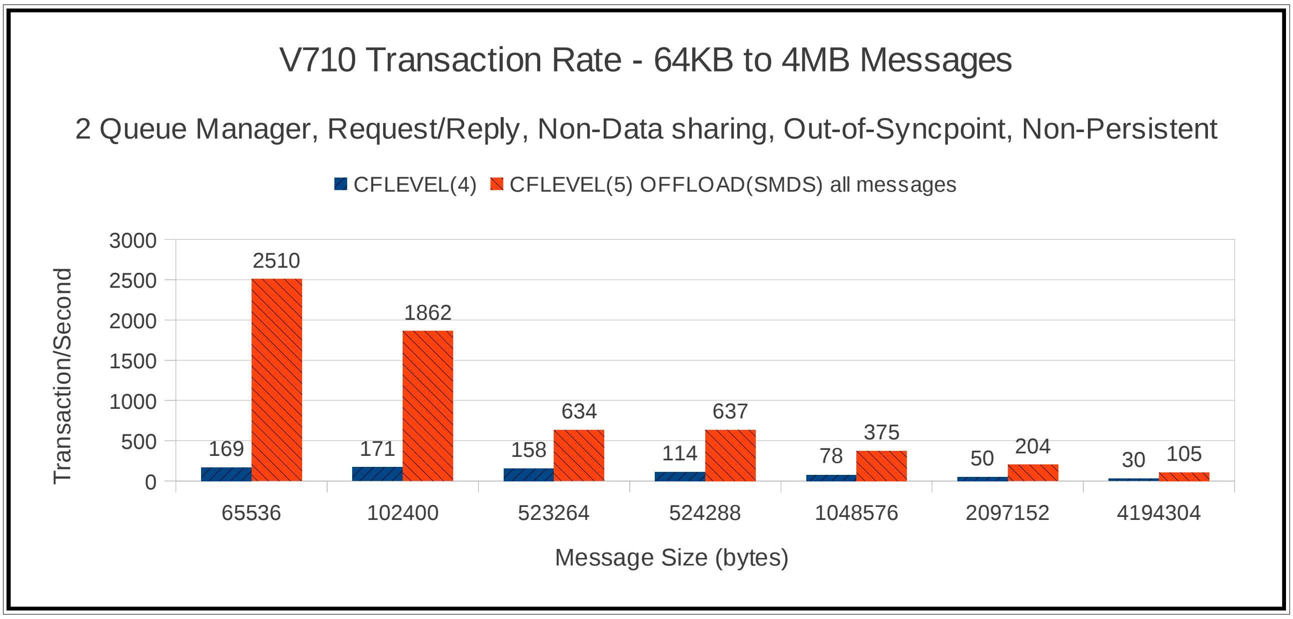


Chart 32: Observed transaction cost – small messages

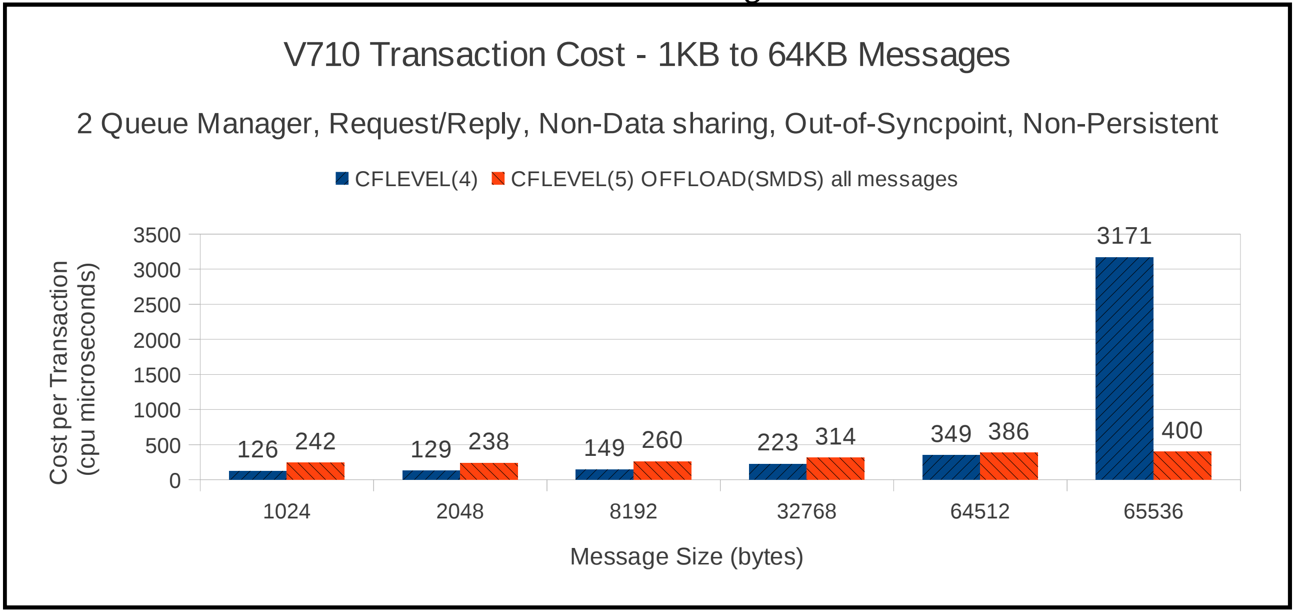
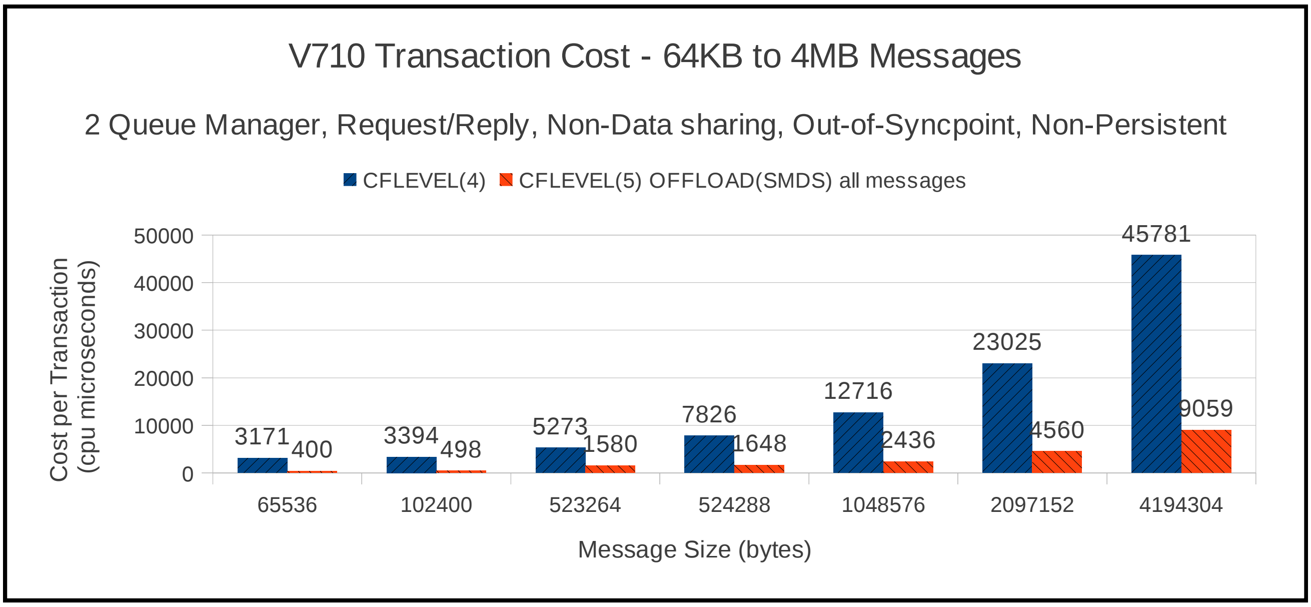


Chart 33: Observed transaction cost – larger messages



As the measurements shown by charts 30 to 33 are non-datasharing, this means that the queue manager performing the MQGET may be the same as the queue manager that performed the MQPUT. As a result some benefit can be gained from the use of shared message buffers but the majority of any gains are more likely to be in putting the message to the getting application's buffer.

Two queue managers – data sharing, non-persistent, in-syncpoint

These tests run a request/reply model for a range of message sizes.
The requesters put a non-persistent request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. The put is out of syncpoint as is the get.
The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are in-syncpoint.
The messages put by applications connected to queue manager 1 will be got by applications connected to queue manager 2 and vice versa. Hence the term “data sharing”.

Chart 34: Achieved transaction rate – small messages

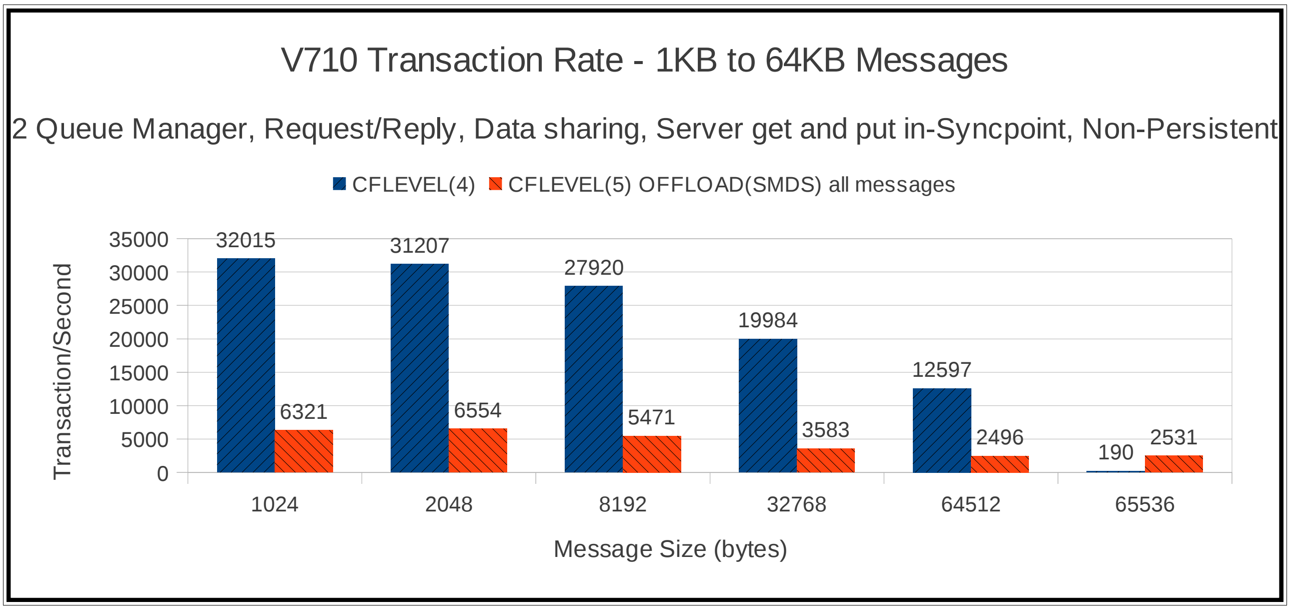


Chart 35: Achieved transaction rate – larger messages

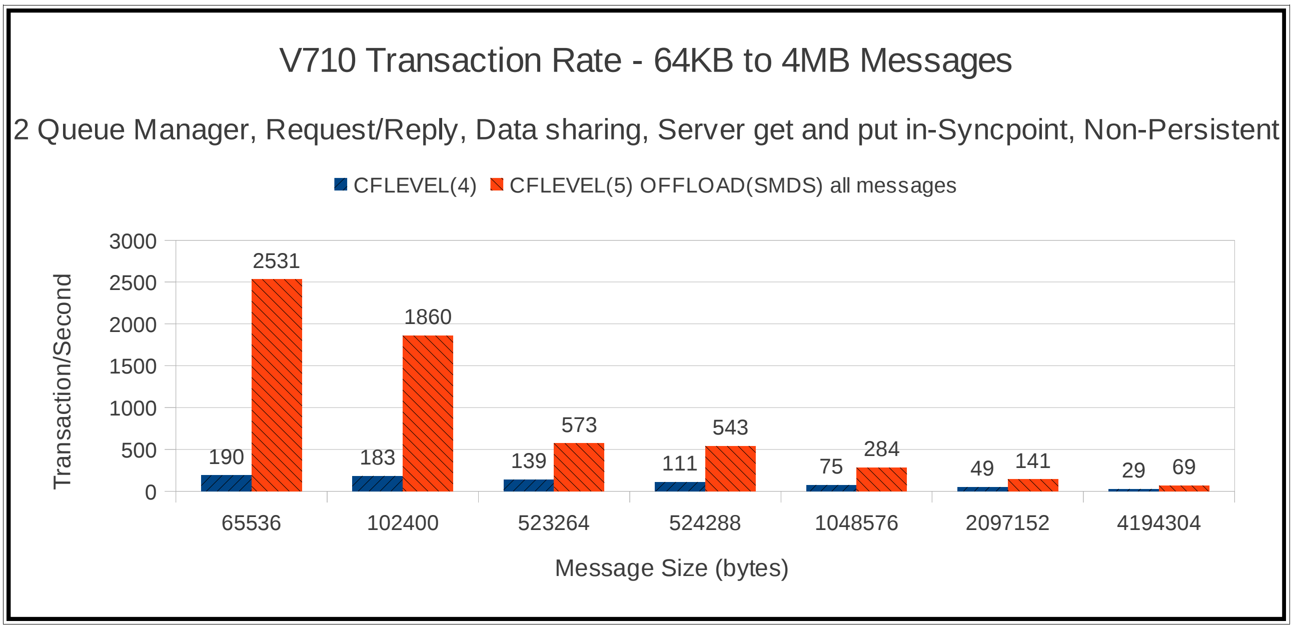


Chart 36: Observed transaction cost – small messages

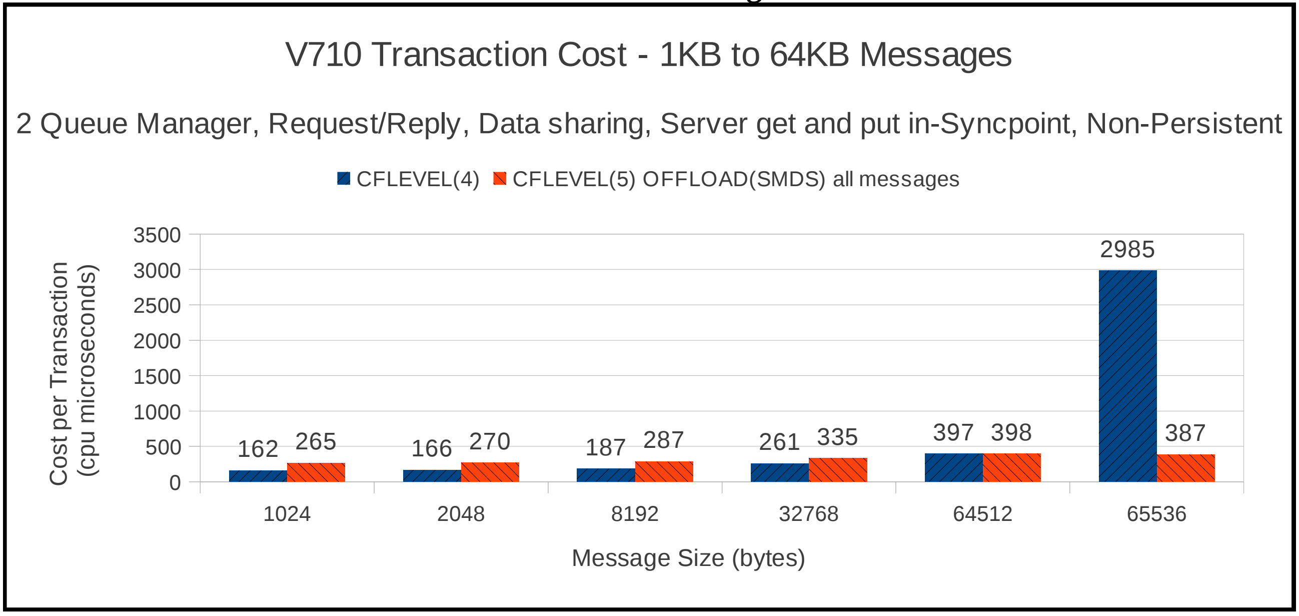
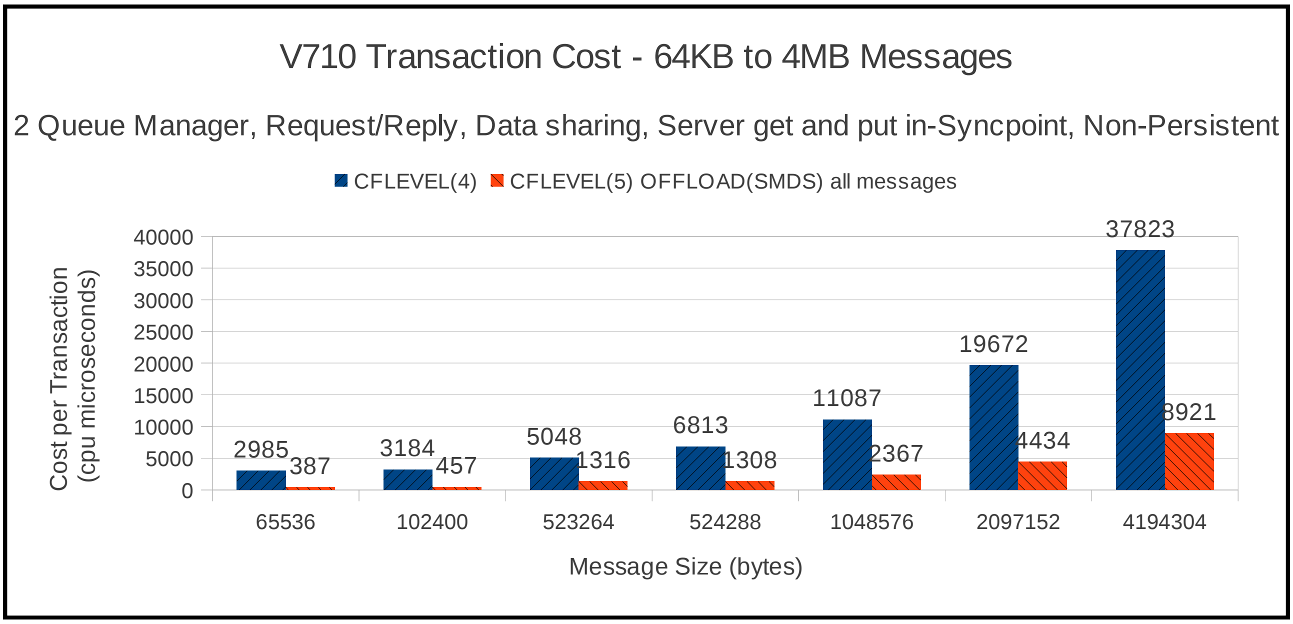


Chart 37: Observed transaction cost – larger messages



Two queue managers – non-data sharing, non-persistent, in-syncpoint

These tests run a request/reply model for a range of message sizes.

The requesters put a non-persistent request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. The put is out of syncpoint as is the get.

The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are in-syncpoint.

In these measurements, a common set of queues are used – all applications putting messages use the same set of request and reply queues. This means that a message put by an application connected to queue manager 1 may be processed by an application connected to either queue manager 1 or queue manager 2.

Chart 38: Achieved transaction rate – small messages

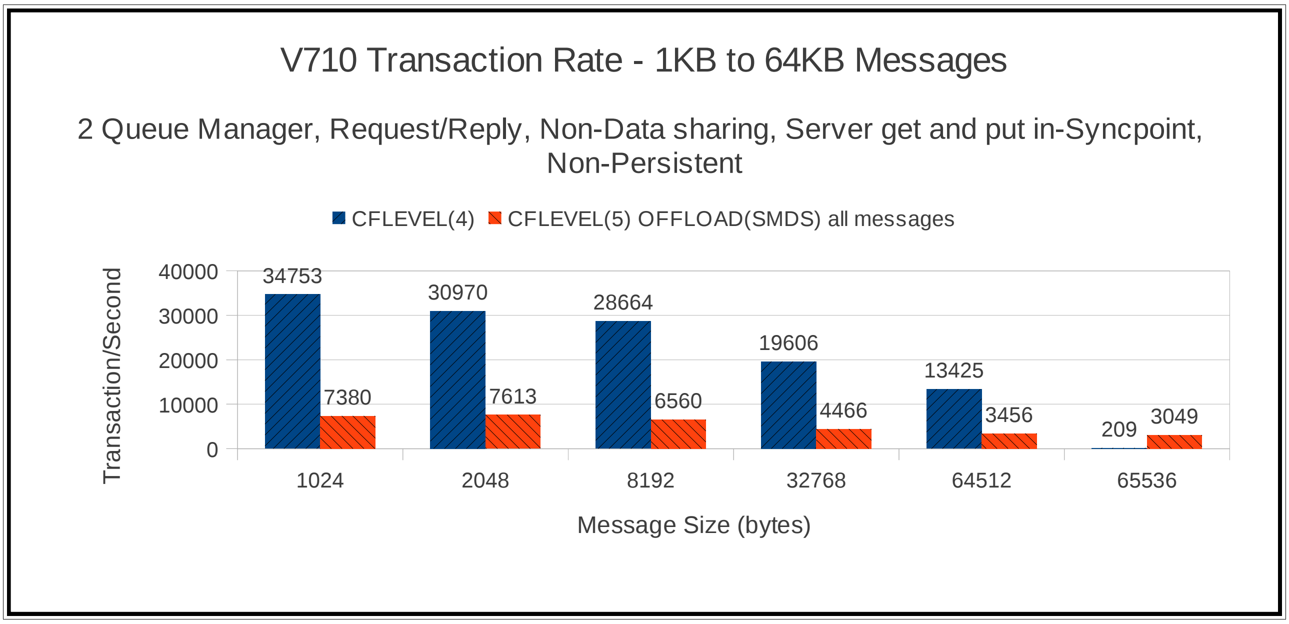


Chart 39: Achieved transaction rate – larger messages

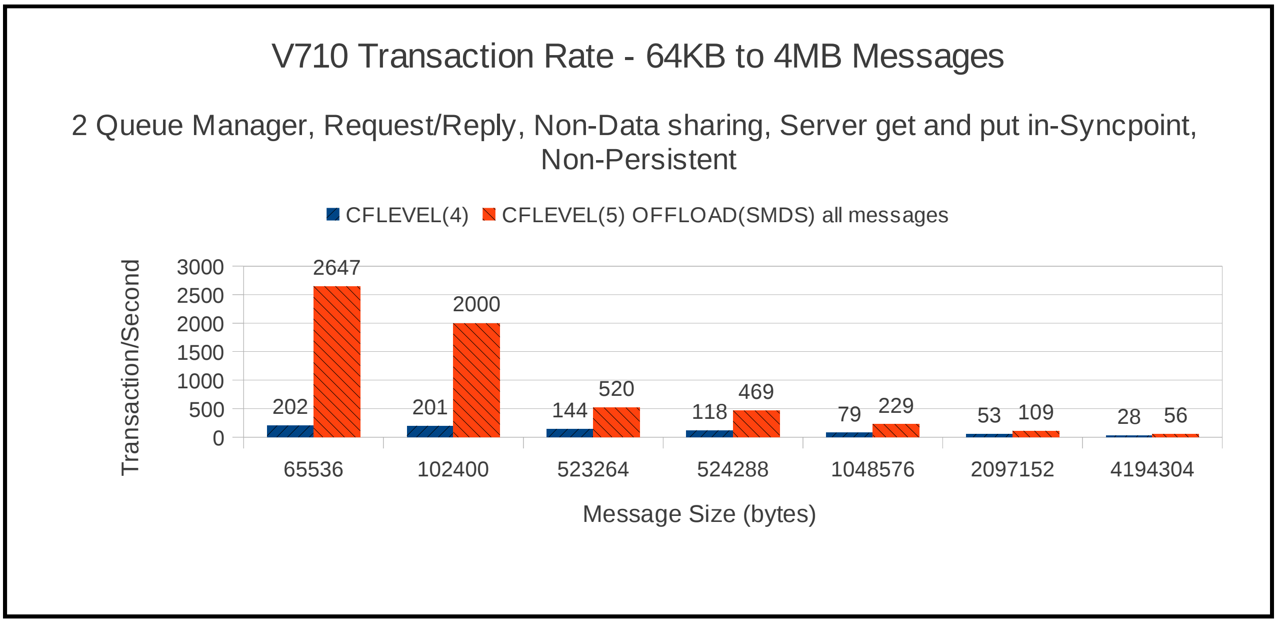


Chart 40: Observed transaction cost – small messages

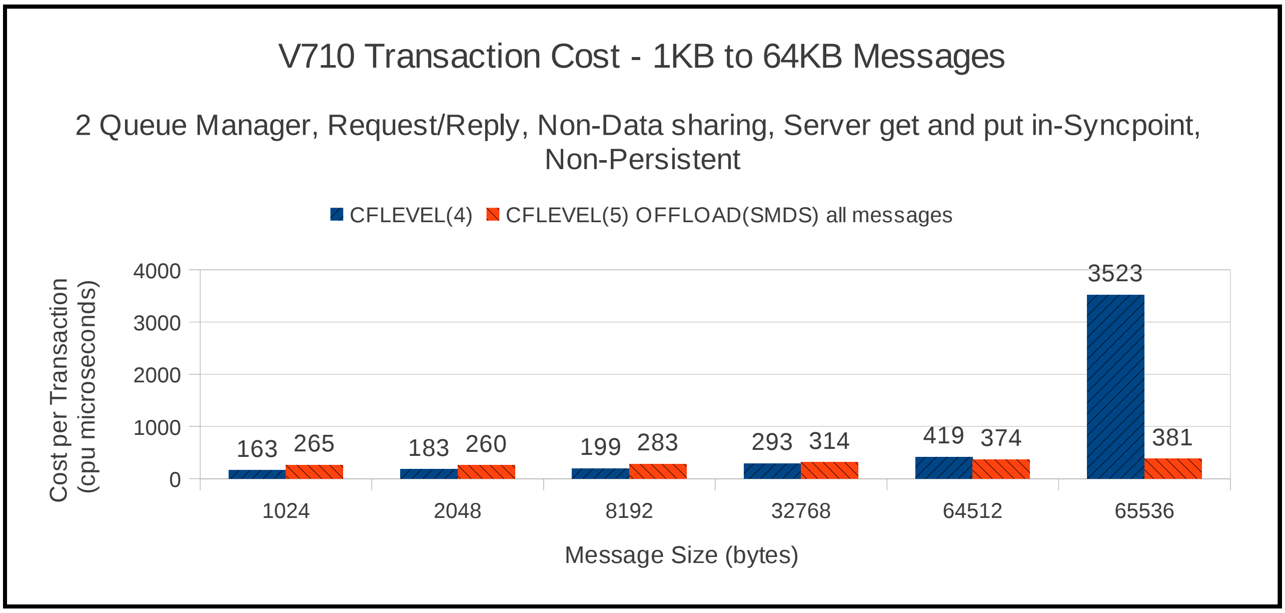
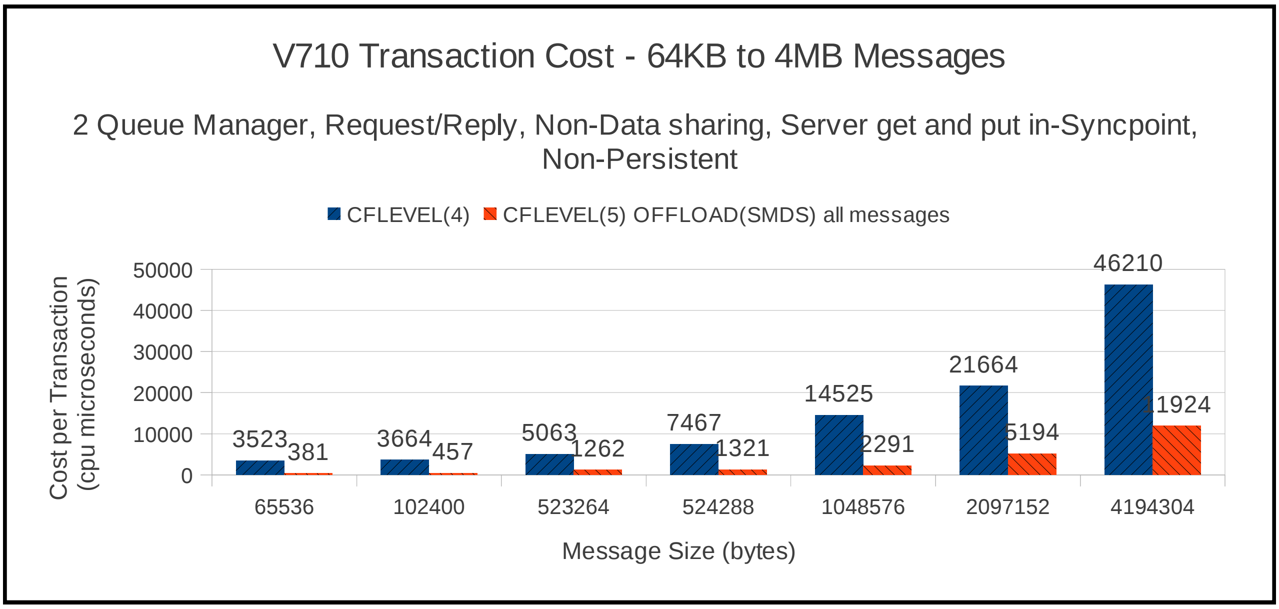


Chart 41: Observed transaction cost – larger messages



Three queue managers – data sharing, non-persistent, out of syncpoint

These tests run a request/reply model for a range of message sizes.

The requesters put a non-persistent request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. The put is out of syncpoint as is the get.

The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are out-of-syncpoint.

The messages put by requester applications connected to queue manager 1 will be got by server applications connected to queue manager 2. Messages put by requester applications connected to queue manager 2 will be got by server applications connected to queue manager 3 and messages put by requester applications connected to queue manager 3 will be got by server applications connected to queue manager 1.

Chart 42: Achieved transaction rate – small messages

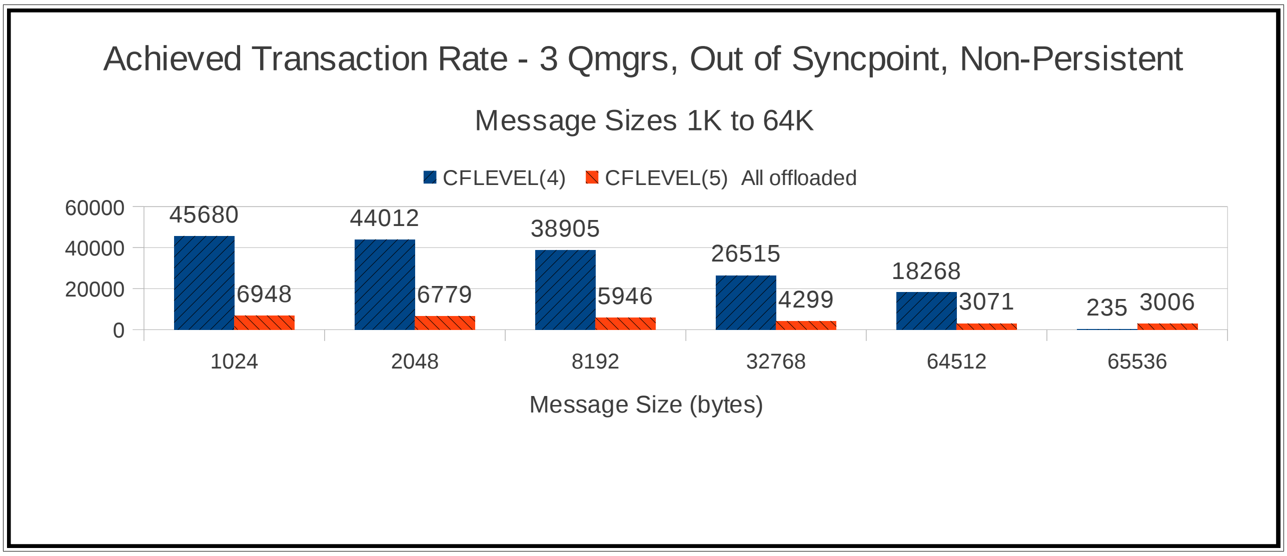


Chart 43: Achieved transaction rate – larger messages

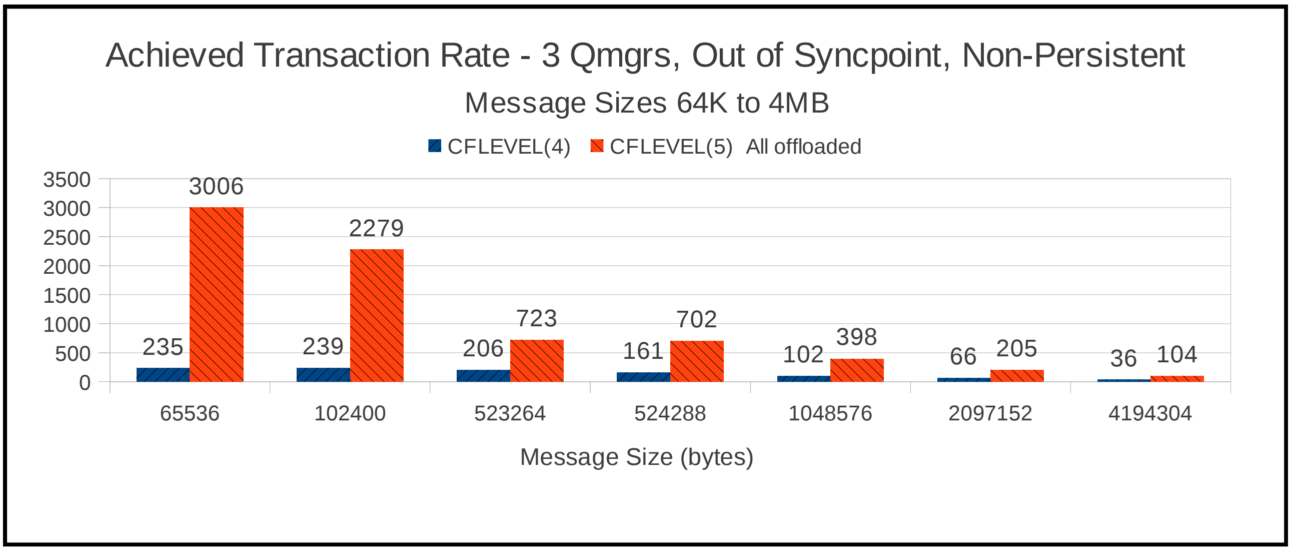
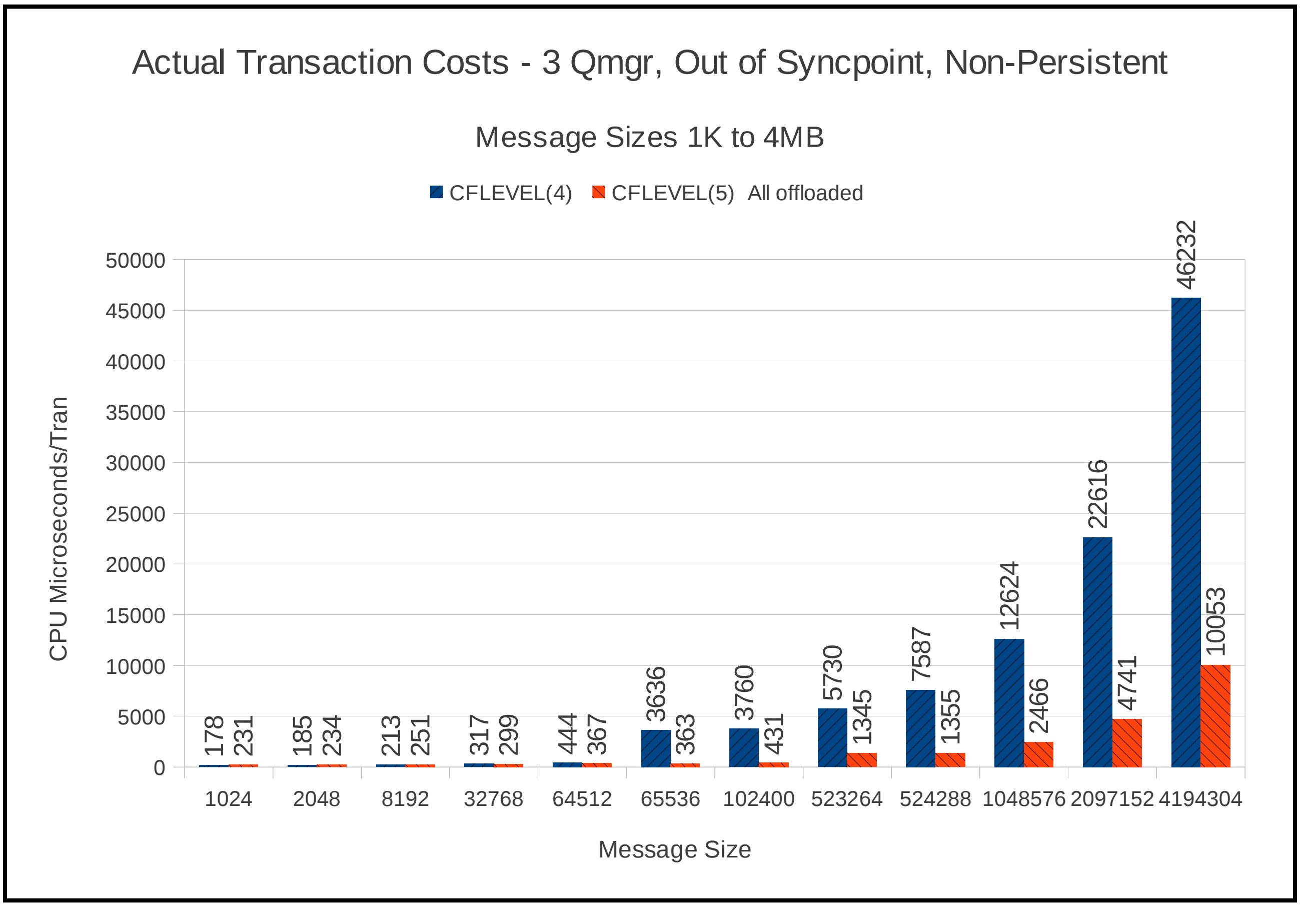


Chart 44: Observed transaction cost



Three queue managers – data sharing, non-persistent, in-syncpoint

These tests run a request/reply model for a range of message sizes.

The requesters put a non-persistent request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. The put is out of syncpoint as is the get.

The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are in-syncpoint.

The messages put by requester applications connected to queue manager 1 will be got by server applications connected to queue manager 2. Messages put by requester applications connected to queue manager 2 will be got by server applications connected to queue manager 3 and messages put by requester applications connected to queue manager 3 will be got by server applications connected to queue manager 1.

