MQ for z/OS Performance Report

Use cases with z/OS Container Extensions (zCX)

December 2020

IBM MQ Performance

IBM UK Laboratories

Hursley Park

Winchester

Hampshire

SO21 2JN

Take Note!

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

First edition, December 2020. This edition applies to IBM MQ for z/OS and the use of zCX.

Notices

INTENDED AUDIENCE

This report is intended for Architects, Systems Programmers, Analysts and Programmers wanting to understand the performance characteristics of **IBM MQ for z/OS when used for specific MQ-related use cases with zCX**. The information is not intended as the specification of any programming interfaces that are provided by IBM MQ. Full descriptions of the IBM MQ facilities are available in the product publications. It is assumed that the reader is familiar with the concepts and operation of IBM MQ.

Summary of Amendments

Date	Changes
2020	Version 1.0 - Initial Version.

Table of contents

Chapter 1

Introduction to zCX

IBM z/OS Container Extensions (zCX) is a new z/OS 2.4 feature that enables clients to deploy zLinux applications in Docker containers on z/OS as part of a z/OS workload.

zCX brings many benefits, but is particularly useful if you need to run a small number of zLinux applications that need to connect into your existing z/OS applications as there is no need for a dedicated zLinux environment or IFLs. zCX Containers can run on zIIPs making them attractive from a pricing perspective.

Running zCX maintains operational control of the Linux environment with z/OS, bringing z/OS qualities of service to the application deployment and does not require the provisioning of separate LPARs or system images.

IBM zCX allows you to bring “Dockerized” Linux applications as close as possible to your enterprise z/OS resources without first having to port lots of open source infrastructure to z/OS.

Chapter 2

Use cases for zCX with IBM MQ

There are a number of MQ specific use cases for zCX, some of which were discussed in the blog "Use cases for MQ in z/OS Container Extensions".

This document will discuss those use cases plus additional ones that have been made possible since IBM MQ Advanced for z/OS VUE version 9.2 was released.

This document is an accumulation of a number of performance reports, blogs as well as investigations into the performance of MQ both with and without zCX. A list of all of the resources that have had provided input to this document is available in the Resources section.

The main use cases where MQ in zCX adds value are:

- As a client concentrator. The blog "Using client concentrators with MQ for z/OS" discusses the use of client concentrators and how they may benefit MQ users on z/OS.
- As a cluster full repository. This is mainly of benefit when the cluster resides fully on z/OS and the cluster repository is growing such that it will be constrained by the z/OS repositories' 2GB limitation.
- As a host system for a fasp.io gateway - where queue managers are located across high latency networks.

Each of these use cases is discussed in further sections.

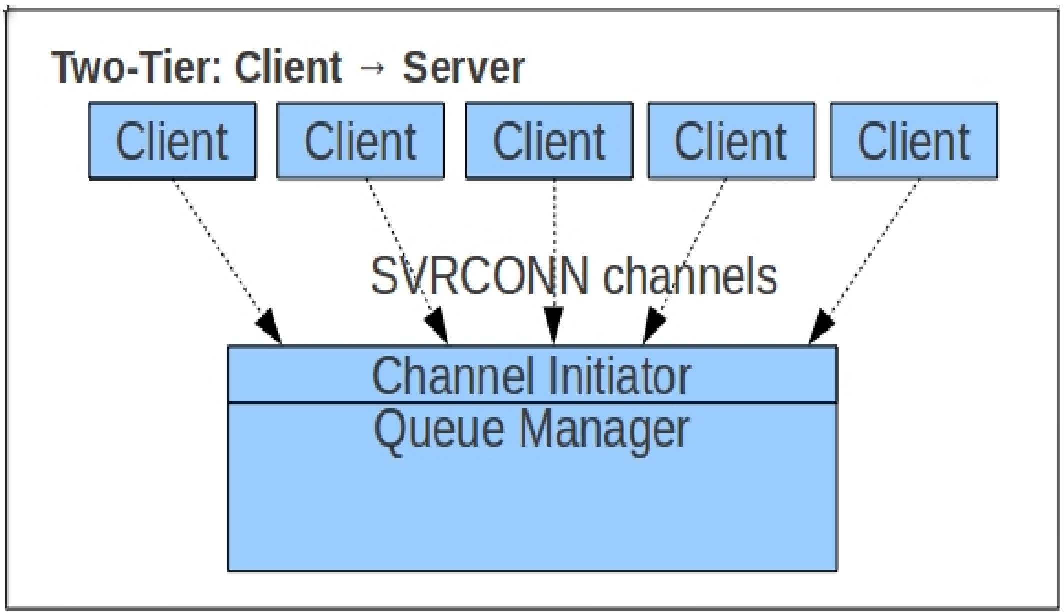
Chapter 3

Client concentrators with MQ for z/OS

Traditionally most applications that used MQ for z/OS ran on z/OS and connected directly to queue managers running on the same LPAR using shared memory - this is known as bindings mode.

For a long time now, more and more applications running on distributed platforms need to connect into MQ for z/OS. Recently that trend has accelerated as many customers want to be able to exploit the unparalleled resilience that MQ for z/OS queue sharing groups provide.

In order to connect an application running on distributed to an MQ for z/OS queue manager, you need to connect over the network, and you would use a server-conn channel to do this. This is shown in the following diagram:



As more applications connect into a queue manager, the queue manager must do more work on their behalf, and therefore uses more CPU. A particularly expensive operation is the initial connect to the queue manager (MQCONN), which may involve a TLS handshake, plus the security checks performed by the queue manager as well as other processing.

If you have a large number of applications that connect to a queue manager, send or receive one message and then disconnect, this can result in very high CPU usage patterns. Similarly applications that poll the queue manager can result in high CPU usage.

z/OS customers are typically interested in how much CPU is being used and how they may reduce the cost, and in part this may be addressed by implementing the appropriate application design patterns, but sometimes this is not enough, which is where client concentrators may provide benefit.

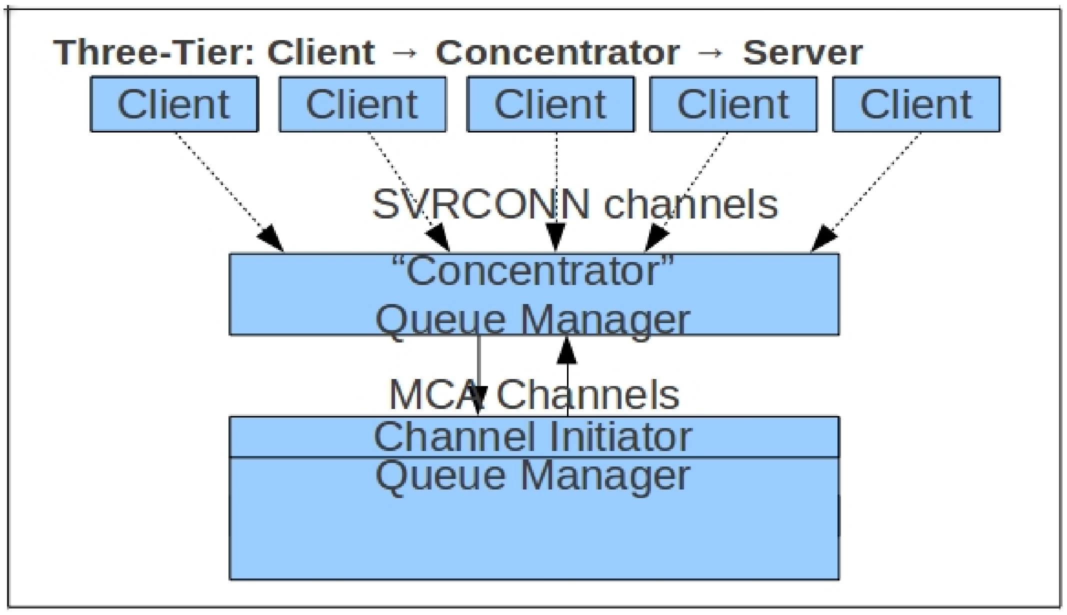
The blog "Cost of connecting to a z/OS queue manager" provides some insight into the cost of the MQCONN into a z/OS queue manager on an IBM z14.

What is a client concentrator?

A client concentrator is a distributed queue manager that acts as a proxy for the z/OS queue manager. The application connects to the distributed queue manager over a server-conn channel and sends and receives messages. The distributed queue manager moves these messages to and from the z/OS queue manager over a small number of sender/receiver channels.

In this way, the distributed queue manager *concentrates* a large number of server-conn channels into a small number of sender/receiver channels, hence client concentrator. Sender/receiver channels are typically long-lived and as such are typically cheaper than using server-conn channels.

This configuration is shown in the following diagram:

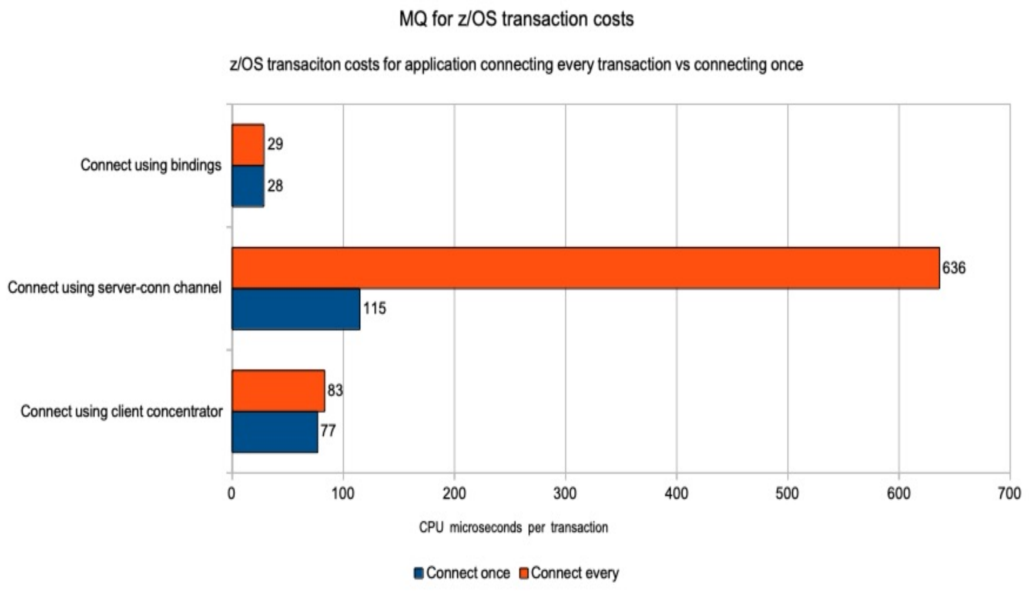


Does a concentrator really make a difference?

Using a client concentrator can result in a significant CPU reduction - but it does depend on the applications usage pattern.

In the following sample chart we compare the cost on z/OS of a “badly” behaved application that is invoked multiple times and on each invocation runs MQCONN, MQPUT, MQGET and MQDISC, against an application that remains connected for multiple MQPUT and MQGET calls. In this example, the application performs 600,000 MQPUTs and MQGETs in each configuration.

For the purposes of this measurement, the client concentrator is hosted on a non-z/OS platform.



In this example. the badly behaved client application that connects to z/OS before each MQPUT costs more than 5 times the well behaved application, when connecting using server-conn channels.

There are circumstances when the client application cannot remain connected for multiple MQPUTs or MQGETs, and in those situations, the cost of using a server conn channel is 8 times that of connecting via a client concentrator.

Why choose one configuration over the other?

There are both performance and operational reasons why one configuration may be most appropriate for any particular customer and the following section aims to highlight some of those considerations.

From the earlier charts, it is easy to assume that you should always use a client concentrator to connect applications to a z/OS queue manager. However those charts only focus on the CPU usage of the z/OS queue manager and there are many other factors to consider:

- Complexity: The client concentrator model is more complicated, simply as a result of increasing the number of queue managers in the operating model. Since they are run on different platforms, they may be managed by different teams.
- Latency: client concentrators add latency. If the application is sending a message and waiting for a response, each message now has two network hops, and must be processed by two queue managers. Messages sent over sender-receiver channels are often batched up, which can add extra latency. If the messages are persistent, they must be logged on each queue manager.
- High availability: having a single client concentrator queue manager is a single point of failure. This is becoming less and less acceptable in today's world. There are many options for making distributed MQ highly available but all options add complexity. Considering that many distributed applications make use of MQ for z/OS in order to take advantage of queue sharing groups, inserting a less available client concentrator may not be considered an optimal solution!
- Security: the client concentrator model is secure, but requires more effort to set up. It is advisable to use TLS on all channels and perform the necessary certificate management. However, this effort tends to be less than required for the security configuration of the server-conn channels.
- Cost: the client concentrator model requires one or more distributed MQ licences plus the need for somewhere to run the distributed queue managers.
- Number of connecting applications: each application connected will use some of the available storage, and the channel initiator has an absolute limit of 9,999 channels. This limit of 9,999 channels may not be achievable depending on a number of factors, and the performance report for "IBM MQ Advanced for z/OS VUE version 9.2" discusses how to determine the number of channels your system may support.

Where should the client concentrator be located?

Any distributed queue manager can be a client concentrator, however there is value in having dedicated client concentrator queue managers and locating them as close to the z/OS queue managers as possible, which will minimise network latency.

The common choice has been to put the client concentrator on Linux on Z, ideally on the same physical hardware as the MQ for z/OS queue manager.

Of course, not every z/OS customer has a Linux on Z installation, and this is where zCX can offer benefit. zCX allows you to run any Linux on Z application, such as distributed MQ, in a container inside a z/OS LPAR. Critically for the client concentrator scenario, the zCX container is in the same LPAR as the MQ for z/OS queue manager, which allows these two related components to run closely together and be managed by the same team, whilst benefitting from the fast in-memory network.

Configuring a concentrator on zCX

For the client concentrator tests, the system used a pre-built MQ docker image. As the measurements were run before fix UJ3302 was available, the MQ level used was 9.1.3, which still used the mqm group.

