How to maximise your single-threaded messaging rate

Objective

IBM MQ is a highly optimised, multi-threaded message broker able to service many concurrent connections. To achieve the best messaging throughput the general advice has always been to avoid the use of single-threaded applications, but we have had several similar enquiries recently that have commented the following:

- "I want the highest messaging rate possible"
- "I can't run more than a single thread, the application is built that way"
- "Why can't MQ match the messaging rate of xyzMQ?"

This report will discuss some configuration and tuning that will provide some answers to those questions; detailing what the performance benefit will be and what compromises to application design might have to be made to achieve those benefits.

Background

The vast majority of the MQ performance reports across distributed and appliance platforms feature a request/response scenario. This type of scenario is frequently used by our customers and provides repeatable results that can be recreated by customers using similar hardware and the configuration provided. The throughput results are measured in Round trips per second, with each round trip requiring two message PUTs and two message GETs. Some messaging provider reports that feature msg/sec headline values would need to be divided by 4 to be comparable with results that IBM publish as round trips/s.

For the purposes of this report, we will be looking at the PUT (send) rate of a producer application sending 2K Non Persistent messages to a single queue. A GETting application consisting of 2 threads will be run local to the QM(using local bindings mode) to consume all messages sent and avoid growing the queue depth during the tests.

The client runs on a single xLinux server, the QM runs on a separate xLinux server connected via a 10Gb network. No latency was added between the client and the QM. Detailed specifications can be found in Appendix A.

The test client for both sending and receiving is CPH (https://github.com/ibm-messaging/mq-cph). The configuration changes (to both QM.ini and cph) will be shown in the testing below, and how to monitor the QM to determine which connection mechanism is in use.

FASTPATH binding, client binding, remote binding, bindings mode, server binding, standard binding

The MQ documentation in this area can be a little confusing and we have multiple overlapping names that make it difficult to accurately describe the scenario and the configuration required to optimize the performance of that scenario. I will try and step through the different scenarios and how they are configured.

a) Standard binding (remote client)

The application runs remote from the QM and communicates over the network with a Message Channel Agent(MCA) process that resides on the QM node. The MCA process then communicates using an IPC(Inter Process Communication) with the QM to perform the tasks required by the client.

No QM tuning configuration required. No cph switches required.

1 thread throughput: 8,900 msg/s

runmqsc output:

```
dis conn(*) all where(APPLTAG eq 'cph')
AMQ8276I: Display Connection details.
   CONN(E358A66500105240)
   EXTCONN(414D5143504552463020202020202020)
   TYPE(CONN)
   PID(9464)                               TID(9)
   APPLDESC(IBM MQ Channel)                APPLTAG(cph)
   APPLTYPE(USER)                          ASTATE(NONE)
   CHANNEL(SYSTEM.DEF.SVRCONN)             CLIENTID( )
   CONNAME(10.20.36.146)                   CONNOPTS(MQCNO_SHARED_BINDING)
   USERID(mqperf)                          UOWLOG( )
   UOWSTDA( )                              UOWSTTI( )
   UOWLOGDA( )                             UOWLOGTI( )
   URTYPE(QMGR)
   EXTURID(XA_FORMATID[] XA_GTRID[] XA_BQUAL[])
   QMURID(0.0)                             UOWSTATE(NONE)
```

b) FASTPATH binding (remote client)

The first tuning configuration that should be applied for most queue managers is to enable FASTPATH binding for the messaging channels. This is the default on the MQ appliance. This allows the MCA (Message Channel Agent) process that handles the incoming work at the QM, on behalf of the remote client to share the same memory space as the QM. The only reason that you wouldn’t enable this configuration is if you wanted to run user defined channel exits and had concerns about their robustness.

Set by editing qm.ini and restarting the QM:

```
Channels:
        MQIBindType=FASTPATH
```

No cph switches required.