Chart 45: Achieved transaction rate – small messages

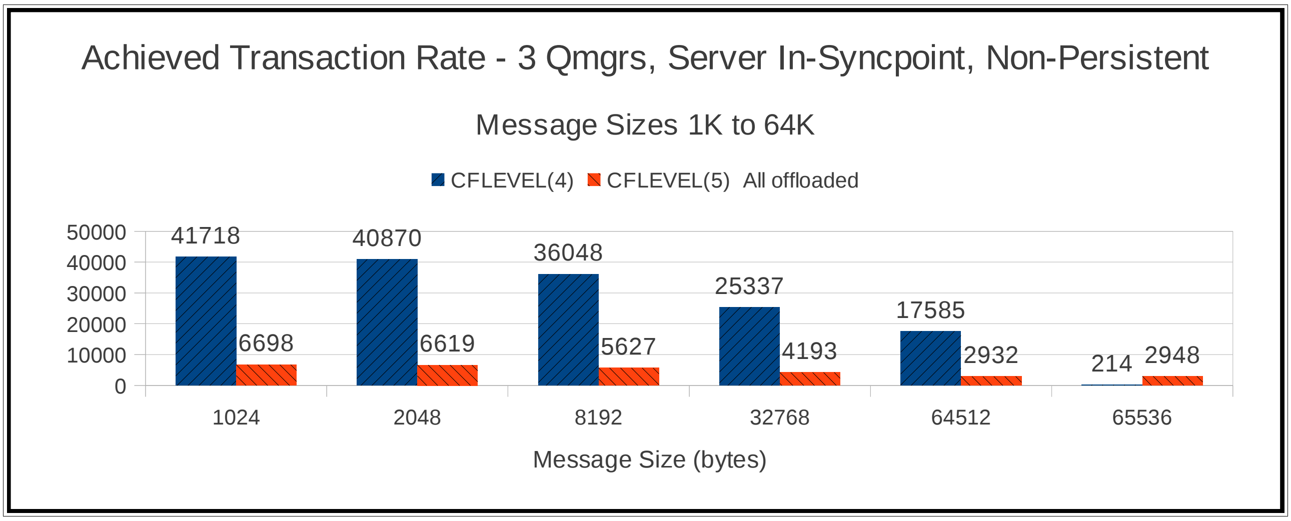


Chart 46: Achieved transaction rate – larger messages

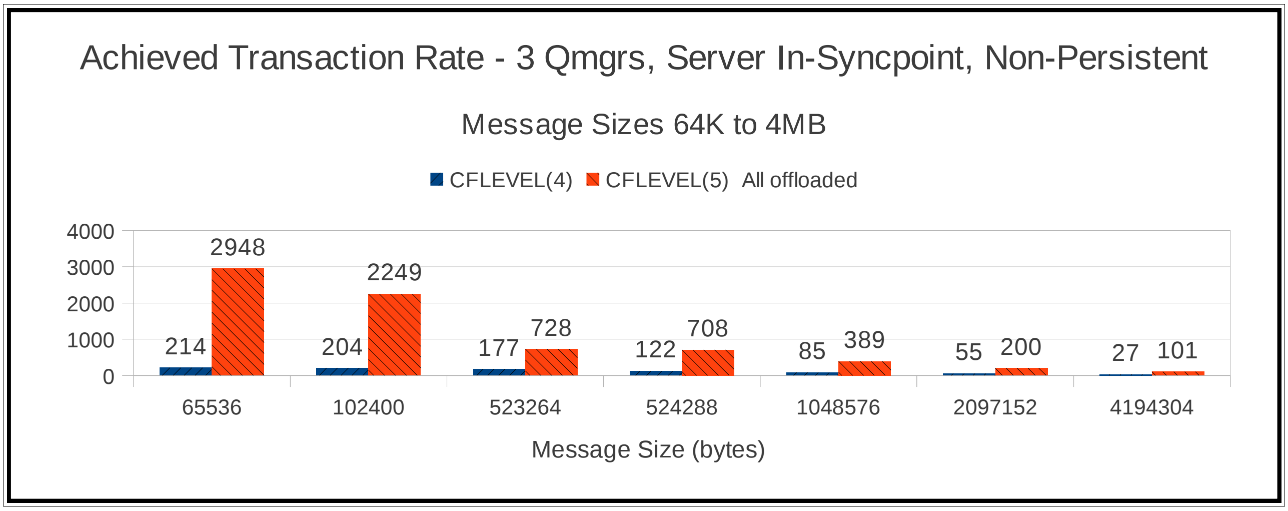
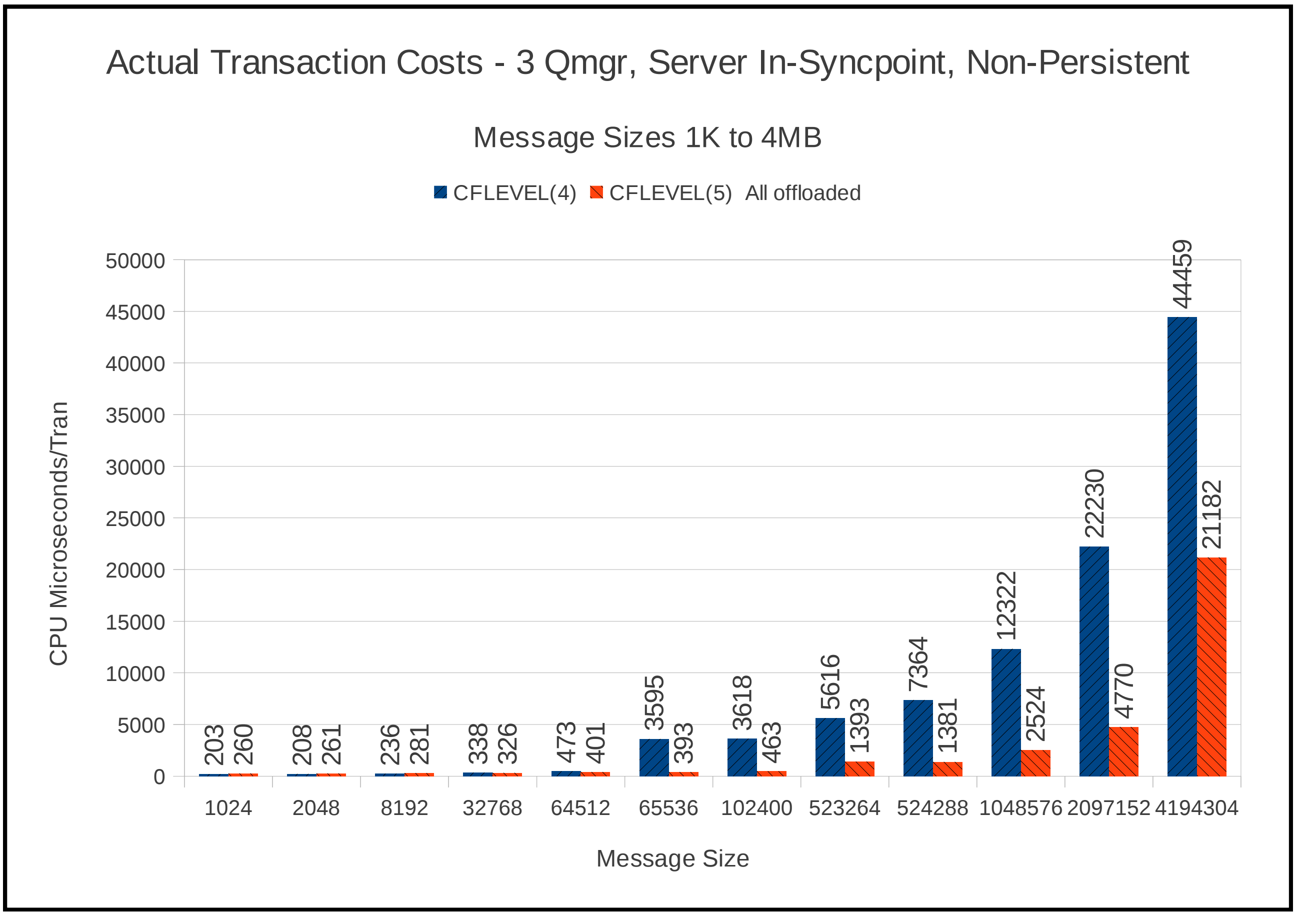


Chart 47: Observed transaction cost



Three queue managers – data sharing, persistent, in-syncpoint

These tests run a request/reply model for a range of message sizes.

The requesters put a persistent request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. The put is in-syncpoint as is the get.

The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are in-syncpoint.

The messages put by requester applications connected to queue manager 1 will be got by server applications connected to queue manager 2. Messages put by requester applications connected to queue manager 2 will be got by server applications connected to queue manager 3 and messages put by requester applications connected to queue manager 3 will be got by server applications connected to queue manager 1.

Chart 48: Achieved transaction rate – small messages

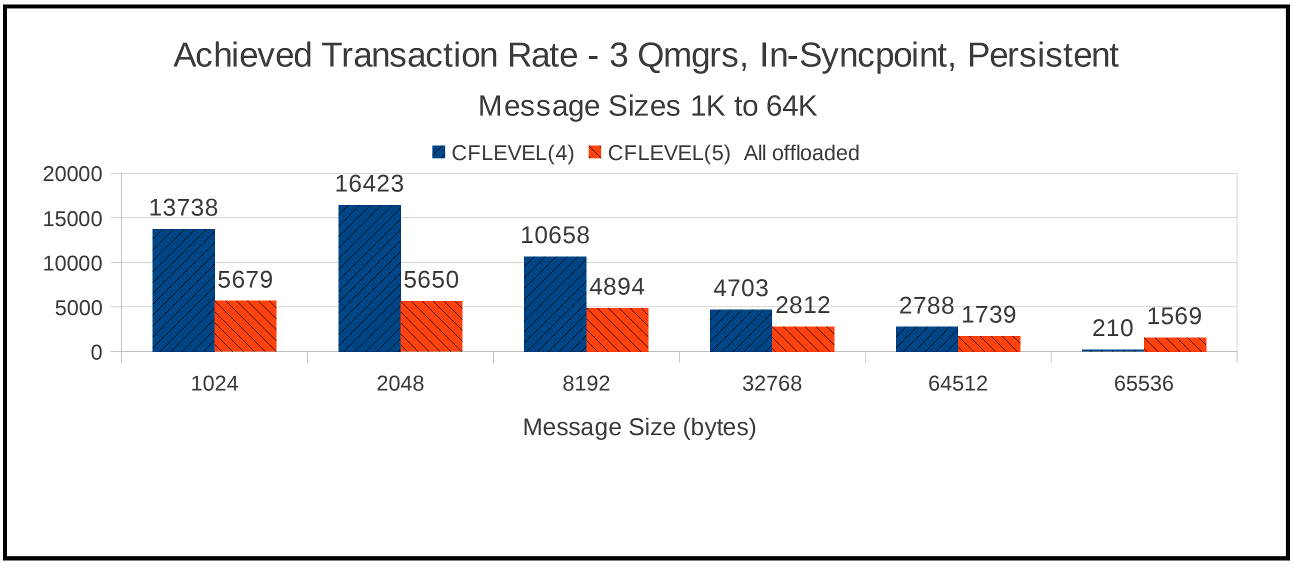


Chart 49: Achieved transaction rate – larger messages

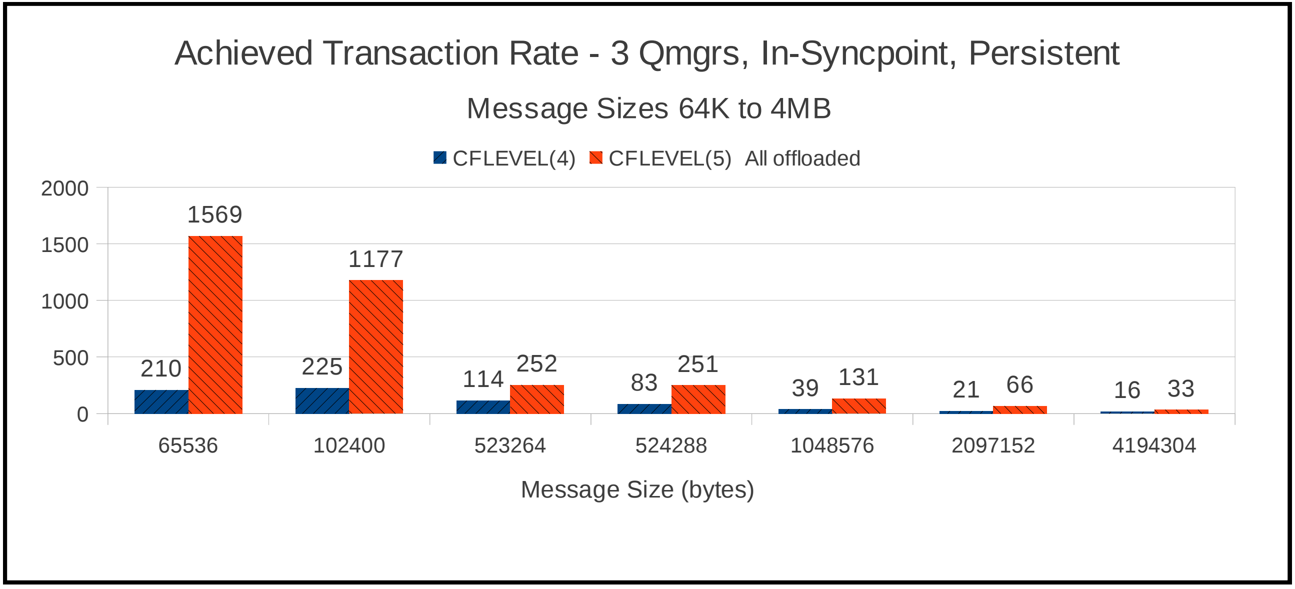
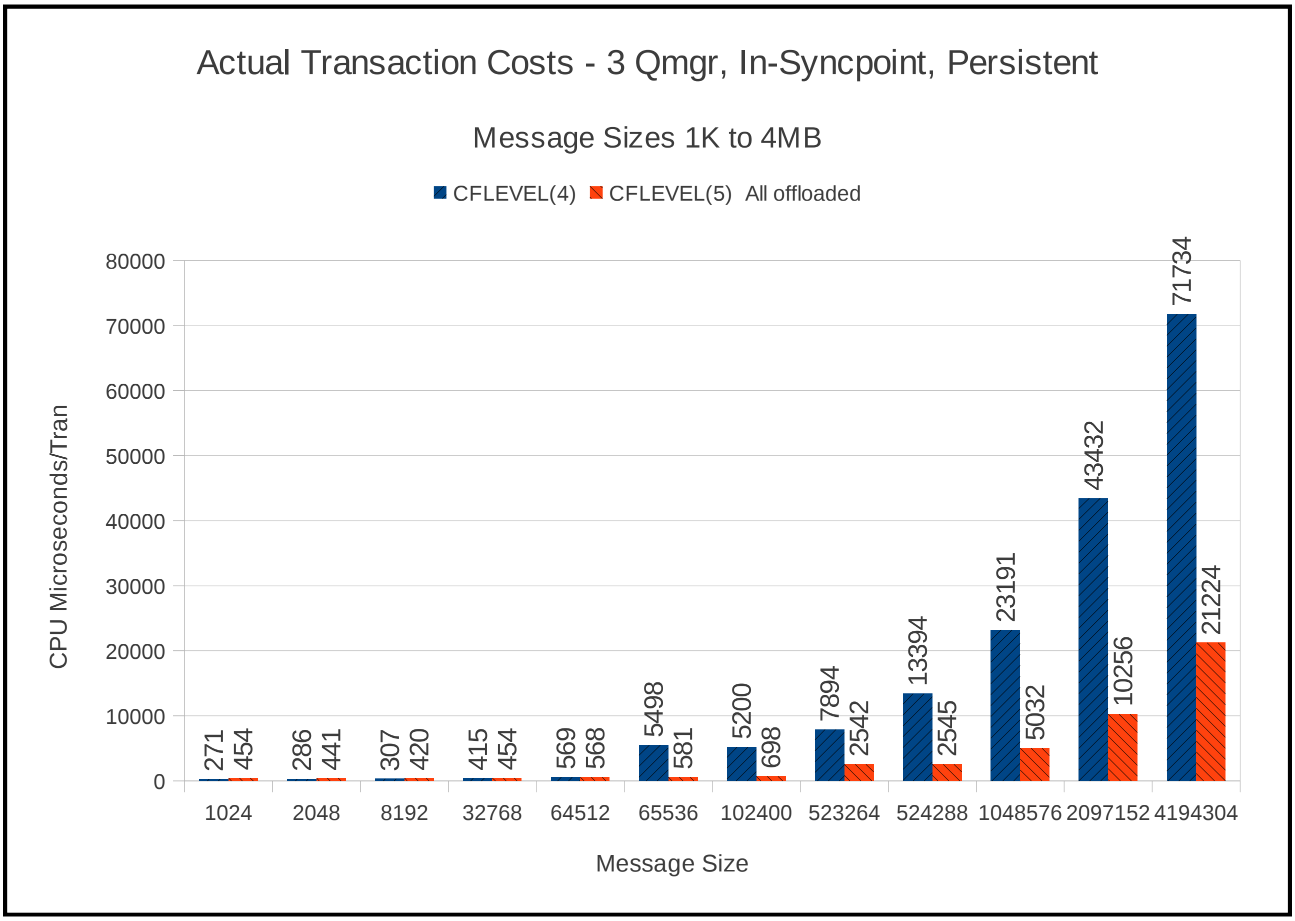


Chart 50: Observed transaction cost



When reviewing the Channel Path Activity report in RMF for the 4MB message size, the following was observed:

CF LEVEL(4):

```
  CHANNEL PATH      UTILIZATION(%)     READ(MB/SEC) WRITE(MB/SEC)        FICON OPERATIONS
ID TYPE    G SHR  PART   TOTAL    BUS   PART  TOTAL   PART  TOTAL       RATE  ACTIVE   DEFER
9C FC_S    9  Y  15.28  46.83  28.48  27.45  81.78  47.66 146.04    2751.1     3.6     0.0
9D FC_S    9  Y  15.22  46.50  28.21  27.28  81.86  47.42 143.84    2737.0     3.5     0.0
9E FC_S    9  Y  14.99  46.34  28.11  27.53  82.03  46.12 142.85    2734.0     3.5     0.0
9F FC_S    9  Y  15.27  46.47  28.30  27.26  80.42  48.11 146.01    2725.1     3.6     0.0
```

CF LEVEL(5):

```
  CHANNEL PATH      UTILIZATION(%)     READ(MB/SEC) WRITE(MB/SEC)        FICON OPERATIONS
ID TYPE    G SHR  PART   TOTAL    BUS   PART  TOTAL   PART  TOTAL       RATE  ACTIVE   DEFER
9C FC_S    9  Y  15.11  46.20  35.81  23.20  69.69  71.67 216.81    1073.8     8.3    10.2
9D FC_S    9  Y  15.17  46.25  35.87  22.51  69.21  72.74 217.77    1068.0     8.3    11.4
9E FC_S    9  Y  14.97  45.87  35.55  22.90  69.64  71.13 214.72    1073.2     8.0    10.6
9F FC_S    9  Y  15.15  46.04  35.69  23.40  69.82  71.88 215.74    1074.0     8.1    11.3
```

This shows that despite the CFLEVEL(5) measurement achieving twice the transaction rate, the channel paths are similarly loaded.

NOTE: The amount of data written per FICON instruction is significantly higher under CFLEVEL(5) when looking at “FICON OPERATIONS RATE x WRITE for this PARTition” e.g:
CFLEVEL 4 channel path “9C” has 47.66MB / 2751.1 = 17.74KB / FICON instruction
CFLEVEL 5 channel path “9C” has 71.67 MB / 1073.8 = 68.35KB / FICON instruction.

Six queue manager – data sharing, non-persistent, server in-syncpoint

These tests run a request/reply model for a range of message sizes.

The requester applications run on 2 queue managers, on LPAR1, and put a non-persistent request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. The put is in-syncpoint as is the get.

The server applications run on the remaining 4 queue managers which are running on 2 other LPARs. The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are in-syncpoint.

Chart 51: Achieved transaction rate

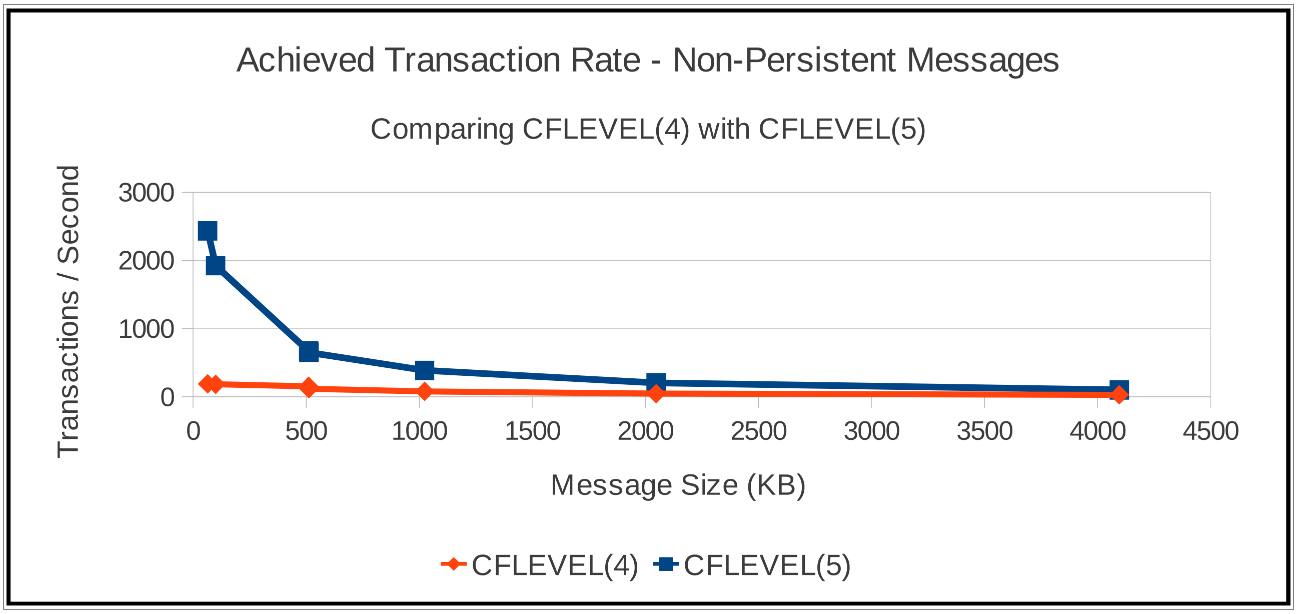
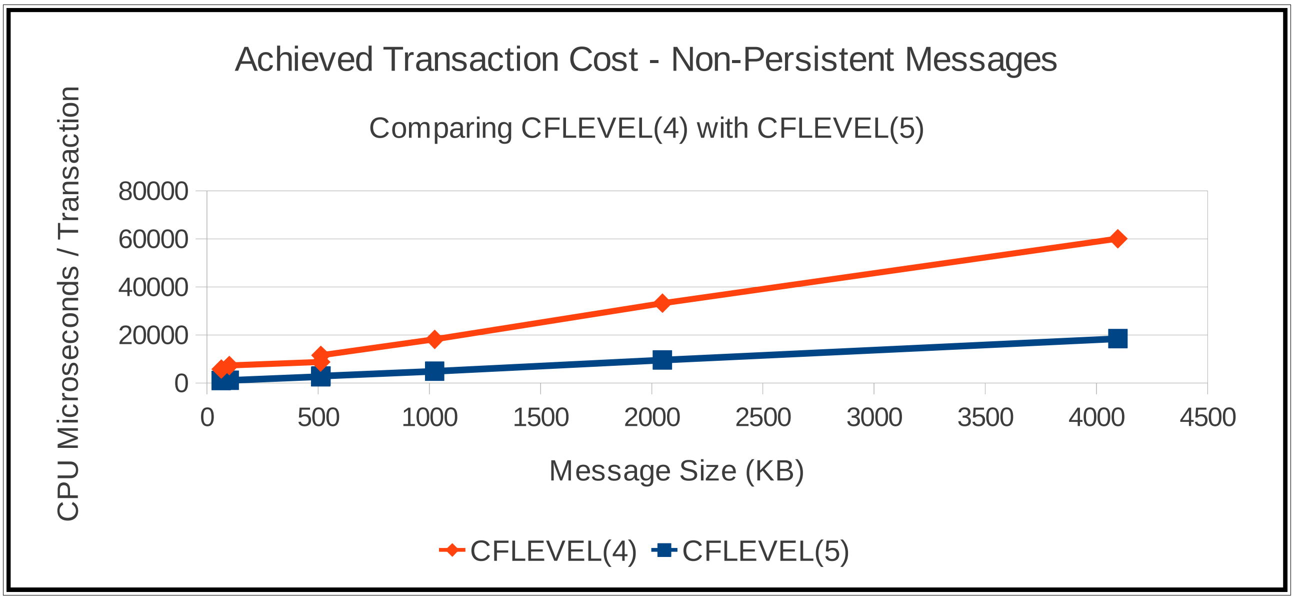


Chart 52: Observed transaction cost



Six queue manager – data sharing, persistent, server in-syncpoint

These tests run a request/reply model for a range of message sizes.

The requester applications run on 2 queue managers, on LPAR1, and put a persistent request message to a shared queue and waits for a reply message on a separate shared queue using CORRELATION-ID. The put is in-syncpoint as is the get.

The server applications run on the remaining 4 queue managers which are running on 2 other LPARs. The servers get the next available message from the request queue and puts a corresponding reply message to the pre-agreed reply queue. These gets and puts are in-syncpoint.

Chart 53: Achieved transaction rate

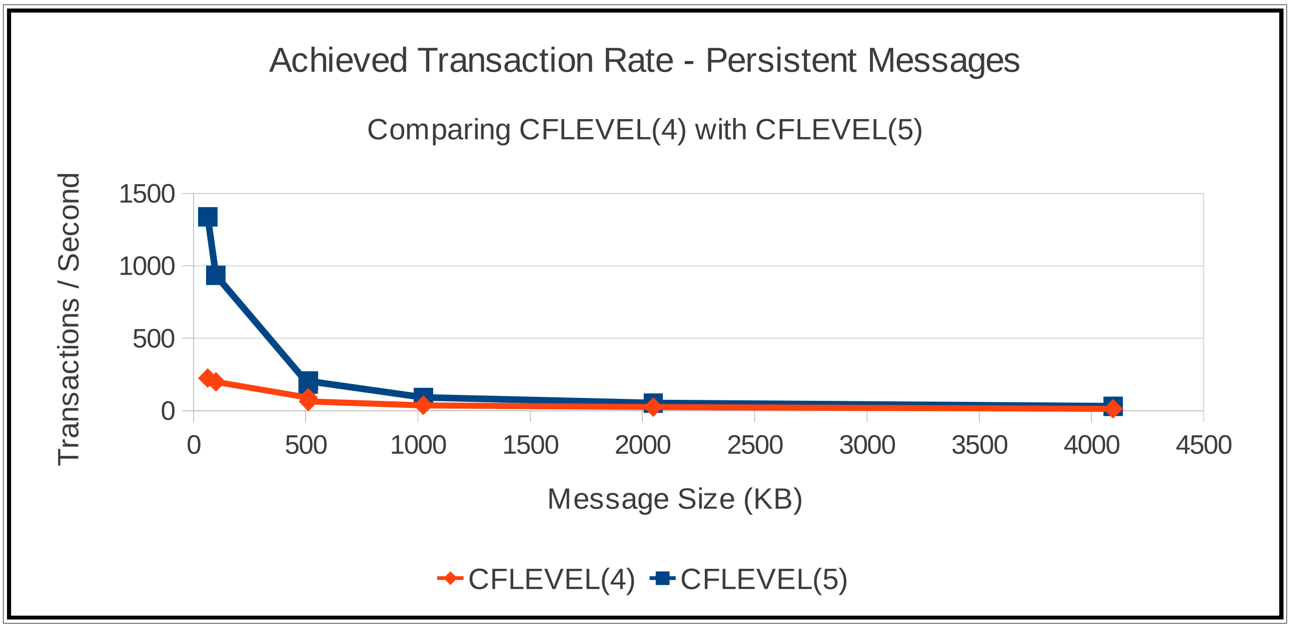
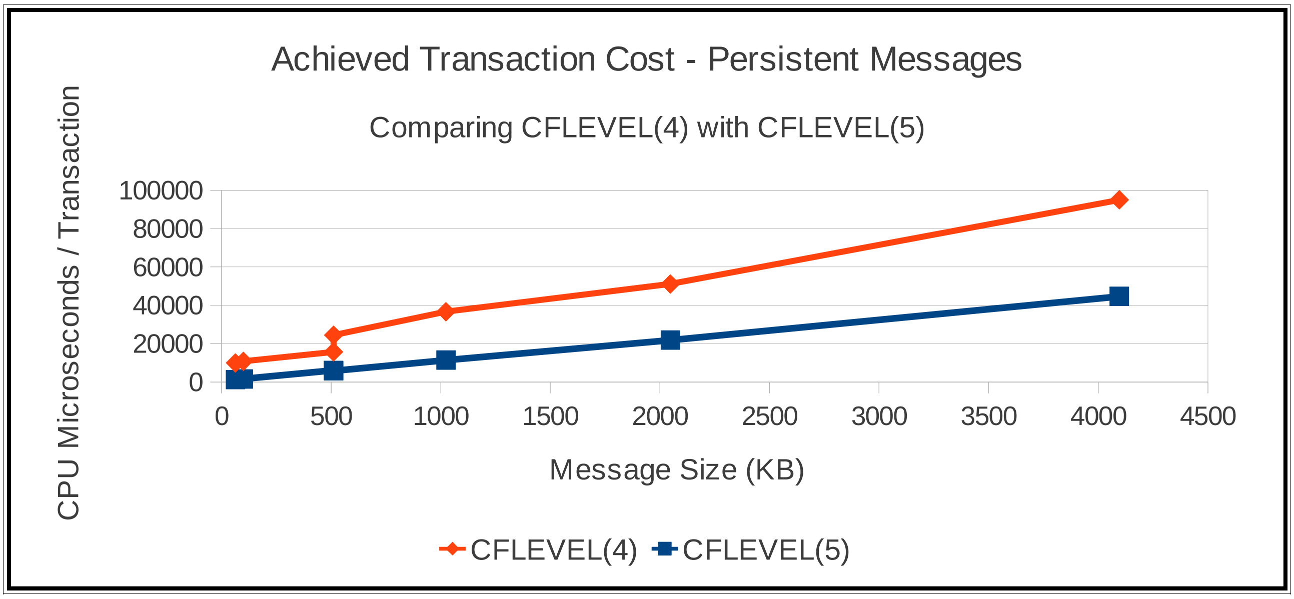


Chart 54: Observed transaction cost



Scaling – non-persistent large shared queue messages

For this type of scenario a single queue manager is created and connects to a queue sharing group with 20 structures.
Request/Reply workload is run on this queue manager, where the measurement begins using one pair of request and reply queues and increments this by one until there are 15 request and 15 reply queues in use.
Each pair of request and reply queues is used by an equal number of request/reply applications.
Typically the measurements begins with 1 requester application and 1 server application using a single request and reply queue pair.
The requester puts a message to the request queue and wait for a corresponding reply message on the reply queue. This is completed out of syncpoint.
The server application action an MQGET-with-wait for the message and then puts a reply message to the reply queue. This is completed within syncpoint.

With this configuration the measurements are run with the least possible contention for messages on the queues and is not necessarily typical of production workload.

Each structure is the same size.
Structures 1 to 15 are CFLEVEL(5) OFFLOAD(SMDS).
Structures 16 to 20 are CFLEVEL(4).
Queues are defined on each application structure such that a scaling workload can be run on 1 structure or multiple structures.

Non-persistent 10KB messages

The following charts compare transaction cost and throughput rates when using 10KB non-persistent messages.
The CFLEVEL(5) structures have been altered to ensure that all messages are offloaded to SMDS, e.g.

```
ALT CFSTRUCT(APPLICATION1) OFFLD1SZ(0K) OFFLD1TH(0) +
                           OFFLD2SZ(0K) OFFLD2TH(0) +
                           OFFLD3SZ(0K) OFFLD3TH(0)
```

Regarding the data displayed in following 2 charts, the following comments apply:
When reviewing the transaction rate chart, it is expected that the rate achieved on the CFLEVEL(4) structure exceeds the CFLEVEL(5) structure as the accesses are only to the Coupling Facility.
The CFLEVEL(4) measurement tails off when the coupling facility reaches a state where it requires additional processors to service the work in a timely manner.
For 10KB messages, the rate at which the I/O subsystem was being driven was sufficient such that benefits were seen by using multiple structures and by implication, multiple shared message data sets.

NOTE: With regard to the transaction cost – the increasing cost of incurred when running using queues defined on the CFLEVEL(4) can partly be attributed to the CF responses becoming asynchronous as it becomes CPU constrained.

Chart 55: Achieved transaction rate

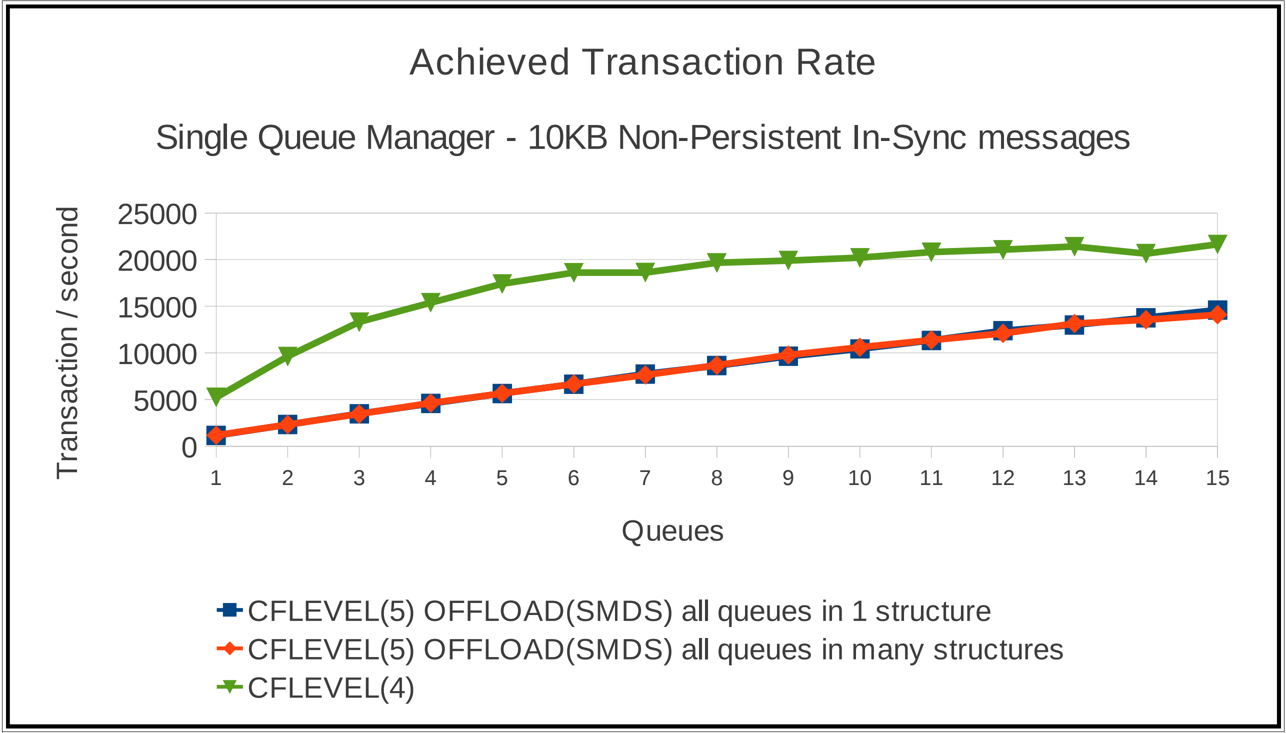
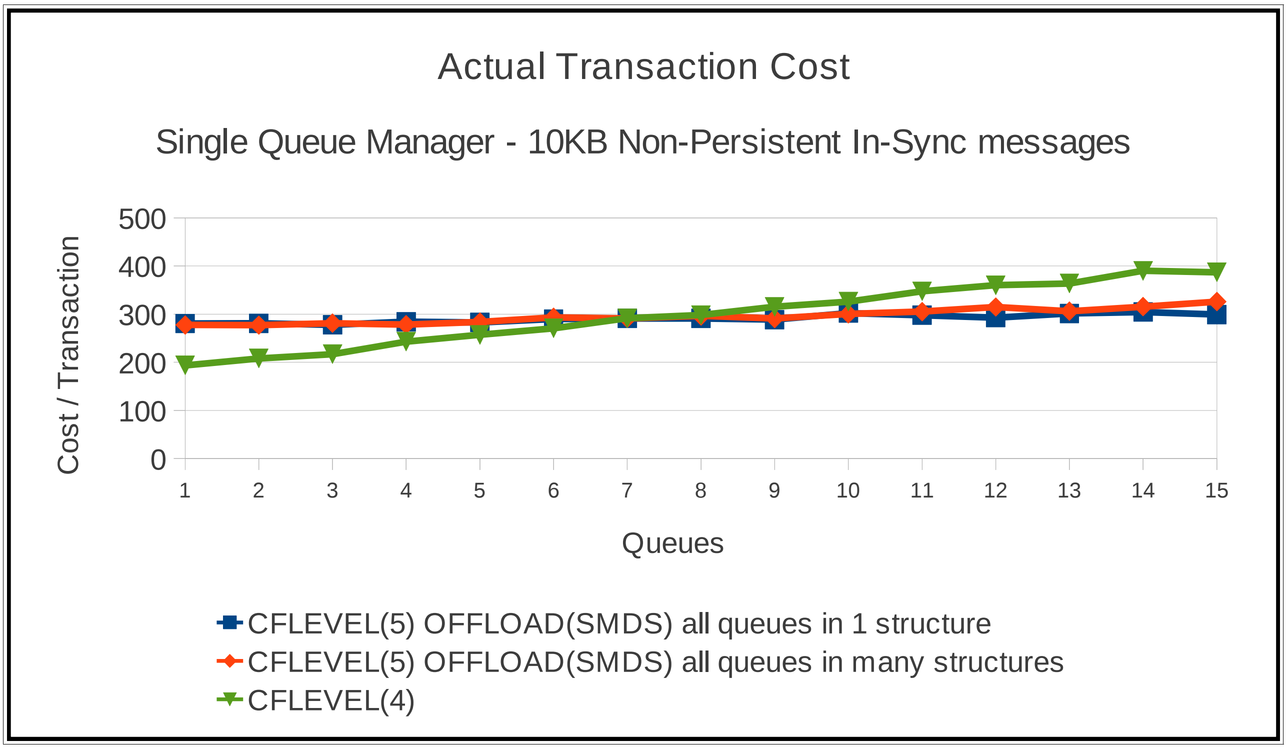


Chart 56: Observed transaction cost



Non-persistent 100KB messages

The following charts compare transaction cost and throughput rates when using 100KB non-persistent messages.