The Dockerfile used to configure the concentrator queue manager contained the following:

```
# Install MQ and define the mqperf user so the clients can connect
# Note version of MQ. All later MQ docker images require vector instruction support as per UJ3302.
# Clients are run under mqperf user id.
# Defining MQ objects into file concentrator.mqsc and then copying to /etc/mqm
FROM ibmcom/mq:9.1.3.0
USER root
RUN useradd mqperf -G mqm && \
echo mqperf:passw0rd | chpasswd RUN pwd
WORKDIR /tmp
RUN echo 'ALT QMGR CHLAUTH(DISABLED)' > concentrator.mqsc \
 && echo "ALT QMGR CONNAUTH(' ')" >> concentrator.mqsc \
 && echo "REFRESH SECURITY TYPE(CONNAUTH)" >> concentrator.mqsc \
 && echo 'DEFINE CHL(CSIM_CHANNEL_TCP) +' >> concentrator.mqsc \
 && echo ' CHLTYPE(SVRCONN) TRPTYPE(TCP) MAXMSGL(104857600)' >> concentrator.mqsc \
 && echo 'DEF QL(VTS1) SHARE DEFSOPT(SHARED) USAGE(XMITQ) MAXDEPTH(99999) REPLACE' >> concentrator.mqsc \
 && echo 'DEF QL(Q_CSIM_REPLY_0) SHARE DEFSOPT(SHARED) MAXDEPTH(99999) REPLACE' >> concentrator.mqsc \
 && echo 'DEF QR(Q_CSIM_IN_0) RNAME(Q_CSIM_IN_0) XMITQ(VTS1) RQMNAME(VTS1) REPLACE' >> concentrator.mqsc \
 && echo 'DEF CHL(VTS1_VTS2) CHLTYPE(RCVR) TRPTYPE(TCP) REPLACE' >> concentrator.mqsc \
 && echo 'DEF CHL(VTS2_VTS1) CHLTYPE(SDR) +' >> concentrator.mqsc \
 && echo " CONNAME('9.20.37.21(2201)') + " >> concentrator.mqsc \
 && echo " XMITQ(VTS1) +" >> concentrator.mqsc \
 && echo ' TRPTYPE(TCP) BATCHSZ(50) NPMSPEED(FAST) BATCHLIM(0) REPLACE ' >> concentrator.mqsc \
 && echo 'STA CHL(VTS2_VTS1)' >> concentrator.mqsc \
 && cp concentrator.mqsc /etc/mqm/
USER mqm
```

The docker image was created using the following command:

```
docker build -tag mqconcentrator
```

The concentrator queue manager was then defined and created using the following commands:

```
docker volume create mqdatavts2

docker run --env LICENSE=accept \
           --env MQ_QMGR_NAME=VTS2 \
           --env AMQ_EXTRA_QM_STANZAS=Channels:MaxChannels=20000 \
           --publish 2202:1414 \
           --publish 9202:9443 \
           --detach \
           --volume mqdatavts2:/mnt/mqm \
           mqconcentrator
```

Note that the `--env AMQ_EXTRA_QM_STANZAS=Channels:MaxChannels=20000` statement allows the concentrator queue manager to accept twice the number of MQ channels that a z/OS queue manager supports.

Concentrator performance measurements

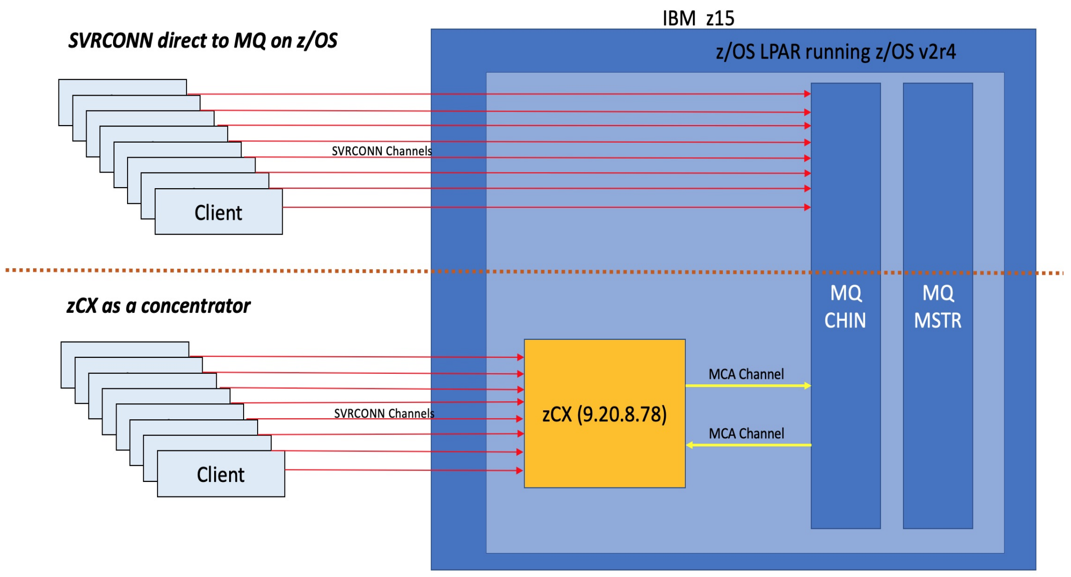
The concentrator performance measurements ask 3 questions:

1. How does the performance differ from clients connecting directly to the z/OS queue manager?
2. Is the zCX concentrator queue manager limited to 9,999 channels?
3. How does message persistence affect the performance?

zCX performance test topology

In the tests in this section, the following test topology was used:

Chart: Topology comparing server-conn to z/OS with server-conn to zCX Concentrator



In the top of the diagram, the clients are connecting directly to the channel initiator address space.

In the bottom of the diagram, zCX acts as a concentrator for the client requests, and forwards those requests to the channel initiator using sender-receiver channels.

How does the performance differ from clients connecting directly to the z/OS queue manager?

In order to answer the first question, we configured the test to connect 8000 clients to the z/OS queue manager. Each of these clients then put and got a 4KB non-persistent message every second, whilst remaining connected for the life-time of the test. These messages were then processed by batch server tasks running on z/OS which put the response message to a common reply queue - and this reply message was then got by the client application.

The test was then repeated with the clients connecting to the concentrator queue manager which would then forward the messages to the z/OS queue manager and receive the response messages.

In particular we are interested in a number of factors:

- How quickly did the clients connect?
- How much does it cost to connect and run the workloads?
- Is the concentrator able to sustain the workload?

As the workload rate is fixed, we know that all clients are connected when the servers are processing 16,000 messages per seconds (8,000 MQGET and 8,000 MQPUT).

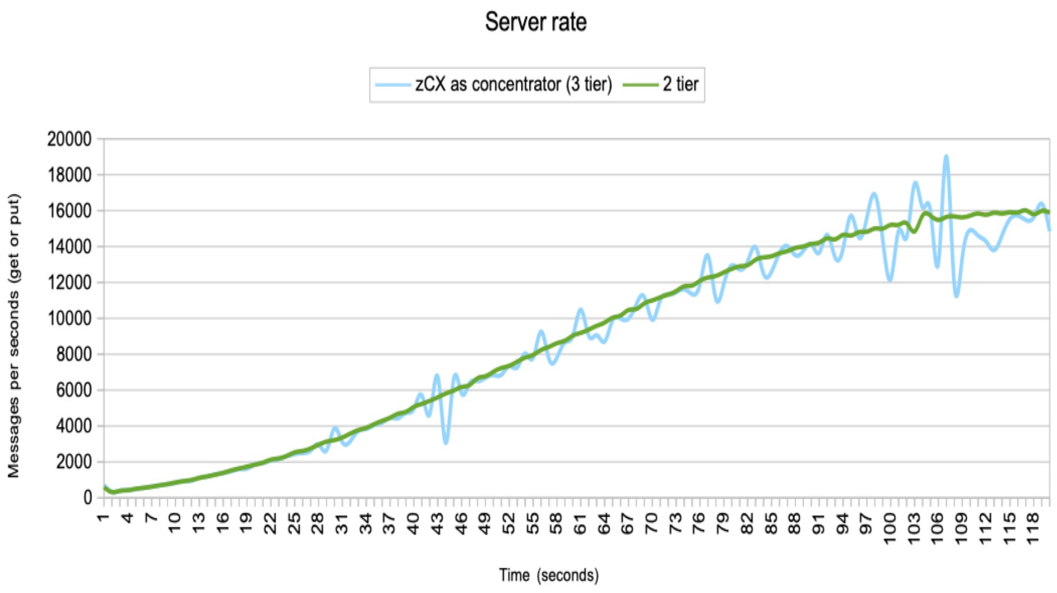
Alternatively we can use the following options:

- For the 2-tier configuration, we can also determine the number of channels using the "`DIS DQM`" command.
- For the 3-tier client concentrator configuration, we can verify the number of channels running using:

```
docker exec -it mqconcentrator bash

echo "DISPLAY CHS(*)" | runmqsc VTS2 | grep ’AMQ8417’ | wc -l
```

The following chart shows the messaging rate for the first 2 minutes of the measurement:



The chart shows a similar increase in messaging rate for both configurations - although the measurement where the clients connect directly is far more smooth than when the messages are connected through the concentrator queue manager on zCX.

In both cases, all 8000 clients were connected within 2 minutes of the workload starting and that rate is likely limited by the test infrastructure.

In terms of workload, we know that the total number of transactions run on the two configurations differed by less than 2%, so that we know in terms of response time, both configurations are similar, i.e. we are able to sustain 1 transaction per second for 8000 clients.

According to the RMF™ Workload report, the zCX configuration was able to offload 99% of the concentrator cost to zIIP. In addition, some of the TCPIP costs are eligible for offload when using zCX.

This means that the transaction costs for the two workloads was as follows:

Table: Transaction cost with 8000 clients

Address Space	Direct	zCX as concentrator	zCX as concentrator Cost after zIIP offload
MSTR	5.2	4.9	4.9
CHIN	127.3	48.2	48.2
TCPIP	13.3	22.3	12.4
Server application	34.5	25.2	25.2
zCX	0	157.5	0.9
TOTAL	180.3	257.8	91.6

Note: Costs are CPU microseconds per transaction.

As the table shows, provided that there are sufficient specialty processors available and the zCX workload is largely offloaded, the overall transaction cost can be reduced significantly.

The channel initiator costs for the zCX configuration are lower as there is significantly less management of tasks to be performed and less contention for processing the messages.

The application costs for the direct connection configuration are higher, primarily in the MQ API, as there is more contention in the buffer pools due to having more tasks attempting to get and put using a small set of queues.

Other benefits to MQ from using a concentrator

1. The storage used by the channel initiator is significantly reduced, needing only to support two MQ channels - one inbound and one outbound.
   - The direct connection using server-conn to the z/OS queue manager used 700 MB of channel initiator storage to support the 8000 channels.
   - When sending and receiving messages onto from the zCX concentrator queue manager, the channel initiators' storage footprint increased by less than 1MB.
2. The storage used by the MSTR address space as it is supporting far less connections - 1 rather than 8000, which resulted in 110 MB less 31-bit storage being used.
3. The amount of data generated due to accounting class(3) being enabled was significantly less - primarily the server applications and the 2 MQ channels, whereas the direct server-conn to channel initiator had 8000 SVRCONN channels written in each SMF interval. This is only relevant if class(3) accounting is enabled by default.
4. The amount of above bar storage used for accounting class(4) in the channel initiator was 40 MB less in the measurement using zCX as a concentrator.

Is the zCX concentrator queue manager limited to 9,999 channels?

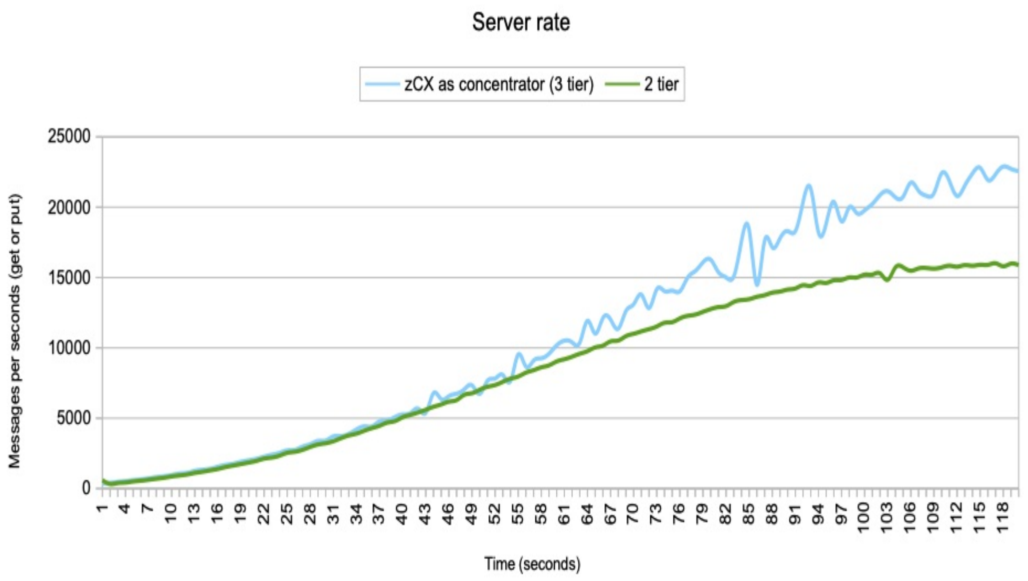
Simply put, no it is not.

The zCX concentrator queue manager is running the distributed MQ code which does not have the same channel limits are MQ for z/OS.

When starting the zCX concentrator queue manager we set the qm.ini Channels stanza as per the "supporting more than 10,000 channels" subsection.

When running with 12,000 clients connecting to the zCX, we saw all of the channels running but in some instances, the workload requirements of 1 transaction per second per client was not always achieved.

Some of this was due to queue contention - by splitting the workload over 2 sets of queues, the number of transactions exceeding the 1 second target reduced significantly, such that we are able to plot a graph of the following messaging rates:



This chart shows the messaging rate reaching 24,000 messages per second.

How does the message persistence affect the performance?

Persistent message performance is affected by the response time from the disk subsystem.