1 thread throughput: 11,700 msg/s

runmqsc output:

```
dis conn(*) all where(APPLTAG eq 'cph')
     1 : dis conn(*) all where(APPLTAG eq 'cph')
AMQ8276I: Display Connection details.
   CONN(3672A665002C0040)
   EXTCONN(414D5143504552463020202020202020)
   TYPE(CONN)
   PID(11777)                              TID(3)
   APPLDESC(IBM MQ Channel)                APPLTAG(cph)
   APPLTYPE(USER)                          ASTATE(NONE)
   CHANNEL(SYSTEM.DEF.SVRCONN)             CLIENTID( )
   CONNAME(10.20.36.146)                   CONNOPTS(MQCNO_FASTPATH_BINDING)
   USERID(mqperf)                          UOWLOG( )
   UOWSTDA( )                              UOWSTTI( )
   UOWLOGDA( )                             UOWLOGTI( )
   URTYPE(QMGR)
   EXTURID(XA_FORMATID[] XA_GTRID[] XA_BQUAL[])
   QMURID(0.0)                             UOWSTATE(NONE)
```

c) Standard binding (local client – client binding)

With the application co-located on the same host as the QM, you can still use the client binding as per the scenario (a) and as before the application would communicate with the MCA over the network stack, the MCA would then communicate via IPC with the QM.

You wouldn't ordinarily run this scenario, because with the application co-located with the QM, a direct 'bindings mode' or 'standard binding' can be used which will always be the more performant configuration as it removes the need for any networking, the MCA process or IPC between MCA and the QM (see sections e and f).

No QM tuning configuration required. No cph switches required.

1 thread throughput: 36,300 msg/s

runmqsc output (as section (a) - though CONNAME will reflect IP address of QM as client is co-located)

d) FASTPATH binding (local client – client binding)

With the application co-located on the same host as the QM, you can still use the client binding as per the previous section and with FASTPATH binding enabled, the MCA again shares the same memory space as the QM.

You wouldn't ordinarily run this scenario, because with the application co-located with the QM, a direct 'bindings mode' or 'standard binding' can be used which will always be the more performant configuration as it removes the need for any networking or MCA process (see next sections).

Set by editing qm.ini and restarting the QM:

```
        Channels:
                MQIBindType=FASTPATH
```

No cph switches required.

1 thread throughput: 41,900 msg/s

runmqsc output (as section (b) - though CONNAME will reflect IP address of QM as client is co-located)

e) Standard binding (local client – standard binding or ‘bindings mode’)

With the application co-located on the same host as the QM, this option allows the application to connect directly to the QM using IPC but the application does not share the same memory space as the QM.

No QM tuning configuration required.

CPH uses the flag `-jt mqb` to enable this mode.

1 thread throughput: 64,700 msg/s

runmqsc output:

```
dis conn(*) all where(APPLTAG eq 'cph')
     3 : dis conn(*) all where(APPLTAG eq 'cph')
AMQ8276I: Display Connection details.
   CONN(9D77A66500530040)
   EXTCONN(414D5143504552463020202020202020)
   TYPE(CONN)
   PID(12508)                              TID(1)
   APPLDESC( )                             APPLTAG(cph)
   APPLTYPE(USER)                          ASTATE(NONE)
   CHANNEL( )                              CLIENTID( )
   CONNAME( )                              CONNOPTS(MQCNO_SHARED_BINDING)
   USERID(mqperf)                          UOWLOG( )
   UOWSTDA( )                              UOWSTTI( )
   UOWLOGDA( )                             UOWLOGTI( )
   URTYPE(QMGR)
   EXTURID(XA_FORMATID[] XA_GTRID[] XA_BQUAL[])
   QMURID(0.0)                             UOWSTATE(NONE)
   CONNTAG(MQCT9D77A66500530040PERF0_2024-01-16_10.13.31cph)
```

Note how there is no CHANNEL or CONNAME defined.

f) FASTPATH binding (local client – fastpath binding)

With the application co-located on the same host as the QM, this option allows the application to share the same memory space as the QM. You would only use this option if you had full confidence in the robustness of your application. Its configuration is defined from within the application using CNO (Connect) options and to share the same address space, the application has to run as the same user as the QM.