Chart 57: Achieved Transaction Rate

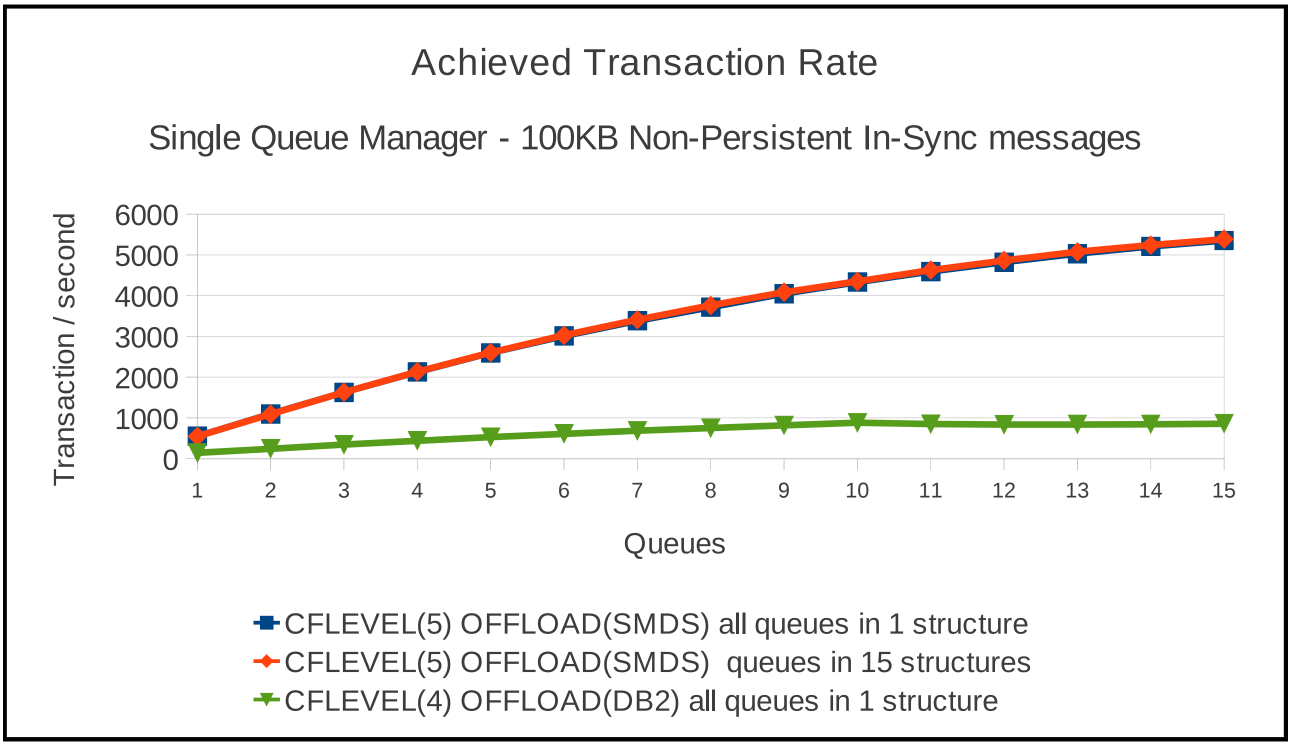
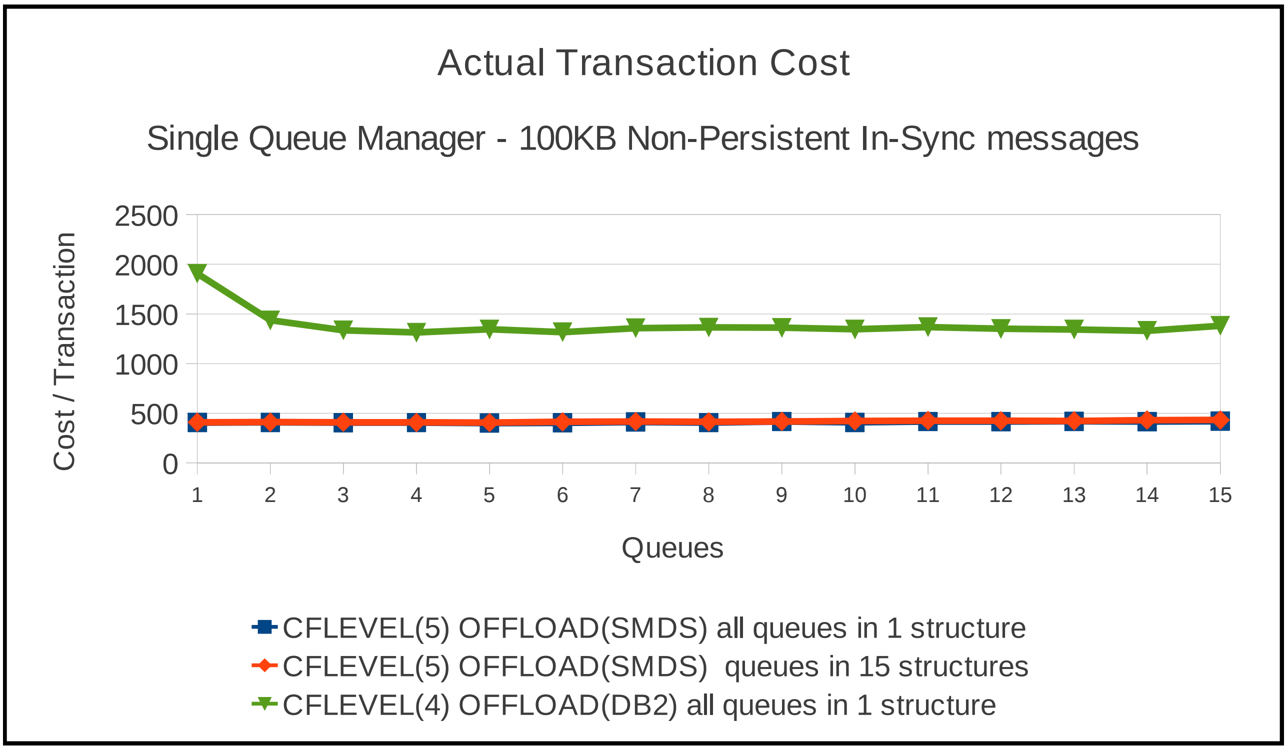


Chart 58: Observed Transaction Cost



Notes on previous 2 charts:

- The achieved transaction rate using a single CFLEVEL(4) structure peaks at approximately 900 per second. This means that there are approximately 180MB/second being written to the DB2 blob tablespaces. As there are 4 tablespaces defined to hold the large messages, this means that each of the 4 channel paths to the I/O subsystem being used, each is writing 45MB per second.
- By comparison, using a single CFLEVEL(5) structure, backed by a single SMDS, the peak transaction rate is 5,350 per second, at significantly less cost. In this measurement, the system is writing over 1,000MB/second to the single SMDS. In this instance, 4 channel paths are being used, each writing 250MB/second.
- When multiple CFLEVEL(5) structures are used to host the queues, there is less contention waiting for the I/O write to complete for a particular volume – the queue manager is able to request I/O is completed on multiple data sets concurrently. The transaction rate can reach 5,400 per second.
- Whether the workload is run on a single CFLEVEL(5) structure or multiple CFLEVEL(5) structures, the cost per transaction is similar and is significantly less than the equivalent CFLEVEL(4) transaction cost.

Non-persistent 1MB messages

The following charts compare transaction cost and throughput rates when using 1MB non-persistent messages.

Chart 59: Achieved transaction rate

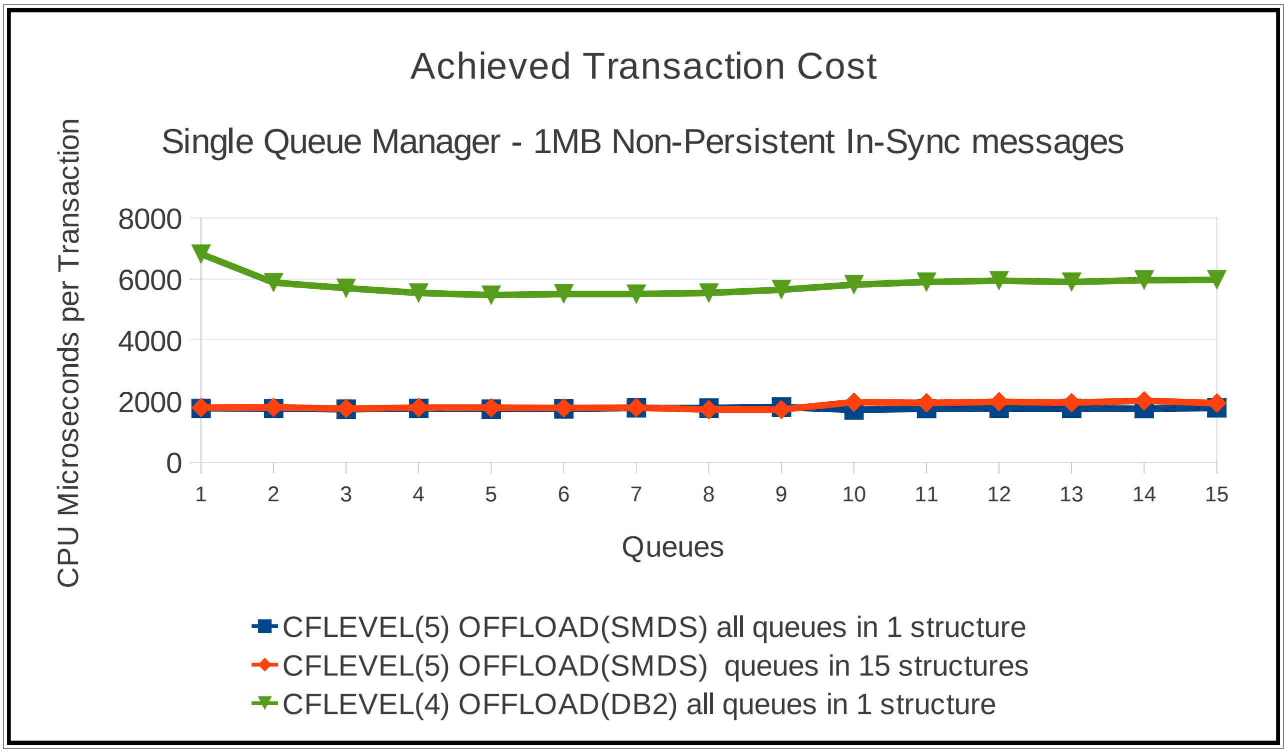
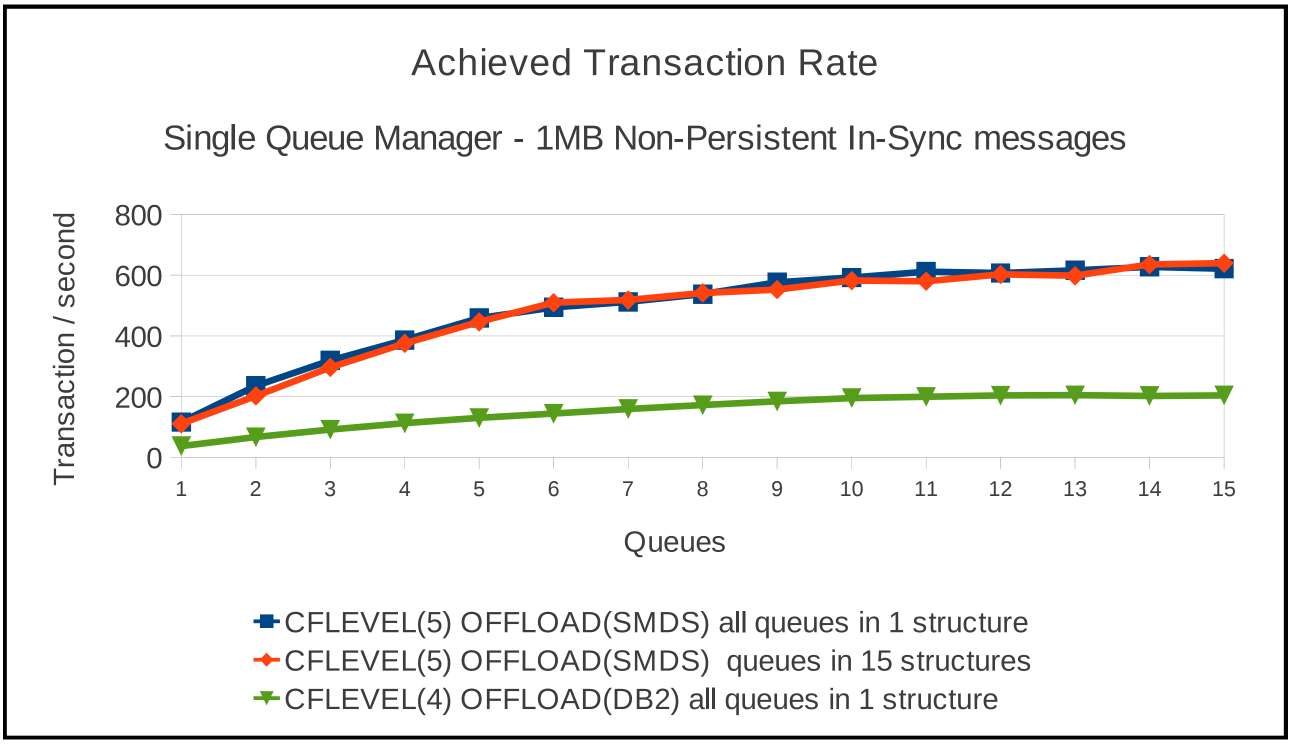


Chart 60: Observed transaction cost



Notes on previous 2 charts:
The achieved transaction rate using a single CFLEVEL(4) structure peaks at approximately 200 per second. This means that there are approximately 400MB/second being written to the DB2 blob tablespaces. Using RMF data, we can determine whether the I/O subsystem is the constraining factor.

Using the RMF “Direct Access Device Activity” reports we can see high activity on DB2's log datasets as well as the 4 volumes used for each of the 4 tablespaces used to hold large WebSphere MQ shared messages.

Extract from “Direct Access Device Activity” report

```
                                                DEVICE
STORAGE  DEV  DEVICE   NUMBER  VOLUME PAV  LCU  ACTIVITY
 GROUP   NUM  TYPE     OF CYL  SERIAL           RATE
DSNLOG1  C009 33909     32760  AADS10 1.0H 0002 1406.44  - DB2 Log Dataset
DSNLOG2  C00A 33909     32760  AADS20 1.0H 0002 1393.75  - DB2 Log Dataset
SGPRIME  C007 33909     65520  AAP001 1.3H 0002 987.105  - MQ LOB Tablespace
SGPRIME  C031 33909     65520  AAP002 1.8H 0003 915.086  - MQ LOB Tablespace
SGPRIME  C032 33909     65520  AAP003 1.3H 0003 925.770  - MQ LOB Tablespace
SGPRIME  C033 33909     65520  AAP004 1.1H 0003 904.537  - MQ LOB Tablespace
```

Using the reported LCUs with the “I/O Queuing Activity” report, we can see that LCUs 0002 and 0003 both use channel paths 9C, 9D, 9E and 9F.
Referring to the “Channel Path Activity” report:

```
-----------------------------------------------------------------------------------------
                                            DETAILS FOR ALL CHANNELS
-----------------------------------------------------------------------------------------
   CHANNEL PATH     UTILIZATION(%)    READ(MB/SEC) WRITE(MB/SEC)        FICON OPERATIONS
 ID TYPE   G SHR  PART  TOTAL    BUS   PART  TOTAL   PART  TOTAL     RATE  ACTIVE   DEFER
 9C FC_S   9  Y  26.07  26.26  14.80   1.97   2.02 116.41 116.41   1654.4     1.5     0.0
 9D FC_S   9  Y  26.12  26.32  14.82   2.18   2.21 116.35 116.36   1654.5     1.5     0.0
 9E FC_S   9  Y  26.10  26.31  14.82   1.94   1.98 116.56 116.57   1655.6     1.5     0.0
 9F FC_S   9  Y  26.17  26.36  14.84   2.01   2.04 116.66 116.67   1655.3     1.5     0.0
```

This report shows us that the particular channel paths used for I/O to the DB2 volumes is not significantly busy (less than 30% utilised).

By comparison, using a single CFLEVEL(5) structure, backed by a single shared message data set, the peak transaction rate is 620 per second, at significantly less cost. In this measurement, the system is writing over 1300MB/second to the single SMDS. Again in this instance, only 4 channel paths are being used, each writing 330MB/second and are each 59% utilised:

```
------------------------------------------------------------------------------------------
                                                   DETAILS FOR ALL CHANNELS
------------------------------------------------------------------------------------------
  CHANNEL PATH     UTILIZATION(%)    READ(MB/SEC) WRITE(MB/SEC)        FICON OPERATIONS
ID TYPE   G SHR  PART  TOTAL    BUS   PART  TOTAL   PART  TOTAL      RATE  ACTIVE   DEFER
9C FC_S   9  Y  59.39  59.86  42.92  10.55  10.59 332.78 332.78    1651.0     8.5     0.0
9D FC_S   9  Y  59.30  59.77  42.76  10.83  10.86 331.22 331.23    1650.1     8.8     0.0
9E FC_S   9  Y  59.33  59.78  42.86  10.34  10.37 332.47 332.48    1650.7     8.4     0.0
9F FC_S   9  Y  59.39  59.84  42.89  10.34  10.37 332.73 332.73    1651.0     8.5     0.0
```

When multiple CFLEVEL(5) structures are used to host the queues, there is less contention waiting for the I/O write to complete for a particular volume ; the queue manager is able to request I/O is completed on multiple data sets concurrently. The transaction rate can exceed 630 per second. Because the datasets are spread across many volumes, a distribution of workload takes place, so that more channel paths can be used, however on our systems we only have 4 channel paths available. When testing with more channel paths we have seen significant improvements in throughput as this increases the capacity of the I/O subsystem and allows the higher transaction rate

```
------------------------------------------------------------------------------------------
                                                    DETAILS FOR ALL CHANNELS
------------------------------------------------------------------------------------------
  CHANNEL PATH     UTILIZATION(%)    READ(MB/SEC) WRITE(MB/SEC)        FICON OPERATIONS
ID TYPE   G SHR  PART  TOTAL    BUS   PART  TOTAL   PART  TOTAL      RATE  ACTIVE   DEFER
9C FC_S   9  Y  58.58  59.04  42.45   0.13   0.16 339.43 339.44    1611.6     8.8     0.0
9D FC_S   9  Y  58.64  59.10  42.50   0.19   0.22 339.76 339.76    1611.5     9.1     0.0
9E FC_S   9  Y  58.65  59.11  42.50   0.18   0.21 339.82 339.83    1611.6     8.8     0.0
9F FC_S   9  Y  58.65  59.12  42.47   0.13   0.16 339.58 339.59    1611.6     9.1     0.1
```

NOTE: In the multiple structure measurement there is less channel path activity for reading. This is due to there being sufficient buffers available for the queue manager to avoid needing to read the message from the dataset.

Whether the workload is run on a single CFLEVEL(5) structure or multiple CFLEVEL(5) structures, the cost per transaction is similar and is significantly less than the equivalent CFLEVEL(4) transaction cost.

Non-persistent 1MB messages – multiple applications per queue pair

By increasing the number of applications using each pair of queues we can determine the maximum rate of writing to our disks.
We would not expect the CFLEVEL(4) measurements to significantly change and as can be seen from the chart following, the peak transaction rate is still at 200 transactions per second.

Chart 61: Achieved Transaction Rate using CFLEVEL(4) with multiple applications

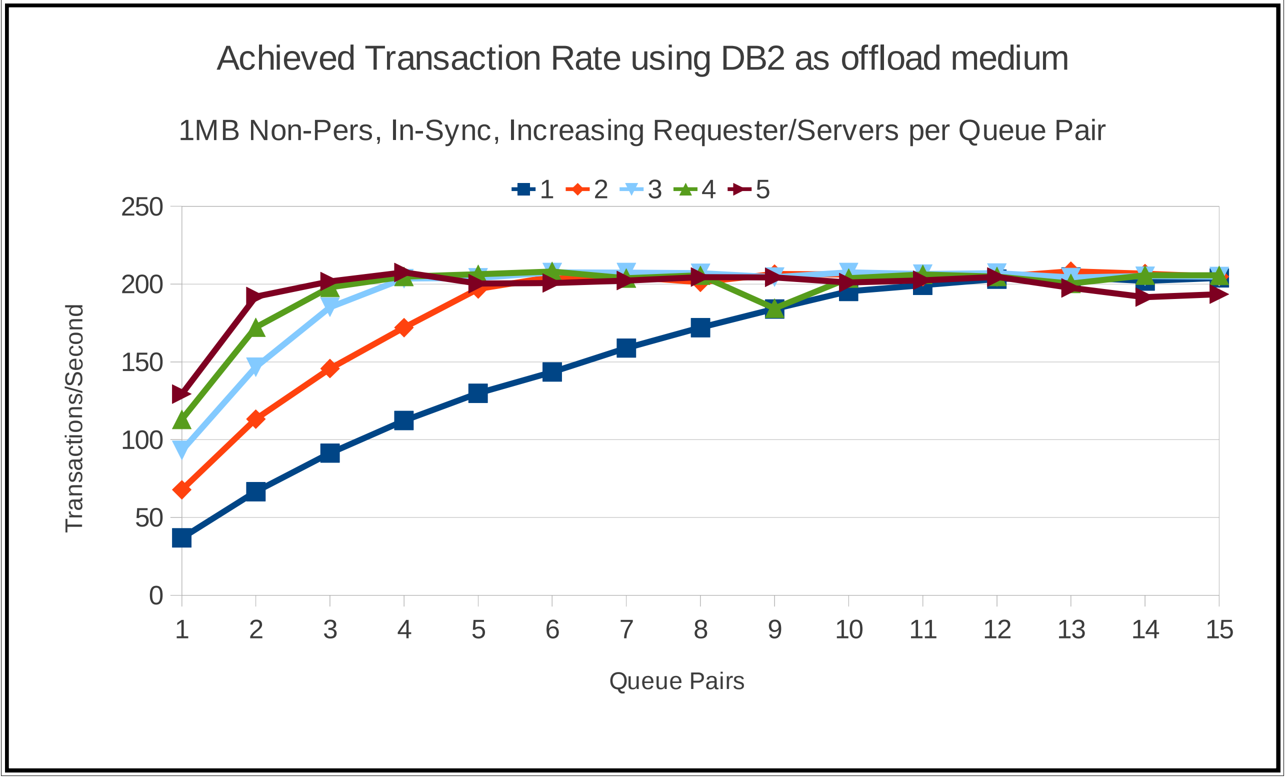
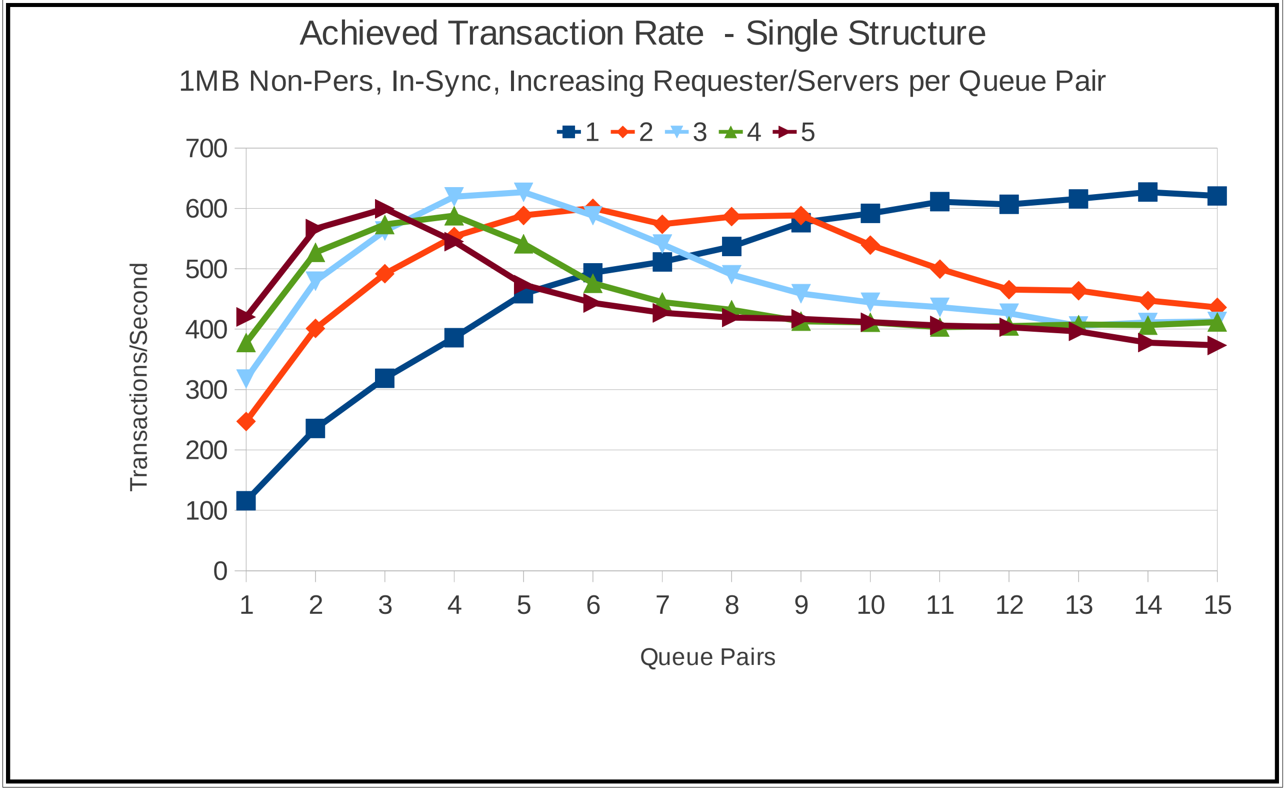


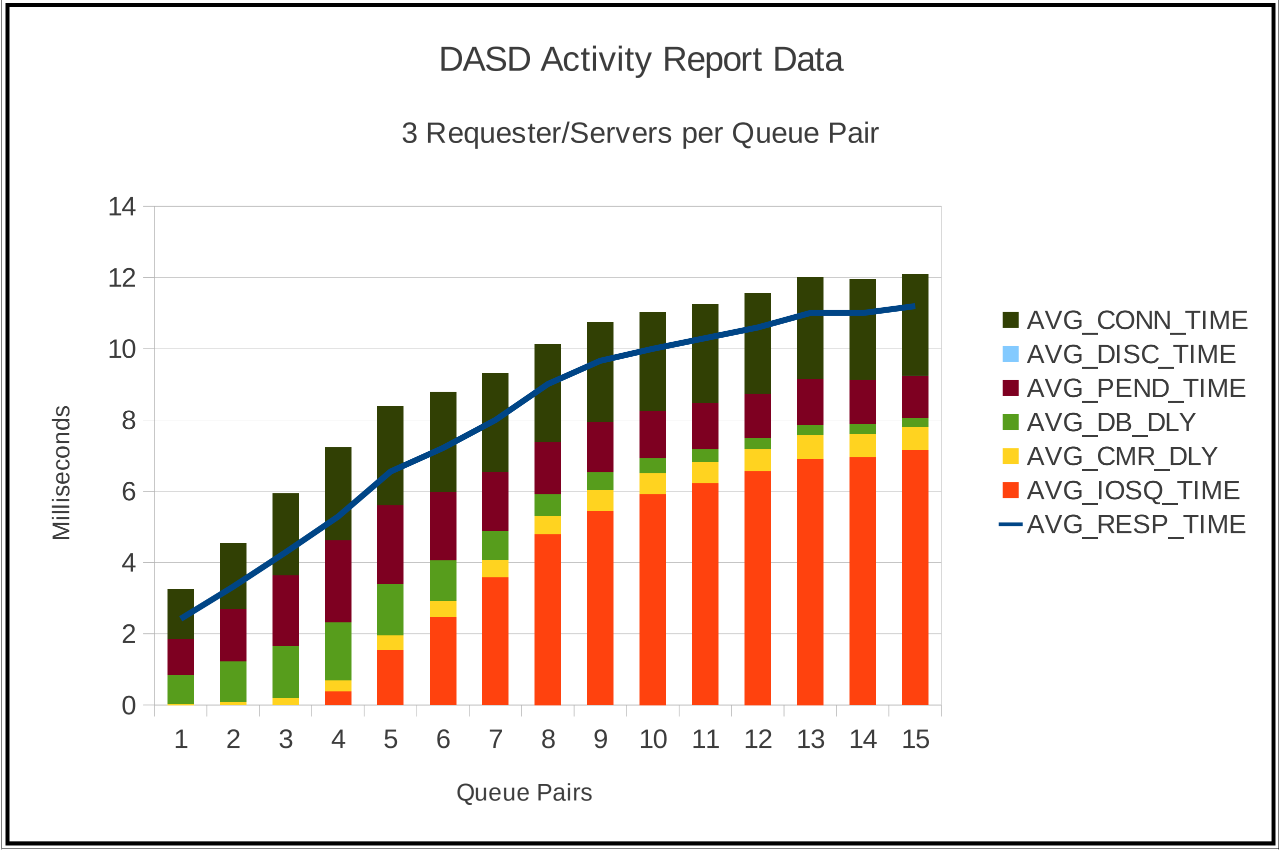
Chart 62: Achieved transaction rate using CFLEVEL(5) offload(SMDS) with 1 structure



When using a single CFLEVEL(5) structure, we would not expect the transaction rate to increase as we are reaching the limits as to how fast we can access a single volume. The chart shows that additional tasks using the queues does not increase the throughput – signifying that the I/O rate on the single volume has peaked.

In this measurement, additional tasks has caused a reduction in achieved throughput to around 400 transactions per second.

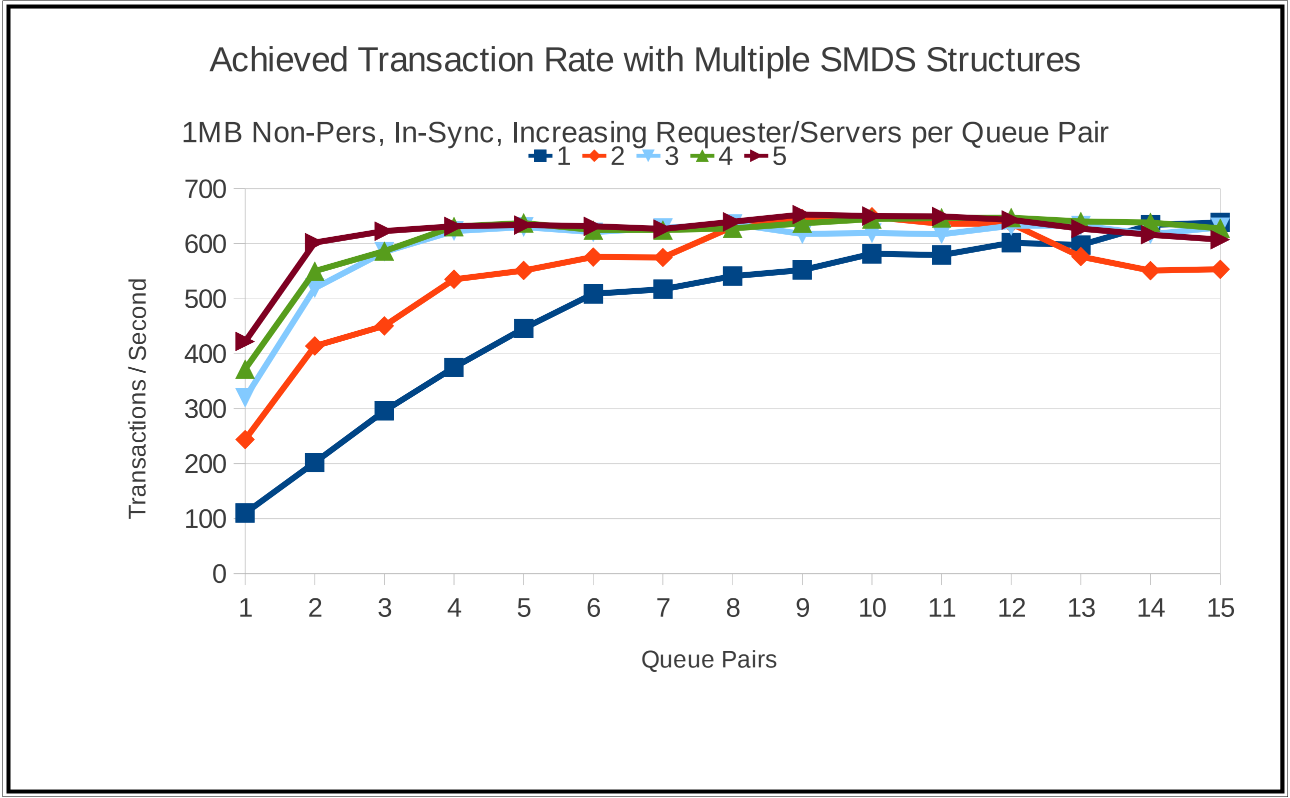
Chart 62-A: DASD activity report for the 3 requester/server per queue pair measurement.



This chart shows the average IOSQ time has significantly increased from the measurement with 5 queue pairs, which corresponds to the peak transaction rate achieved.
The Average IOSQ time is defined as “The average number of milliseconds an I/O request must wait on an IOS queue before a SSCH instruction can be issued.” in the RMF manuals.

The CFLEVEL(5) multiple structures measurement should increase throughput until becoming constrained by channel path usage.
The following chart shows a peak rate of 650 transactions per second which is not significantly higher than the peak throughput achieved for the single structure measurements.

Chart 63: Achieved transaction Rate using CFLEVEL(5) offload(SMDS) with multiple structures



In this measurement, the multiple SMDS datasets are spread across 3 volumes – AAPG00, AAPG10 and AAPG30.
As the number of queue pairs in use increases, the average IOSQ time for each of these volumes rose to over 20 milliseconds. By comparison, the single structure measurement peaked at an average IOSQ time of 7 milliseconds.

Appendix A – Regression

When performance monitoring WebSphere MQ for z/OS version 7.1.0, a number of different categories of tests are run, which include:

- Private Queue
- Shared Queue
- Moving messages using MCA Channels
- Moving messages using Clustered Channels
- Client
- Bridges and Adaptors
- Trace

These tests are run against V6.0.0, V7.0.1 and V7.1.0 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 a zEnterprise 196 (2194-779) which was configured as described in Appendix B.

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

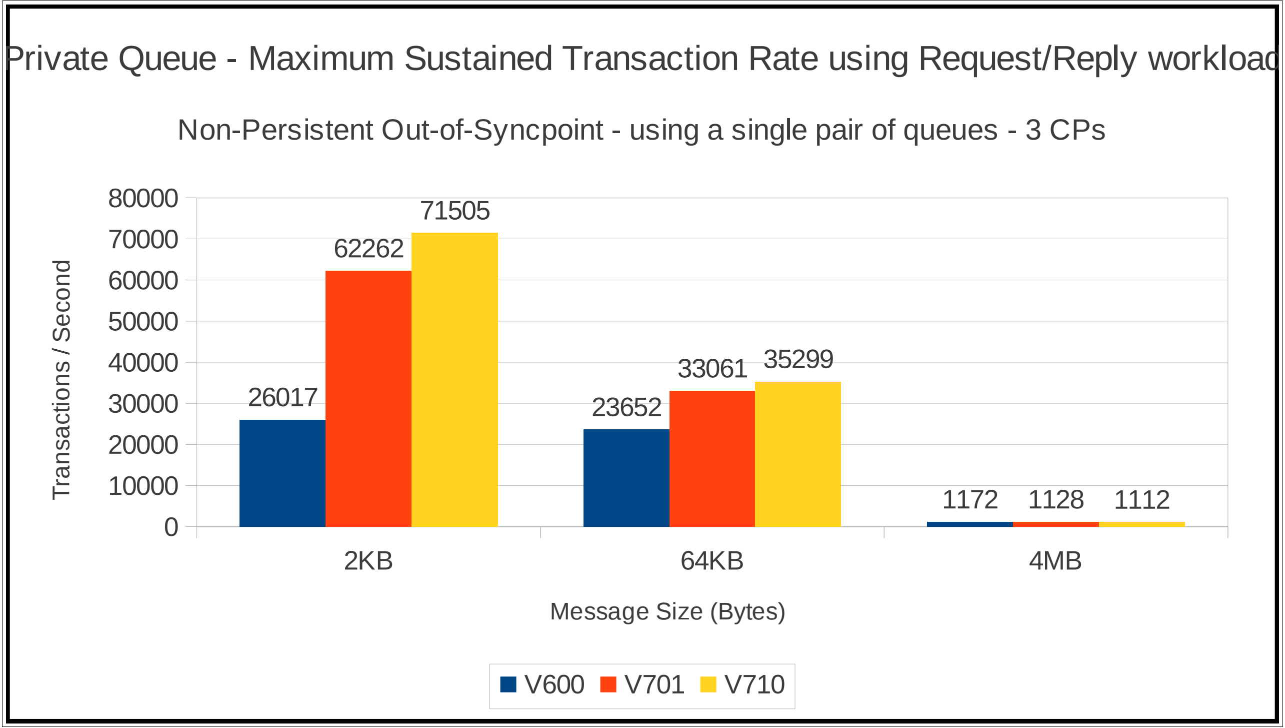
Regression – Private Queue

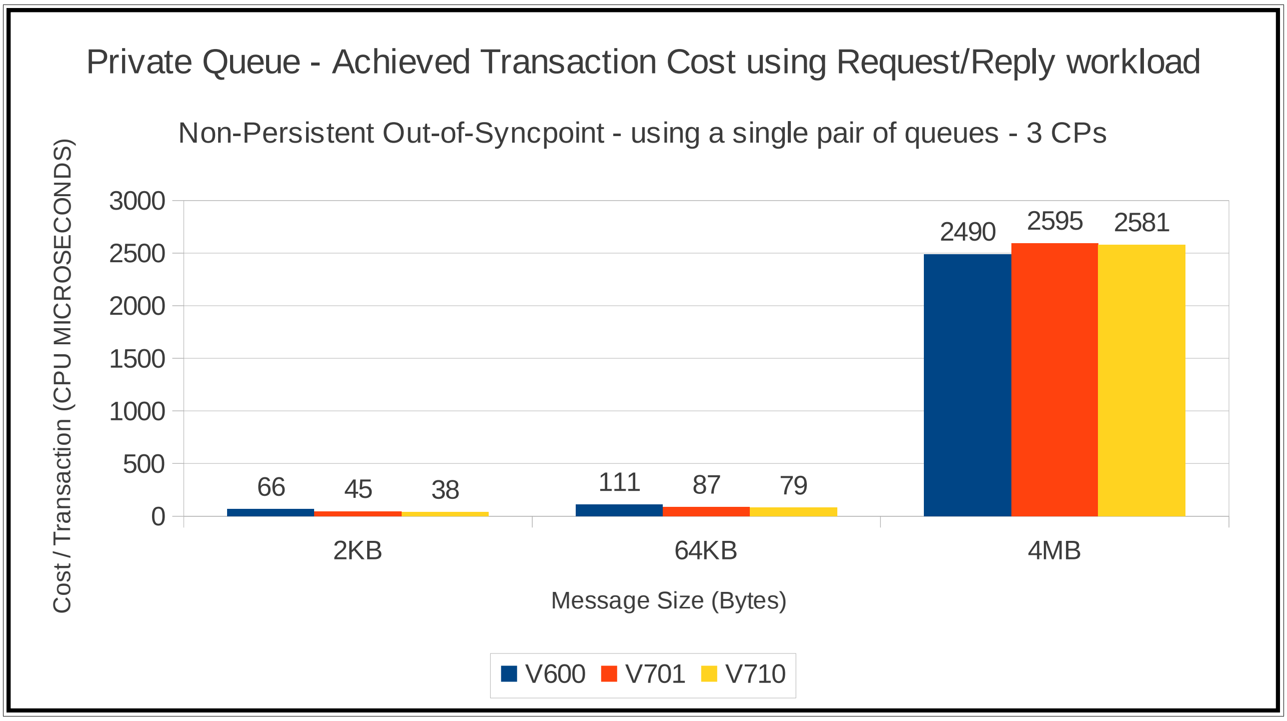
Non-Persistent Out-of-Syncpoint workload

Maximum Throughput on 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 an MQGET-with-wait. The messages are got and put out of syncpoint.