To give some indication of the disk response times within the container, we installed the `ioping` application in the MQ container and ran some basic measurements.

To determine the response time of the disks, we used the following command:

```
ioping -c 20 -D -s <size> /tmp
```

Note that we specify "-D" to use direct I/O, although sequential operations showed similar latency.

In these measurements, we used the following sizes: 4K, 32K and 256K. Additionally we ran with zHPF both enabled and disabled.

Table: "ioping" response times (CPU microseconds)

Size	zHPF=YES	zHPF=NO
4K	173	215
32K	259	343
256K	875	1100

The default z/OS guidance is to have zHPF enabled where possible, and as the table above demonstrates there is a clear benefit to running with zHPF in the zCX environment.

How does ioping compare with MQ I/O?

As we have seen earlier, the ioping response for 32K of data was 259 microseconds.

Using the MQ utility `amqsrua -m <qmgr> -c DISK -t Log -n 1` whilst running a workload with 32KB, the output was:

```
Log - physical bytes written 703823872 35840850/sec
Log - logical bytes written 659445800 33580984/sec
Log - write latency 700 uSec
Log - write size 34716
```

MQ's log process is more complex than the ioping utility and as a result will typically see longer write latency. If swapping is occurring, this can impact the MQ log tasks' response time, as potentially can other containers that are using the zCX disks.

In addition, the process of MQ writing a log record can result in multiple I/Os in the zCX environment, and in our measurements this overhead was of the order 3 to 9 times that which the MQ log task reported.

How does the zCX concentrator allow workloads to scale?

The ability of MQ to scale, whether using z/OS or distributed, is important and the MQ Performance group provides reports on the mqperf github repository.

With regards to hosting MQ in zCX, it is necessary to ensure that there is sufficient resource to process the additional load on the z/OS LPAR, whether that is CPU (zIIP or general purpose processors), load on disk, and network. RMF on z/OS allows for an overview of the system, and the use of additional monitoring utilities can provide insight into the performance of the zCX and any running containers.

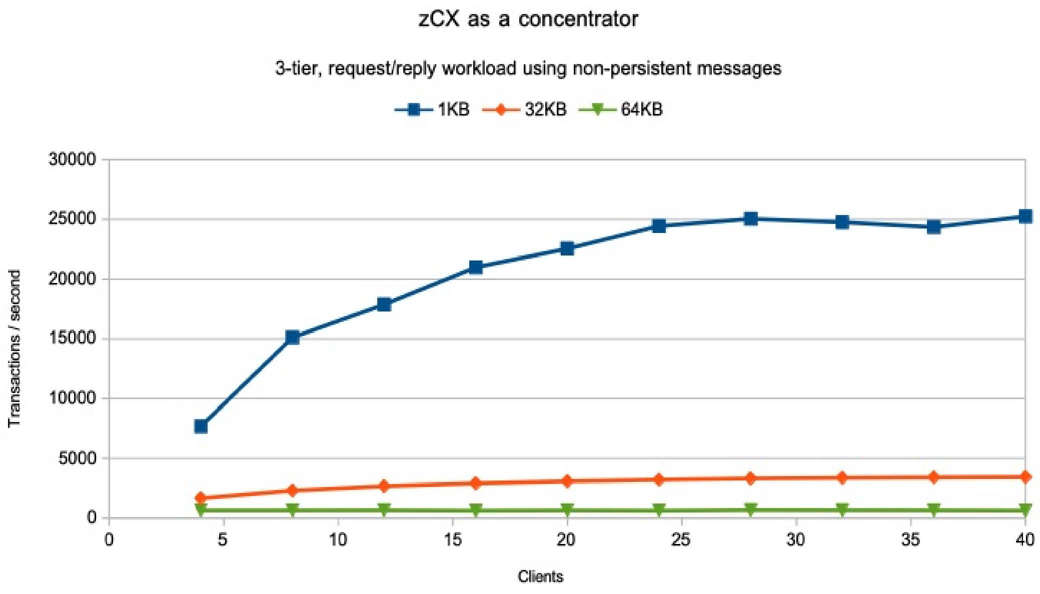
To give some indication of how the 3 tier configuration might perform, we provide the following examples using a small but increasing set of client applications to drive a request/reply workload through the zCX concentrator into the z/OS queue manager, where the messages will be served by a fixed set of batch applications.

Each configuration offers 3 message sizes: 1KB, 32KB and 64KB.

The workloads are run using non-persistent messages and subsequently persistent messages.

Non-persistent scaling measurement

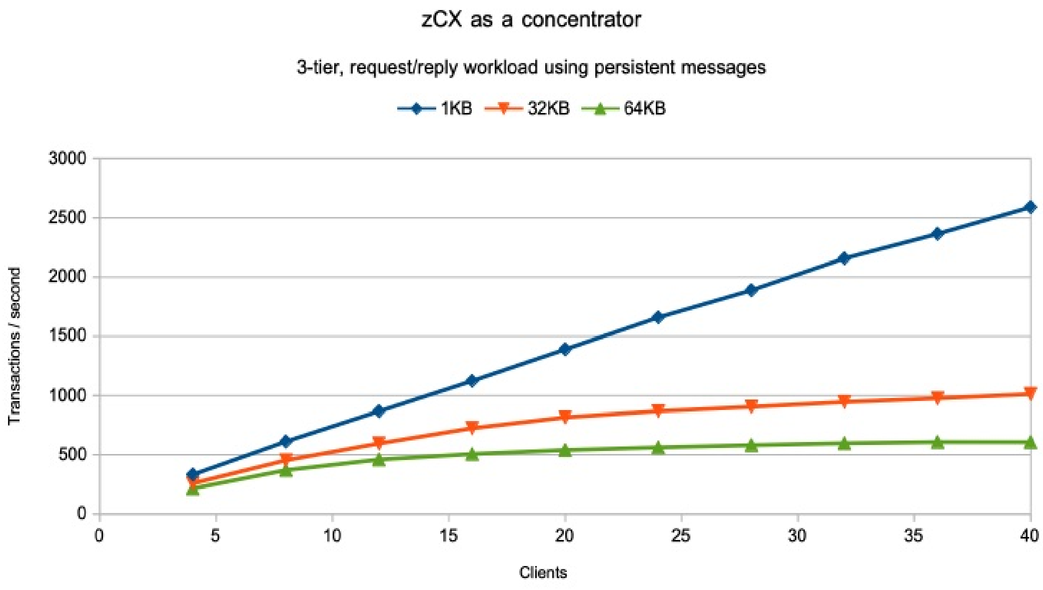
Chart: Increasing client applications with non-persistent messages



In this instance, the throughput is limited by network capacity, rather than CPU.

Persistent scaling measurement

Chart: Increasing client applications with persistent in-syncpoint messages



The 1KB workload continues to scale beyond 40 clients as neither zCX nor z/OS queue managers are at their log limits, whereas the 32KB and 64KB workloads are both approaching the point where their log tasks are fully utilised.

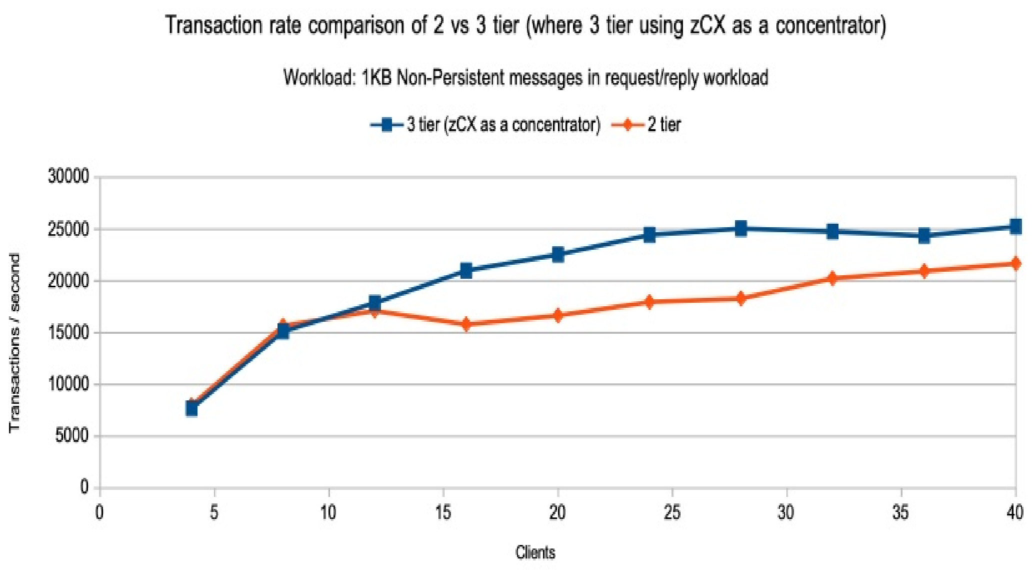
How does zCX concentrator scalability compare with 2-tier configurations?

As the previous section demonstrated the scalability of the zCX as a concentrator configuration, it might be useful to compare those results with the equivalent measurements where the clients connect directly to the z/OS queue manager.

This section compares the performance of the 2 and 3 tier configurations for both non-persistent and persistent messages for the 3 message sizes: 1KB, 32KB and 64KB.

Comparing non-persistent scalability performance

Chart: Comparing 1KB non-persistent throughput



Notes on chart:

The most notable difference in the performance with the 1KB non-persistent messages is that the 2-tier configuration shows worse performance than the 3-tier configuration once 16 or more clients are driving workload. In this instance, the 2-tier configuration is impacted by the increased number of interactions between the client and the queue manager, in the form of additional MQCB and MQCTL flows. The small message size combined with the significant increase in flows between client and queue manager, compared with the zCX queue manager to z/OS queue manager flows, result in the performance flattening out much sooner in the 2 tier configuration.

Queue manager to queue manager communication using MCA channels is more efficient than client to queue manager, and this is most obvious with smaller messages.

In both the 32KB and 64KB non-persistent workloads the performance of the 2 and 3 tier configurations is very similar - the time spent processing the larger message payload is sufficient to offset the impact of the additional client to queue manager flows, particularly in this low-latency environment.

Chart: Comparing 32KB non-persistent throughput

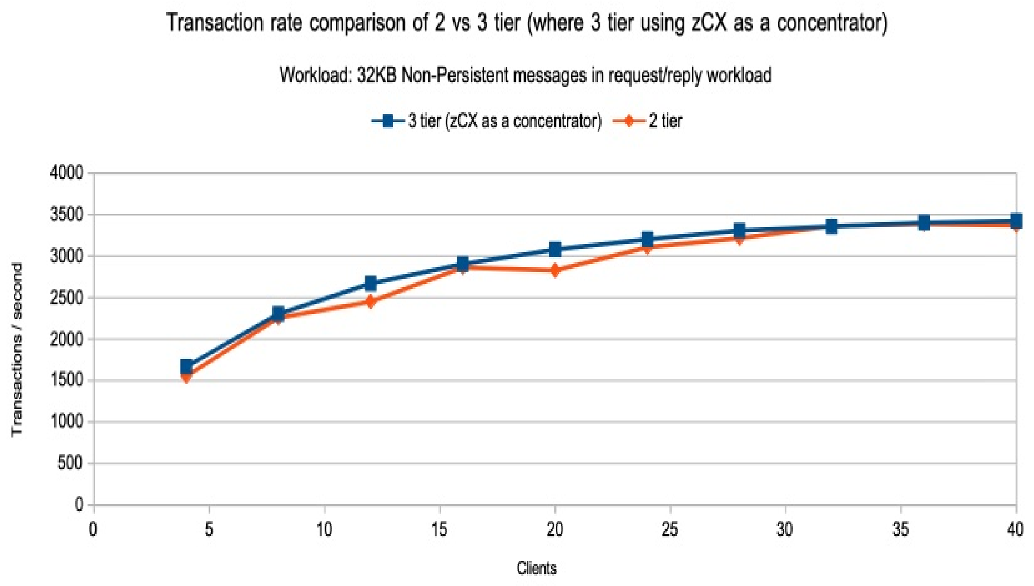
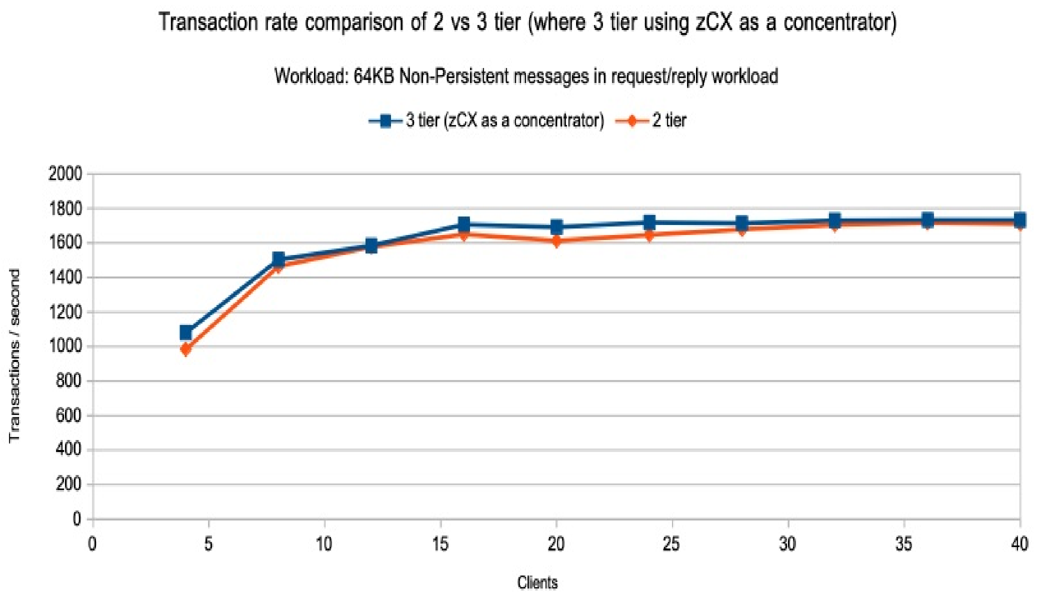
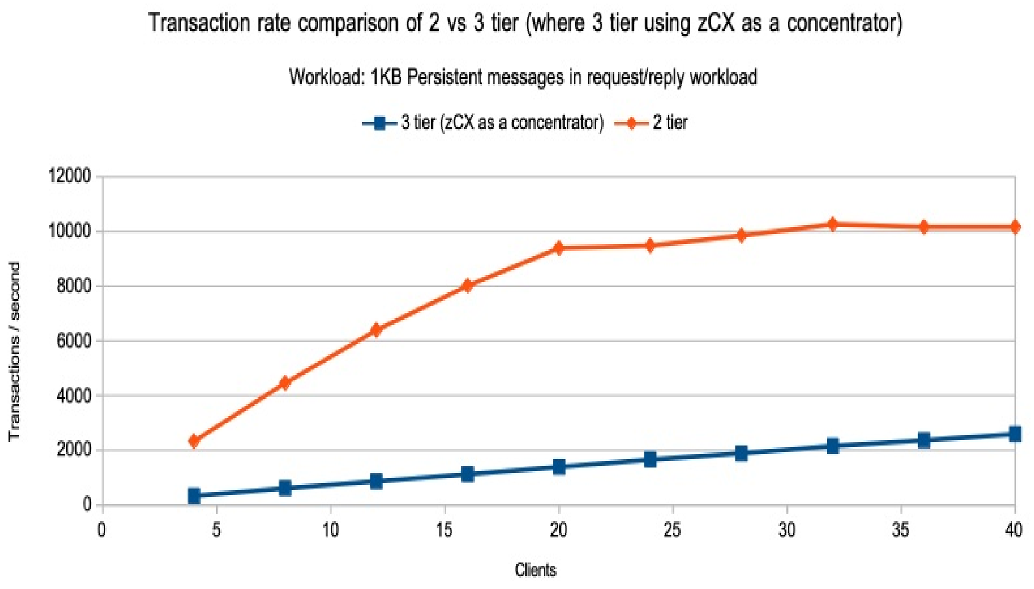


