Publish-Subscribe Performance Report

IBM MQ V8

Version 1 — April 2015

Rowan Lonsdale

IBM MQ Performance
IBM UK Laboratories
Hursley Park
Winchester
Hampshire

Please take Note!

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

Version 1 — April 2015

This version applies to *IBM MQ V8* (and to all subsequent releases and modifications until otherwise indicated in new editions).

Note to U.S. Government Users
Documentation related to restricted rights.
Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule contract with IBM Corp.

Notices

DISCLAIMERS
The performance data contained in this report was measured in a controlled environment. Results obtained in other environments may vary significantly.

You should not assume that the information contained in this report has been submitted to any formal testing by IBM.

Any use of this information and implementation of any of the techniques are the responsibility of the licensed user. Much depends on the ability of the licensed user to evaluate the data and to project the results into their own operational environment.

WARRANTY AND LIABILITY EXCLUSION
The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

In Germany and Austria, notwithstanding the above exclusions, IBM's warranty and liability are governed only by the respective terms applicable for Germany and Austria in the corresponding IBM program license agreement(s).

ERRORS AND OMISSIONS
The information set forth in this report could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; any such change will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time and without notice.

INTENDED AUDIENCE

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

LOCAL AVAILABILITY

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

ALTERNATIVE PRODUCTS AND SERVICES

Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

USE OF INFORMATION PROVIDED BY YOU

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

TRADEMARKS AND SERVICE MARKS

The following terms used in this publication are trademarks of their respective companies in the United States, other countries or both:

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

Other company, product, and service names may be trademarks or service marks of others.

EXPORT REGULATIONS

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

Preface

Target audience

The report is designed for people who:

- Will be designing and implementing publish-subscribe solutions using IBM MQ V8.
- Want to understand the publish-subscribe performance limits of IBM MQ V8.
- Want to understand what actions may be taken to tune publish-subscribe messaging in IBM MQ V8.

The reader should have a general awareness of IBM MQ in order to make best use of this report.

This report should be considered a supplement to the main IBM MQ V8 performance reports[1], as these contain many useful analyses of various aspects IBM MQ, many of which are relevant to publish-subscribe messaging as well as point-to-point. This report will not cover such aspects in detail, and instead focus on areas specific to publish-subscribe messaging.

Feedback on this report

We welcome constructive feedback on this report.

- Does it provide the sort of information you want?
- Do you feel something important is missing?
- Is there too much technical detail, or not enough?
- Could the material be presented in a more useful manner?

Specific queries about performance problems on your IBM MQ system should be directed to your local IBM Representative or Support Centre.

Please direct any comments of this nature to WMQPG@uk.ibm.com

[1] IBM MQ performance reports for various platforms can be found at: www-01.ibm.com/support/docview.wss?uid=swg27007150

Table of Contents

List of Figures

List of Charts & Tables

1 — Introduction

1.1 Publish-subscribe in MQ

While publish-subscribe is architecturally distinct from point-to-point messaging, in MQ the former is built on top of the latter. Understanding a little about how MQ implements publish-subscribe messaging can help to answer some questions about how the performance of these two types of messaging are related. In particular, it will inform how this report measures the performance of MQ: it will concentrate on aspects distinct to publish-subscribe, and you should supplement it with the main IBM MQ Performance Report for your desired platform to cover other considerations.

It is assumed that anyone reading this report is already familiar with how the basic point-to-point and publish-subscribe messaging models work. However, for clarity, we'll recap briefly:

- In **point-to-point** messaging, a message-producing application puts messages to a named destination, called a *queue*, from which one or more consuming applications can retrieve them. Each message is only retrieved once, so if more than one consuming application is reading from the queue, they will each retrieve a disjoint subset of the messages that have been put there. (Figure 1)

- In **publish-subscribe** messaging, a message-producing application (called a *publisher*) *publishes* messages on a *topic*. One or more applications may *subscribe* to that topic, and each will receive all messages that are published on it. (Figure 2)

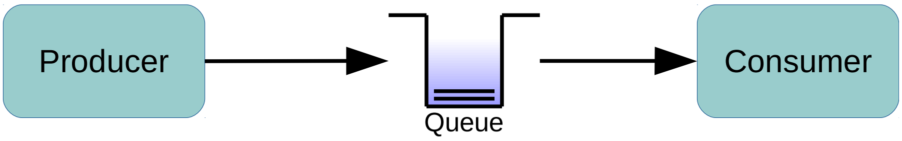

Figure 1: Point-to-point messaging model

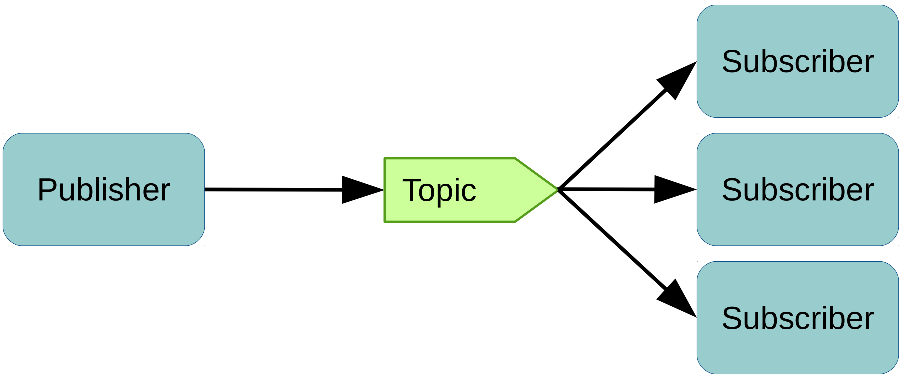

Figure 2: Publish-subscribe messaging model

In MQ, publish-subscribe messaging is facilitated by the *publish-subscribe engine*. Each subscription is given an associated *subscriber queue*, and subscribing applications retrieve messages from this queue just like in point-to-point messaging. When a message is published to a topic, the *publish-subscribe engine* determines which subscriptions are interested in that topic, and puts a copy of the message on the associated *subscriber queue* of each of them. Subscriptions may be created by applications, or alternatively, they may be created administratively in the queue manager.