Scalability of request/reply model across multiple queues

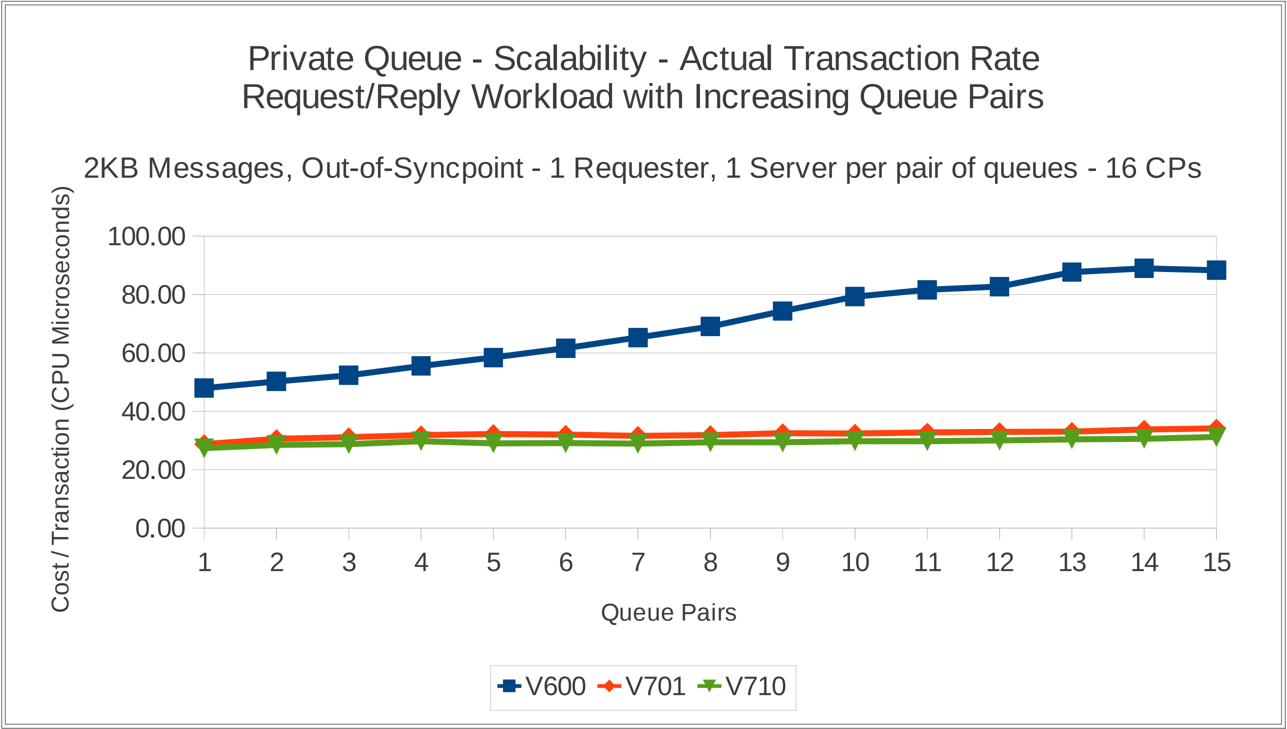
The queue manager is configured with 16 pagesets – 0 through 15 and a corresponding number of buffer pools.

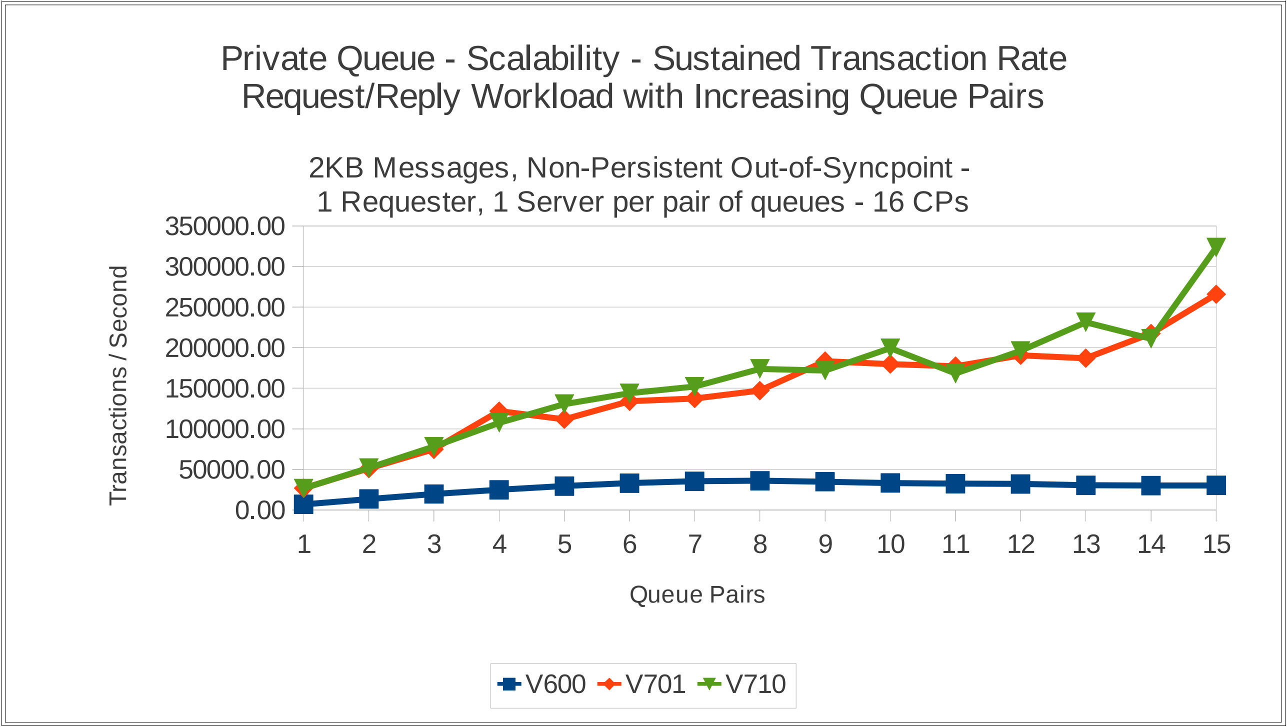
On each of pagesets 1 to 15, a pair of request and reply queues are defined. The test starts up 1 requester and 1 server task accessing the queues on pageset 1 and runs a request/reply workload. At the end, the test starts a second pair of requester and server tasks which access the queues on page set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

The requester and the server tasks specify NO_SYNCPOINT for all messages.

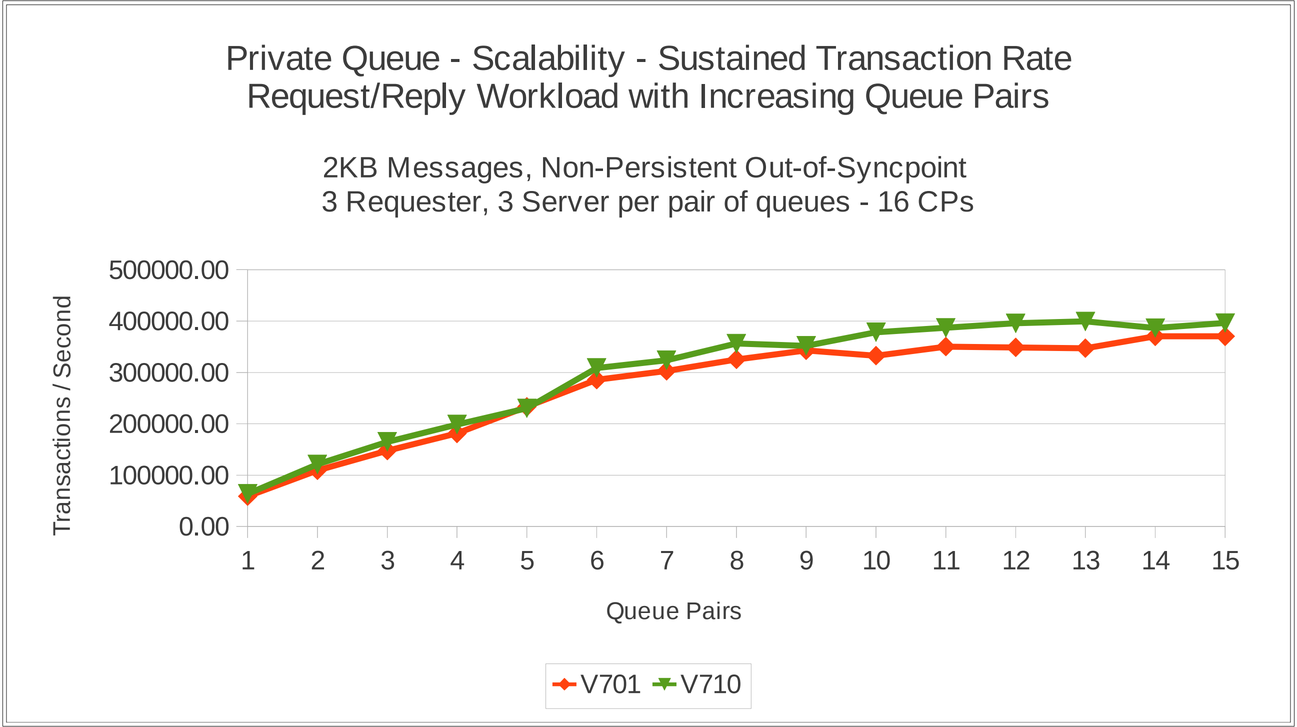
The measurements are run on a single LPAR with 16 dedicated processors online on the zEnterprise 196 (2817) used for testing.

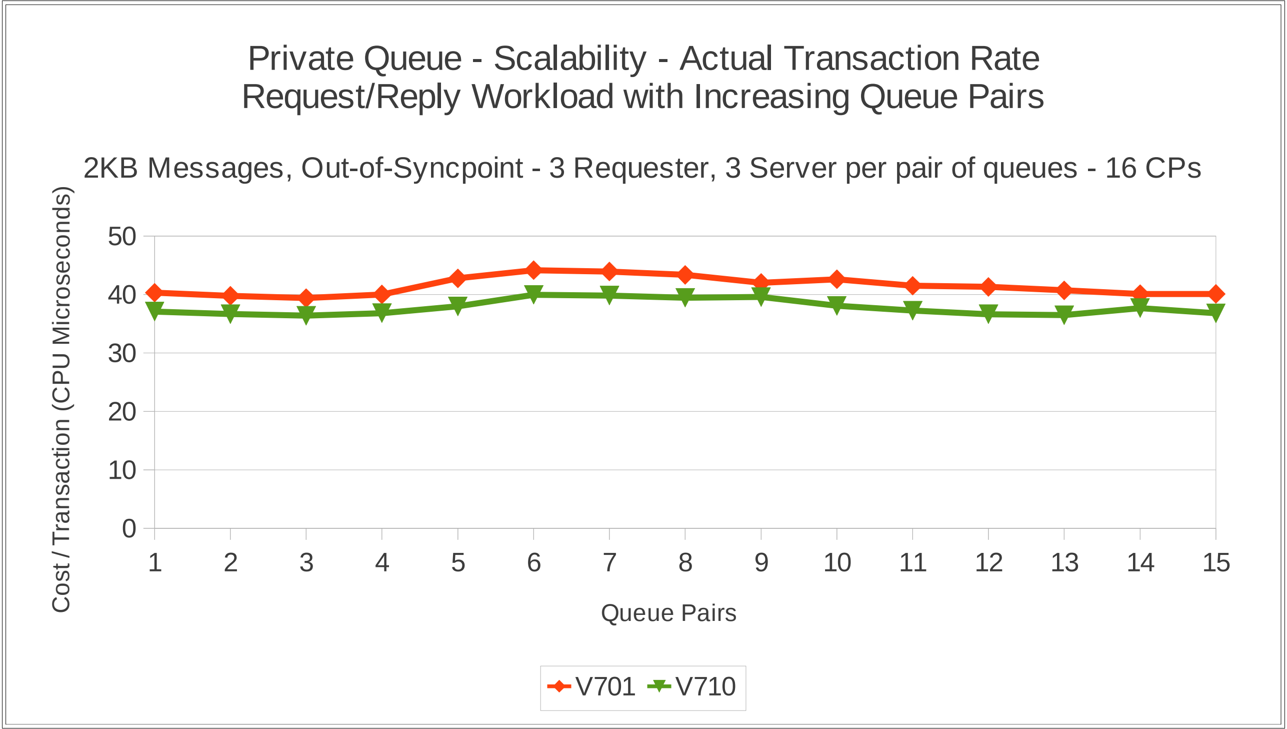
The following chart shows the performance of V6.0.0 for small messages (where multiple messages can fit on a single page) does not scale on the fast hardware. In releases prior to V7.0.0, the scavenger task would run on a fixed interval. As the zEnterprise 196 runs significantly faster than previous generations of mainframe, the scavenger task was unable to keep pace with the workload and the workload spilled over into pageset usage. As a result, the requester application had to slow its rate of put.


Private Queue - Scalability - Actual Transaction Rate
Request/Reply Workload with Increasing Queue Pairs
2KB Messages, Out-of-Syncpoint - 1 Requester, 1 Server per pair of queues - 16 CPs
Cost / Transaction (CPU Microseconds)
100.00
80.00
60.00
40.00
20.00
0.00
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Queue Pairs
V600
V701
V710


Private Queue - Scalability - Sustained Transaction Rate
Request/Reply Workload with Increasing Queue Pairs
2KB Messages, Non-Persistent Out-of-Syncpoint -
1 Requester, 1 Server per pair of queues - 16 CPs
Transactions / Second
350000.00
300000.00
250000.00
200000.00
150000.00
100000.00
50000.00
0.00
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Queue Pairs
V600
V701
V710

By increasing the number of requester and server tasks accessing each pair of queues, we can drive the 16-way LPAR at close to 100% of capacity. This gives the transaction rate and costs shown in the following charts:

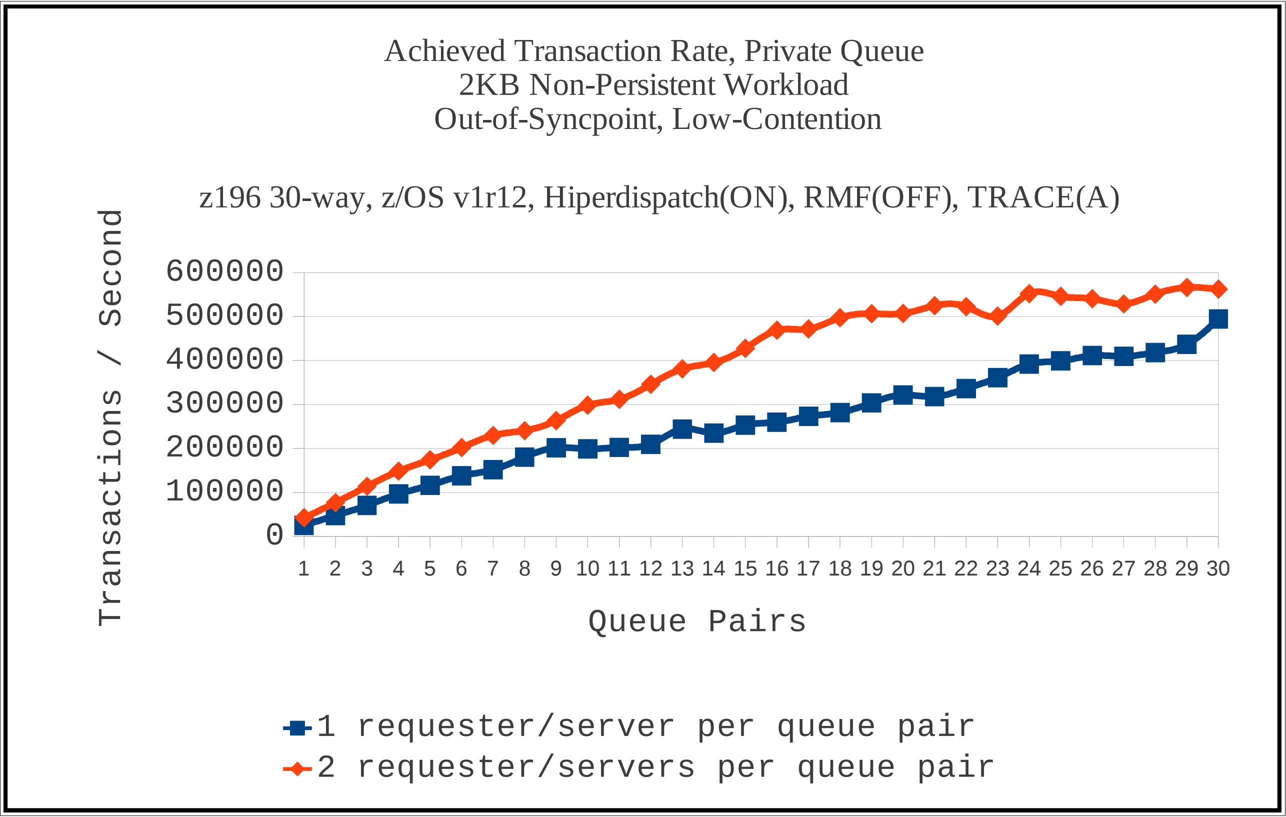




Each transaction involves the requester putting a message, the server getting a message and putting a reply and then the requester getting the reply. This means that each transaction is a total of 2 MQPUTs and 2 MQGETs.

The preceding 2 charts show that a single queue manager is able to drive 400,000 transactions per second – or 800,000 non-persistent messages per second on a 16-way LPAR.

By increasing the number of processors available and using multiple request/reply queues per pageset, we have driven a V.71.0 queue manager to process 1.1 million messages per second (or 550,000 transactions per second) on a 30-way LPAR as shown in the chart following:



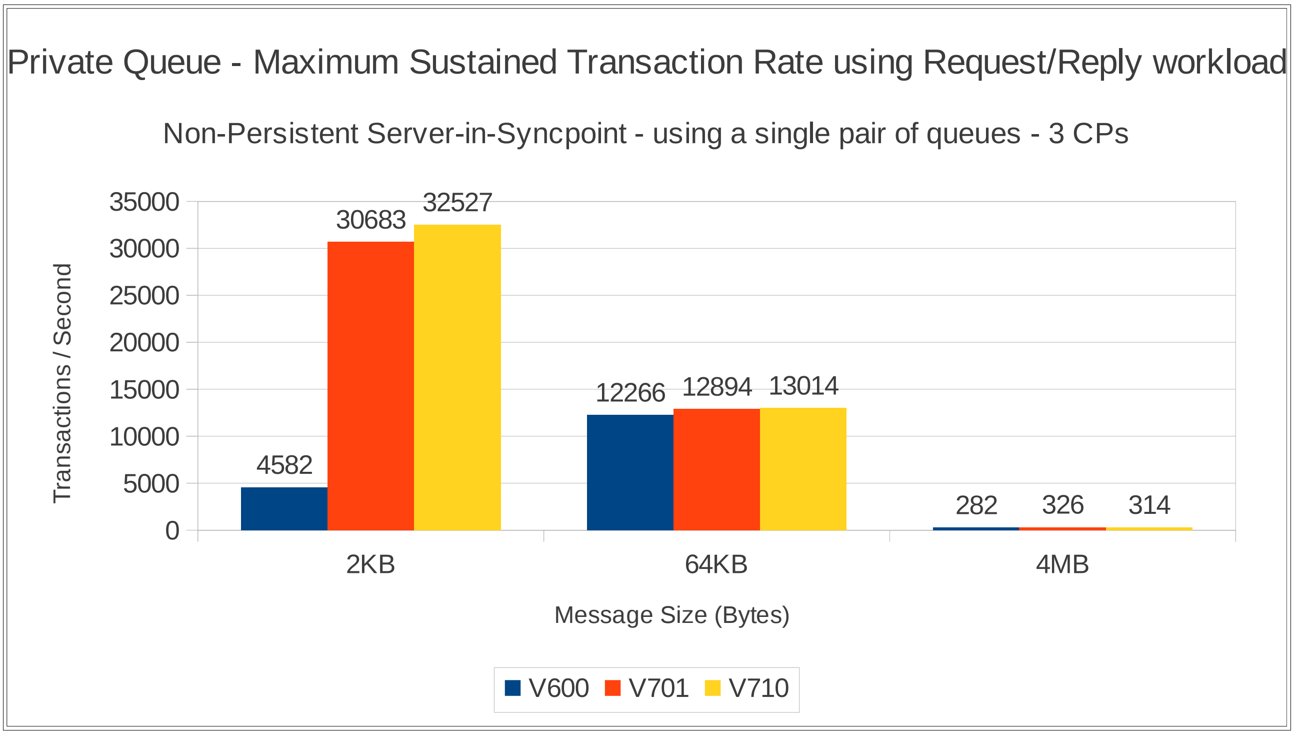
Non-persistent server in-syncpoint workload

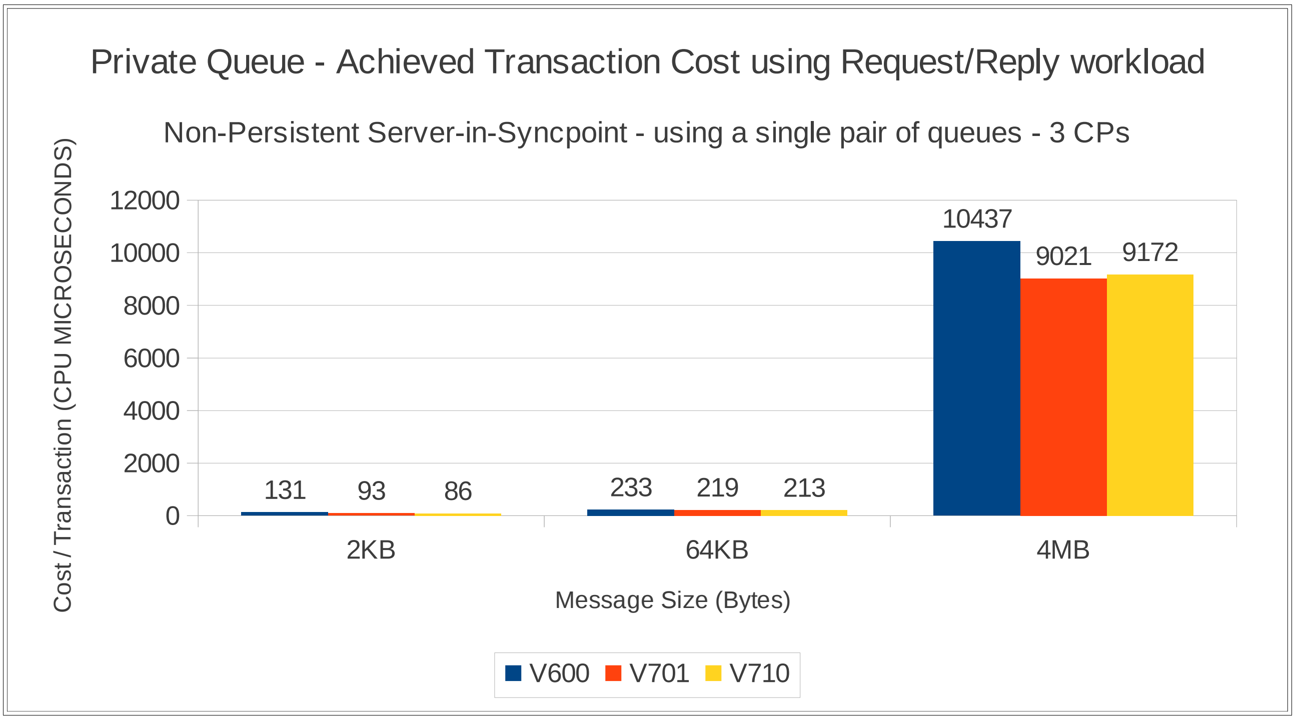
Maximum throughput on single pair of request/reply queues

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 got 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 use 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 an MQGET-with-wait. The messages are got and put in syncpoint with 1 MQGET and 1 MQPUT per commit.

As in the out of syncpoint workload, the performance of V6.0.0 does not look good. In releases prior to V7.0.0, the scavenger task would run on a fixed interval. As the zEnterprise 196 runs significantly faster than previous generations of mainframe, the scavenger task was unable to keep pace with the workload and the workload spilled over into pageset usage. As a result, the requester application needed to slow its rate of put.





Scalability of Request/Reply model across multiple queues

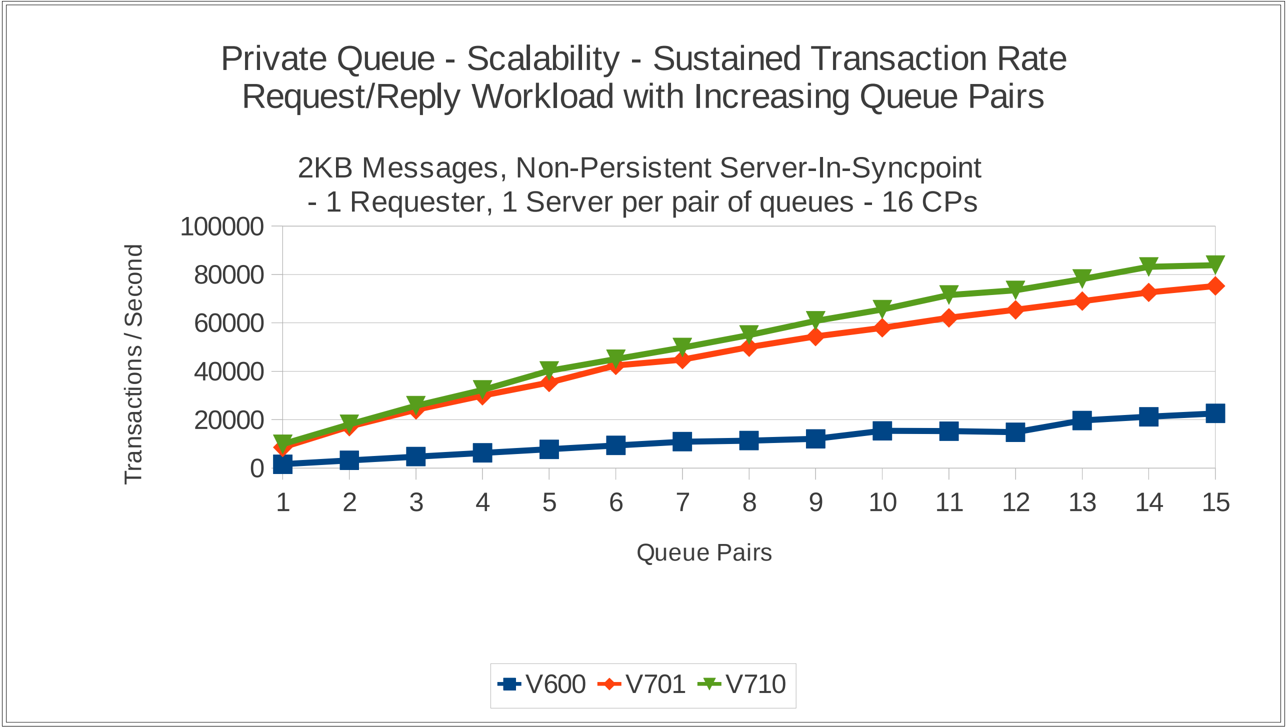
The queue manager is configured with 16 pagesets – 0 through 15 and a corresponding number of buffer pools.

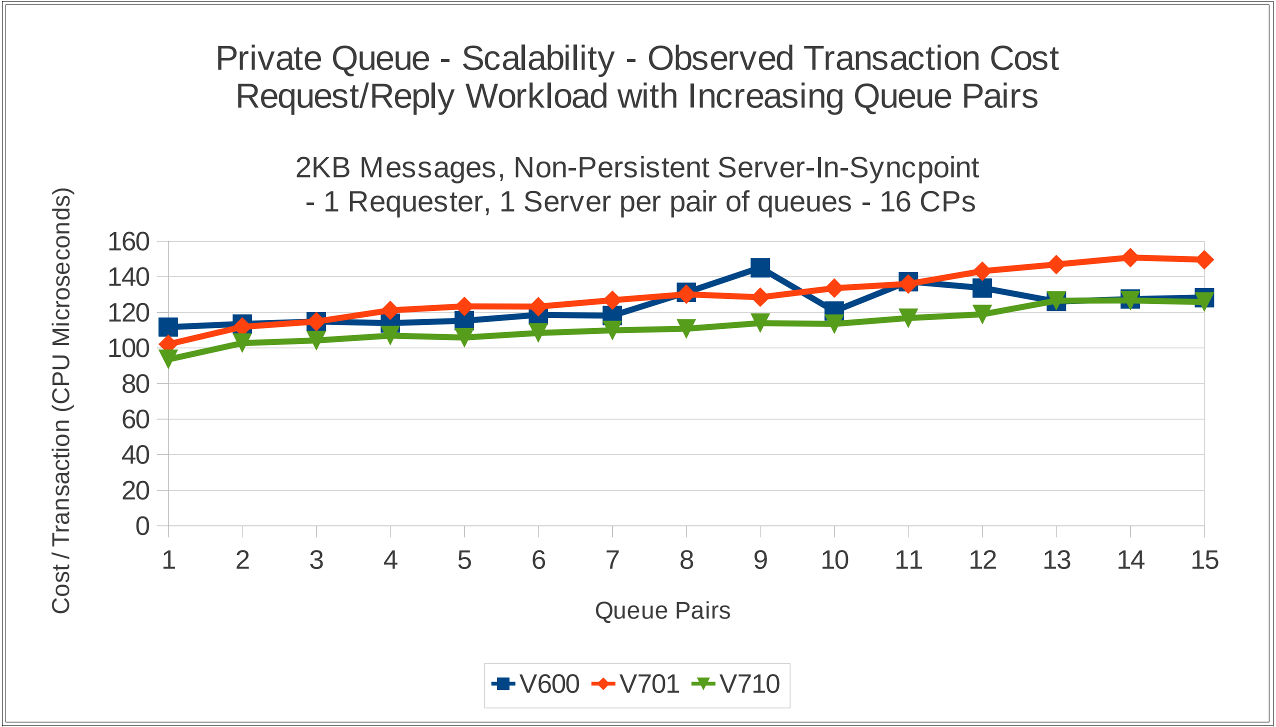
On each of pagesets 1 to 15, a pair of request and reply queues are defined. The test starts up 1 requester and 1 server task accessing the queues on pageset 1 and runs a request/reply workload.

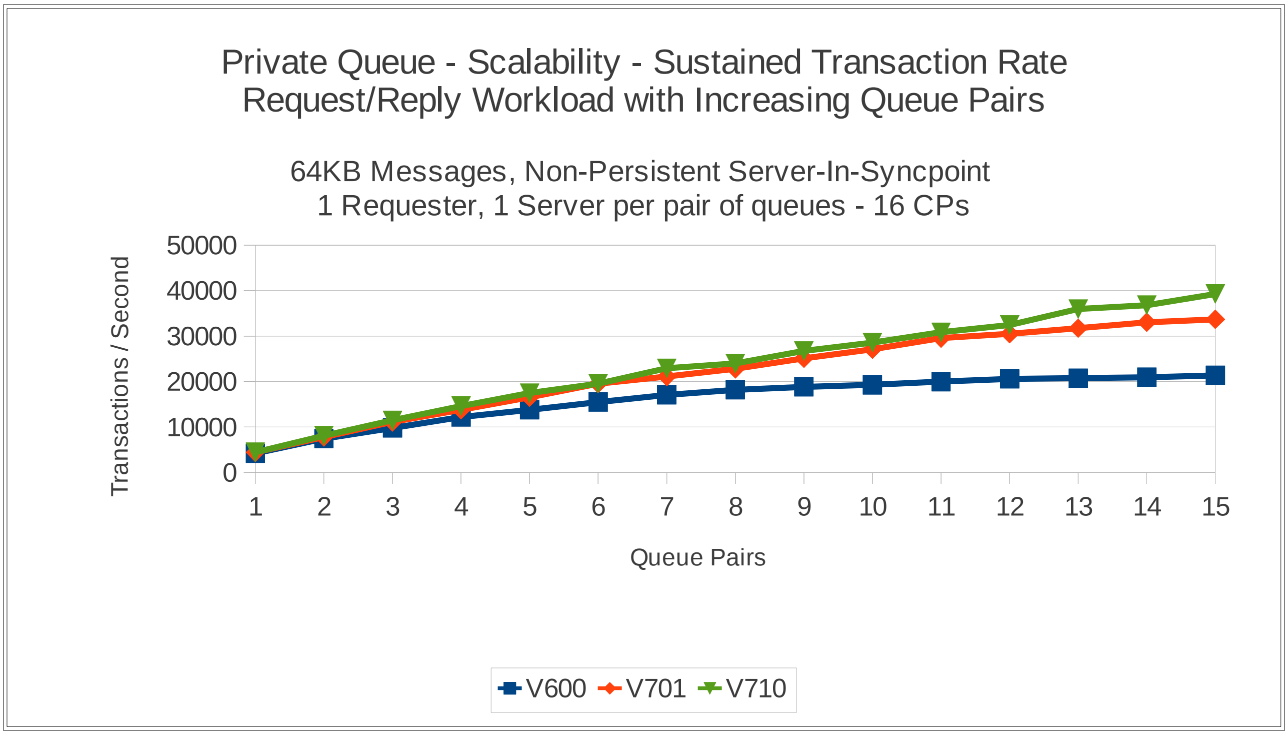
At the end, the test starts a second pair of requester and server tasks which access the queues on page set 2 and so on until there are 15 requester and 15 server tasks using queues on all 15 pagesets with application queues defined.

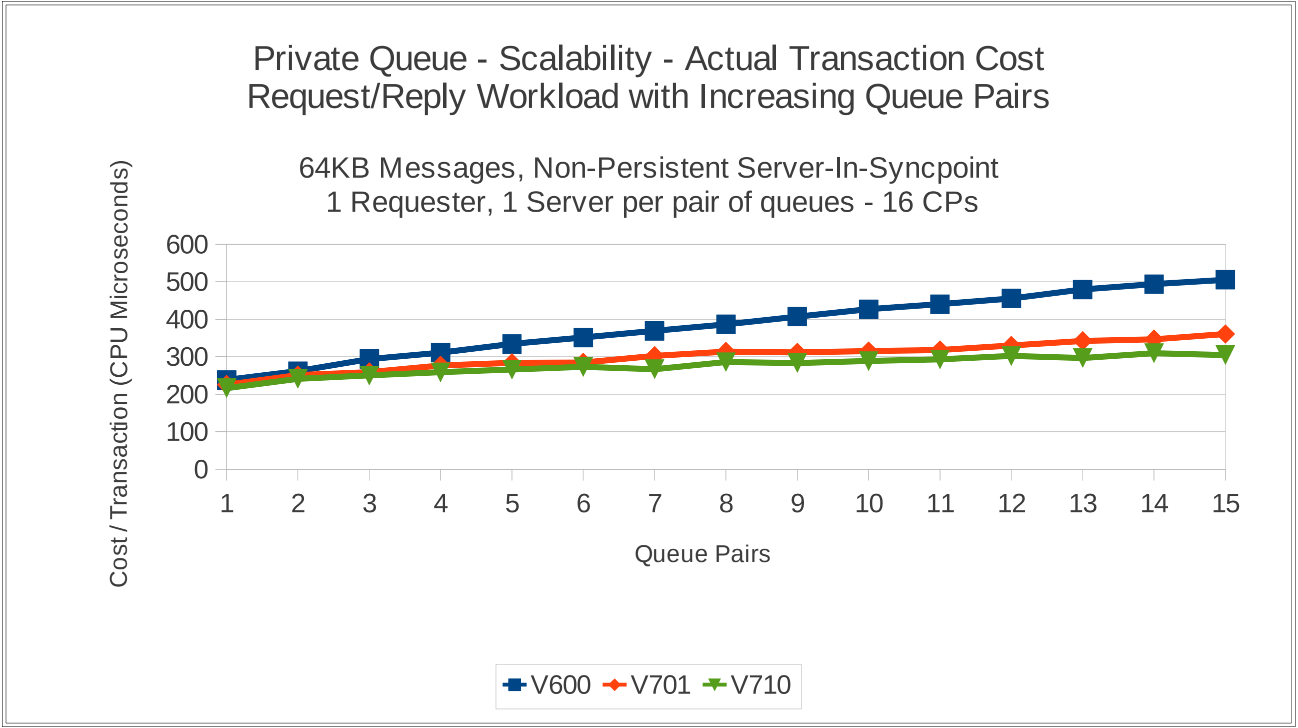
The requester tasks specify NO_SYNCPOINT for all messages and the server tasks get and put messages within syncpoint.

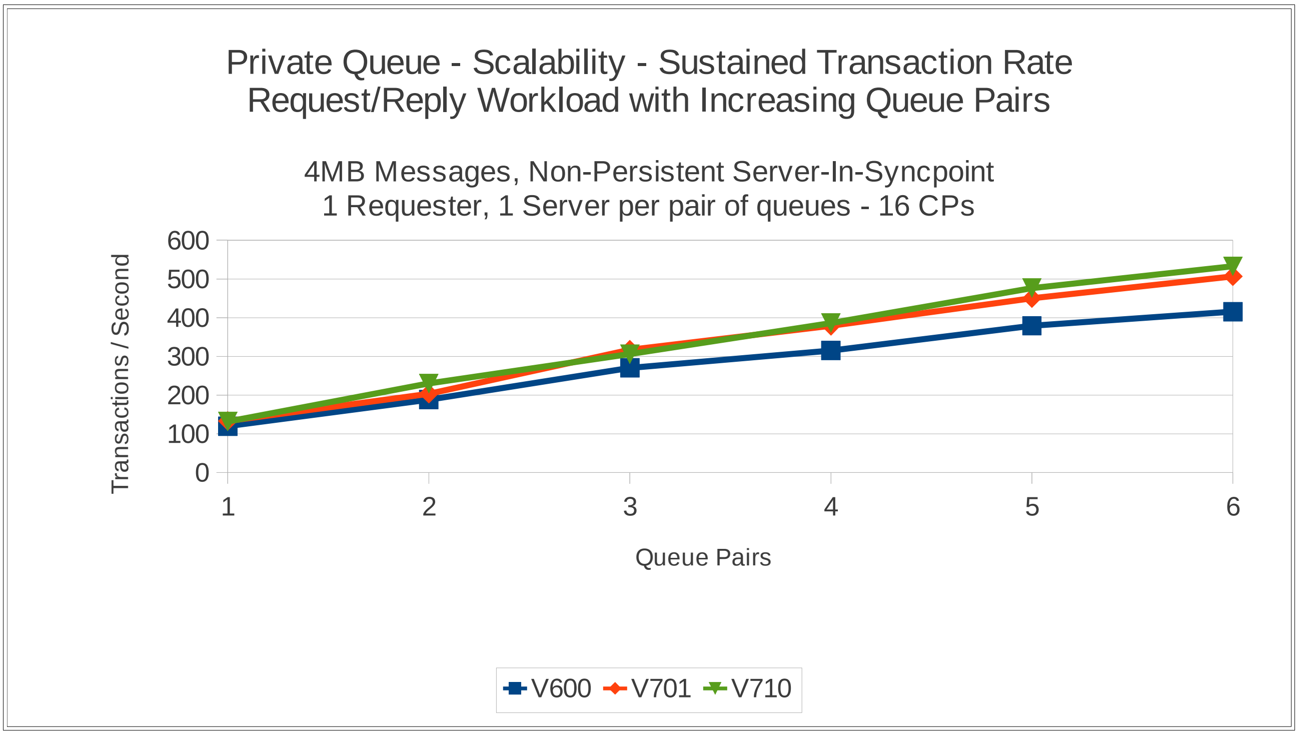
The measurements are run on a single LPAR with 16 dedicated processors online on the 2817 (z196) used for testing.