Chart: Comparing 64KB non-persistent throughput



Comparing persistent scalability performance

Chart: Comparing 1KB persistent throughput



Notes on chart:

The additional latency from the zCX queue manager having to log the transaction data makes a significant difference to the throughput of the 3-tier configuration.

For both 32KB and 64KB messages, the 3-tier configurations demonstrate increased throughput as the workload increases, yet the 2-tier configuration actually sees a decrease in throughput with increased client tasks. This decrease in throughput is related to increased contention at the queue, buffer pool, and log task level. Changing the workload to use multiple queue spread over multiple buffer pools would help improve the performance.

Chart: Comparing 32KB persistent throughput

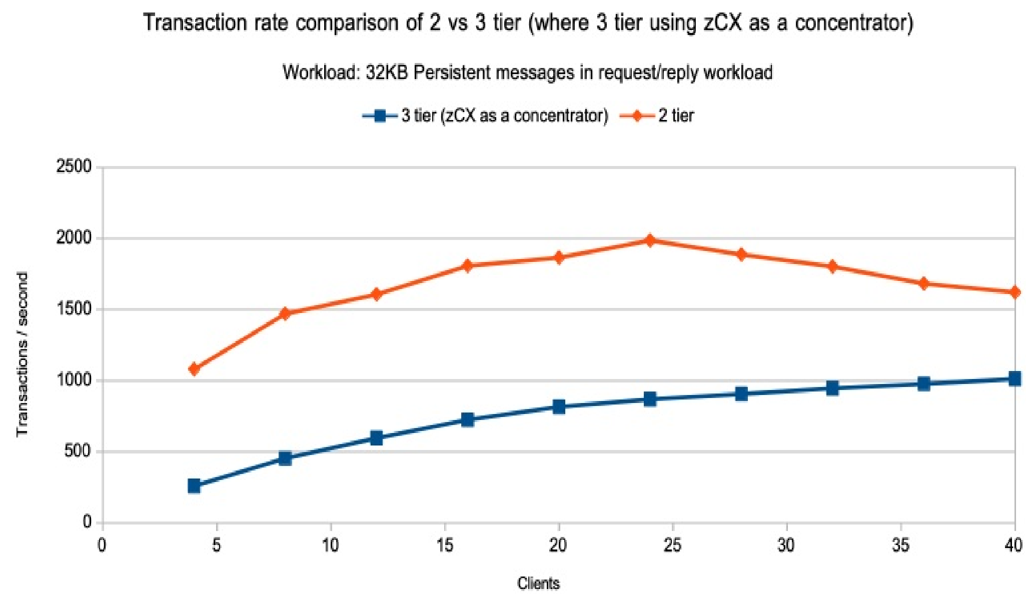
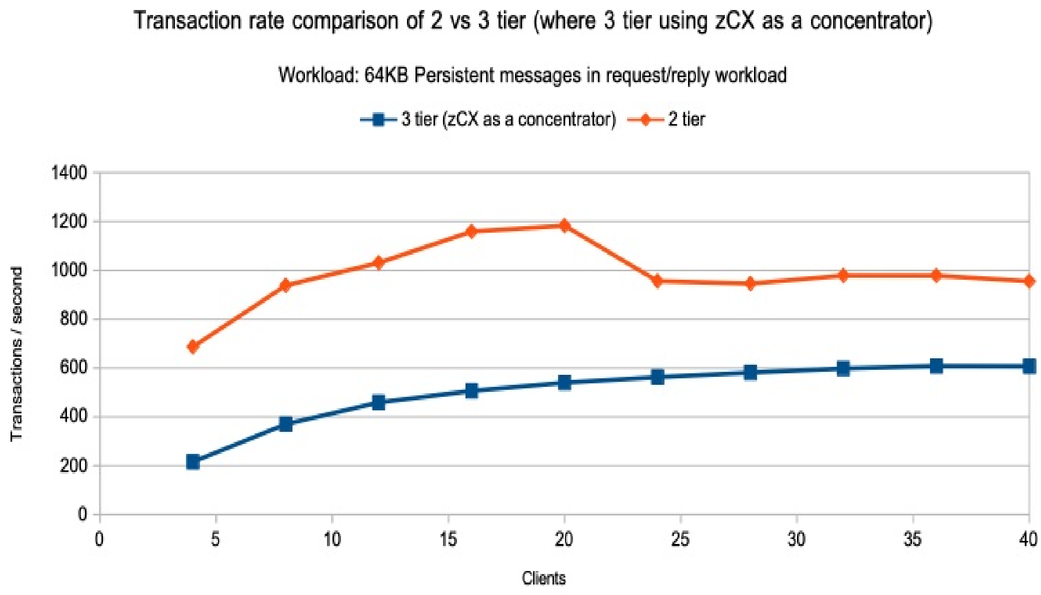


Chart: Comparing 64KB persistent throughput



Tuning the concentrator queue manager

Supporting more than 10,000 channels

With MQ for z/OS' absolute limit of 9,999 channels, the ability of a concentrator queue manager to allow many more thousands of clients to connect can provide a useful mechanism to increase the number of clients able to connect to the z/OS queue manager.

As was indicated earlier, including the `--env AMQ_EXTRA_QM_STANZAS=Channels:MaxChannels=20000` statement in the command to create and start the concentrator queue manager can allow up to 20,000 channels to run.

Remember that the channels running between the concentrator and the z/OS queue manager are included in this total.

Persistent messages and client concentrators

If the workload passing through the client concentrator is largely persistent, you may benefit from increasing the size of the log files.

This can be achieved by specifying a variant of the `--env AMQ_EXTRA_QM_STANZAS=Log:LogFilePages=32768` statement.

Multiple tuning options

It is possible to specify multiple options from different stanzas, such as the Channels and Log stanzas by using the vertical bar "|" to combine into a single --env parameter, e.g.

```
--env AMQ_EXTRA_QM_STANZAS=Channels:MaxChannels=20000|Log:LogFilePages=32768,LogPrimary=64
```

Chapter 4

Cluster repository

Overview of clustering

Distributed queuing means sending messages from one queue manager to another. The receiving queue manager can be on the same machine or another; nearby or on the other side of the world. It can be running on the same platform as the local queue manager, or can be on any of the platforms supported by IBM MQ. You can manually define all the connections in a distributed queuing environment, or you can create a cluster and let IBM MQ define much of the connection detail for you.

You can group a set of queue managers in a cluster, and when you do this, the queue managers can make the queues they host available to other queue managers in the cluster without need for explicit channel definitions, remote-queue definitions, or transmission queues for each destination.

Benefits of using clusters

Clustering provides two key benefits:

1. Clusters simplify the administration of IBM MQ networks, which usually require many object definitions for channels, transmit queues and remote queues to be configured. This situation is especially true in large, potentially changing, networks where many queue managers need to be interconnected. This architecture is particularly hard to configure and actively maintain.
2. Clusters can be used to distribute the workload of message traffic across queues and queue managers in the cluster. Such distribution allows the message workload of a single queue to be distributed across equivalent instances of that queue located on multiple queue managers. This distribution of the workload can be used to achieve greater resilience to system failures, and to improve the scaling performance of particularly active message flows in a system. In such an environment, each of the instances of the distributed queues have consuming applications processing the messages.

How messages are routed in a cluster

You can think of a cluster as a network of queue managers maintained by a conscientious systems administrator. Whenever you define a cluster queue, the systems administrator automatically creates corresponding remote-queue definitions as needed on the other queue managers.

You do not need to make transmission queue definitions because IBM MQ provides a transmission queue on each queue manager in the cluster. This single transmission queue can be used to carry messages to any other queue manager in the cluster. You are not limited to using a single transmission queue. A queue manager can use multiple transmission queues to separate the messages going to each queue manager in a cluster. Typically, a queue manager uses a single cluster transmission queue. You can change the queue manager attribute DEFCLXQ, so that a queue manager uses a different cluster transmission queue for each queue manager in a cluster. You can also define cluster transmission queues manually.

All the queue managers that join a cluster agree to work in this way. They send out information about themselves and about the queues they host, and they receive information about the other members of the cluster.

To ensure that no information is lost when a queue manager becomes unavailable, you specify a minimum of two queue managers in the cluster to act as *full repositories.* These queue managers store a full set of information about all the queue managers and queues in the cluster. All other queue managers in the cluster only store information about those queue managers and queues with which they exchange messages. These queue managers are known as *partial repositories.*

Full repository and partial repository

Typically, two queue managers in a cluster hold a full repository. The remaining queue managers all hold a partial repository.

A partial repository contains information about only those queue managers with which the queue manager needs to exchange messages. The queue managers request updates to the information they need, so that if it changes, the full repository queue manager sends them the new information. For much of the time, a partial repository contains all the information a queue manager needs to perform within the cluster. When a queue manager needs some additional information, it makes inquiries of the full repository and updates its partial repository. The queue managers use the `SYSTEM.CLUSTER.COMMAND.QUEUE` queue to request and receive updates to the repositories.

Why host a full repository in zCX?

As the MP16 “Capacity Planning and Tuning Guide” section “What is the capacity of my channel initiator task?” details, the cluster cache in a z/OS queue manager stores data about cluster subscriptions.

This data is only allocated if there are any cluster channels defined, and so exists in both partial and full repository queue managers.

The storage usage may vary, depending on the setting of the CLCACHE attribute. The initial cache size will be 2MB - but if CLACHE is set to DYNAMIC, the cache may grow. Should CLCACHE be set to STATIC and the cache space is exhausted, a restart of the channel initiator will ensure the storage allocated to the cluster is increased to give some spare capacity.

Since the cluster cache for z/OS queue managers is defined in 31-bit storage, it is limited to 2GB, and this 2GB must be shared with other users of channel initiator storage including MQ channels and tasks such as adapters.

For very large clusters, it is possible that the z/OS queue manager would not have sufficient storage available to host a full repository.

By contrast, a queue manager running in zCX or indeed on a distributed platform uses 64-bit storage for the cluster cache, and as a result can store significantly more cluster information.

There is no particular benefit to hosting the full repository in zCX as opposed to a z/OS queue manager unless the repository is expected to contain many millions of cluster objects or cluster subscriptions.

If your cluster is expected to be very large, there are several reasons why you might choose to host a full cluster repository in zCX:

- There are no distributed queue managers in the cluster that are appropriate for the role of full repository.
- The distributed queue managers in the cluster are located at distance, and you would prefer to have a full repository co-located with your z/OS queue managers.

zCX as a cluster repository performance measurements

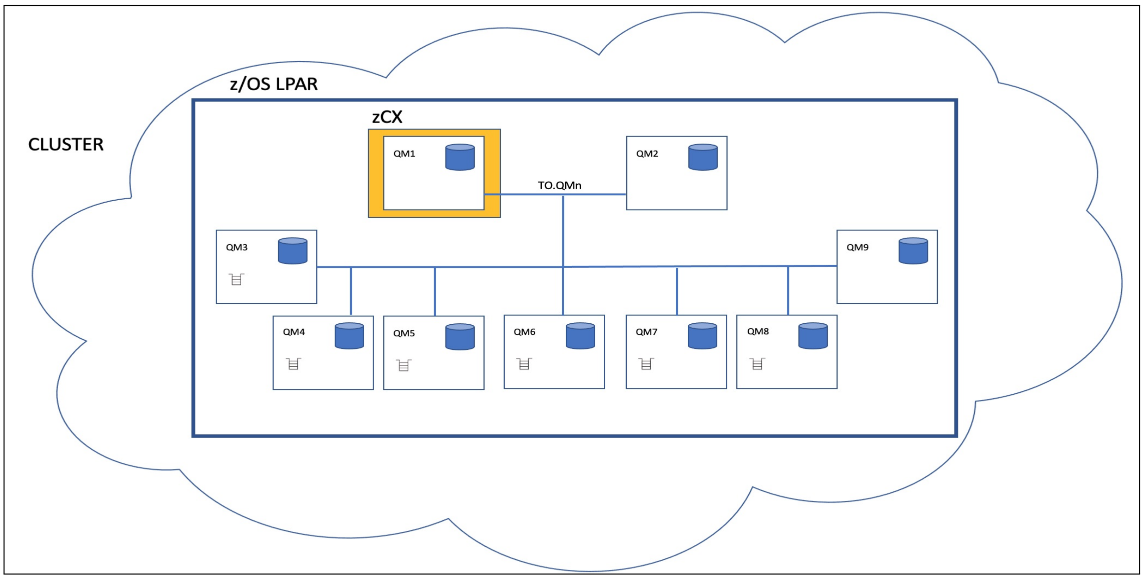
The cluster repository performance measurements ask 2 questions:

1. Does the performance differ between the initialisation of a z/OS repository and a zCX repository?
2. Is the first subscription (MQOPEN) of a cluster queue affected by the location of the full repository?

zCX repository performance test topology

For the performance tests run, the following test topology was used:

Chart: Test topology when using the zCX as a cluster repository



Note:

- Cluster contains 9 queue managers - QM1 through QM9.
- QM1 and QM2 host full repositories of information about the queue managers and queues in the cluster.
- QM1 is running in zCX, all other queue managers are running natively in z/OS.
- QM3 through QM8 host some cluster queues, that are accessible to any other queue manager in the cluster.
- Each queue manager has a cluster-receiver channel called TO.qmgr on which it can receive messages.
- Each queue manager also has a cluster-sender on which it can send information to one of the repository queue managers.
  - QM2 through QM9 sends to the repository at QM1.
  - QM1 sends to the repository at QM2.