An MQI application needs to set:

`MQCNO.Options |= MQCNO_FASTPATH_BINDING;`

CPH uses the flag `-jf true` to enable this option. `-jt mqb` must also be set.

1 thread throughput: 90,500 msg/s

runmqsc output:

```
dis conn(*) all where(APPLTAG eq 'cph')
     3 : dis conn(*) all where(APPLTAG eq 'cph')
AMQ8276I: Display Connection details.
   CONN(9D77A66500694045)
   EXTCONN(414D5143504552463020202020202020)
   TYPE(CONN)
   PID(12793)                              TID(1)
   APPLDESC( )                             APPLTAG(cph)
   APPLTYPE(USER)                          ASTATE(NONE)
   CHANNEL( )                              CLIENTID( )
   CONNAME( )                              CONNOPTS(MQCNO_FASTPATH_BINDING)
   USERID(mqm)                             UOWLOG( )
   UOWSTDA( )                              UOWSTTI( )
   UOWLOGDA( )                             UOWLOGTI( )
   URTYPE(QMGR)
   EXTURID(XA_FORMATID[] XA_GTRID[] XA_BQUAL[])
   QMURID(0.0)                             UOWSTATE(NONE)
   CONNTAG(MQCT9D77A66500694045PERF0_2024-01-16_10.13.31cph)
```

Note that the userid of the client application had to be the same as the QM (mqm)

So you’ve now seen the performance benefits that can be achieved by enabling FASTPATH, co-locating the client with the QM and using bindings mode connections. If your clients are remote, and you wish to improve the performance of scenario (b) above, there are still some alternative methodologies that you might consider:

- Asynchronous Put
- Distributed Queuing (QM->QM message transfer)
- Client/channel compression

Asynchronous Put

Performing messaging in a single thread for a default MQI configuration requires the MQPUT to wait for a response from the QM to acknowledge message receipt before subsequent messages can be sent. This requires a full line turnaround (Msg sent followed by receipt of Ack) in between each message, and thus messaging rate is heavily dependent (and restricted) by the latency of the connection to the QM.

If an application->QM ping time is 1ms, then the theoretical maximum message rate a single thread could achieve would be 1000 msg/sec; though once you allow for application code, networking and QM processing, the realistic maximum would be much less.

Asynchronous Put can be enabled for a destination, and this allows the application to send subsequent messages without waiting for acknowledgements from the QM. This avoids the line turnaround cost but puts a cost on the application should it wish to a) determine if not all messages were received, and b) resend those messages if required.

The application can query about the number of messages received by the QM and the status of those messages by issuing an MQSTAT call. The application would have to store messages (at potentially some risk of loss) and incorporate the logic to resend any that weren’t successfully received by the QM.

Set by editing the local queue definition:

```
runmqsc QM
ALTER QLOCAL(QUEUE1) DEFPRESP(ASYNC)
```

No QM tuning configuration required. No cph switches required. This is the same scenario as section(a) with the client located remotely. The number of responders has been increased to 10 threads to ensure that the queue does not accumulate messages.

1 thread throughput: 51,000 msg/s

After enabling FASTPATH (See section (b) above), the result is:

1 thread throughput: 63,000 msg/s

As you can see, that's quite a difference when the line turnaround can be avoided. This is especially suitable for scenarios where the data being flowed is temporal and it might not matter for an odd message to be lost, as a later message would soon supersede it. This style of application would avoid having to store and potentially resend messages which didn't arrive.

Summary of results

Pulling together all of the results from the sections above, the table below shows what rate was achieved for a single threaded cph application putting 2K non-persistent messages on a single MQ queue.

Remember that these rates are what can be achieved on the hardware and infrastructure the tests were run on here, your own results will of course be different.