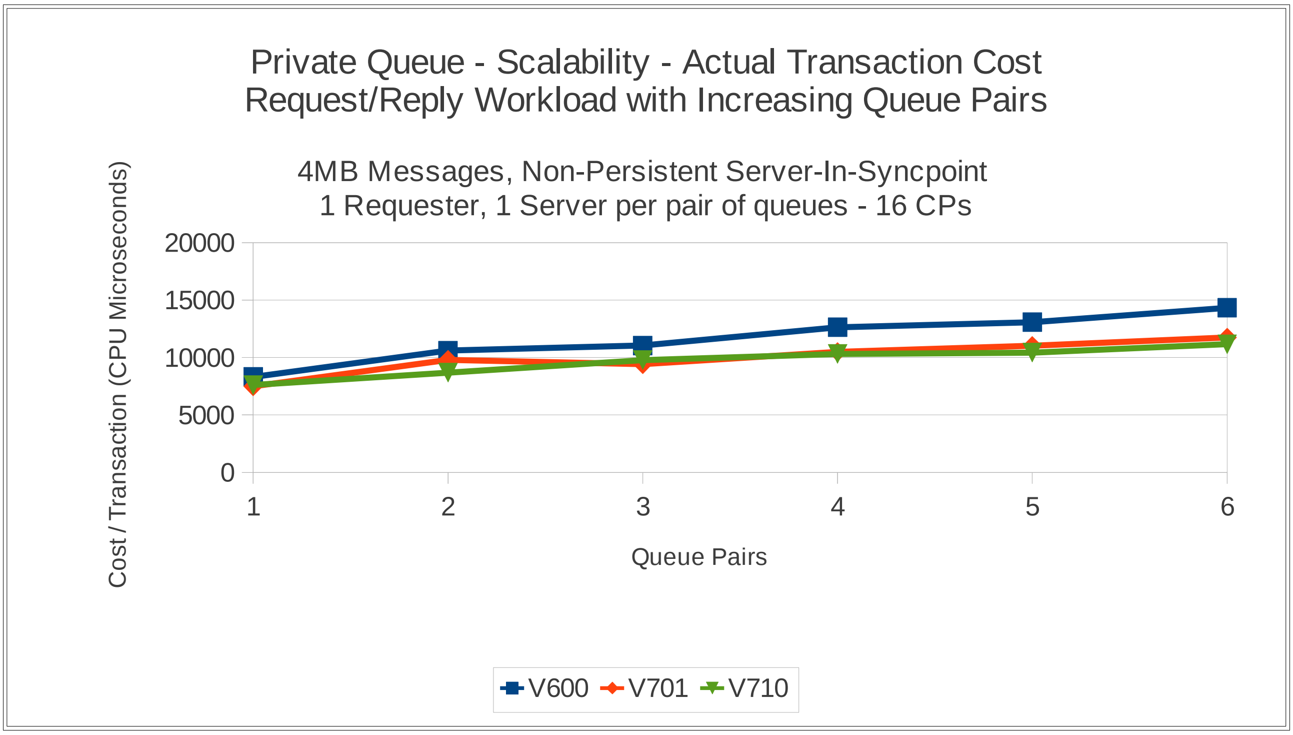

Private Queue - Scalability - Sustained Transaction Rate
Request/Reply Workload with Increasing Queue Pairs
2KB Messages, Non-Persistent Server-In-Syncpoint
- 1 Requester, 1 Server per pair of queues - 16 CPs
Transactions / Second
100000
80000
60000
40000
20000
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Queue Pairs
V600
V701
V710


Private Queue - Scalability - Observed Transaction Cost
Request/Reply Workload with Increasing Queue Pairs
2KB Messages, Non-Persistent Server-In-Syncpoint
- 1 Requester, 1 Server per pair of queues - 16 CPs
Cost / Transaction (CPU Microseconds)
160
140
120
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Queue Pairs
V600
V701
V710


Private Queue - Scalability - Sustained Transaction Rate
Request/Reply Workload with Increasing Queue Pairs
64KB Messages, Non-Persistent Server-In-Syncpoint
1 Requester, 1 Server per pair of queues - 16 CPs
Transactions / Second
50000
40000
30000
20000
10000
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Queue Pairs
V600
V701
V710


Private Queue - Scalability - Actual Transaction Cost
Request/Reply Workload with Increasing Queue Pairs
64KB Messages, Non-Persistent Server-In-Syncpoint
1 Requester, 1 Server per pair of queues - 16 CPs
Cost / Transaction (CPU Microseconds)
600
500
400
300
200
100
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Queue Pairs
V600
V701
V710


Private Queue - Scalability - Sustained Transaction Rate
Request/Reply Workload with Increasing Queue Pairs
4MB Messages, Non-Persistent Server-In-Syncpoint
1 Requester, 1 Server per pair of queues - 16 CPs
Transactions / Second
600
500
400
300
200
100
0
1
2
3
4
5
6
Queue Pairs
V600
V701
V710

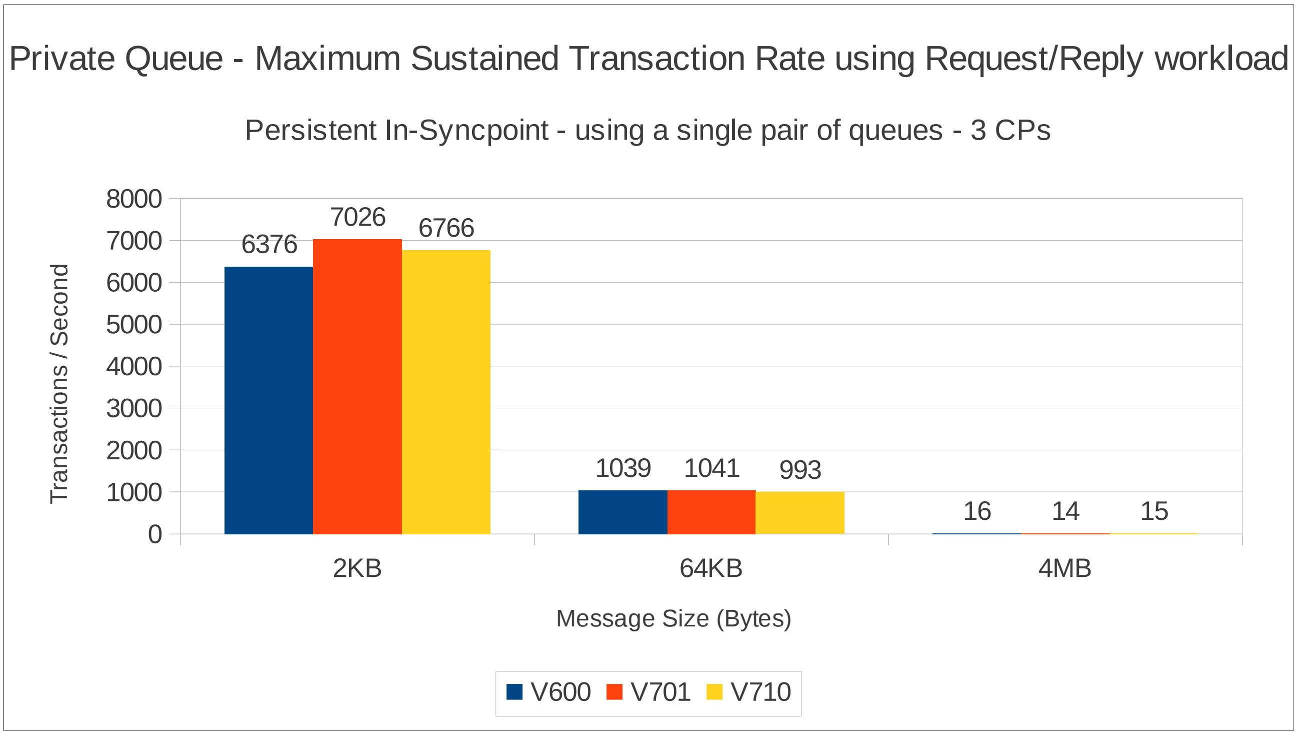

Private Queue - Scalability - Actual Transaction Cost
Request/Reply Workload with Increasing Queue Pairs
4MB Messages, Non-Persistent Server-In-Syncpoint
1 Requester, 1 Server per pair of queues - 16 CPs
Cost / Transaction (CPU Microseconds)
20000
15000
10000
5000
0
1
2
3
4
5
6
Queue Pairs
V600
V701
V710

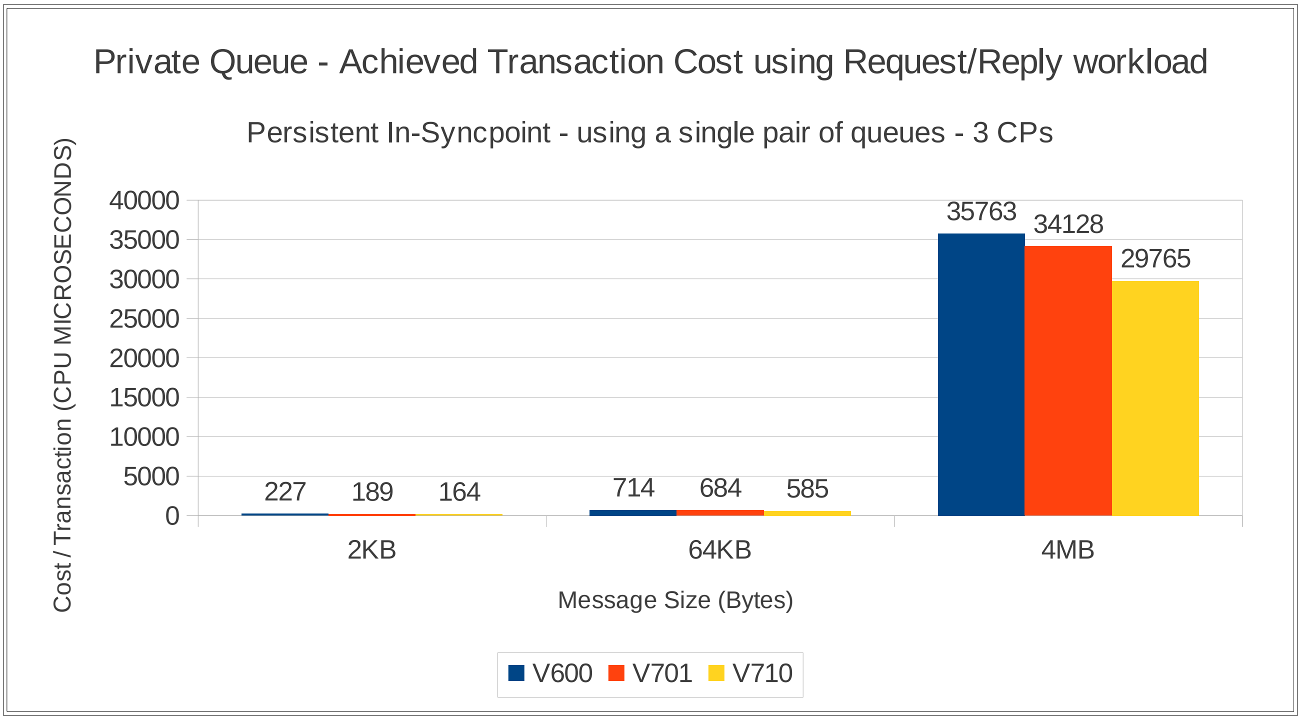
Persistent in-syncpoint workload

Maximum throughput on 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 got the message, they put another message to the request queue. The messages are put in-syncpoint and got in-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 an MQGET-with-wait. The messages are got and put in syncpoint with 1 MQGET and 1 MQPUT per commit.




Private Queue - Achieved Transaction Cost using Request/Reply workload
Persistent In-Syncpoint - using a single pair of queues - 3 CPs
Cost / Transaction (CPU MICROSECONDS)
40000
35000
30000
25000
20000
15000
10000
5000
0
227
189
164
714
684
585
35763
34128
29765
2KB
64KB
4MB
Message Size (Bytes)
V600
V701
V710

Regression – shared queue - CFLEVEL(4)

Non-persistent out-of-syncpoint workload

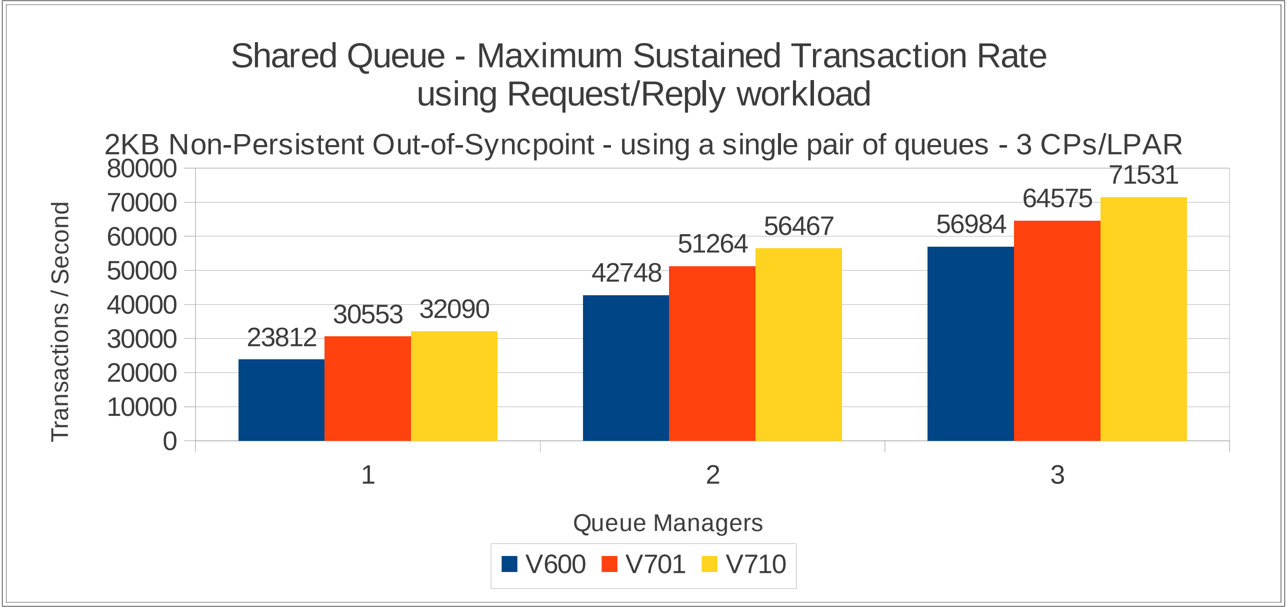
Maximum throughput on single pair of request/reply queues

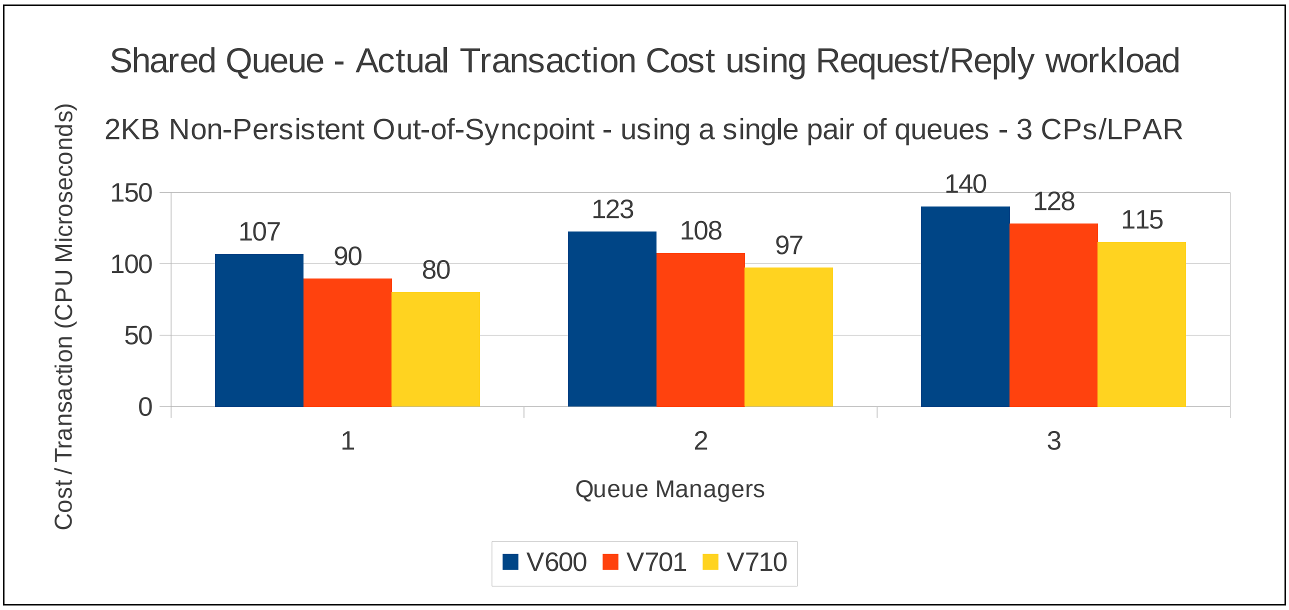
The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have got the message, they put another message to the request queue. The messages are put and got out of syncpoint.

There are 4 batch server tasks that use 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 an MQGET-with-wait. The messages are got and put out of syncpoint.

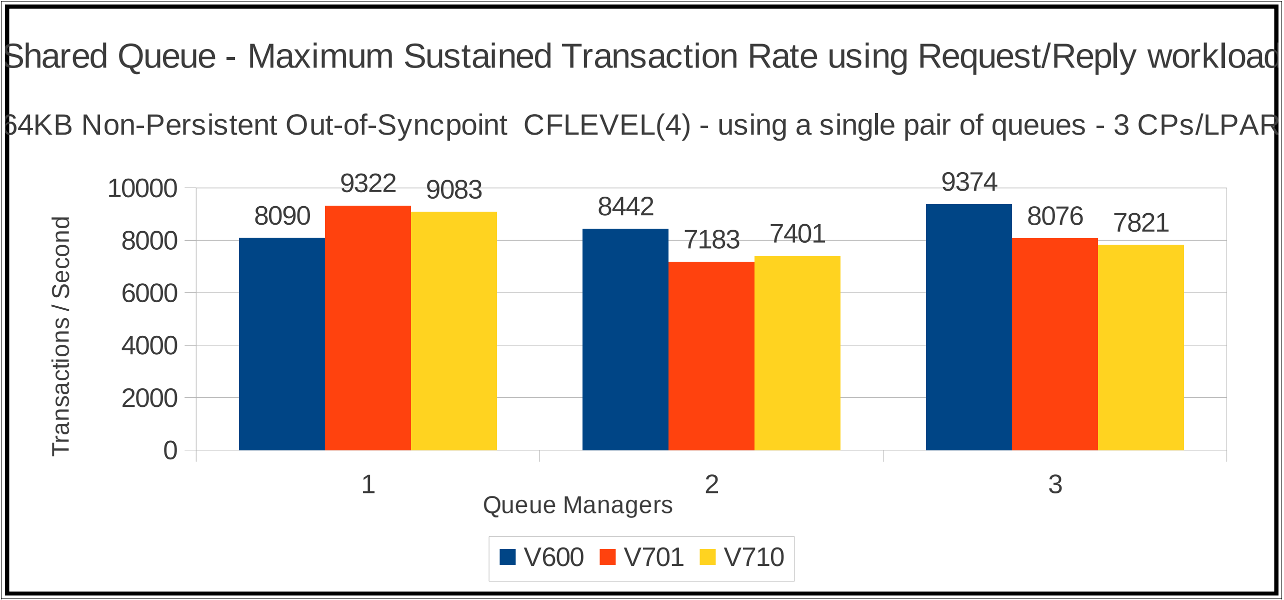
An increasing number of queue managers are allocated to process the workload. Each queue manager has 5 requester and 4 server tasks.

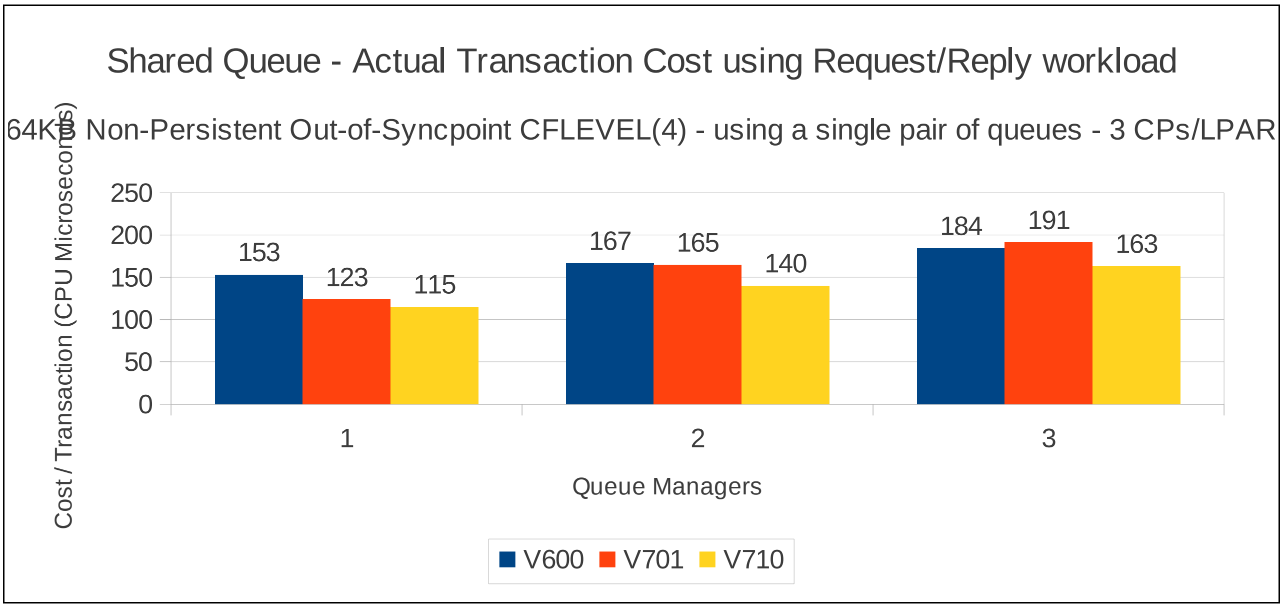
2KB messages



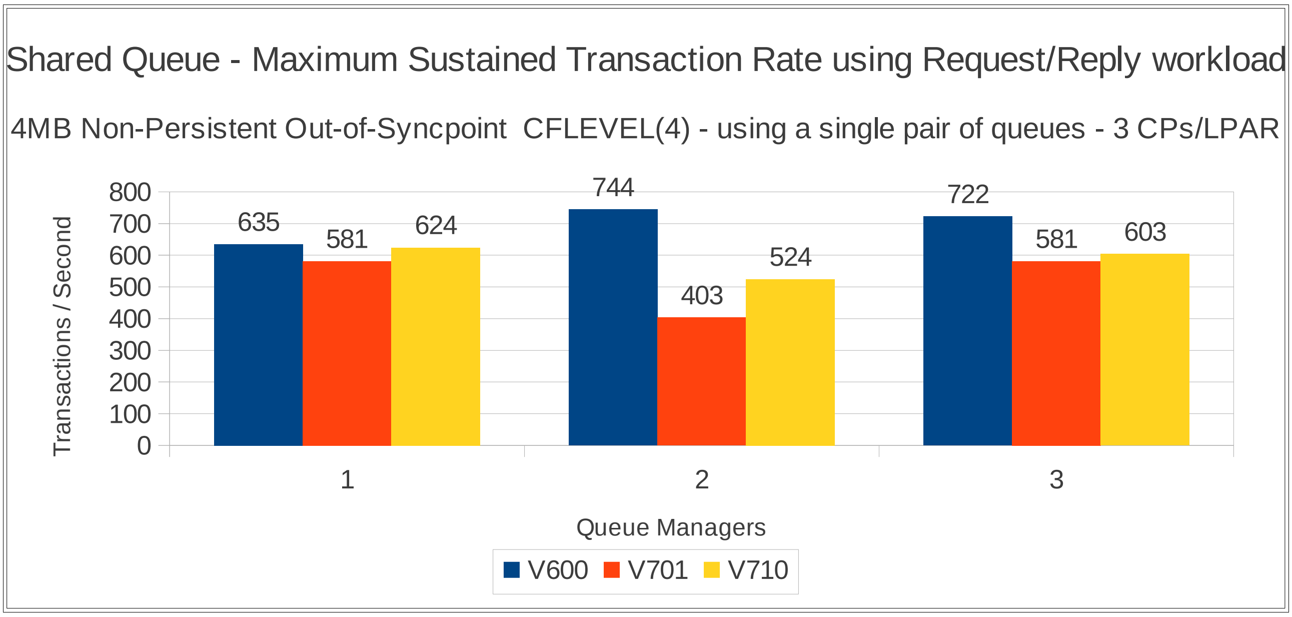


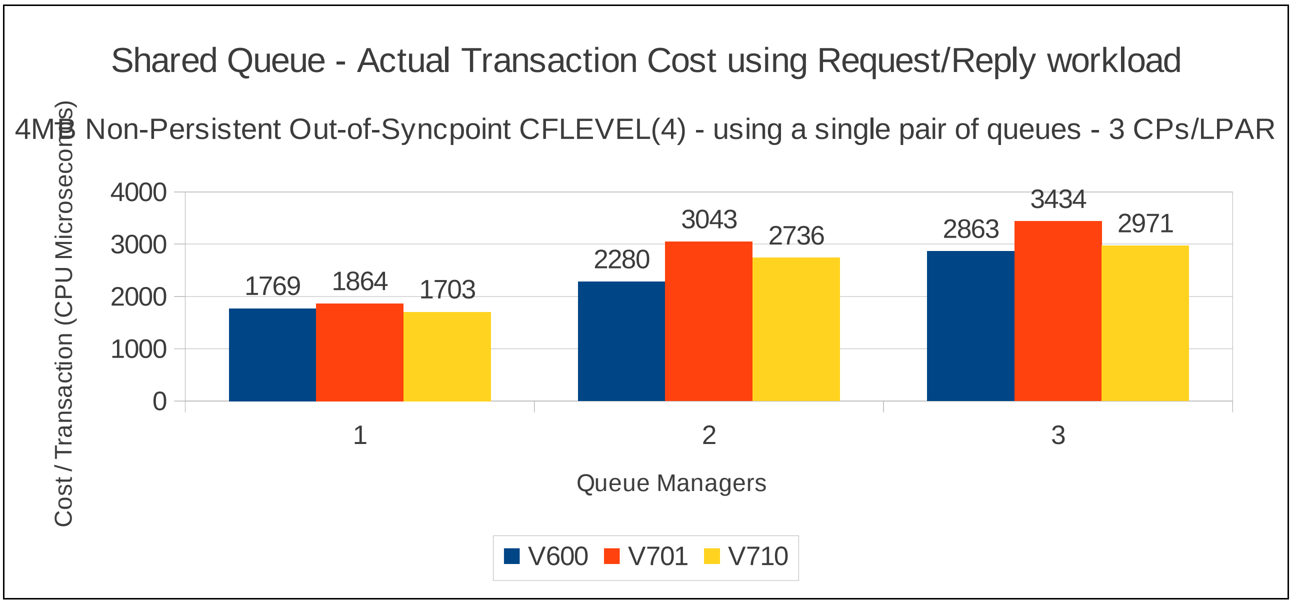
64KB messages





4MB messages





Non-persistent server in-syncpoint workload

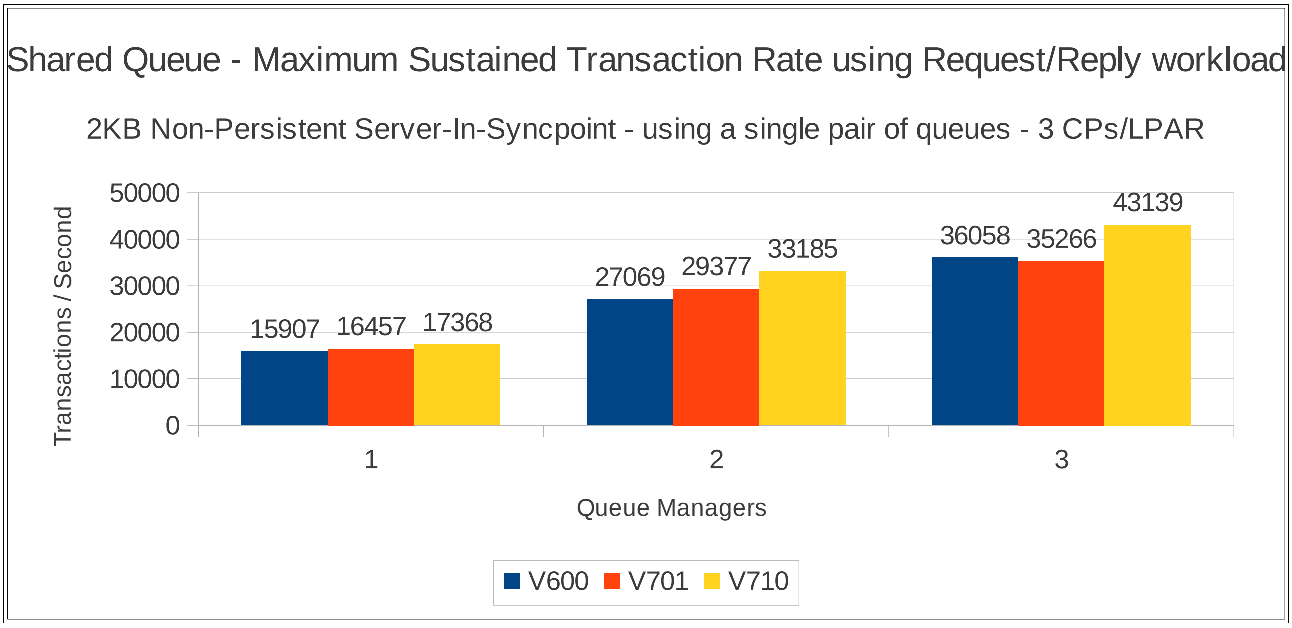
Maximum throughput on single pair of request/reply queues

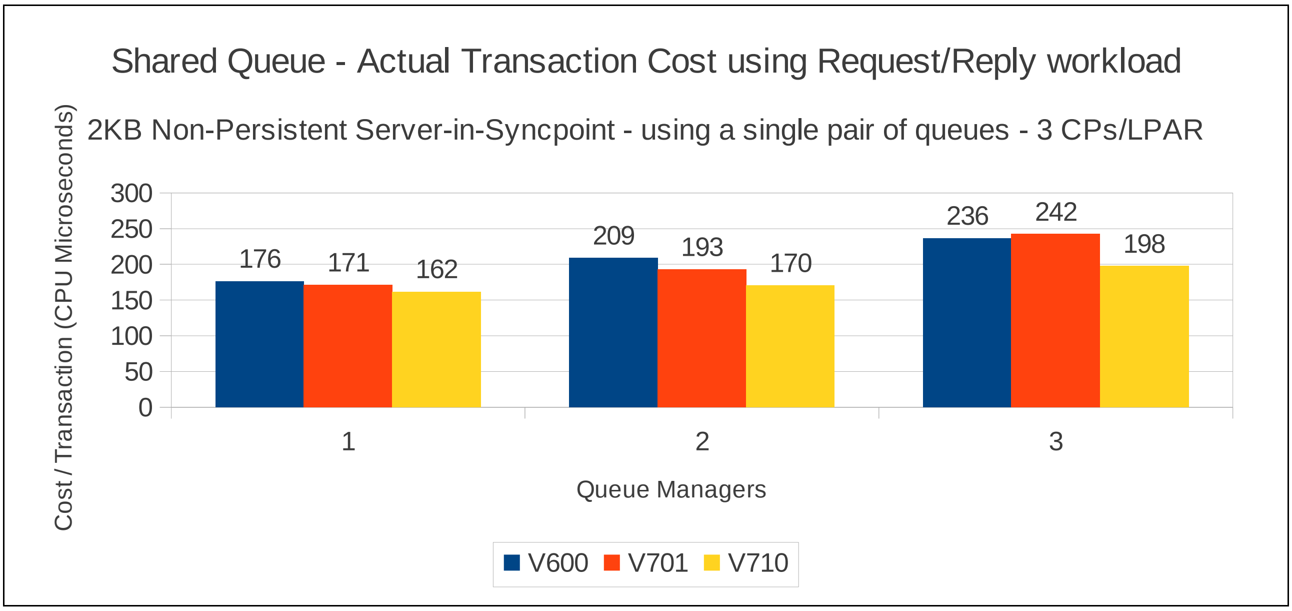
The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have got the message, they put another message to the request queue. The messages are put and got out of syncpoint.

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 an MQGET-with-wait. The messages are got and put in-syncpoint.

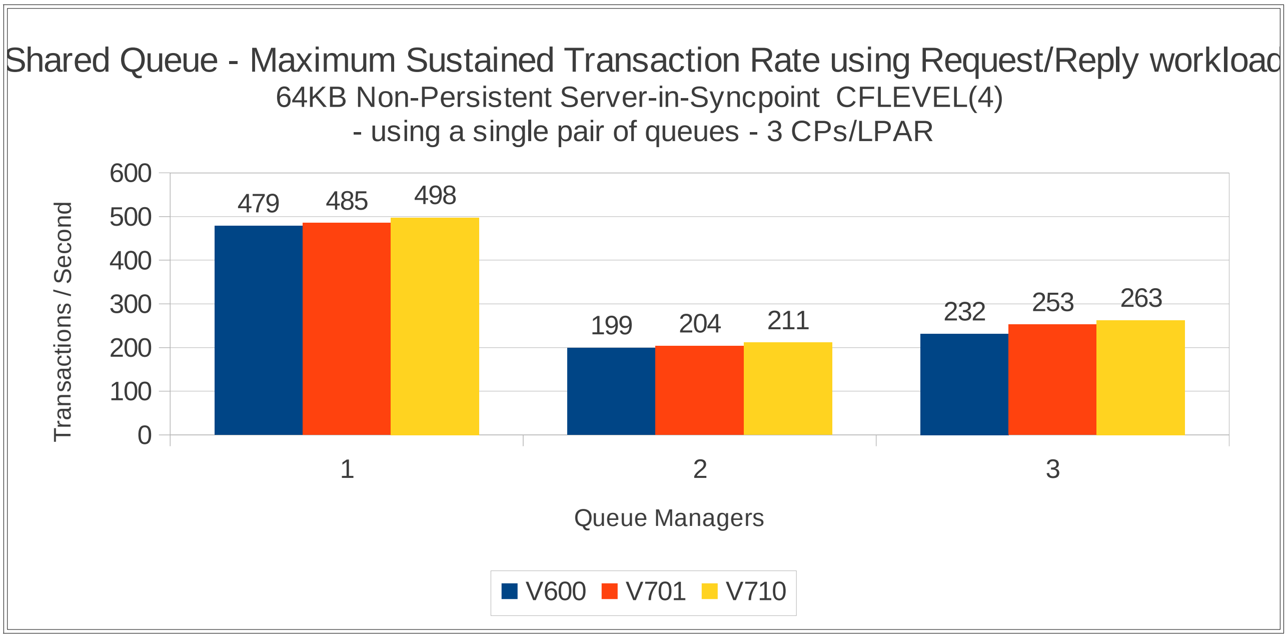
An increasing number of queue managers are allocated to process the workload. Each queue manager has 5 requester and 4 server tasks.