Does the performance differ between the initialisation of a z/OS repository and a zCX repository?

What do we mean by the initialisation of the repository?

On each queue manager in the cluster, there are a number of `SYSTEM.CLUSTER.*` queues. These queues are described in detail in the default cluster objects section in the MQ Knowledge Center.

Of particular interest are the following queues:

`SYSTEM.CLUSTER.REPOSITORY.QUEUE`

- each queue manager in a cluster has a local queue of this name. This queue is used to store all the full repository information. This queue is not normally empty.

`SYSTEM.CLUSTER.COMMAND.QUEUE`

- each queue manager in a cluster has a local queue of this name, which is used to transfer messages to the full repository. The message contains any new or changed information about the queue manager, or any requests for information about the other queue managers. This queue is normally empty.

In the performance tests run, we have defined cluster queues on each of the partial repositories, *except QM9*. This results in messages being sent to the full repositories, and they will reside on the `SYSTEM.CLUSTER.COMMAND.QUEUE` until they have been processed by the repository manager.

For the purposes of these performance measurements, we regard the cluster "ready for business" when the `SYSTEM.CLUSTER.COMMAND.QUEUE` has `CURDEPTH(0)` on each of the full repository queue managers.

In this section we compare the time taken for the repository queue managers for this initialisation to complete such that the cluster is ready for business.

Why does this matter?

It might be expected that initial load of the full repository is only performed once, i.e. at initialisation, but when administration needs to be performed, such as using the `RESET CLUSTER` or `REFRESH CLUSTER` commands, the longer the `SYSTEM.CLUSTER.COMMAND.QUEUE` takes to reach `CURDEPTH(0)`, the longer it is before the cluster is once more "ready for business".

Initialisation time

In this measurement, we define 1,000,000 cluster queues on each of 6 partial repository queue managers (QM3 through QM8). This means there is a total of 6 million cluster queues in the cluster, but only 1 million unique queue names.

The information about these cluster queues are shared with the repository queue managers and the time taken for the `SYSTEM.CLUSTER.COMMAND.QUEUE` to reach depth zero is reported below.

- **z/OS** 7 minutes elapsed with a maximum depth of 50,000 messages.
- **zCX** 25 seconds elapsed, with a maximum depth in excess of 500,000 messages.
  - As a result of the COMMAND queue reaching such a high queue depth, the amount of space used to host those messages reached 1.2GB in the zCX volume.[1]

[1] Usage for the mqdata volume was determined using the command: `docker system df -v`.

Is the first subscription (MQOPEN) of a cluster queue affected by the location of the full repository?

When a cluster queue is defined on a local queue manager, the information is shared with the repository queue manager. If any other queue manager in the cluster hosts a queue of the same name, the full repository queue manager will return this information to the local queue manager, thus ensuring that the partial repository queue manager is aware of **all** valid destinations of that queue name.

If an application connects to the local queue manager and attempts to open a queue that is known to the partial repository, the resolution is near instantaneous.

If however, the partial repository queue manager does not hold information on such a named queue, it will require a resolution from one of the full repository queue managers.

The location of the full repository queue manager may have some impact on how long that resolution takes.

In our measurements, all of the queue managers are co-located in the same physical LPAR, but we compare the average resolution time when using either the ZCX or the z/OS repository queue manager.

To ensure we are confident of accessing the desired repository, we stop the queue manager of the alternate repository. Note that this does mean we are running with a single repository for a short period of time - which is not recommended.

In the performance measurements run, the application connects to QM9 and attempts to MQOPEN (subscribe) and then MQCLOSE a number of cluster queues that are defined on other queue managers in the cluster.

Regardless of the location of the repository queue manager (zCX or z/OS), each resolution took 1 second to complete, i.e. 1000 MQOPEN/MQCLOSEs took 1000 seconds.

By contrast, when attempting the same process once the local queue managers' partial repository was up to date, the same 1000 MQOPEN/MQCLOSEs was completed in less than 2 seconds.

Note that if the cluster resolution takes too long, MQ will return MQRC 2085 "Unknown object name".

zCX as a repository - things to watch out for

The primary benefit of using zCX as a full repository is to be able to store more cluster information than a z/OS queue manager acting in the same role.

In order to do this, ensure that there is sufficient disk space and memory to host the large repository queue manager.

Our original configuration defined the zCX with 10GB of disk space and 4GB of memory. As the system was defined for testing purposes we had multiple images stored and when defining the objects for the repository, we reached the point where the system ran out of disk space - primarily due to the high depth of `SYSTEM.CLUSTER.COMMAND.QUEUE`.

This resulted in the container hosting the queue manager failing with `AMQ6125E An internal MQ error has occurred`.

Attempting to restart the container using "`docker restart zcxrepos`" reported:

```
Error:  endpoint with name zcxrepos already exists in network bridge.
```

For this particular use case, we restarted the zCX address space and recreated the repository container.

Check that there is sufficient space:

- Command "`docker system df`" will display the allocation of space currently in use.
  - The -v (verbose) option provides additional useful information which may help determine which components are using the space.
- The command "`du -h .`" issued when running inside the MQ container can indicate which directories are sizeable.

In our test that ran out of space in zCX, we saw allocations, by queue, of:

- `SYSTEM.CLUSTER.COMMAND.QUEUE` used 1.7GB of disk space
- `SYSTEM.CLUSTER.REPOSITORY.QUEUE` using an additional 100MB.

Note that the `SYSTEM.CLUSTER.COMMAND.QUEUE` having a high queue depth that requires such a large amount of storage should be unusual, and was due to the rapid loading of the cluster objects. A cluster repository that grows more organically would not typically see such a high queue depth. The reason for highlighting this particular case is that the resolution to the system running out of space required an outage to this cluster repository queue manager.

Chapter 5

Aspera fasp.io gateway

IBM MQ Advanced for z/OS VUE version 9.2 only

The IBM® Aspera fasp.io gateway provides a fast TCP/IP tunnel that can significantly increase network throughput for IBM MQ.

The Aspera gateway can be used to improve the performance of queue manager channels. It is especially effective if the network has high latency or tends to lose packets, and it is typically used to speed up the connection between queue managers in different data centres.

However, for a fast network that does not lose packets there is a decrease in performance when using the Aspera Gateway, so it is important to check network performance before and after defining an Aspera Gateway connection.

A queue manager running on any entitled platform can connect through an Aspera gateway. The gateway itself is deployed on Red Hat®, Ubuntu Linux® or Windows. This means that the gateway can be deployed on Linux on Z or in a z/OS Container Extension (zCX). zCX is a particularly good candidate for running the gateway as it ensures the queue manager and the gateway are as close as possible.

When deploying the Aspera fasp.io gateway in a z/OS Container Extension, the CPU used is largely eligible for offload to zIIP. In our measurements, it was typical to see in excess of 98% of the zCX costs to be eligible for zIIP offload.

When deploying the fasp.io gateway other than in zCX, consider that there may be additional latency when communicating between the MQ channel initiator and the gateway.

Aspera fasp.io gateway performance highlights

The following section details some of the performance environments where use of the Aspera fasp.io gateway can provide significant improvements to throughput, in particular:

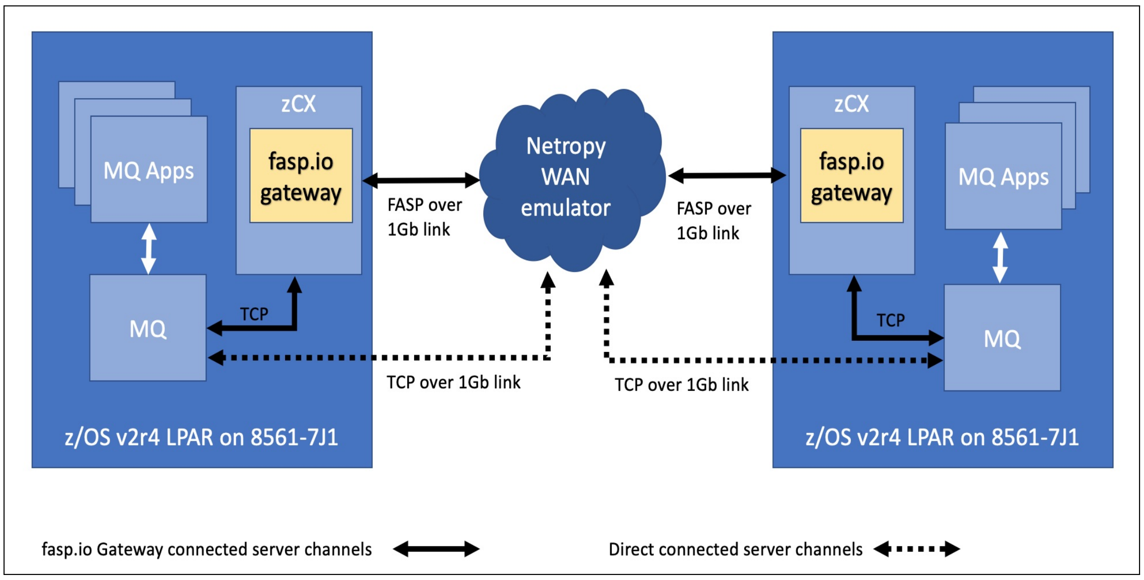
- Streaming-type workloads with large messages - up to 72 times throughput improvement.
- Streaming-type workloads with small messages - up to 11 times throughput improvement.
- Request/Responder-type workloads with medium messages - up to 60% throughput improvement.
- Reduction in overall transaction cost with the use of zIIPs.
- Reduced impact from unreliable networks.

Aspera fasp.io gateway test configuration

The Aspera fasp.io gateway benefit can be seen in networks with high latency and/or a tendency to lose packets. To simulate these environments, we use a Netropy 10G4 WAN emulator, which we configure dynamically to simulate a range of network configurations.

In all tests shown in this section, the following test topology is used.

Chart: Test topology when using the Aspera fasp.io Gateway



Note:

- All communication, whether using the Aspera fasp.io gateway or not, is routed through the Netropy WAN emulator.
- All communication is limited to 1Gb / sec.
- The fasp.io gateway on each LPAR is running in a zCX.

Workload configuration

There are two models of workload that we have considered for use with the Aspera fasp.io Gateway.

1. Uni-directional (streaming) - send/receive workload, simulating Queue Replication with 1 capture task and 1 apply task. These tests are modelled on the streaming tests in the Regression section of this document.
2. Bi-directional (Request/Responder) workload utilising 1 pair of server channels with multiple applications. These tests are modelled on the request/reply workloads using 1 channel pair in the Regression section of this document.

Each of these workloads is measured in 5 configurations, with different latency and packet loss, as below:

- **N0:** 0ms network latency (no packet loss)
- **N1:** 5ms network latency (no packet loss)
- **N2:** 25ms network latency (no packet loss)
- **N3:** 40ms network latency (0.1% packet loss)
- **N4:** 50ms network latency (0.5% packet loss)

An additional configuration, N5, is used on occasion to demonstrate the impact from packet loss on high latency networks.

- **N5:** 50ms network latency (no packet loss)

Note: Latency is applied to both directions, so request/responder workloads will be affected by 2 times the latency.

Aspera fasp.io gateway streaming workload

The streaming workloads use 2 message sizes, 10KB and 1MB, to simulate online and batch processing models respectively.

In each case, the MQ channels are configured to have a batch size of 200 and sufficient buffers are available to minimise impact from I/O to MQ page set on both the sending and receiving queue managers.

10KB streaming workload

In a streaming workload, smaller messages, such as 10KB, tend to result in worse performance when using the Aspera fasp.io gateway on relatively low-latency networks.

As the latency increases, such as in the N2 configuration, parity is achieved with the TCP/IP configuration, but as the latency and packet loss begins to occur, the fasp.io gateway shows significant improvement in throughput.

- **N3**: 4.8x TCP performance
- **N4**: 11.75x TCP performance

The following charts shows the sustained channel throughput in MB/second across a single MQ channel and the cost per transaction.

The first chart demonstrates the improved performance as the network latency and packet loss increases. The second chart compares the cost of running the workloads and demonstrates the benefits that may be achieved provided sufficient zIIP capacity is available.

Chart: Achieved channel throughput when streaming 10KB persistent messages.