Scenario	**Rate**
Standard binding (remote client)	8,900 msg/s
FASTPATH binding (remote client)	11,700 msg/s
Standard binding (local client – client binding)	36,300 msg/s
FASTPATH binding (local client – client binding)	41,900 msg/s
Standard binding (local client – standard binding or 'bindings mode')	64,700 msg/s
FASTPATH binding (local client – fastpath binding)	90,500 msg/s
Asynchronous Put (remote client)	51,000 msg/s
Asynchronous Put (remote client with FASTPATH)	63,000 msg/s

I've got a high latency link between the client and QM, what are my options?

Using additional concurrent clients would normally be enough to increase throughput in high latency scenarios, as you then have multiple threads of execution sending/receiving data over the high latency link. If you are single threaded, then one way of increasing throughput is to send the messages asynchronously as discussed in the section above; the second way is to use an additional MQ QM which then delivers the message to the destination QM via distributed queuing.

Adding 500µs of delay between our client and QM, the throughput of section(b) would drop to 1650 msg/s

Enabling asynchronous put in that same scenario, allows the throughput to achieve 52,000 msg/s

If we then install a QM local to the client and configure distributed queueing to deliver the messages to the remote QM, the client can get acknowledgement quickly and leave MQ to manage the high latency transmission.

With the same amount of latency, throughput of 36,000 msg/s can be achieved (this is the delivery rate to the local QM and the consumption rate of the remote QM)

As the latency increases, you will likely need to increase the maximum depth of the transmission queue, increase the MQ batch size and/or add throttling to the client application.

Now that we have a QM local to the client again, we can use local bindings to make the client handoff even faster. This increases the throughput rate to: 57,000 msg/s but also requires the channel batch size to be increased (to 200) to ensure that messaging workload can be transmitted to the remote QM whilst avoiding excessive queueing at the transmission queue.

If the latency between the QM rises further, then additional configuration or application changes might be required. For a 2ms latency, a rate of 52,000 msg/s was achieved, with a batch size of 500; with the larger batch size increasing the average end-to-end message processing latency.

Network restriction – Use MQ client compression to maximise throughput

This would usually only be relevant to tests with concurrent workload but could apply to the single threaded case in extreme cases. MQ can apply compression to the message payload when the MQ client is communicating with a remote QM; and also between two QM in the distributed queueing topology.

Both scenarios will add a little CPU and latency as the payload is compressed and uncompressed, but where the network environment is heavily restricted, this mechanism can be used to provide an overall increase in throughput. In my testing with a repeating payload the ZLIBFAST encryption offered a good balance between speed and compression rate, offering up to 6x throughput improvement. From MQ 9.4 an additional compression algorithm is available (LZ4FAST) which can provide further gains. The performance gain you receive will depend on whether the data can be easily compressed (text-based message data such as XML and JSON will compress better than binary data), and how restrictive the network bottleneck is that you are trying to overcome.

Conclusions

In most scenarios MQ performs well with little or no tuning required by application users. In scenarios that inherently limit performance (single/low threadedness, high latency, network bottlenecks); some reconfiguration of your scenario/topology may be required to achieve specific SLA (Service Level Agreement) requirements.

This document discusses a number of approaches to improving performance in those scenarios, how users could test these topologies (using open source tooling – see Appendix B) and apply them to their own environments.

Author

The author of this whitepaper is Sam Massey who works in the MQ Performance Team at the IBM UK Laboratory, Hursley. If you have any questions or comments on this paper, please contact him at smassey@uk.ibm.com

Appendix A

The three machines used for the performance tests in this report have the following specification:

Category	Value
Machine	x3550 M5
OS	Red Hat Enterprise Linux Server 7.3
CPU	2x12 (2.6Ghz)
RAM	128GB RAM
Network	10Gb/40Gb Ethernet
Disks	2x 480GB SSD
RAID	ServeRAID M5210 (4GB Flash RAID cache)

Appendix B

The tooling used to generate this report is available here: https://ibm-messaging.github.io/mqperf/testharness.html