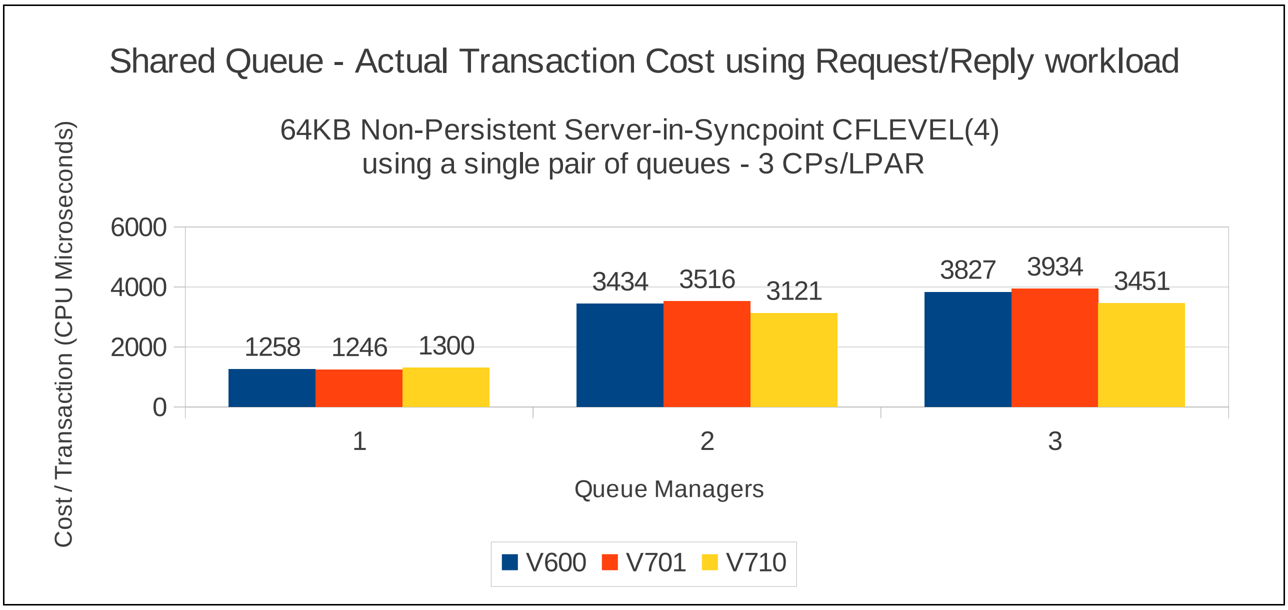
2KB messages



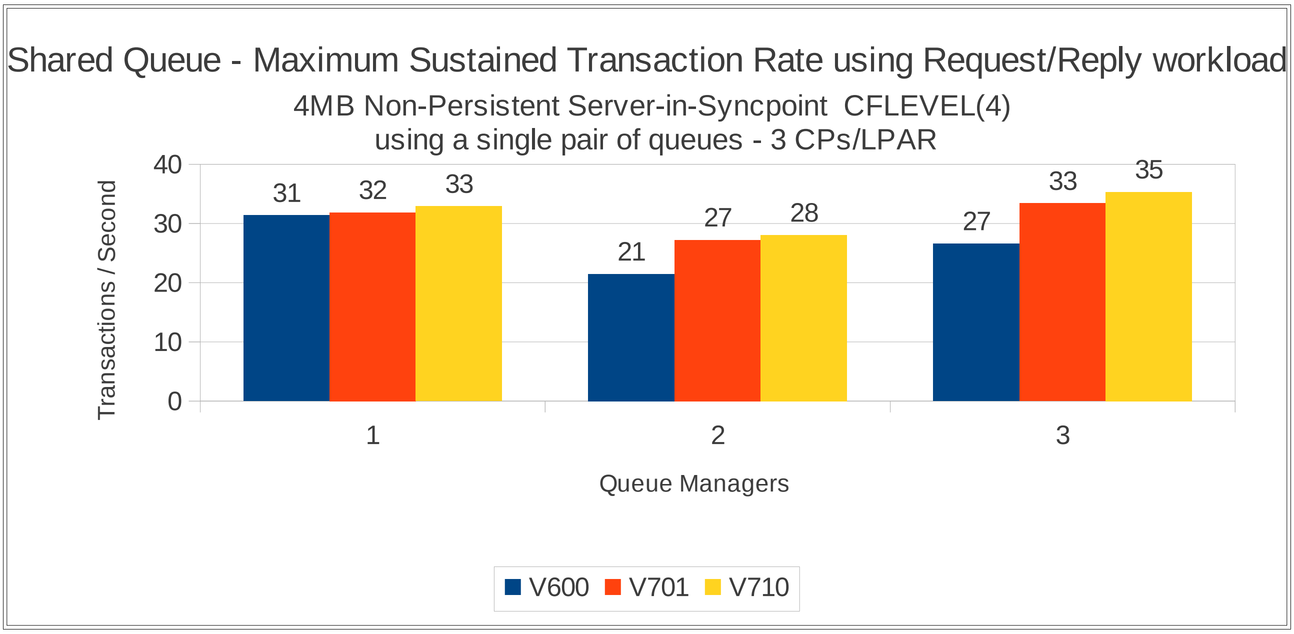


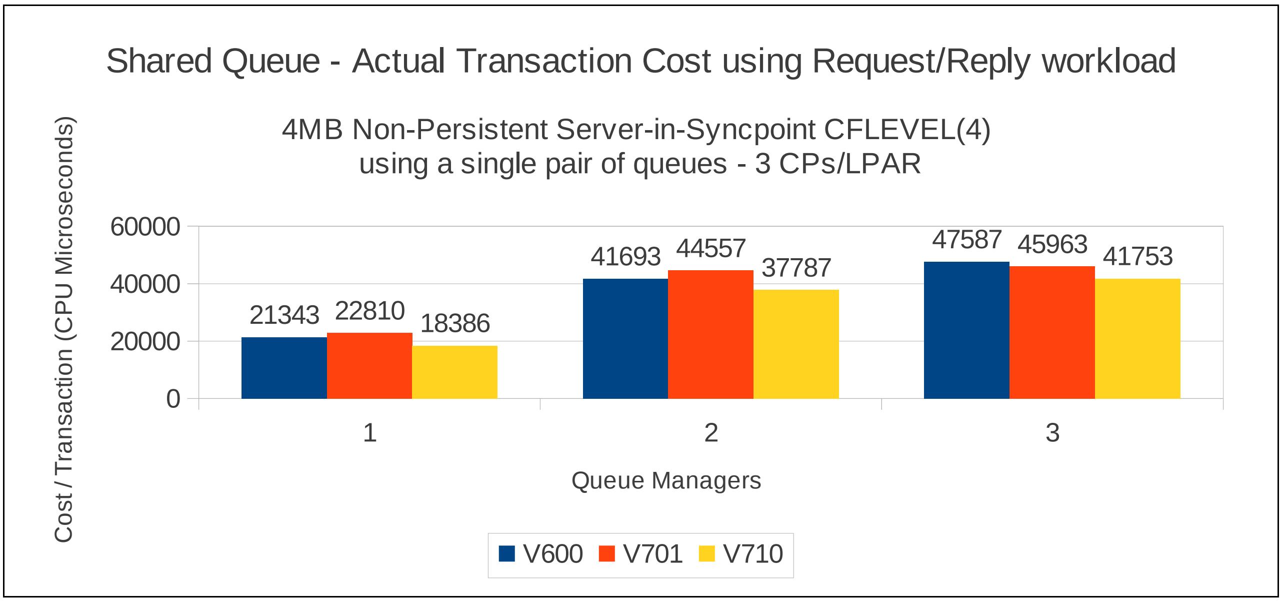
64KB messages





4MB messages





Data sharing non-persistent server in-syncpoint workload

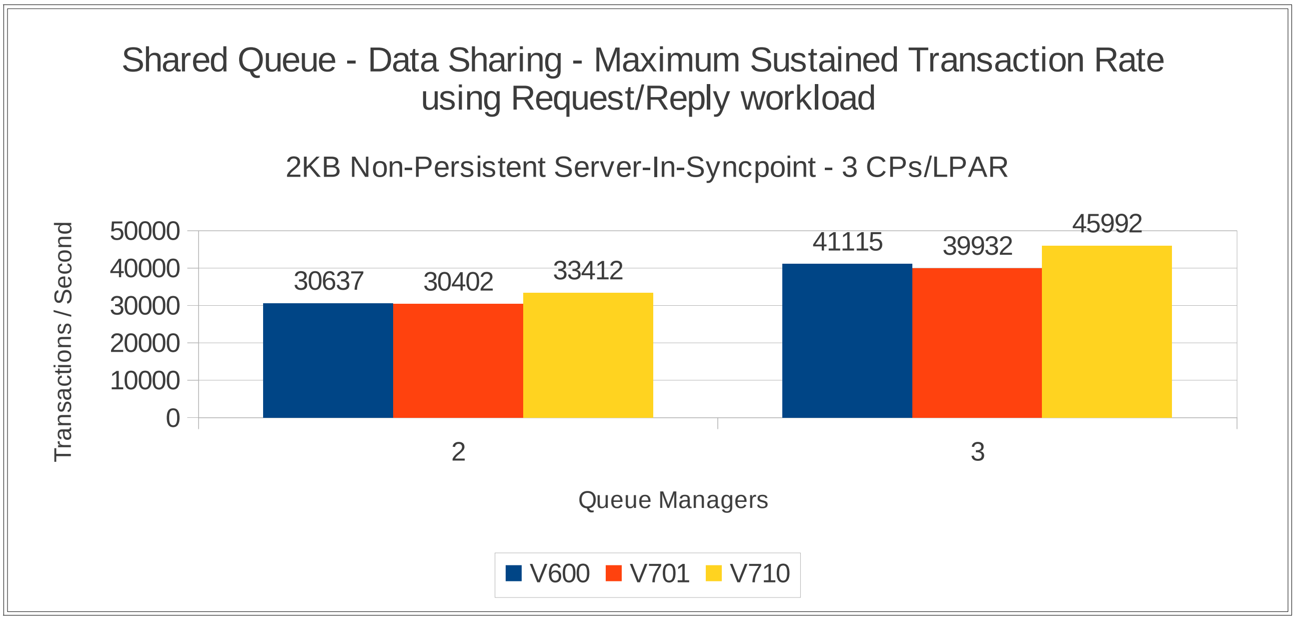
The previous shared queue tests are configured so that any queue manager within the queue sharing group can process messages put by any particular requester application. This means that the message may be processed by a server application on any of the available LPARs. Typically, the message is processed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

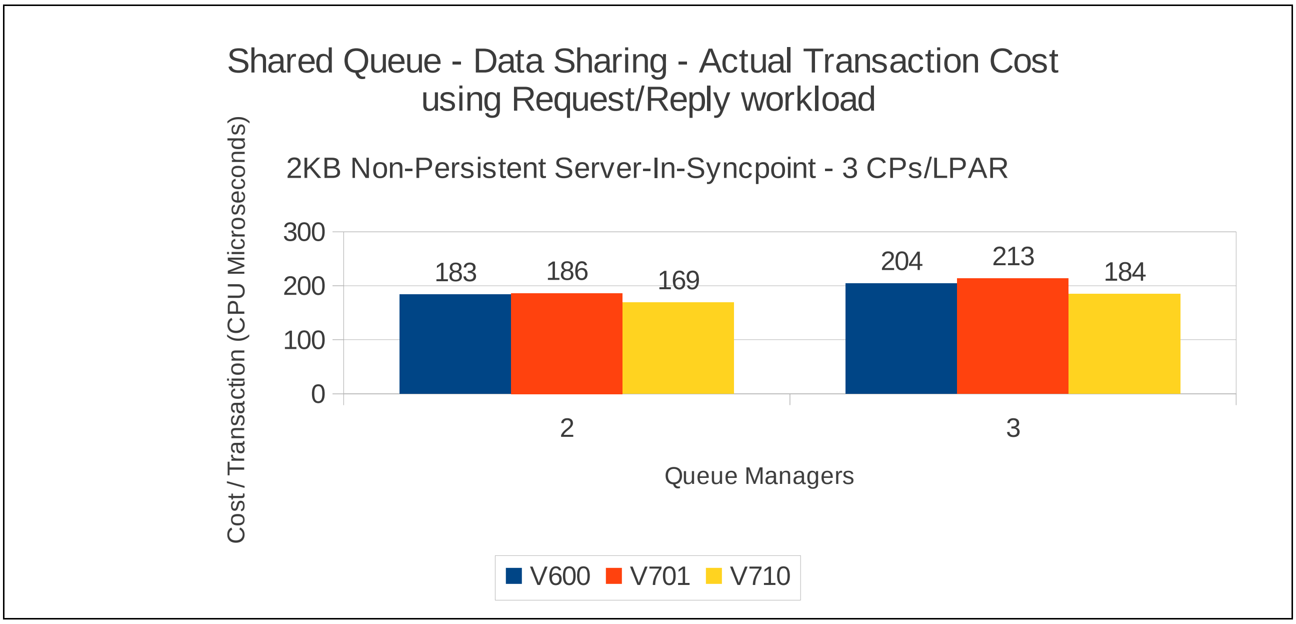
In the following tests, the message can only be processed by a server application on a separate LPAR. This is achieved by using multiple pairs of request/reply queues.

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

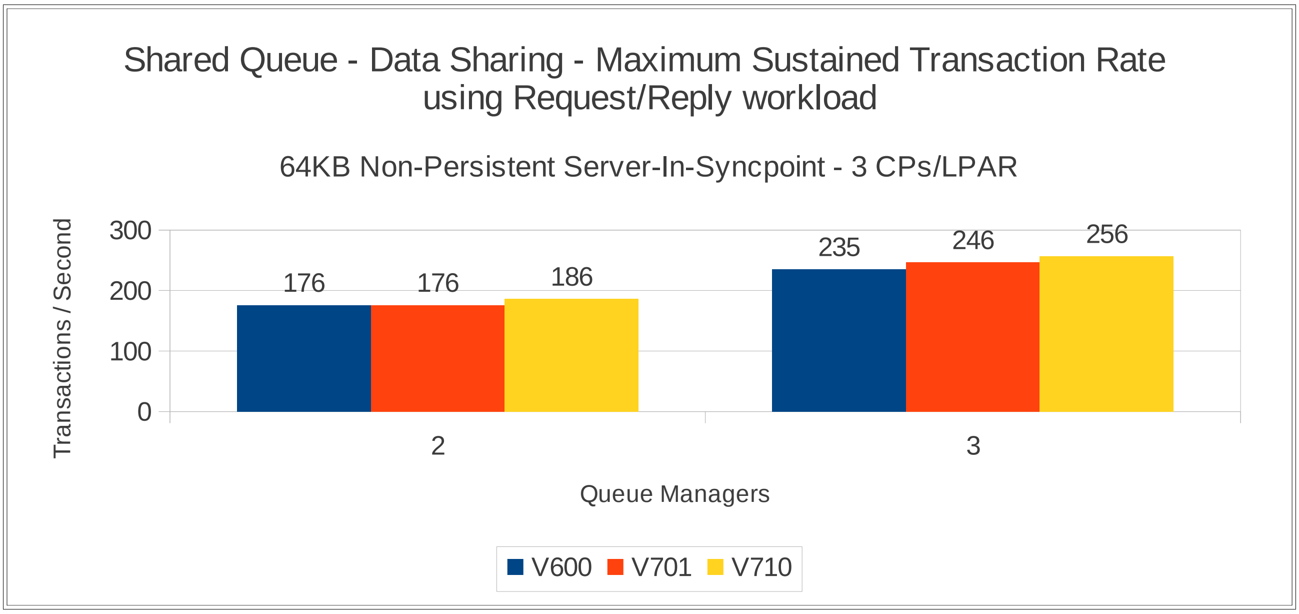
There are 4 batch server tasks on each queue manager that use MQGET-with-wait calls on the request queue of one of the remote queue managers, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into an MQGET-with-wait. The messages are got and put in-syncpoint.

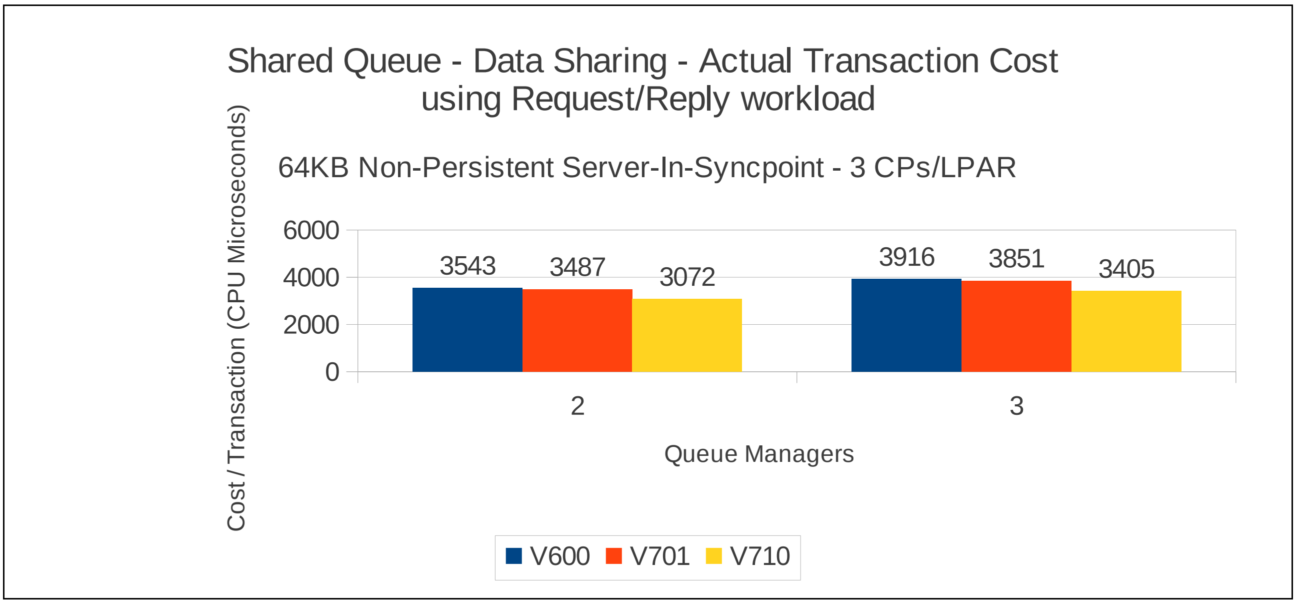
2KB messages





64KB messages





Regression – moving messages across channels

The regression tests for moving messages across channels e.g. sender-receiver channels, are designed to drive the channel initiator such that the network is the constraining factor. Therefore the tests use non-persistent messages out-of-syncpoint, so that they are not constrained by logging capacity, etc.

Within the channel tests there are measurements with small numbers of channels e.g 1 to 4, which was suitable for driving the network to capacity and also tests with 10 to 50 channels.

For each of the test types, the channels are used both with and without SSL encryption enabled.

The SSL tests are configured to use “TRIPLE_DES_SHA_US” encryption on the Crypto-Accelerator processor. The SSLRKEYC attribute is set so that 1MB of data can flow across the channel before renegotiating the key.

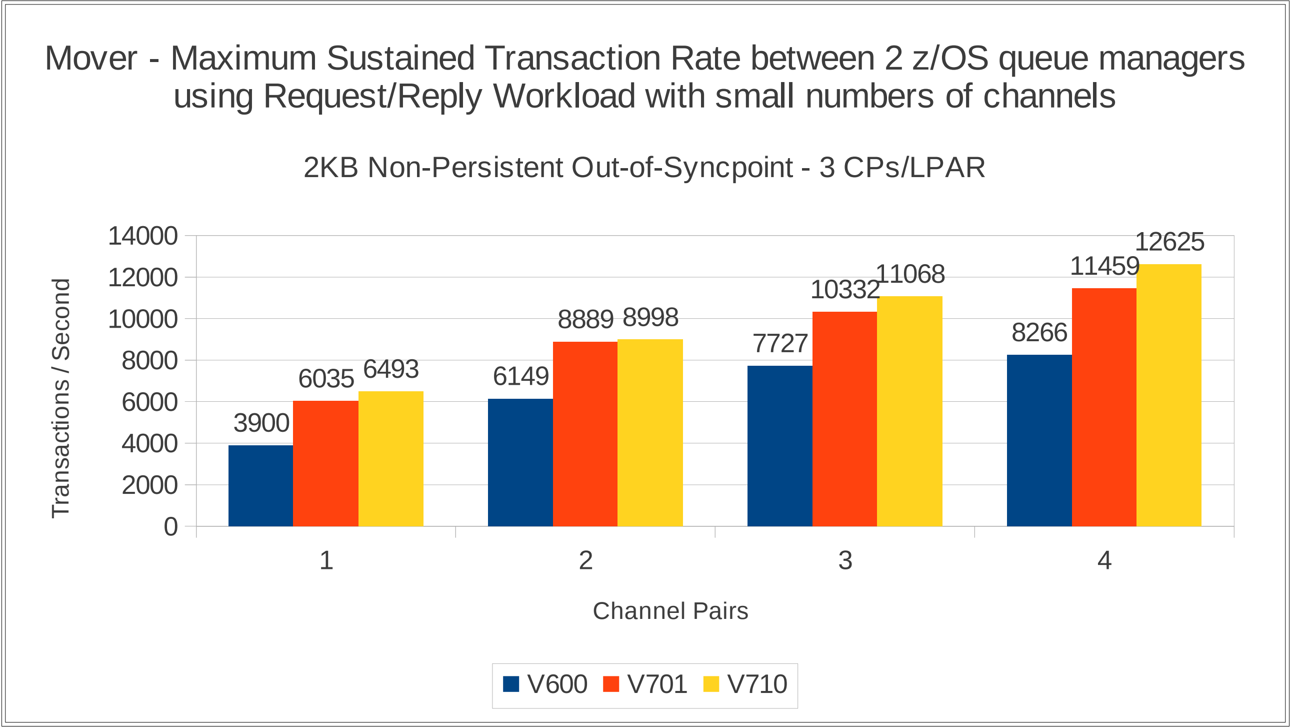
For further guidance on channel tuning and usage, please refer to the supportpac MP16 – Capacity Planning and Tuning Guide.

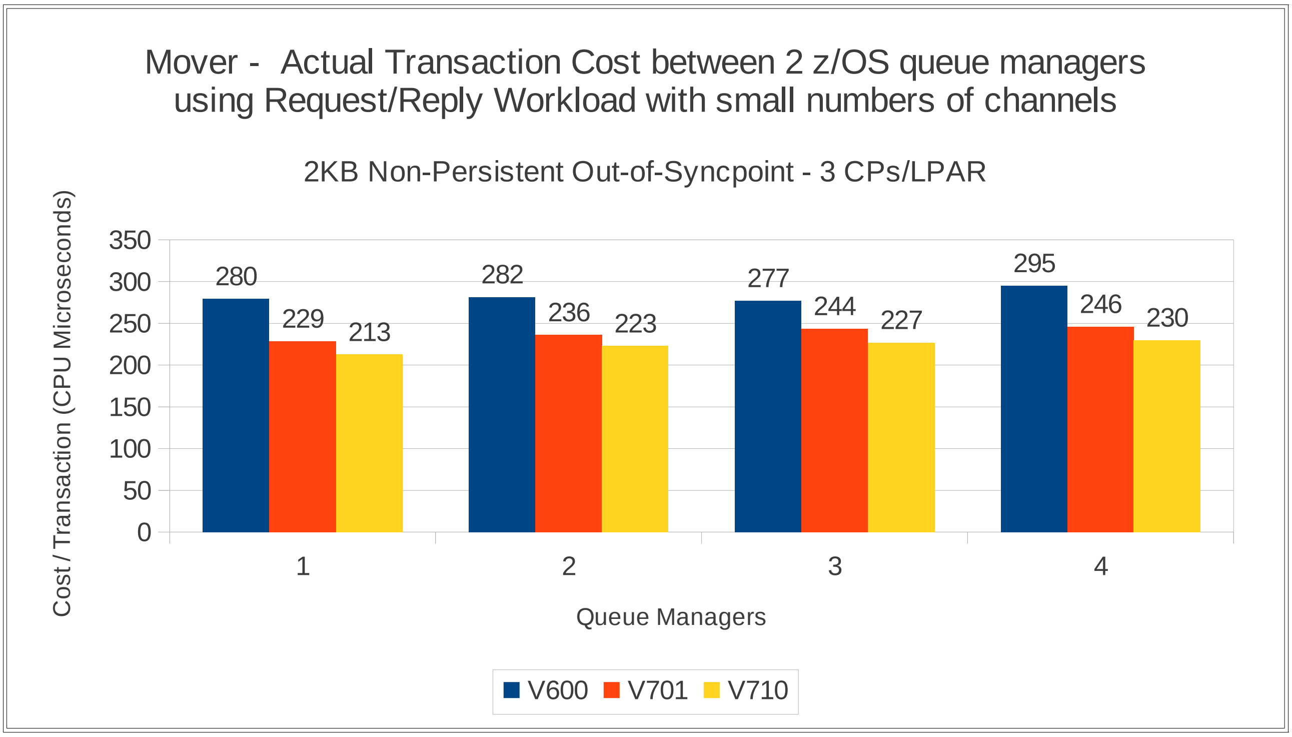
The measurements using 1 to 4 channels use batch applications to drive the workload at each end of the channel.

The measurements using 10 to 50 channels use long-lived CICS transactions to drive the workload. This means that each CICS application will put and get thousands of messages before ending. This model means that we are not including the cost of starting a CICS transaction, opening and closing queues and the teardown of the transaction at the end of the workload.

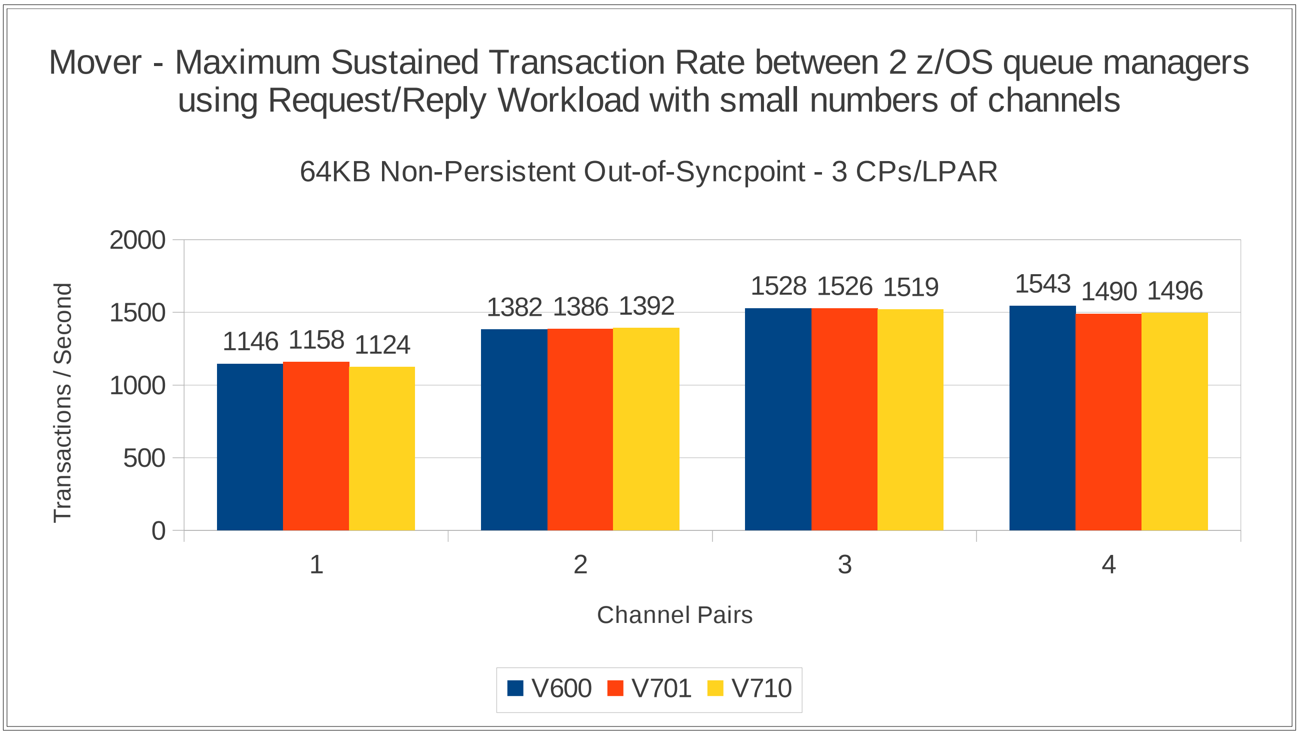
Non-persistent out-of-syncpoint – 1 to 4 sender-receiver channels