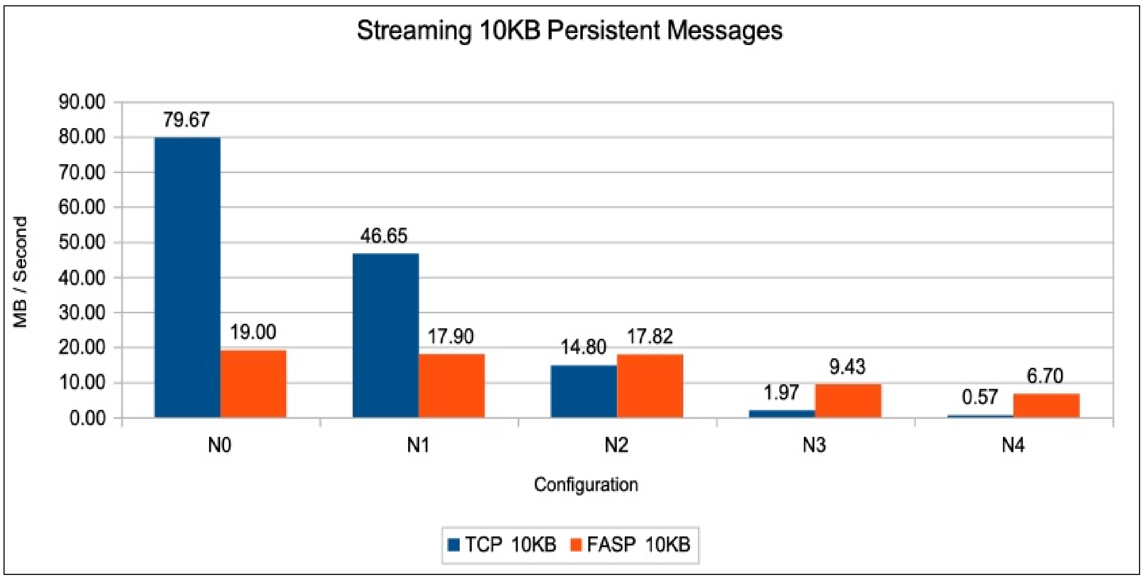
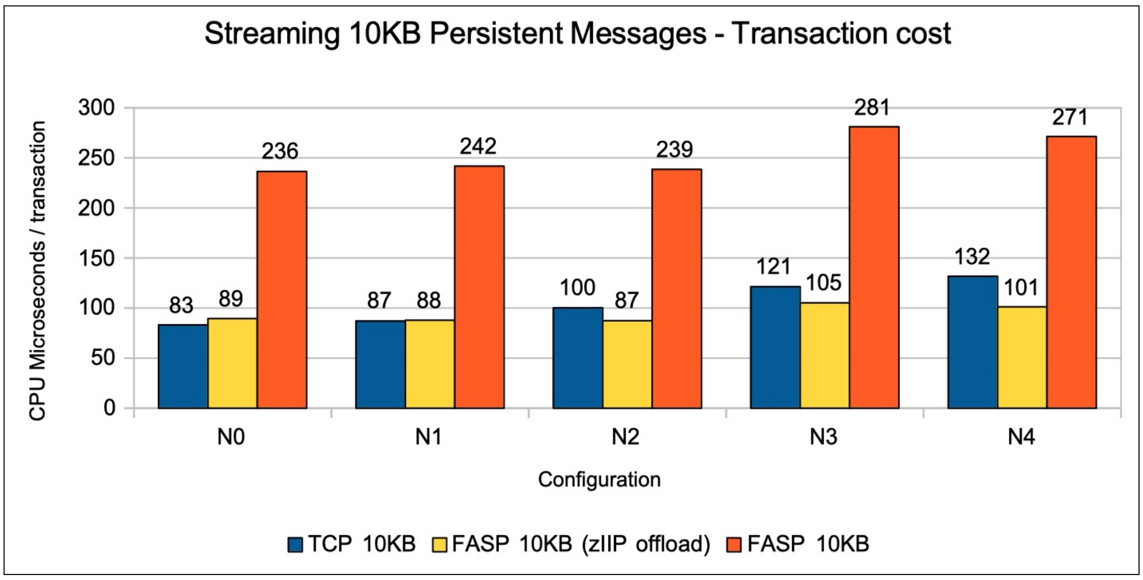


Chart: Actual transaction cost when streaming 10KB persistent messages.



Notes on transaction cost chart:

The transaction cost is calculated using the data from the RMF Workload report and includes the costs for the MQ MSTR, channel initiator, zCX, TCP/IP and application address spaces.

The costs reported for the "FASP 10KB (zIIP offload)" columns are based on the reported zIIP-eligibility for those address spaces and are the optimum costs based on those calculations.

1MB streaming workload

In a streaming workload, larger messages, such as 1MB with larger achieved batch sizes (XBATCHSZ) tend to result in performance that is comparable with TCP/IP even in low latency networks, and far exceed the performance of TCP/IP on high-latency networks or those suffering from packet loss.

- **N1**: +17% TCP performance
- **N2**: 3.6x TCP performance
- **N3**: 22.4x TCP performance
- **N4**: 72.25x TCP performance

The following charts shows the sustained channel throughput in MB/second across a single MQ channel and the cost per transaction.

The first chart demonstrates the improved performance as the network latency and packet loss increases. The second chart compares the cost of running the workloads and demonstrates the benefits that may be achieved provided sufficient zIIP capacity is available.

Chart: Achieved channel throughput when streaming 1MB persistent messages.

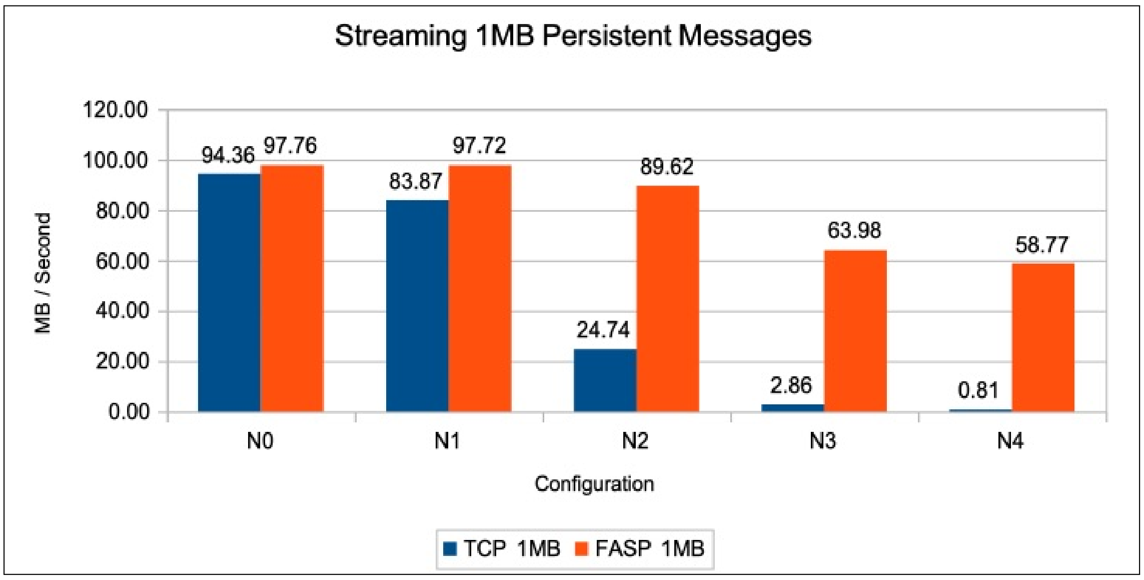
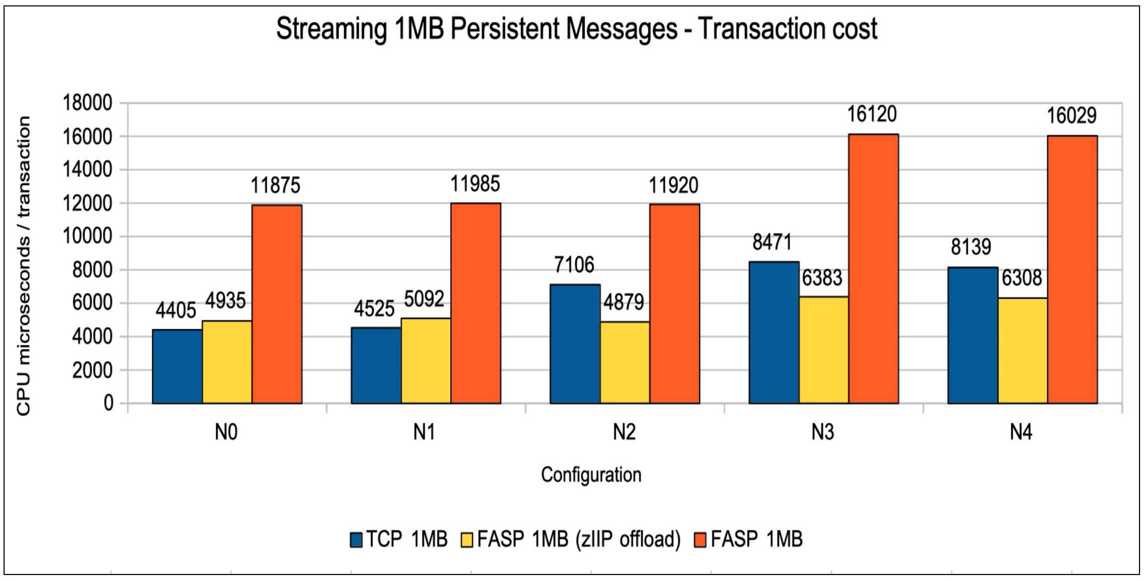


Chart: Actual transaction cost when streaming 1MB persistent messages.



Notes on transaction cost chart:

The transaction cost is calculated using the data from the RMF Workload report and includes the costs for the MQ MSTR, channel initiator, zCX, TCP/IP and application address spaces.

The costs reported for the "FASP 1MB (zIIP offload)" are based on the reported zIIP-eligibility for those address spaces and are the optimum costs based on those calculations.

Aspera fasp.io gateway request/responder workload

32KB request/responder

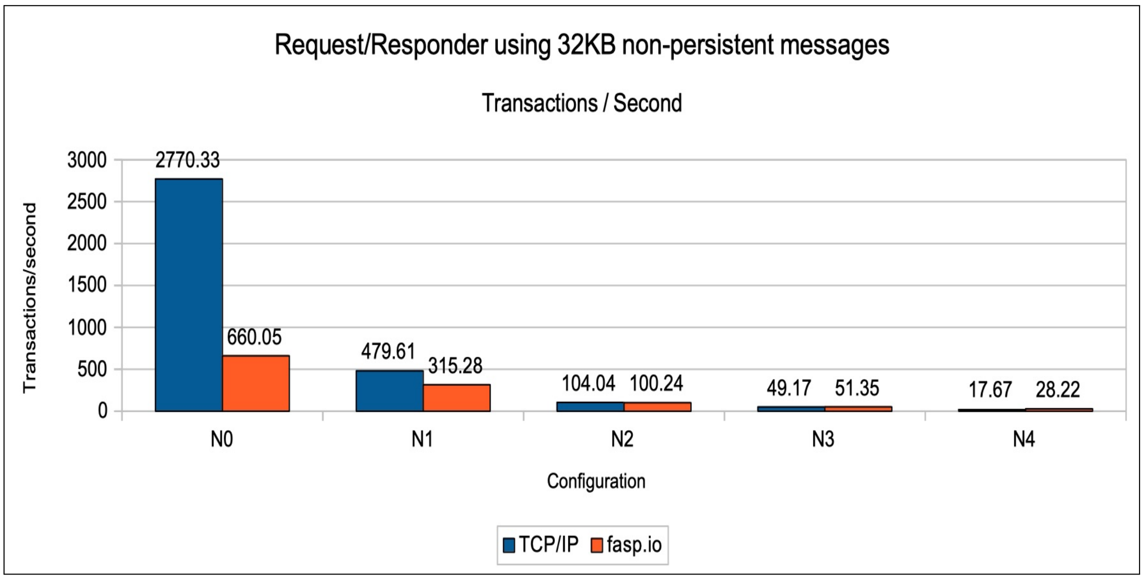
Request/Responder workloads do not benefit from using the fasp.io gateway to the same extent as streaming workloads.

There are many factors which impact the performance of a request/responder workload when routing the network traffic through a pair of fasp.io gateways, and these include:

- Batch size - this is the number of messages in the batch, and the amount of data flowing over the channel between end-of-batch flows.
- Message size.

In the following example, the measurement uses 32KB messages where a small number of requester applications each put a message and wait for a reply of equivalent size.

Chart: Achieved transaction rate with 32KB messages in request/responder workload.

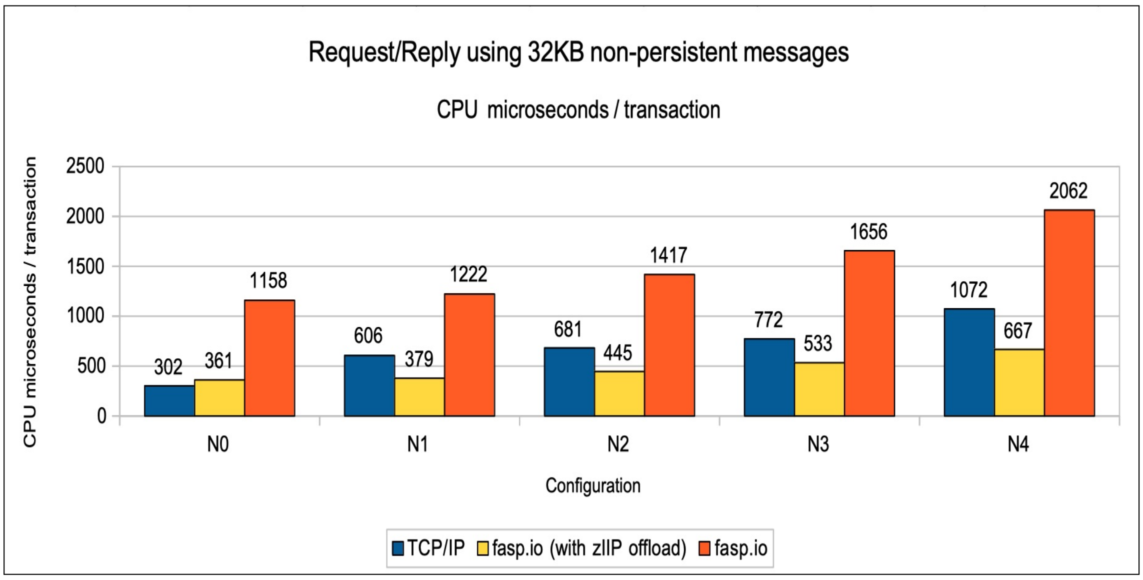


Notes on chart:

At configurations N2 and N3, the fasp.io configuration is able to achieve parity with the TCP/IP configuration. As both latency and packet loss increases, the benefits of the Aspera fasp.io gateway become more obvious, such that with configuration N4, 50ms latency (in each direction) and 0.5% packet loss, the throughput improves by 60%.

When zIIP offload is not available, the fasp.io gateway hosted on zCX configuration shows costs that are between 2-3 times that of the TCP/IP configuration. Providing sufficient zIIP resource can bring CPU savings in excess of 20% over the comparable TCP/IP configuration.

Chart: Actual transaction cost with 32KB messages in Request/Responder workload.



Notes on chart:

The chart shows the fasp.io cost in two modes; when running the gateway on zCX using general purpose processors and again when the zIIP-eligible work is fully offloaded. Once network latency is introduced, provided there is sufficient capacity to offload the zCX work to zIIP, the fasp.io workload can offer a reduction in overall CPU cost.

The increase in cost in the TCP/IP configuration as latency increases is primarily incurred in the network layer, whether in the TCP/IP address space or MQ's dispatcher tasks in the channel initiator.

The increase in cost in the fasp.io configuration as latency increases is again primarily in the network layer, but in this configuration the cost is incurred by TCP/IP and the zCX address spaces - and as a result of zCX being largely zIIP eligible, the overall transaction cost is lower than the equivalent TCP/IP measurement when zIIP-offload is fully utilised.

How much impact is there from packet loss on high latency networks?

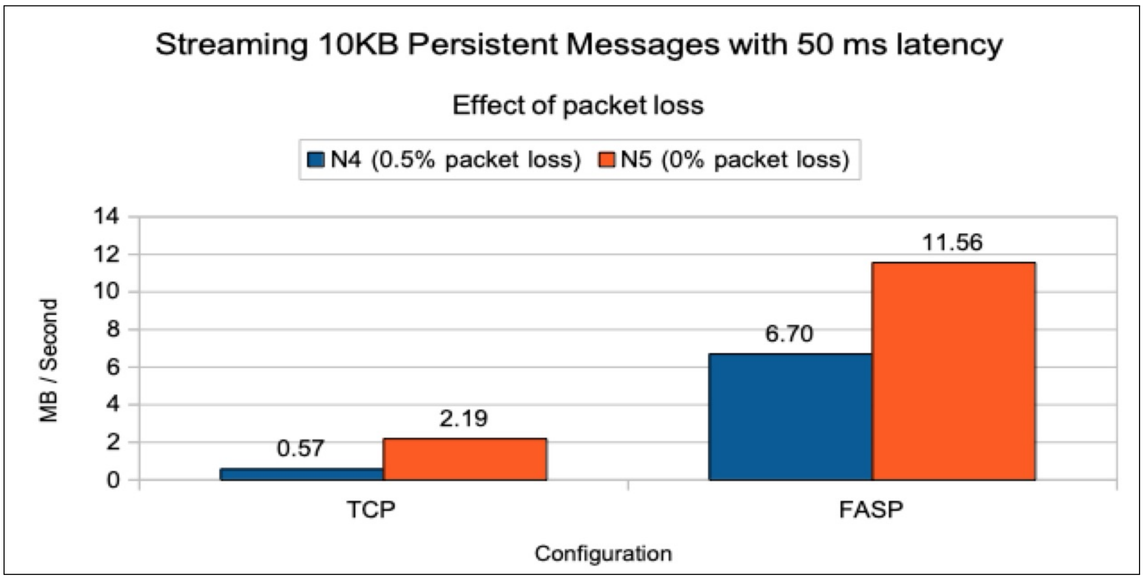
TCP/IP is a conversational-type protocol such that packet loss is noticed relatively quickly and results in data being re-sent. This can result in TCP/IP being unable to optimise its send and receive buffers to be able to benefit from dynamic right sizing.

By contrast, the fasp.io gateway uses a proprietary user defined protocol (UDP) which sends the data and at certain points performs scrutiny on the received data and in the event of lost data, requests that data is re-sent.

As a result, networks that experience packet loss see a larger relative impact to the TCP/IP performance than the fasp.io gateway performance.

If the streaming tests are repeated such we can compare the N4 (0.5% packet loss) and N5 (0% packet loss) configurations, we observe the performance changes as per the following 2 charts:

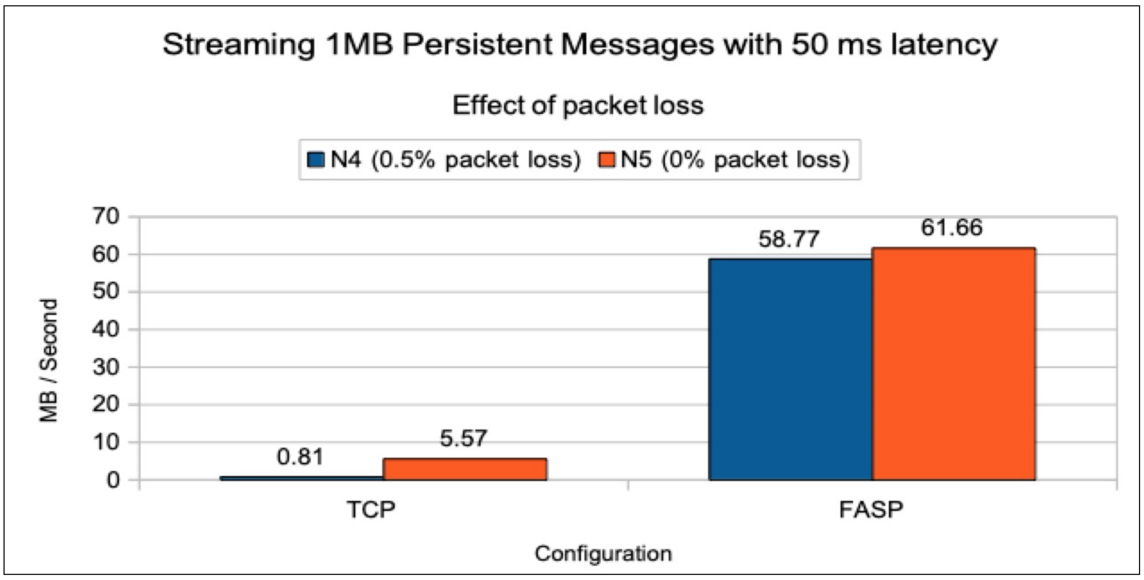
Chart: Zero loss channel throughput when streaming 10KB persistent messages.



For the N5 configuration with 0% packet loss:

- TCP/IP throughput increased from 0.57 MB/second to 2.19 MB/second - an increase of 3.8x or 1.6 MB/second.
- fasp.io throughput increased from 6.7 MB/second to 11.56 MB/second - an increase of 1.72x or 4.86 MB/second.

Chart: Zero loss channel throughput when streaming 1MB persistent messages.



For the N5 configuration with 0% packet loss:

- TCP/IP throughput increased from 0.81 MB/second to 5.56 MB/second - an increase of 6.8x or 4.75 MB/second.
- fasp.io throughput increased from 58.77 MB/second to 61.66 MB/second - an increase of 4% or 2.89 MB/second.

Does achieved batch size affect Aspera fasp.io gateway performance?

In our measurements, the batch confirm flow was the major inhibitor to the fasp.io performance, particularly on low-latency networks. You can see the impact in the NETTIME value, which shows the network cost of the flow.

The following table shows the expected NETTIME values based on the latency and the reported NETTIMEs for all five configurations.

Table: NETTIME - Expected vs Achieved

Configuration	Expected	TCP/IP actual	fasp.io actual
N0	< 1	1	5-6
N1	10	10	16-18
N2	50	50	54-58
N3	80	80-100	90-97
N4	100	120-180	110-130

Note on table: The expected, actual TCP/IP and actual fasp.io times are reported in milliseconds and are as reported by the DISPLAY CHSTATUS MQSC command.

The expected latency is double the latency configured, as the confirm flow requires data to flow in both directions.

In the fasp.io configurations, the NETTIME was typically 4+ milliseconds longer than the equivalent TCP/IP configuration - until configuration N3 where the impact of packet loss affected the TCP/IP configuration far more significantly.

Note that it is not clear whether the additional time reported by NETTIME is an artifact of hosting the gateway on zCX or is directly related to the fasp.io gateway.

Mitigating impact of BATCHSZ on fasp.io gateway

In the higher latency configurations, the achieved batch size (XBATCHSZ) indicated that the batches were not full. Given the significant impact of the end of batch flow, ensuring each batch contained more messages resulted in the transaction rate more than doubling, e.g. for the request/responder workload using 32KB messages, the N4 configuration originally achieved 28 transactions per second with XBATCHSZ(34), but when achieving XBATCHSZ(50) the transaction rate increased to 63 (2.25x) with the side effect of increased latency per round-trip.

This can be achieved in a number of ways:

- Increase BATCHSZ - only of benefit if the XBATCHSZ already matches the BATCHSZ. If the channel sends a wide range of message sizes, it may be beneficial to use the BATCHLIM attribute to ensure that the amount of data (MB) is not excessive such that the target buffer pool is sufficiently large to hold at least one full batch without writing to page set.
- Increase BATCHINT - to keep the channel batch open for longer. Be aware that increasing this can result in increased response times because the batches last longer and messages will remain uncommitted for longer.

Does message size affect Aspera fasp.io gateway performance?

Yes, but to a far less extent than the achieved batch size.

In a request/responder-type workload with a 5 millisecond latency, as per configuration N1, we found that messages up to 32KB achieved a transaction rate approximately 33% lower when using the fasp.io gateway rather than TCP/IP.

Messages between 64KB and 100KB achieved a transaction rate 12-20% higher in the TCP/IP configuration when measured with 5 milliseconds of latency (as per N1).

How does channel compression affect the Aspera fasp.io gateway?

IBM z15 introduced on-chip compression, which replaced the optional zEnterprise Data Compression feature on IBM z14 and earlier platforms.

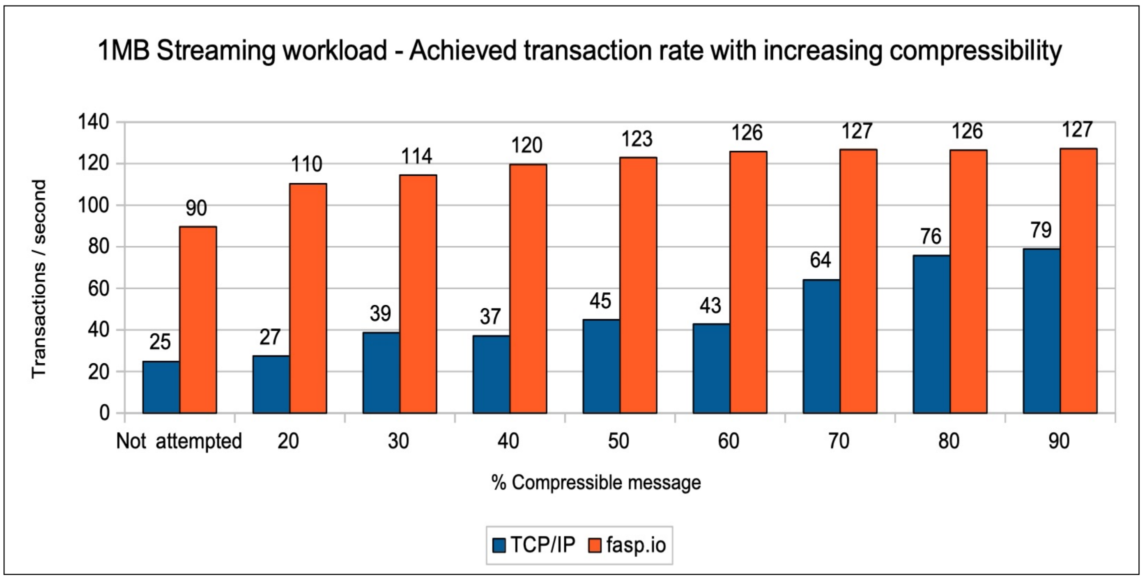
Since IBM MQ version 8.0, MQ has supported the use of hardware compression on channels, and this can be enabled by setting `COMPMSG(ZLIBFAST)`. It should be noted that if hardware compression or the data to be compressed falls below the minimum threshold for hardware compression, then compression and decompression will be performed by standard general purpose processors by software functions. Further detail relating to the performance of channel compression on z15 can be found in the IBM MQ for z/OS on z15 performance report.

One of the more common use-cases for channel compression is on high latency networks such as those where the fasp.io gateway also provides significant benefit, so can channel compression be used in conjunction with the fasp.io gateway? The answer is yes - and indeed in our measurements the threshold for seeing benefits from channel compression is lower than when compressing and sending data over networks without the gateway.

In the following examples, the 1MB streaming workload is run, where the message payload is incrementally more compressible - from 20 to 90% compressible, i.e. at its most compressible, the 1MB message results in 100KB of data being sent over the network.

For simplicity's sake, the measurements are run only on configuration N2 (25 milliseconds of latency).