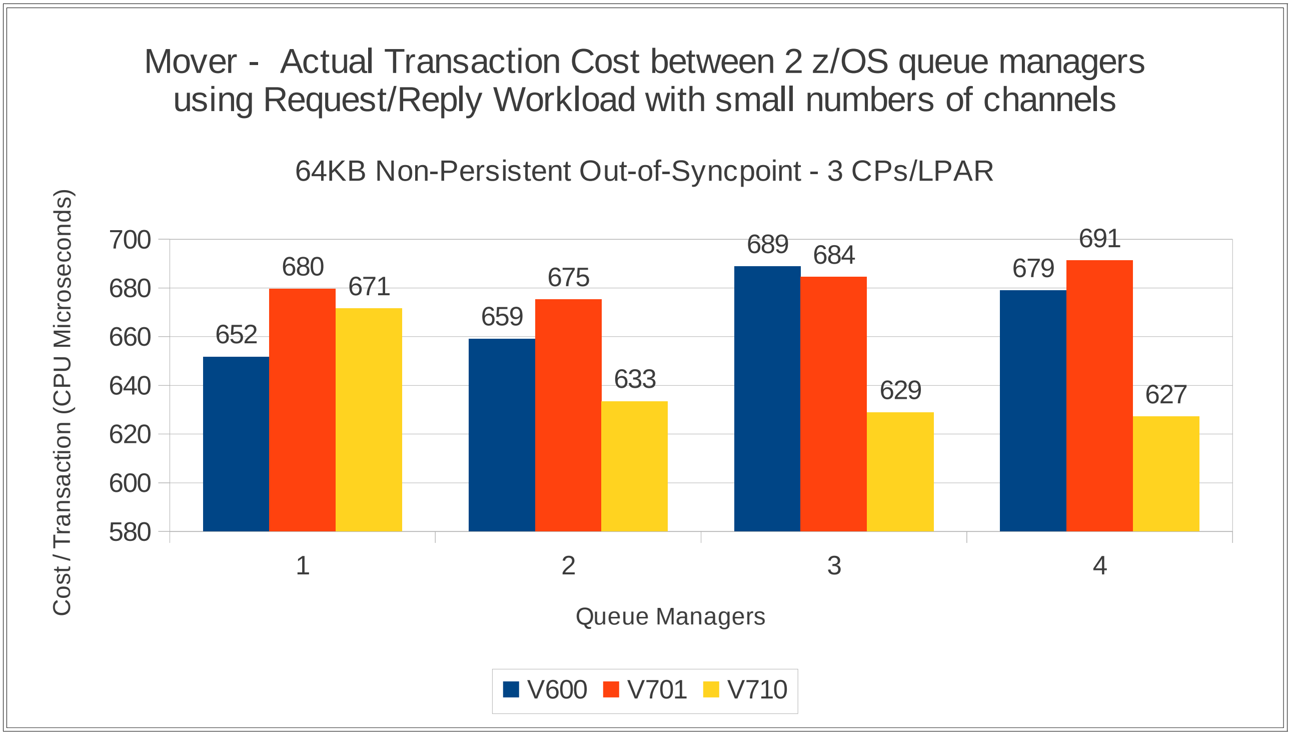
2KB messages



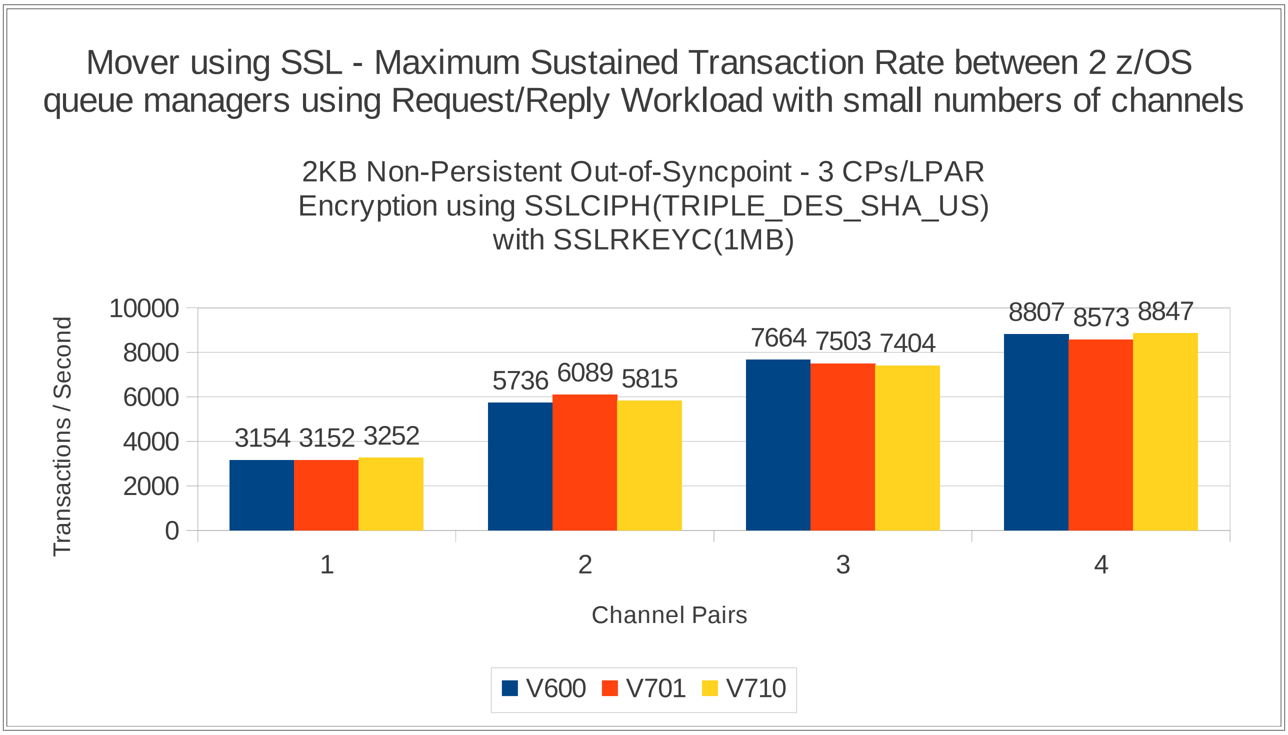


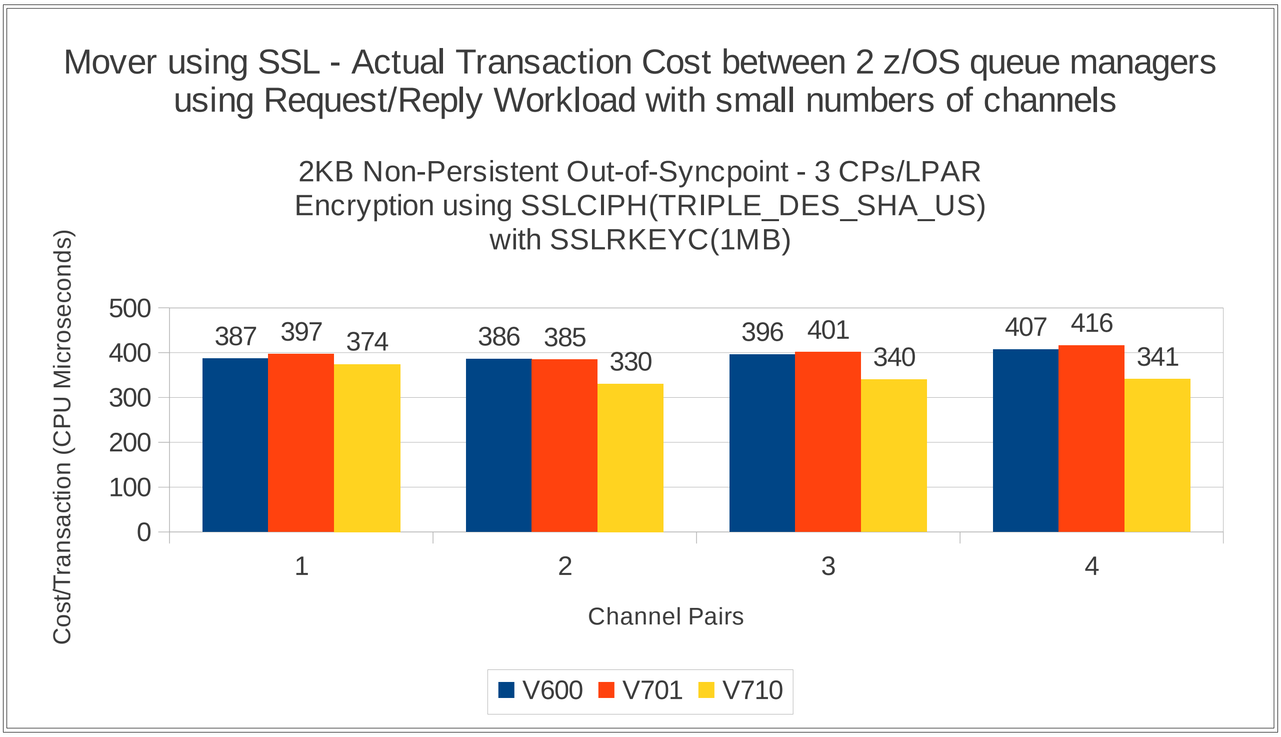
64KB messages



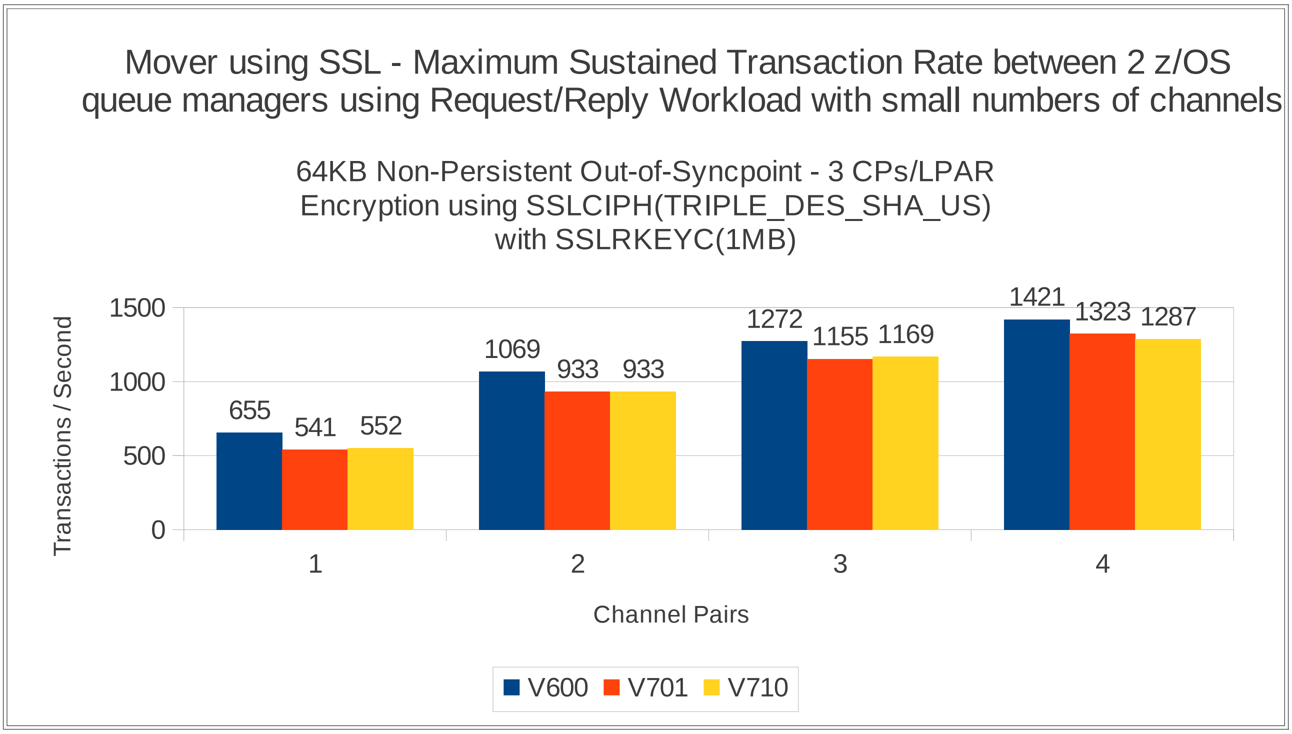


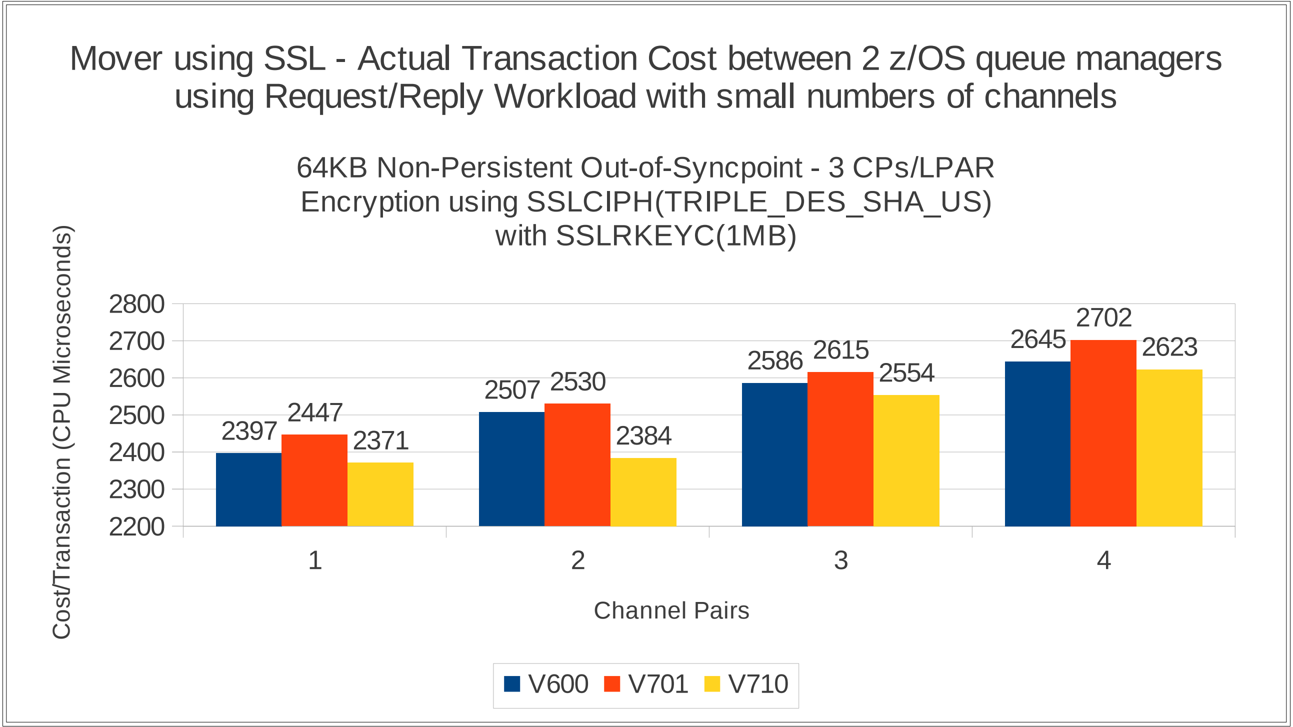
2KB messages using SSL channels





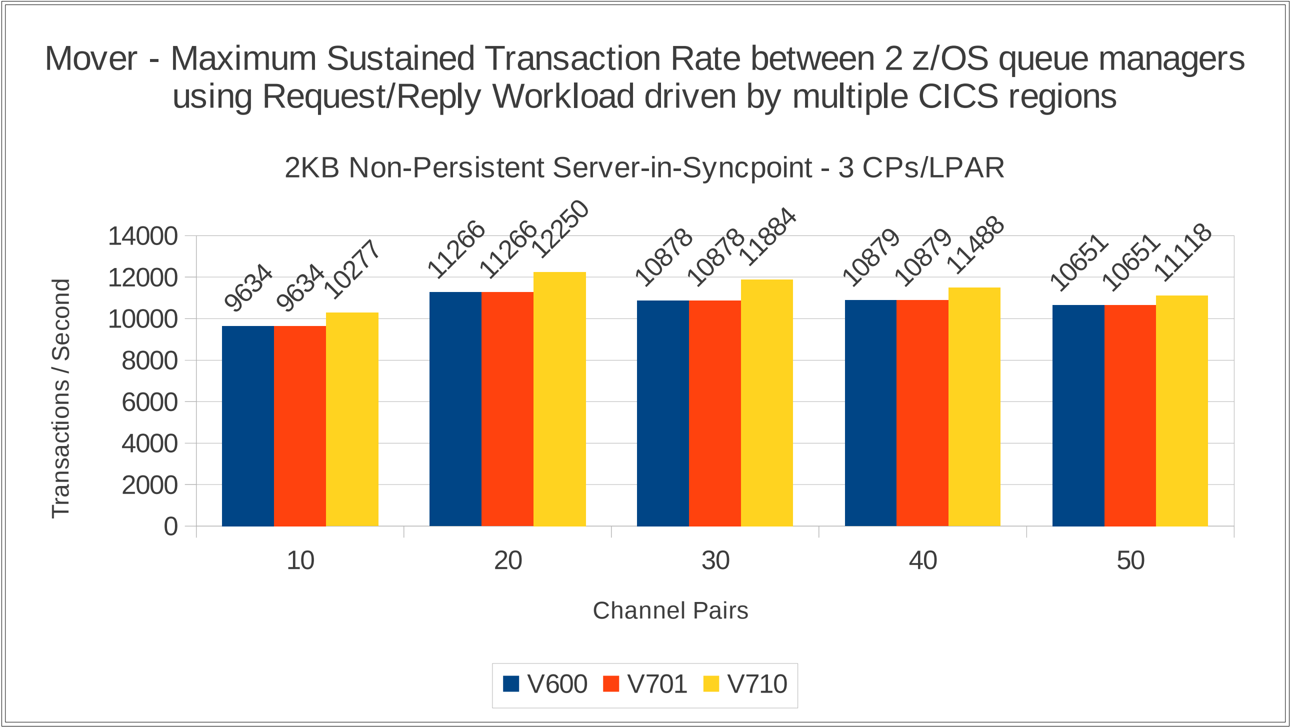
64KB messages using SSL channels

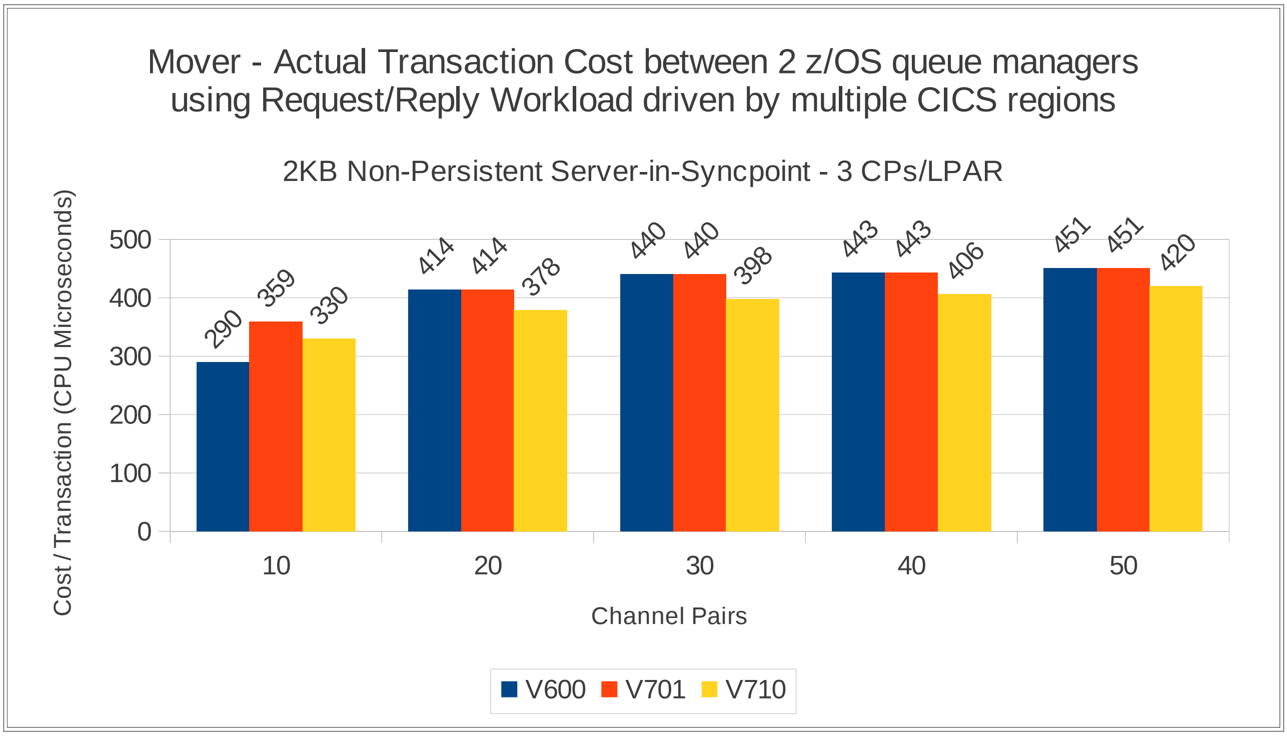




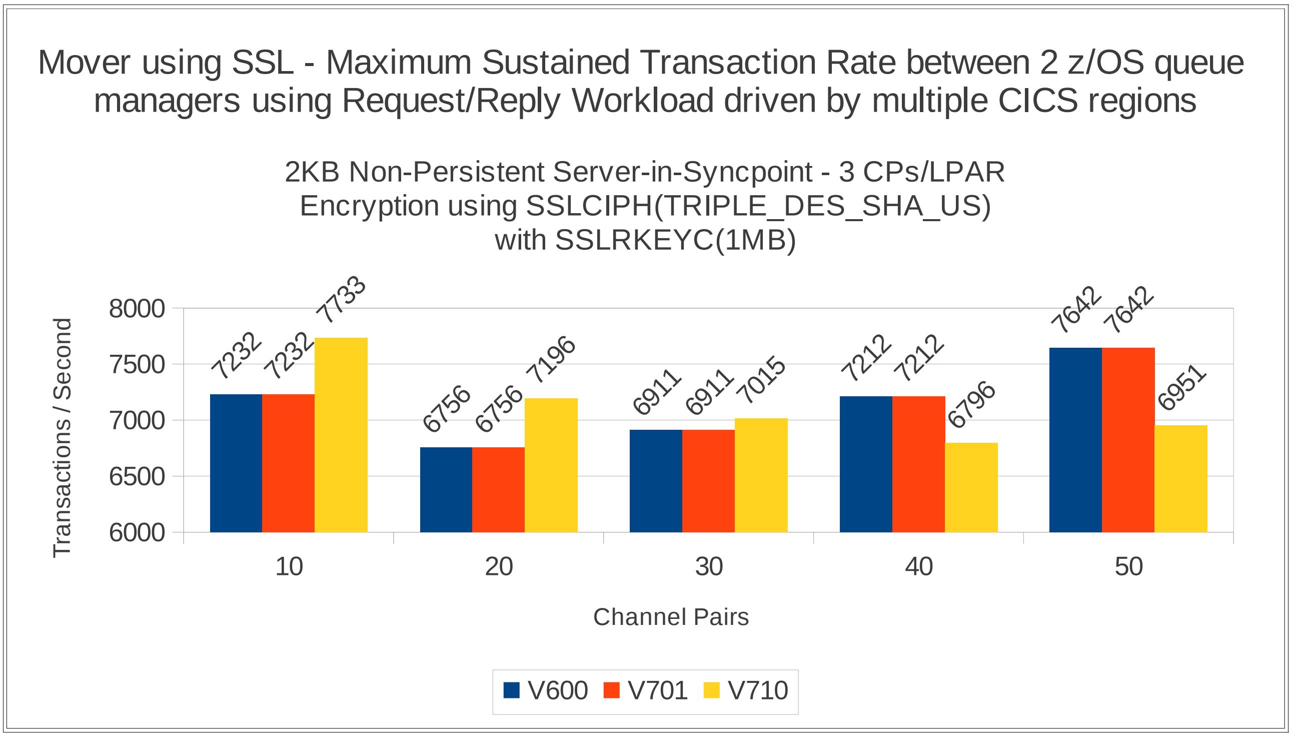
Non-persistent out-of-syncpoint – 10 to 50 sender-receiver channels

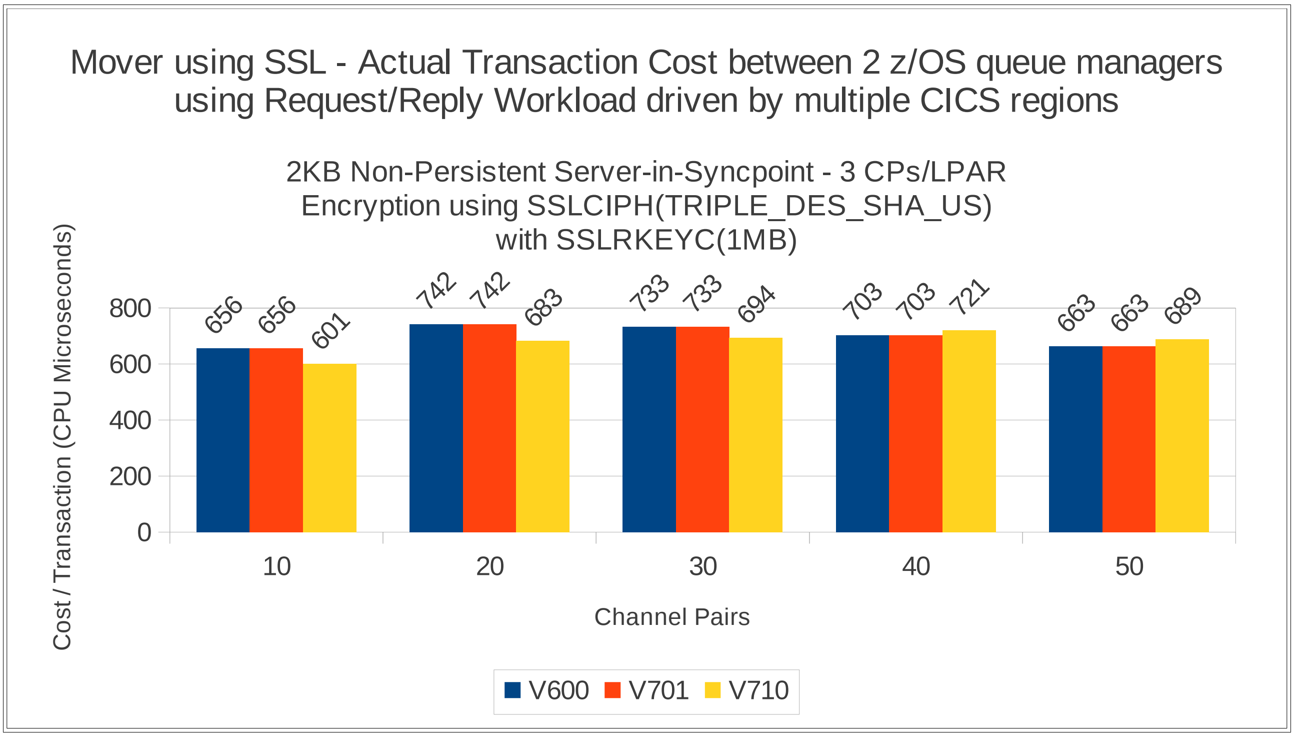
2KB messages





2KB messages using SSL channels





Regression – moving messages across cluster channels

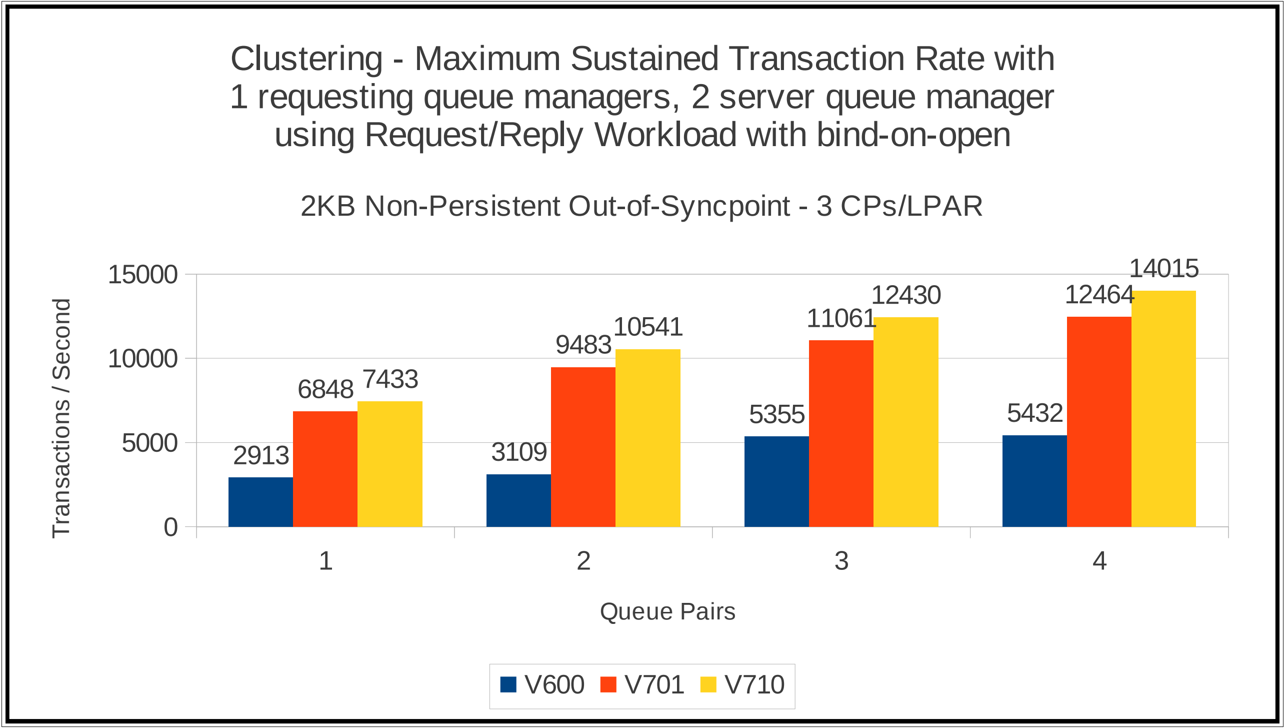
The regression tests for moving messages across cluster channels are relatively simple, providing multiple destinations for each message put.

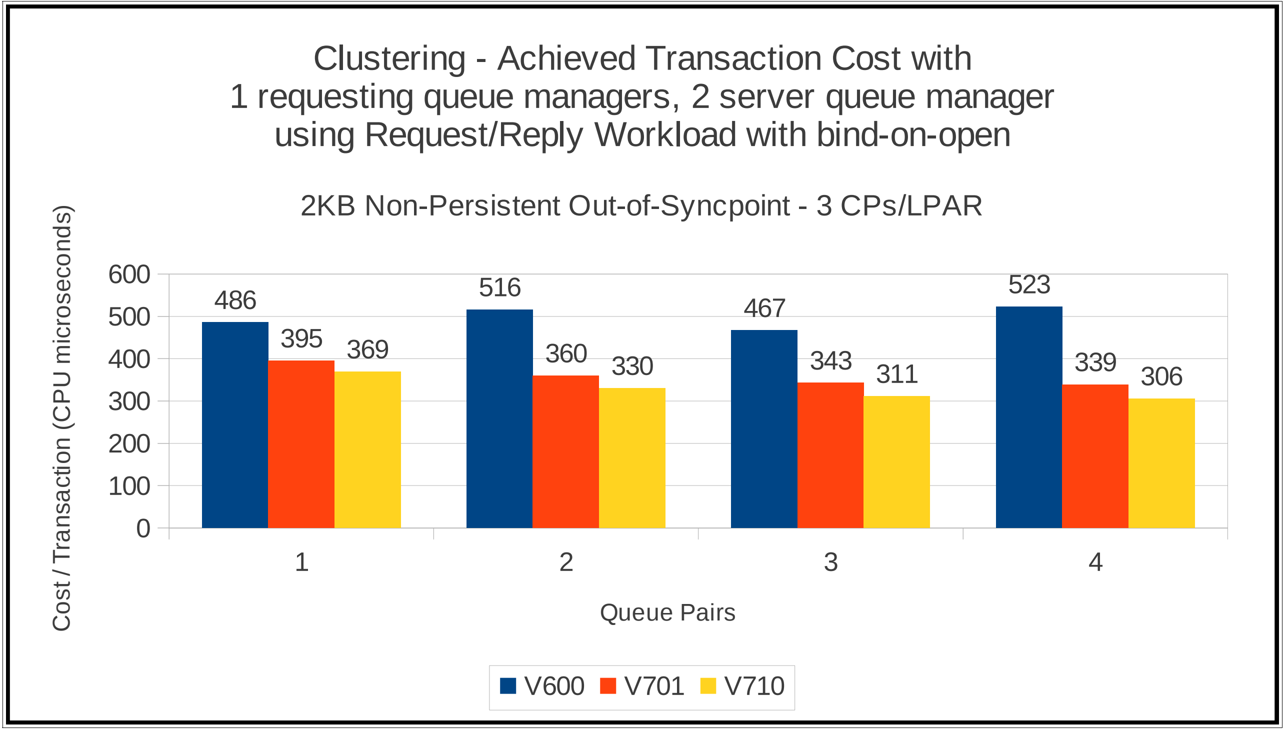
The cluster has 3 queue managers – one for the requester workload and two for the server workload. The queue managers hosting the server workload are both full repository queue managers.

A set of requester tasks is run against one queue manager and the application chooses either bind-on-open or bind-not-fixed. 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.

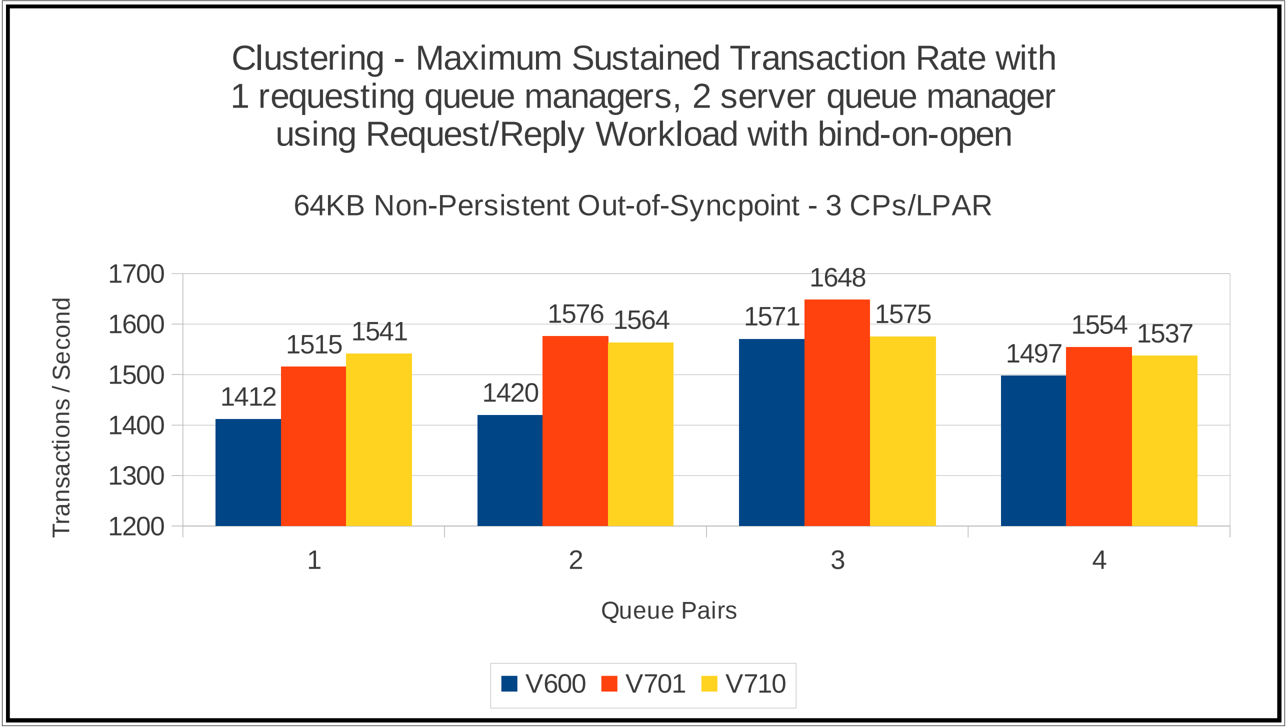
An increasing number of queues (with an corresponding increase in applications) are used.

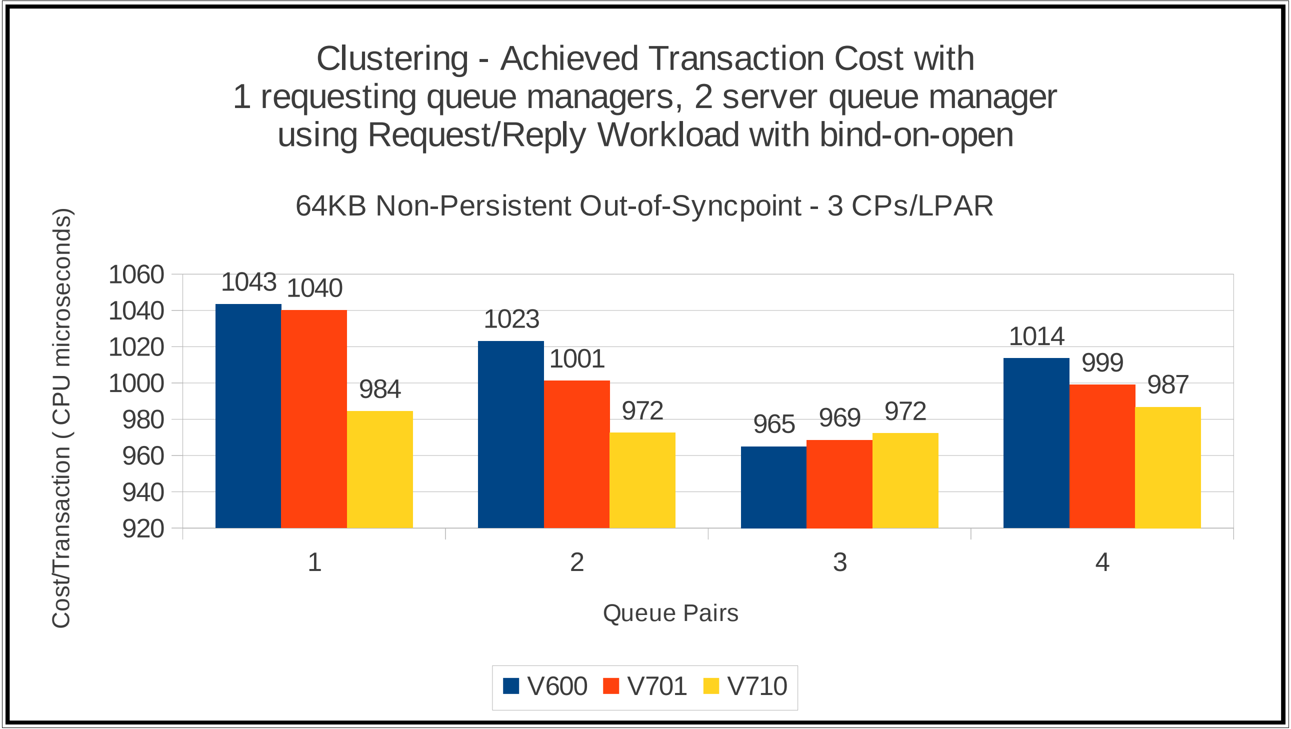
2KB messages – bind-on-open



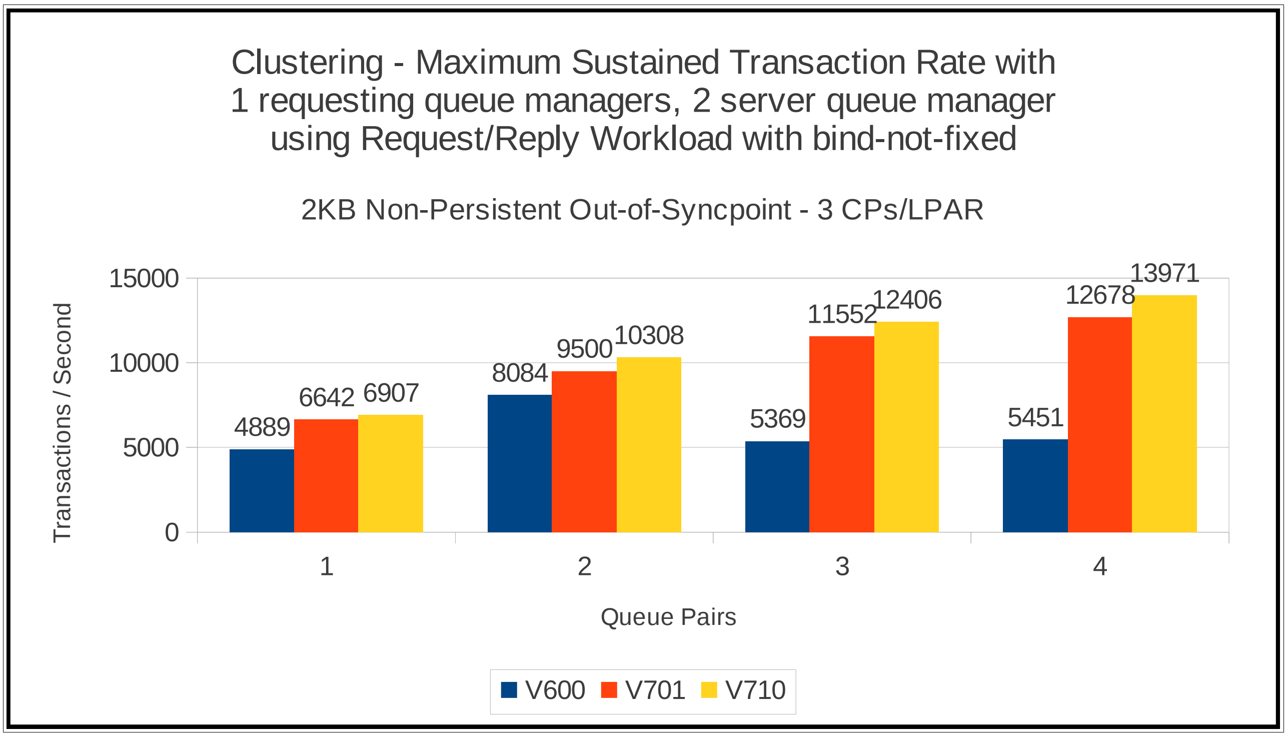


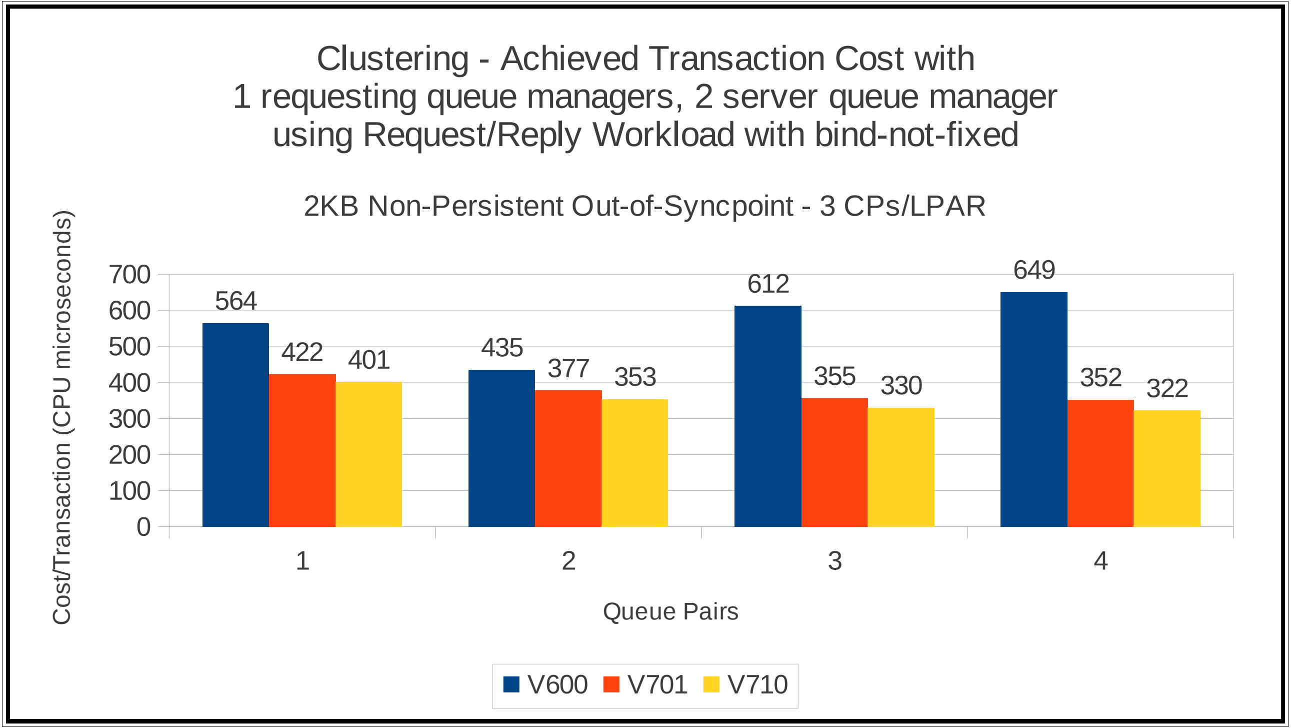
64KB messages – bind-on-open



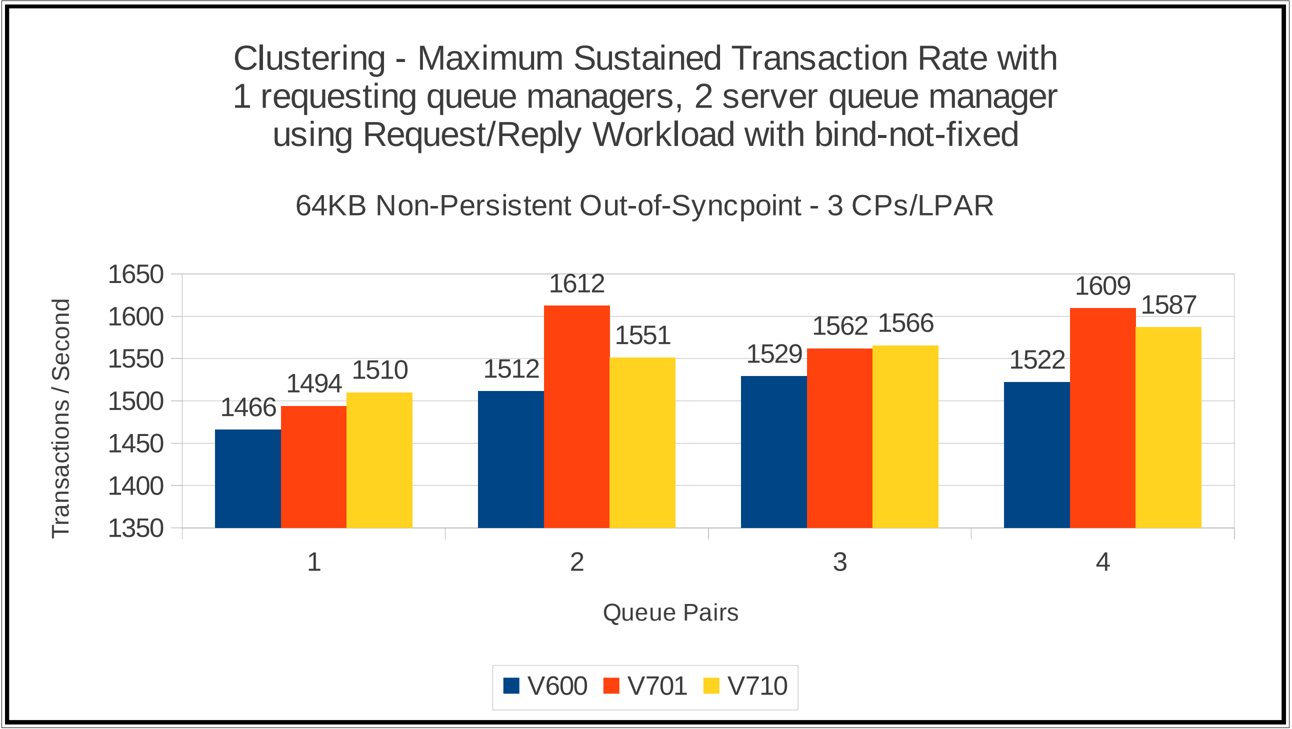


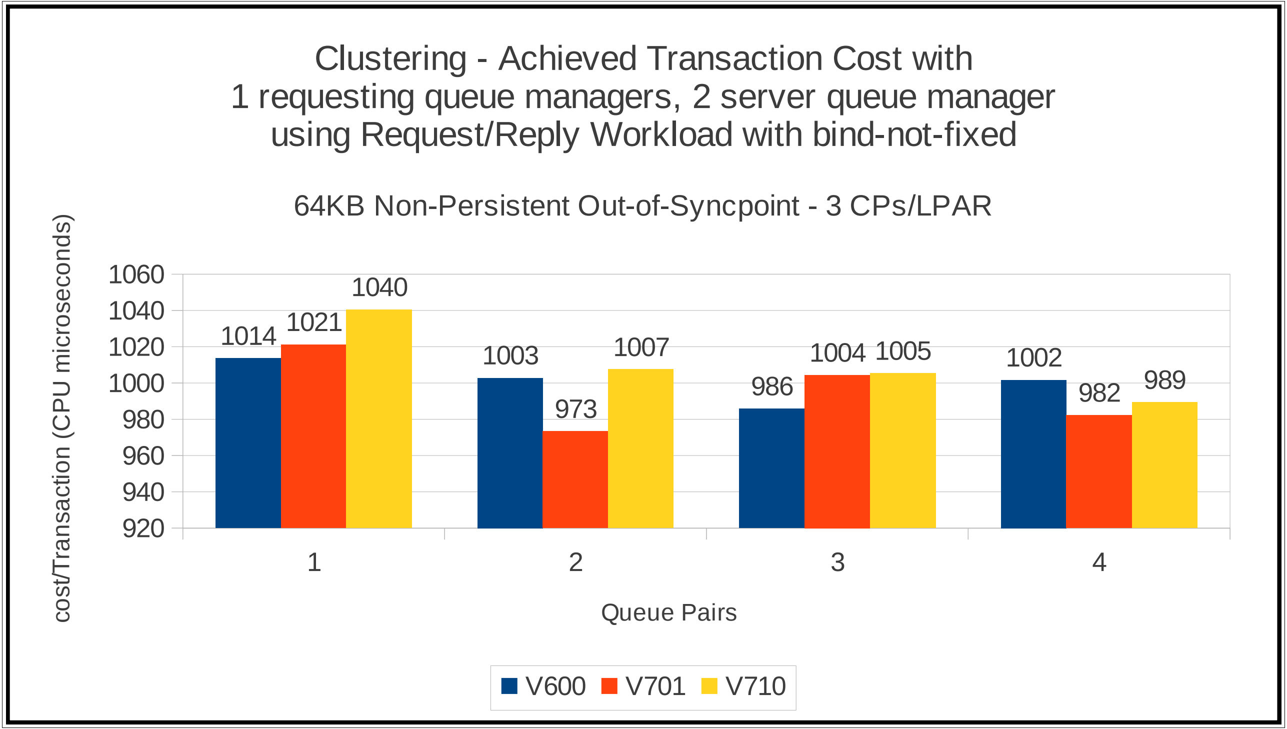
2KB messages – bind-not-fixed





64KB messages – bind-not-fixed





Regression – 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 is used, with a corresponding increase in the number of putting and getting clients.

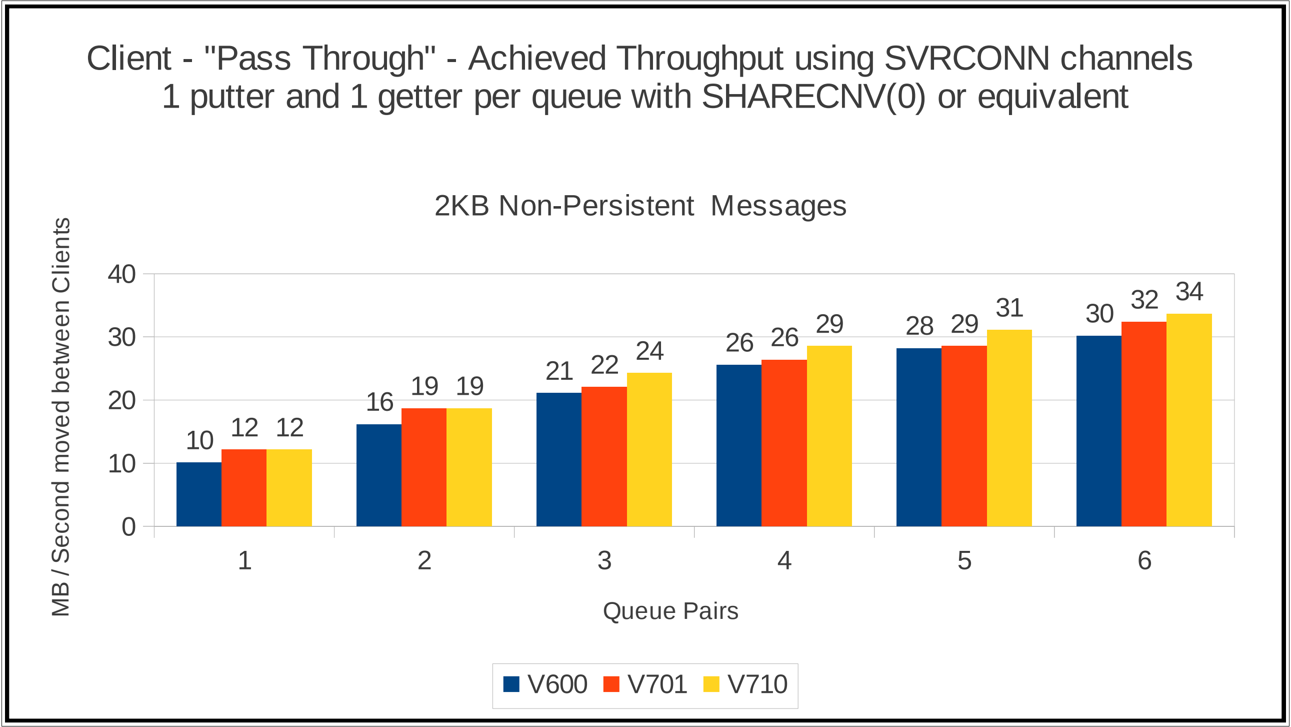
2 sets of tests are run – the first uses SHARECNV(0) on the SVRCONN channel to run in a mode comparable to that used on WebSphere MQ V6. The second uses SHARECNV(1) so that function such as asynchronous puts and asynchronous gets are used via the DEFPRESP(ASYNC) and DEFREADA(YES) queue options.