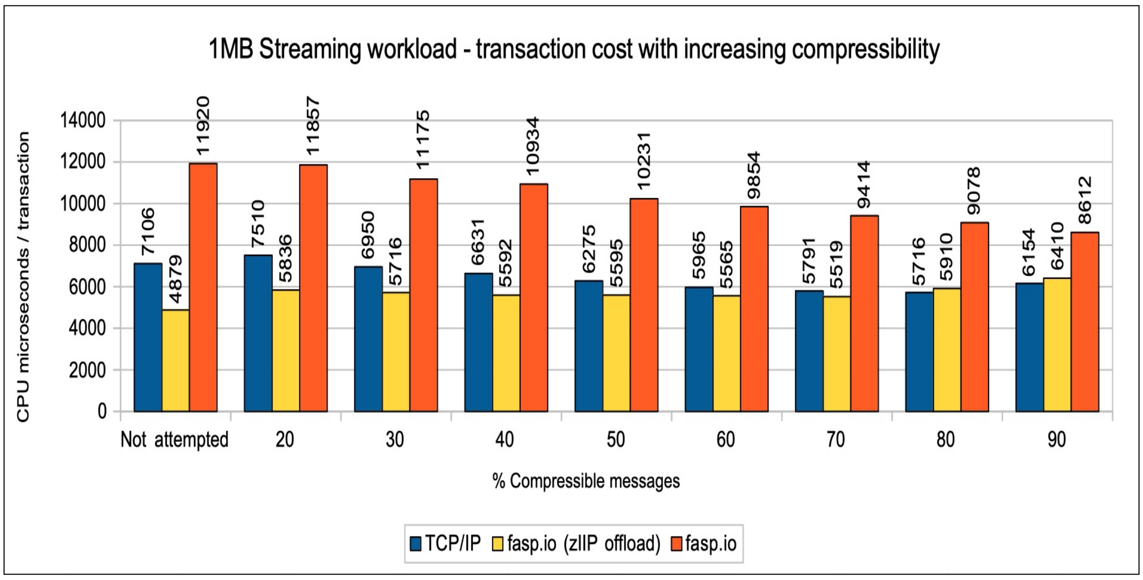
Chart: Achieved transaction rate - Streaming 1MB workload with 25 millisecond latency and increasing compressibility.



Note that the "not attempted" column is the baseline measurement where `COMPMSG(NONE)` is defined.

With regards to transaction rate, the TCP/IP measurements see a 3 times improvement in throughput with messages that are highly compressible, but even those highly compressible messages were unable to achieve more than 62% of the equivalent fasp.io measurement.

Chart: Transaction cost - Streaming 1MB workload with 25 millisecond latency and increasing compressibility.



Notes on transaction cost chart:

The transaction cost is calculated using the data from the RMF Workload report and includes the costs for the MQ MSTR, channel initiator, zCX, TCP/IP and application address spaces.

The costs reported for the "fasp.io (zIIP offload)" columns are based on the reported zIIP-eligibility for those address spaces and are the optimum costs based on those calculations.

Appendix A

System configuration

IBM MQ Performance Sysplex running on z15 (8561-7J1) configured thus:

LPAR 1: 1-32 dedicated CP processors, plus 2 zIIP, 144GB of real storage.

LPAR 2: 1-3 dedicated CP processors, 44GB of real storage.

LPAR 3: 1-10 dedicated CP processors, plus 2 zIIP, 44GB of real storage.

Default Configuration:

3 dedicated processors on each LPAR, where each LPAR running z/OS v2r4 FMID HBB77C0.

Coupling Facility:

- Internal coupling facility with 4 dedicated processors.
- Coupling Facility running CFCC level 24 service level 00.18.
- Dynamic CF Dispatching off.
- 3 x ICP links between each z/OS LPAR and CF

DASD:

- FICON Express 16S connected DS8870.
- 4 dedicated channel paths (shared across sysplex)
- HYPERPAV enabled.

System settings:

- zHPF disabled by default.
- HIPERDISPATCH enabled by default.
- LPARs 1 and 3 configured with different subnets such that tests moving messages over channels send data over 1/10GbE performance network.
  - SMC-R enabled by default between LPARs 1 and 3.
  - SMC-D enabled by default between LPARs 1 and 3.
- zEDC compression available by default - used with MQ channel attribute COMPMSG(ZLIBFAST).
- Crypto Express7 features configured thus:

  - 1 x Accelerator, shared between LPARs 1,2 and 3.
  - 2 x Coprocessor on LPAR1.
  - 1 x Coprocessor on LPAR2.
  - 2 x Coprocessor on LPAR3.

IBM MQ trace status:

- TRACE(GLOBAL) disabled.
- TRACE(CHINIT) disabled.
- TRACE(S) CLASS(1,3,4) enabled where supported.
- TRACE(A) CLASS(3,4) enabled where supported.

IBM MQ queue manager tuning:

- MQ Knowledge Center section "Planning your channel initiator" discusses the ability to force MQ to use specific TCP buffer sizes.
- Apply using commands in CSQINP2 or directly issuing the commands to the queue manager:
  - # Set the TCP buffers to the maximum size
  - RECOVER QMGR (TUNE CHINTCPRBDYNSZ 2091752)
  - RECOVER QMGR (TUNE CHINTCPSBDYNSZ 2091752)

zCX configuration

- 2 zCX systems defined, one on LPAR1 and one on LPAR3.
- Memory: 12GB - initially configured with 4GB but this showed some swapping under load. Using 4K pageable storage as the system was not under load and these micro-benchmarks did not see particular benefit from large frames.
- Storage: 20GB - as reported by `df -h .`
- CPU: 4 - zIIP available in both single and multi-thread mode (MT=1 and MT=2).
- Network: 1GbE

General information:

- Client machines:
  - 2 x IBM SYSTEM X5660 each with 12 x 2.6GhZ Processor, 32GB memory
- Client tests used a 1/10GbE performance network.
- Other IBM products used:
  - IBM CICS TS 5.5.
  - Db2 for z/OS version 12.
  - IMS 15.
  - IBM MQ Advanced for z/OS VUE version 9.2 with latest service applied as of December 2020.

Appendix B

TCP/IP profile changes

When configuring our systems to use the fasp.io gateway over high latency networks, there were a number of TCP profile changes that we were encouraged to make to benefit streaming workloads.

These include the following changes to the SYS1.TCPPARMS(PROFxx) profiles.

IPCONFIG	
SEGMENTATIONOFFLOAD	Specifies whether the stack should offload TCP segmentation for IPv4 packets to OSA-Express features. The TCP segmentation offload support transfers the overhead of segmenting outbound data onto the individual packets to QDIO-attached OSA-Express devices that support this function.
TCPCONFIG	
TCPMAXRCVBUFRSIZE 2048K	Set to the maximum of 2MB.
TCPMAXSENDBUFRSIZE 2048K	Set to the maximum of 2MB.
TCPSENDBUFRSIZE 2048K	Set to the maximum of 2MB.
TCPRCVBUFRSIZE 2048K	Set to the maximum of 2MB.
INTERFACE GE1N	Interface dedicated link to Netropy
INBPERF DYNAMIC WORKLOAD VMAC	Determine how processing of inbound traffic for QDIO is performed. **DYNAMIC** setting causes host stack to dynamically adjust the timer-interrupt values while the device is active and in use. **WORKLOADQ** specifies that QDIO inbound workload queuing (IWQ) is enabled for inbound traffic. **VMAC** is required with the WORKLOADQ setting to enable QDIO inbound workload queuing.
BEGINROUTES	
ROUTE x HOST y GE1N MTU 1492	Add a static route to the remote zCX via the dedicated Netropy link.

Appendix C

ENOBUFS errors

When performing the initial testing of the Aspera fasp.io gateway running in zCX, there were intermittent MQ channel errors reported.

Upon reviewing the SYSLOG at the time of the channel error, the following messages were observed:

```
09:18:56.67 STC05397 00000080 IVT5592I CSM FIXED STORAGE AT CONSTRAINED LEVEL
09:18:56.71 STC05397 00000080 IVT5565I CSM FIXED STORAGE SHORTAGE RELIEVED
09:18:57.76 STC62854 00000090 +CSQX206E @VTS1 CSQXRCTL Error sending data, 283
                 283 00000090 channel VTS1_VTS2_0001
                 283 00000090 connection 9.20.8.78
                 283 00000090 (queue manager VTS2)
                 283 00000090 TRPTYPE=TCP RC=00000462 (ENOBUFS) reason=76690324
```

The `IVT5592I` message is written when CSM FIXED memory usage has exceeded 80% of the CSM FIXED storage limit and is approaching the 85% constrained level.

Use the `D NET,CSM` command to determine current storage limits and usage, e.g.

```
IVT5536I TOTAL ALL SOURCES 19284K 29120K 48404K
IVT5538I FIXED MAXIMUM = 240M FIXED CURRENT = 42537K
IVT5541I FIXED MAXIMUM USED = 42537K SINCE LAST DISPLAY CSM
IVT5594I FIXED MAXIMUM USED = 212065K SINCE IPL
```

In this example, we are using 207MB out of the 240MB available, i.e. 86% of the maximum. Unfortunately it was not possible to predict how much CSM FIXED storage would be required, and we attempted both 360MB and 480MB before settling on 1GB - and the maximum usage reached 427MB for our workloads.

Note: the guidance is that the maximum used is 50% or less than the maximum available.

To increase the CSM FIXED storage, there are 2 options:

1. Dynamically - issue `MODIFY NET,CSM,FIXED=xxxM` to alter the fixed storage.
2. For a persistent change, update the `SYS1.PARMLIB(IVTPRM00)` member and set the attribute `FIXEDM MAX(xxxM)` accordingly. This change will require an IPL to be activated.

Appendix D

Guest Memory Page Frame Size

zCX APAR **OA59573** available as of September 30, 2020:

- Provides the ability to back the Linux memory with 2G fixed, 1M fixed or 4K fixed pages.
    - Larger pages reduce Translation Lookaside Buffer misses and Translation Table sizes.
    - 2G and 1M pages save about 8MB of Translation Table space for every 2GB of memory.
    - This is not Linux Hugepage support - Linux Hugepages are not currently supported.
- Has additional performance improvements.

The zCX Performance Considerations (GSE 2020) document offers the following guidance on how to choose a page size:

- 2G fixed pages
  - Best performance:
    - Reduces TLB misses and page table storage as one 2GB page contains 524,288 4K pages and 2048 1MB pages.
  - Least flexible:
    - The storage is pre-allocated at IPL time via the 2G LFAREA parameter and cannot be used for any storage pool.
    - If storage consumption is an issue, then it may not be the best choice for instances that come and go, as others may not be able to use the 2G memory.
    - Cannot be used when z/OS is a z/VM guest.
- 1M fixed pages
  - Improved performance over 4K but noticeably less than 2G.
    - Reduces TLB misses and page table storage as a 1M page contains 256 4K pages.
    - On z/VM, provides a dramatic performance improvement over 4K frames and is highly recommended.
  - Good flexibility:
    - Good choice for appliances that come and go as storage can be reused as 4K if needed.
    - 1M LFAREA is only a maximum value as there is no dedicated 1M page pool.
- 4K frames
  - Worst performance - not recommended as a first choice unless a sandbox appliance.
  - Best availability - add as a second choice in case your first choice is unavailable.
- Recommendations:
  - Use 2G pages for appliances that are always up or when reusing memory is not an issue.
  - Pick of combination of sizes starting with your first choice and the system will use the first one that is available (i.e. 2G, 4K).
  - Automate for cases where the best size was not available but should be.

Note: In the measurements in this document, we used a variety of page frame sizes and saw little benefit, however the environment for each of the measurements was such that only the minimum required containers were running.

Appendix E

Monitoring of zCX environment

It is important to monitor your zCX appliance to ensure the best performance is achieved.

Avoid Linux Swapping

If the containers inside a zCX instance needs more virtual memory that expected, paging to the defined swap disks takes place on the Linux level, inside the zCX instance. This might lead to performance impacts.

Use of the "`free -ht`" command can quickly show whether there is any use of the swap disks e.g.

```
                total       used       free     shared  buff/cache   available
Mem:             7.6G       3.7G       152M       670M        3.8G        3.1G
Swap:            1.9G        53M       1.9G
Total:           9.5G       3.7G       2.0G
```

In this example, there is only a small level of swapping, but this was resolved by increasing the amount of memory available to the zCX appliance.

If you have configured a separate storage group for the zCX swap data set, the RMF DASD report will show activity against that storage group if Linux swapping is occurring. Alternatively use of the SMF 42 subtype 6 data can show the zCX SWAP data set performing I/O.

Monitoring utilities

For general monitoring of the appliance, we enabled system monitoring by following the instructions in:

- zCX Monitoring with Grafana
- How to monitor multi zCX appliances using Grafana

This supercedes the guidance in IBM Redbooks® "Getting started with z/OS Container Extensions" section 7.6 "Configuring Grafana to monitor zCX Containers".

This meant installing the following four components in separate containers:

- **Node-Exporter** which exposes metrics about the Linux operating system.
- **cAdvisor** exposes metrics about containers.
- **Prometheus** collects the data of the preceding components.
- **Grafana** visualises the data that it pulls from Prometheus.

These images are not insignificant in size, approximately 2.4GB at the time of writing, and you should ensure that you have sufficient space to install and run these components in addition to the applications to be monitored.

The IBM zCX for z/OS Monitoring dashboard was then added to monitor the MQ containers.

Note: Running the monitors full-time with the default options, used between 2 to 4 CPU seconds per 60 second interval and as such we would suggest that the once initial evaluation of the MQ configuration is complete, unless there is specific reason to monitor the appliance, that the monitoring containers are stopped.

Alternatively the cAdvisor container can be configured to reduce the polling interval, which will reduce the cost. For example using the `--housekeeping_interval=10s` option reduced the cost of running the monitor to approximately 0.6 CPU seconds per 60 second interval.

Appendix F

Future configurations

The following list mentions performance options that have not been tested but *may* offer some benefit:

- Software striping of ZCX DATA dataset.
- Pervasive encryption of configuration, user data, diagnostics data, VSAM linear data sets and zCX instance directory zFS file system.
  - Note that by default, zCX instance root file systems and swap data volumes that are backed by VSAM linear data sets are encrypted by Linux, based on LUKS encryption. Pervasive encryption is not recommended for these linear VSAM data sets.

Appendix G

Resources

To find further information on z/OS container extensions and the use cases with IBM MQ for z/OS, the following resources are available:

- IBM Knowledge Center - Container Extensions.
- zCX Container Extensions guide.
- IBM Redbooks® "Getting started with z/OS Container Extensions".
- zCX Performance Considerations (GSE 2020).
- Blog - "Use cases for MQ in z/OS Container Extensions".
- IBM Redbooks® "IBM z/OS Container Extensions (zCX) use cases".
- Blog - Using client concentrators with MQ for z/OS.
- Blog - Cost of connecting to a z/OS queue manager.
- Report - Persistent messaging performance in IBM MQ for Linux.