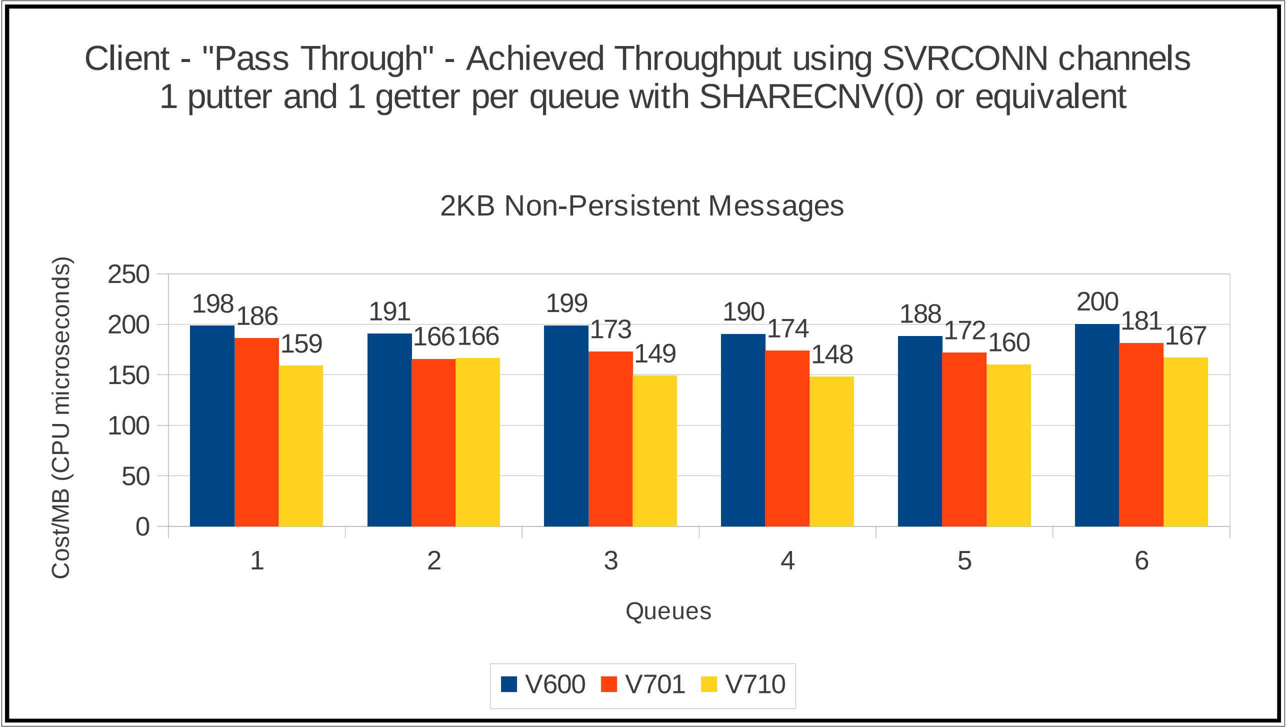
As WebSphere MQ for z/OS version 6 does not support the SHARECNV channel attribute, the SHARECNV(1) charts do not show values for V6.0.0.

Choosing which SHARECNV option is appropriate can make a difference to performance and is discussed in more detail in the supportpac MP16 – “Capacity Planning and Tuning Guide”.

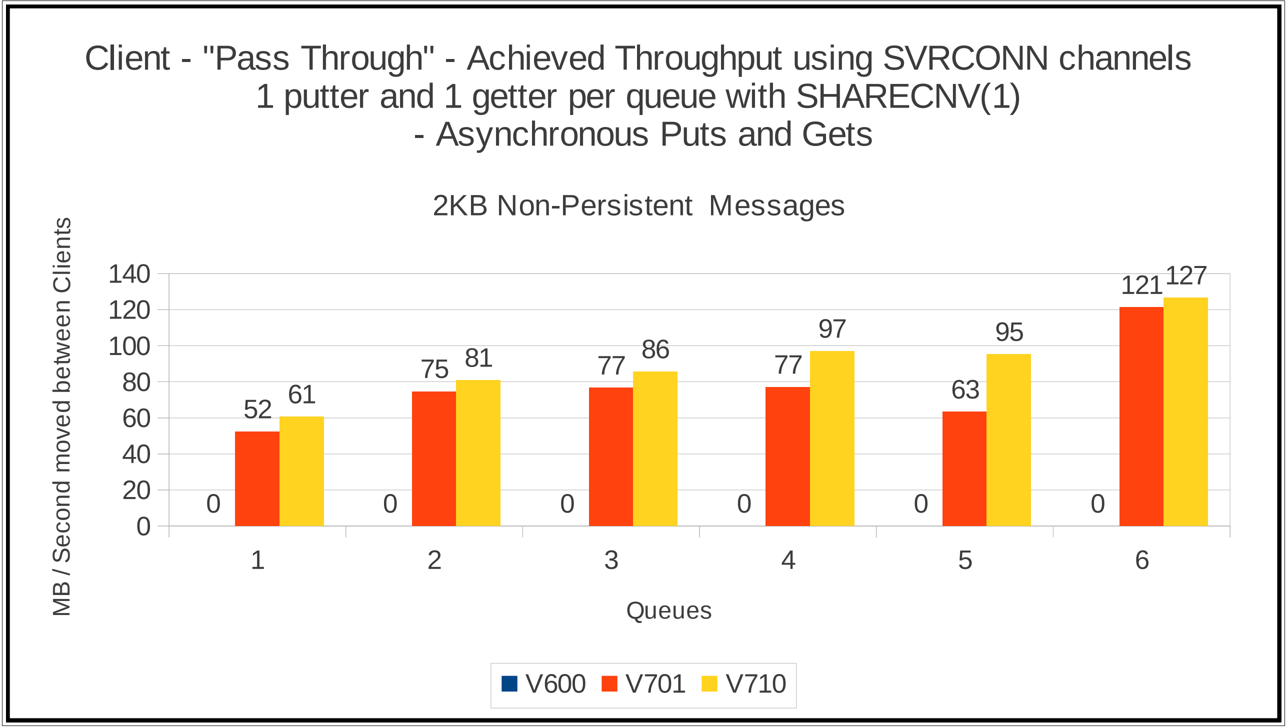
NOTE: The rate and costs are based on the number of MB of data moved per second rather than the number of messages per second.

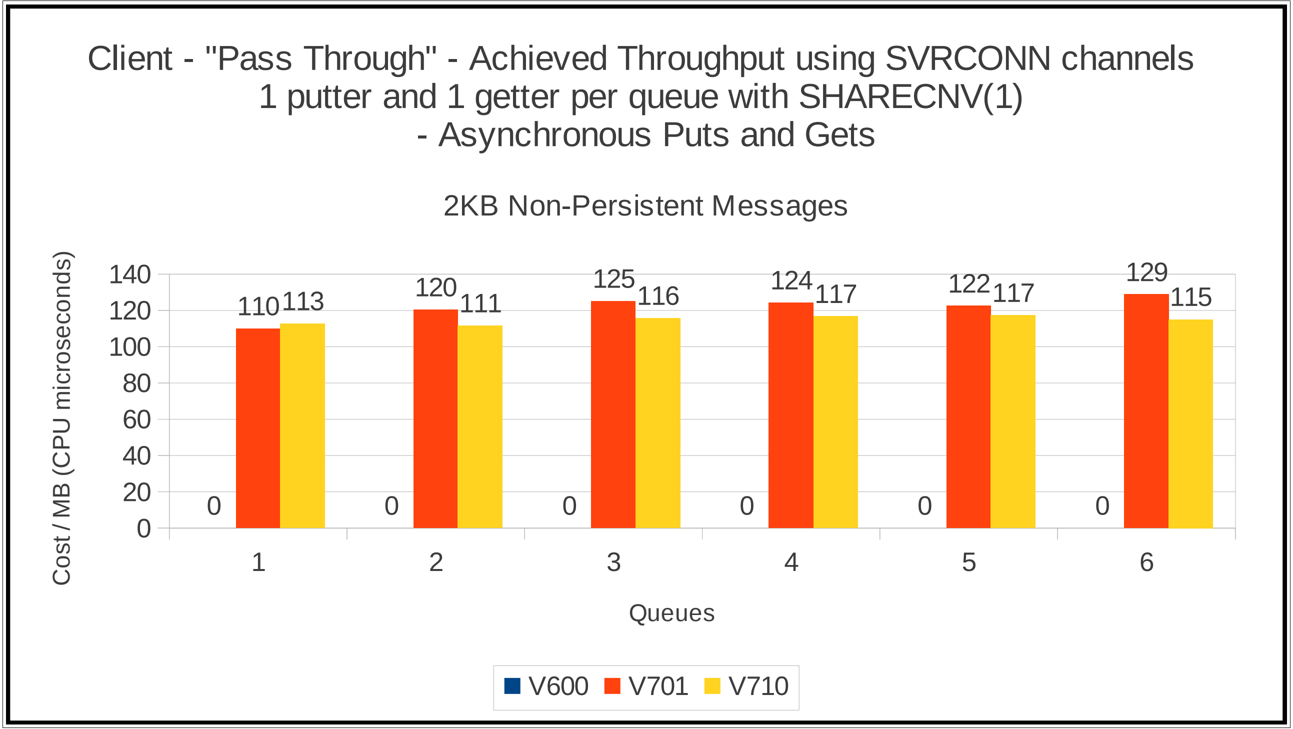
Client pass through tests using SHARECNV(0)





Client pass through tests using SHARECNV(1)





Regression – IMS bridge

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

The IMS region has been configured as detailed in the supportpac MP16 using the recommendations in the section “IMS Bridge: Achieving Best Throughput”.

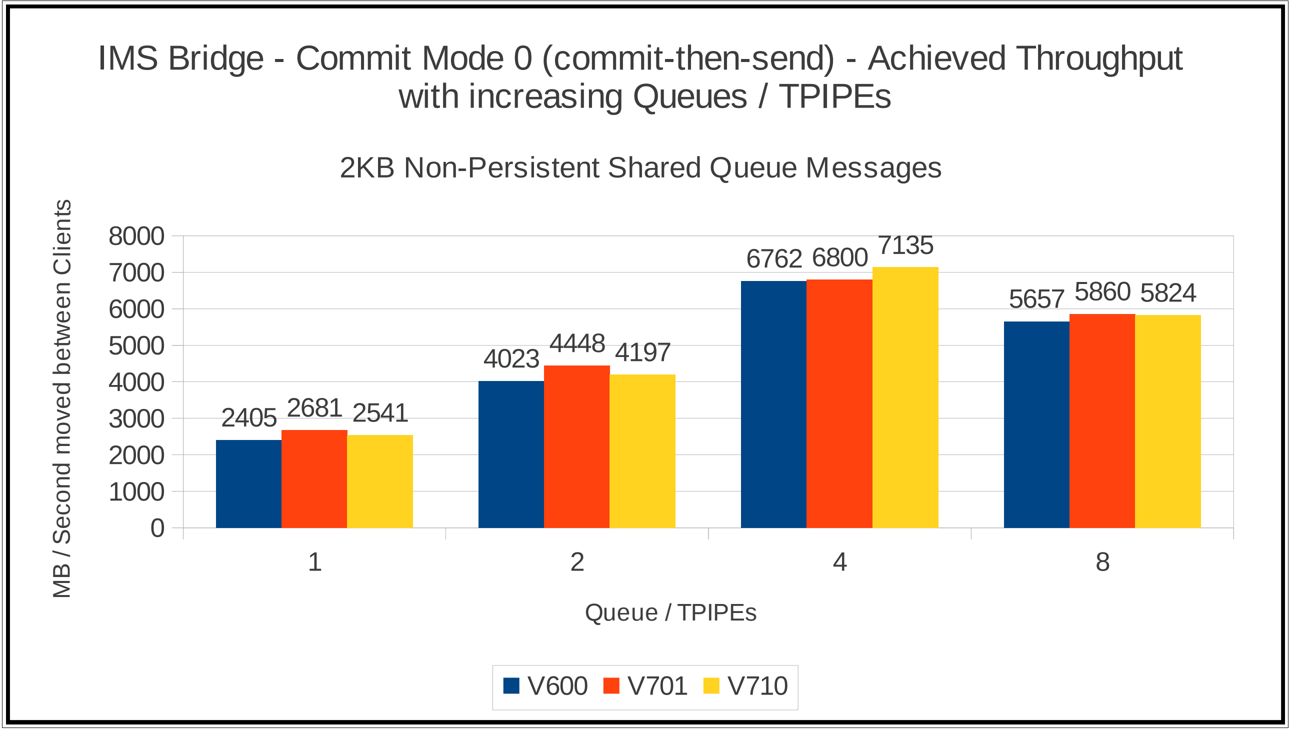
NOTE: 16 MPRs are more than really required for the 1, 2 and 4 TPIPE tests but they are available for a consistent configuration across the test suite.

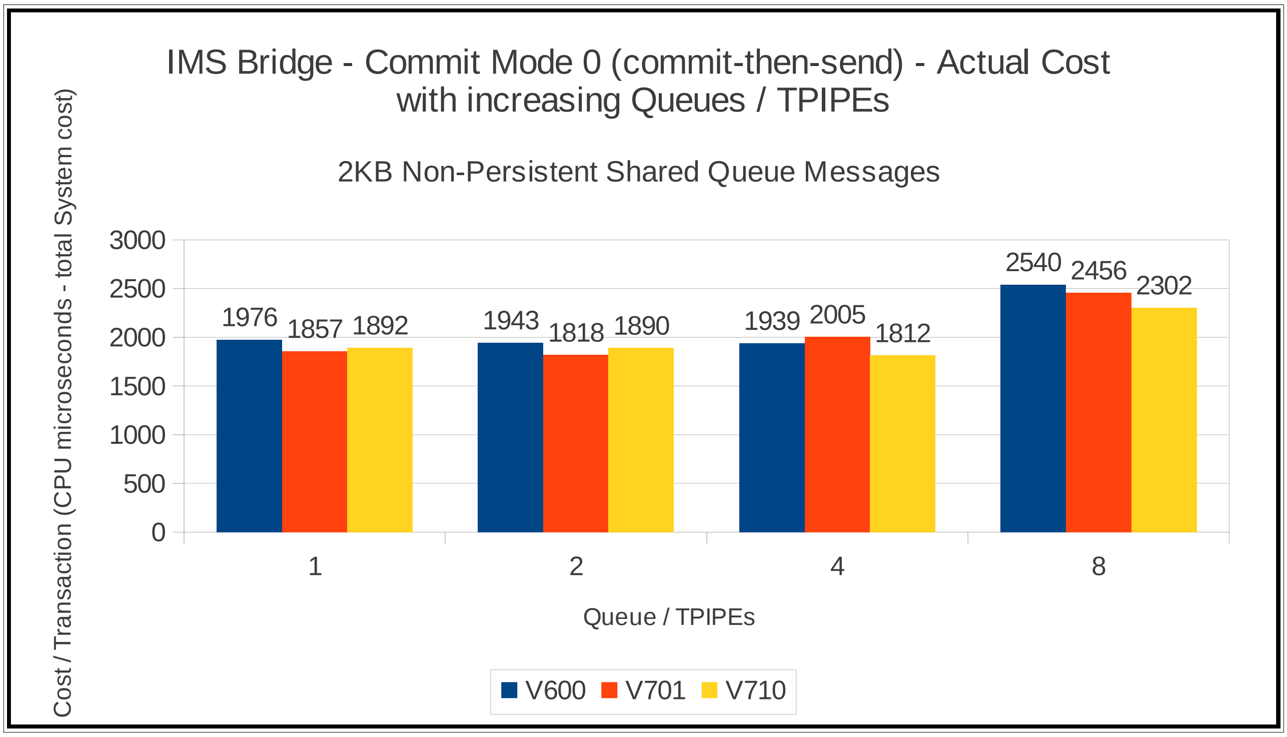
There are 8 queues defined in the QSG that are configured to be used as IMS Bridge queues.

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.

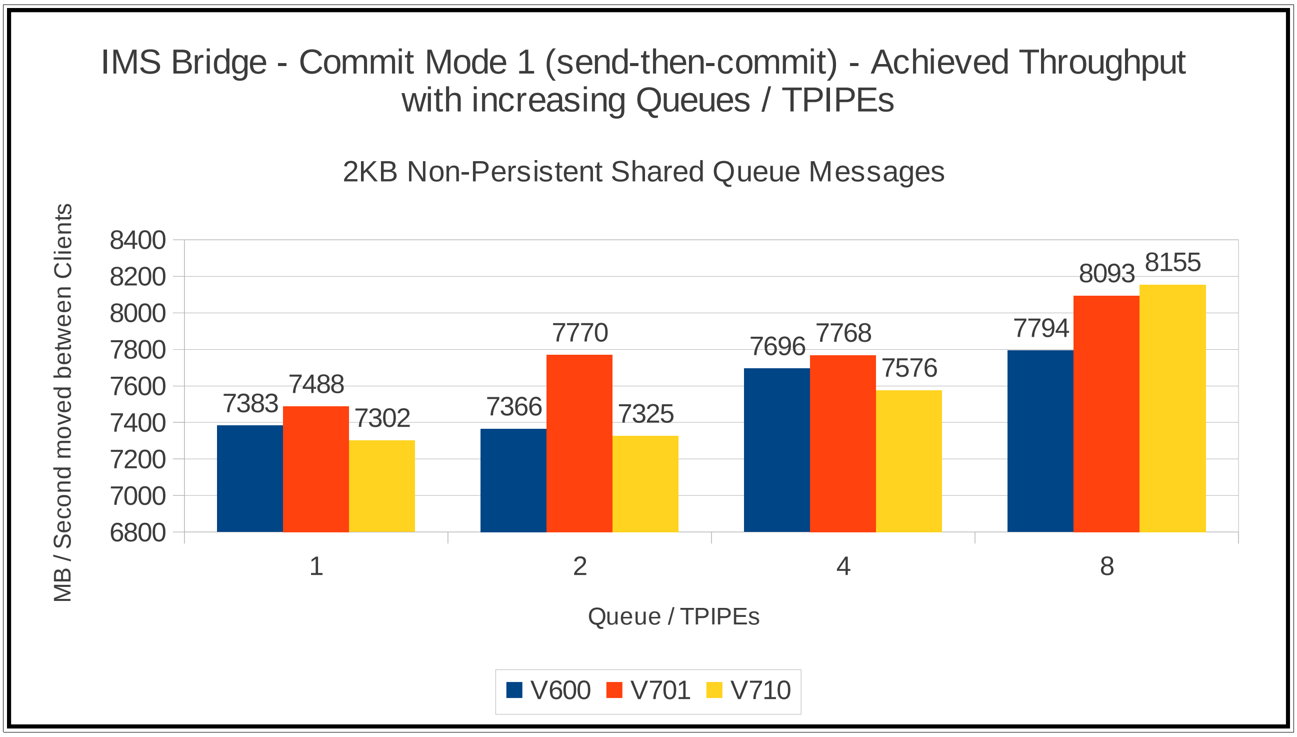
Tests are run using both Commit Mode 0 (Commit-Then-Send) and Commit Mode 1 (Send-Then-Commit)

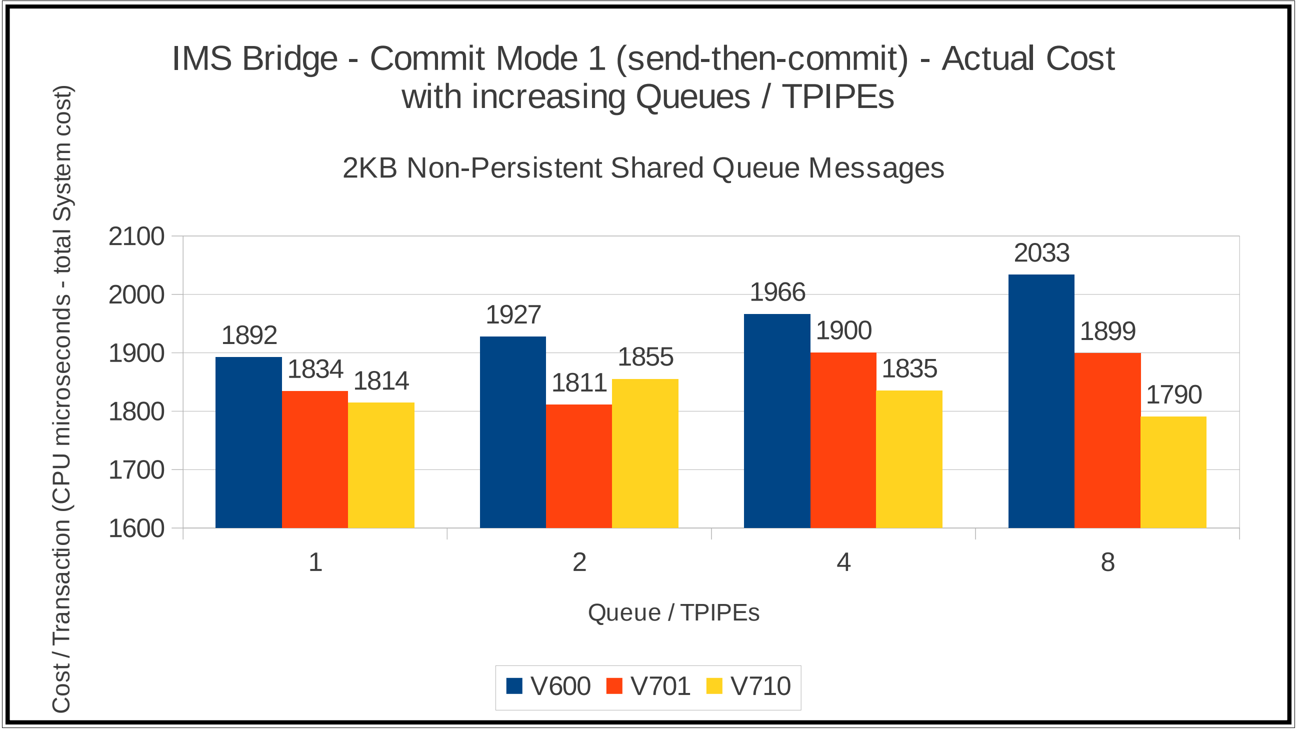
Commit mode 0 (commit-then-send)





Commit mode 1 (send-then-commit)





Regression – Trace

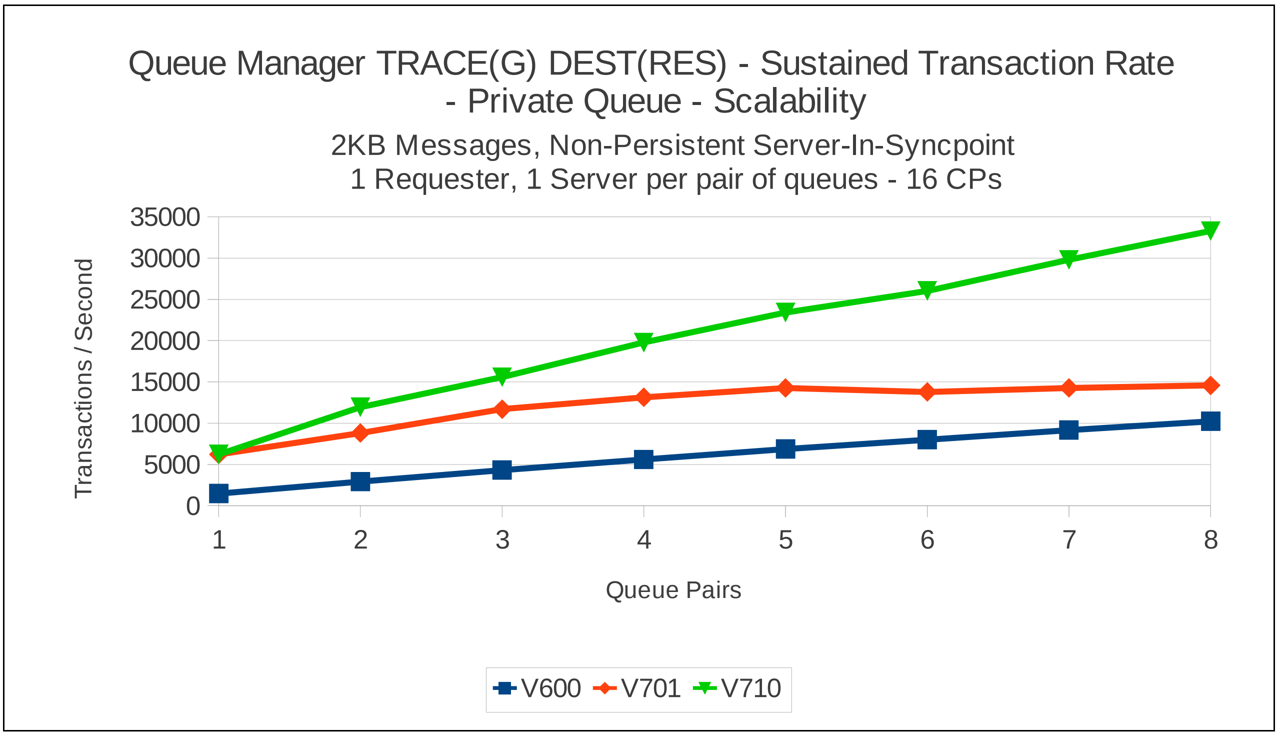
The regression tests for trace cover both queue manager global trace and the channel initiator trace.

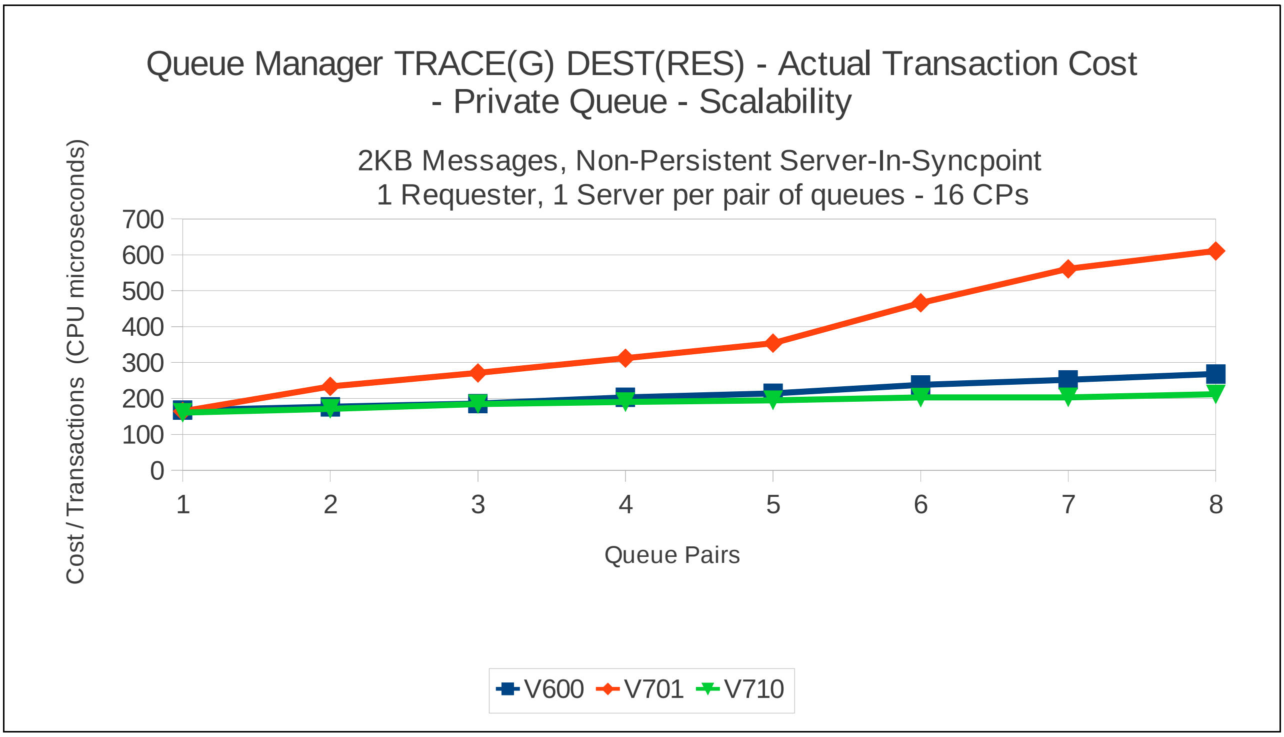
Queue manager global trace

The queue manager global trace tests are a variation on the private queue non-persistent 2KB scalability tests with TRACE(G) DEST(RES) enabled.

In V7.1.0, the global trace uses 64-bit storage which allows the queue manager to store a trace record for each thread and reduces contention on the trace storage. As a result the throughput achieved is significantly improved over previous releases of WebSphere MQ for z/OS.

It appears that the V6.0.0 costs are significantly lower than those observed in V7.0.1, but this is offset as the transaction rate achieved in V6.0.0 is significantly less due to the way the queue manager stored small messages and as a result the V600 workload had to be artificially constrained to prevent messages spilling into pageset usage. In turn, this meant that the V6.0.0 workload did not reach the level of workload where contention for the trace storage was observed.


Queue Manager TRACE(G) DEST(RES) - Sustained Transaction Rate - Private Queue - Scalability
2KB Messages, Non-Persistent Server-In-Syncpoint
1 Requester, 1 Server per pair of queues - 16 CPs
Transactions / Second
35000
30000
25000
20000
15000
10000
5000
0
1
2
3
4
5
6
7
8
Queue Pairs
V600
V701
V710

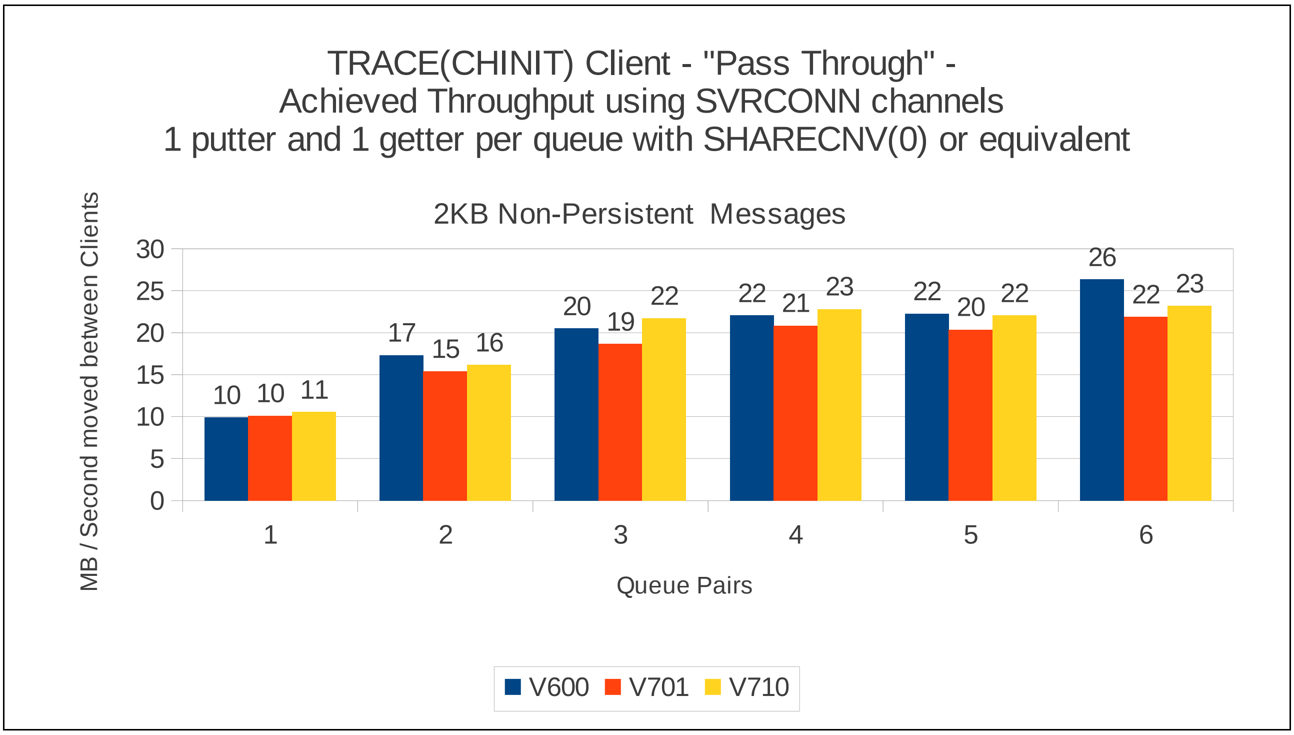

Queue Manager TRACE(G) DEST(RES) - Actual Transaction Cost - Private Queue - Scalability
2KB Messages, Non-Persistent Server-In-Syncpoint
1 Requester, 1 Server per pair of queues - 16 CPs
Cost / Transactions (CPU microseconds)
700
600
500
400
300
200
100
0
1
2
3
4
5
6
7
8
Queue Pairs
V600
V701
V710

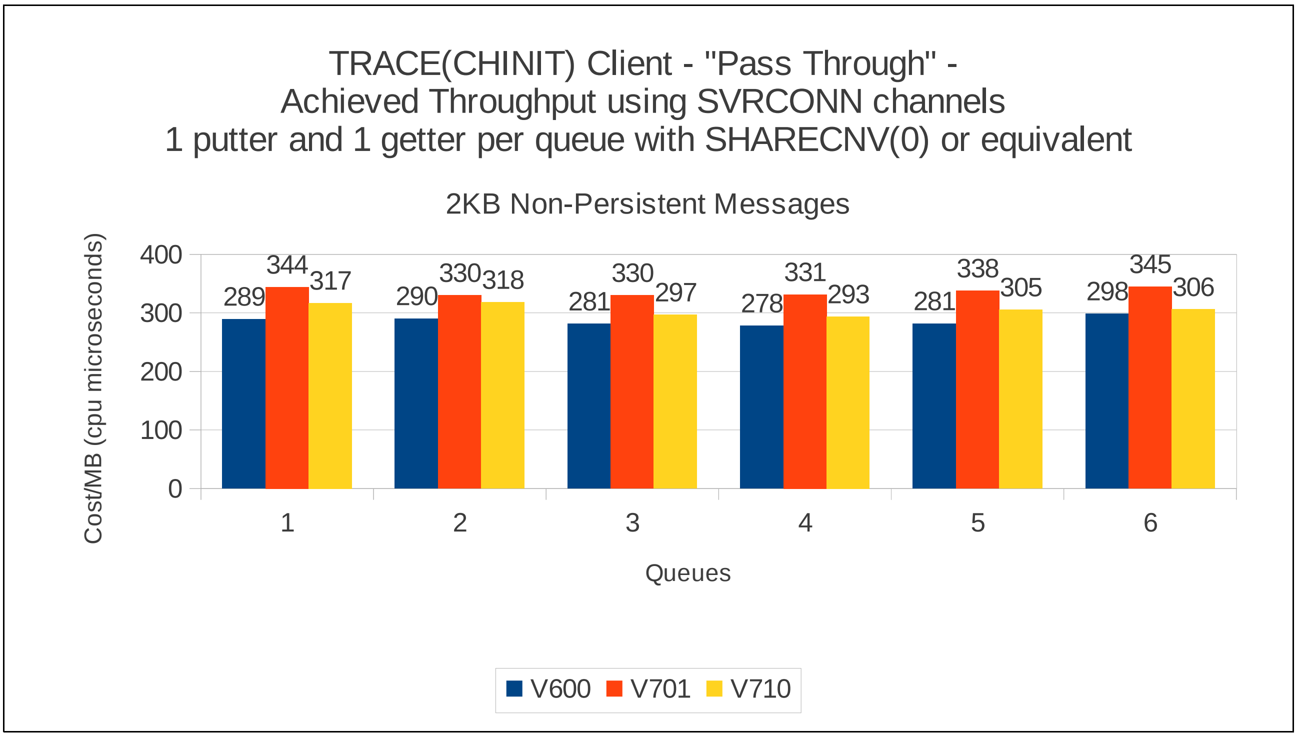
Channel initiator trace

The channel initiator trace tests are a variation on the moving messages across SVRCONN regression tests using channels with SHARECNV(0) and TRACE(CHINIT) enabled.

In V7.0.1 a significant increase in the number of trace points recorded was added, which had the effect of increasing the overhead of channel initiator trace over V6.0.0.

In V7.1.0, the trace routines are called more efficiently than in V7.0.1 but there are still more trace points being recorded than V6.0.0.


TRACE(CHINIT) Client - "Pass Through" -
Achieved Throughput using SVRCONN channels
1 putter and 1 getter per queue with SHARECNV(0) or equivalent
2KB Non-Persistent Messages
MB / Second moved between Clients
0
5
10
15
20
25
30
10 10 11
17 15 16
20 19 22
22 21 23
22 20 22
26 22 23
1
2
3
4
5
6
Queue Pairs
V600
V701
V710


TRACE(CHINIT) Client - "Pass Through" -
Achieved Throughput using SVRCONN channels
1 putter and 1 getter per queue with SHARECNV(0) or equivalent
2KB Non-Persistent Messages
Cost/MB (cpu microseconds)
0
100
200
300
400
289 344 317
290 330 318
281 330 297
278 331 293
281 338 305
298 345 306
1
2
3
4
5
6
Queues
V600
V701
V710

Appendix B – System configuration

MVS: zEnterprise 196 (2817-779) configured thus:

LPAR 1

Between 1 and 16 dedicated CP processors

LPAR 2

Between 1 and 3 dedicated CP processors

LPAR 3

Between 1 and 5 dedicated CP processors.

Default configuration:

3 dedicated processors on each LPAR

Coupling Facility

Internal coupling facility with 3 shared processors.
Priority weighted as 5 times shared LPAR processors (against non-performance sysplex's).
Dynamic Dispatching switched OFF

DASD
FICON-connected DS8800
4 dedicated channel paths (shared across sysplex)
HYPERPAV enabled.

System Settings:

Each MVS image running z/OS v1r12
Coupling facility running CFCC level 17.
ZHPF available, but by default is configured as disabled.
HIPERDISPATCH enabled by default.
Tests moving messages between LPARs that were on different TCP/IP subnets

Trace Status:

- TRACE(GLOBAL) disabled.
- TRACE(S) enabled
- TRACE(A) CLASS(3) enabled

General Information

- Client machine was:
  - IBM SYSTEM X3850 - 6 x 3.16GHz Processor, 6Gb Memory
- Client tests used a 1Gb performance network

- Other IBM products used:
  - CICS V6.6 CTS4.2
  - IMS v10
  - WebSphere MQ v6.0.0 with latest service applied as of May 2011.
  - WebSphere MQ v7.0.1 with latest service applied as of May 2011.