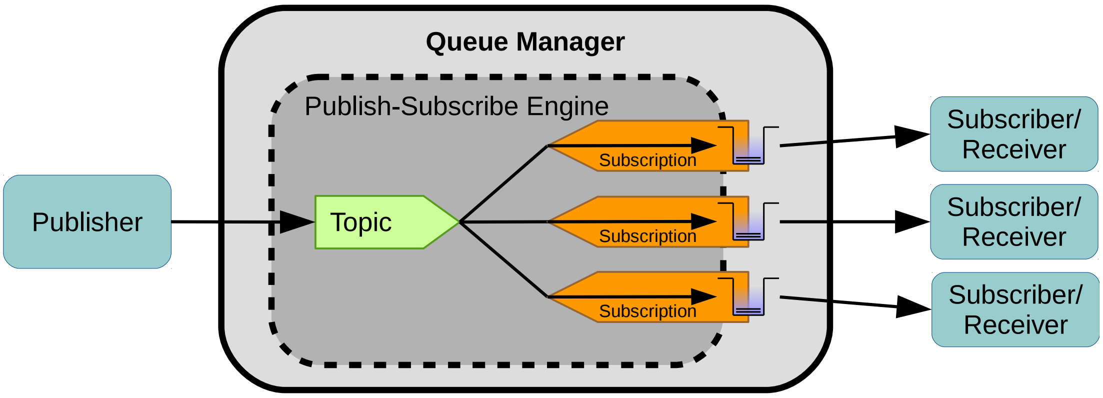

Figure 3: Publish-subscribe architecture inside MQ

This design means that many components of the message flow – such as delivery of messages to and from the queue manager, retrieval of messages, any handling of the message payload, and any persistence – are identical for both publish-subscribe and point-to-point messaging. Hence, this report will focus mainly on how the behaviour of the publish-subscribe engine (as shown in Figure 3) affects performance. Other aspects of performance are covered by the main MQ V8 Performance Reports, many of which are also relevant to publish-subscribe scenarios.

1.2 Terminology

In the results and charts in this report, several concepts will be used to describe and measure performance:

- **Publish rate –** the total number of publications per second processed by the queue manager.

- **Total message rate –** the sum of the total number of messages delivered to the queue manager per second, and the total number retrieved from the queue manager per second. For example, if a publisher were to publish to a single topic at a rate of 100 messages per second (the publish rate), and that topic had 2 subscribers, then the total message rate would be 300 messages per second.

- **Mean publish time –** the average amount of time a publisher application takes between starting one publish and starting the next. Mathematically, we have:

$$\mathit{mean\ publish\ time} = \frac{\mathit{number\ of\ publishers}}{\mathit{publish\ rate}}$$

1.3 Executive Summary

In general, publish-subscribe messaging in IBM MQ V8 is improved in virtually all areas wen compared with V7.5:

- The pure overhead cost of the publish-subscribe engine has reduced by up to 34% (section 2).

- The amount of time each subscriber adds to the length of a publish has decreased by around 10% in all scenarios, leading to a 10-14% increase in the total message rate achievable using a single publisher (section 3).

- In scenarios where publish-subscribe messaging can be compared with point-to-point, we see that for local, non-persistent messaging publish-subscribe is about 35% slower than point-to-point, and uses about 50% more CPU. However, for persistent messaging, publish-subscribe is only 14% slower, and uses 17% more CPU, and with clients communicating remotely over the network, there is virtually no difference between publish-subscribe and point-to-point (section 4).

- IBM MQ V8 also sees big improvements in non-persistent messaging of up to 25%, and up to 44% in persistent messaging, when compared with V7.5, which benefits publish-subscribe, but also point-to-point (section 4).

For a more detailed summary of these results, see Chart 6 on page 18, Chart 15 on page 28, and Chart 23 on page 34.

The remaining sections of this report present a number of studies which, as well as illustrating the results above, exemplify some of the constraining factors and pitfalls one might come across when implementing a publish-subscribe infrastructure using IBM MQ, and provides guidance on how these can be mitigated or avoided.

- Section 2 looks purely at the cost of delivering messages to a queue manager for publishing.

- Section 3 considers the effect the number of subscribers has on how long each publication takes, and hence the rate at which a publishing application can operate.

- In section 4, publish-subscribe messaging is compared directly with point-to-point, and we also consider the CPU resources required for large-volume publish-subscribe messaging.

- Section 5 looks at how publish-subscribe messaging using the JMS protocol compares with using IBM MQ's own native MQI interface.

- Finally, in section 6, we briefly look at publish-subscribe messaging in an IBM MQ cluster, and the new publish-subscribe clustering feature in V8: routed topics.

2 — The Pure Cost of Publishing

In publish-subscribe scenarios, it may often be the case that there are applications publishing to topics for which there are no subscribers. It's reasonable to ask what the cost of such 'null-publishes' is, in order to establish whether a large number of 'uninteresting' messages flowing through a queue manager might hamper the delivery of those messages that are actually subscribed to. Also, as illustrated in section 1.1 and Figure 3, the publication itself is the most significantly different part of publish-subscribe messaging in MQ that differs from point-to-point. Other aspects, such as delivery of messages to and from subscriber queues use the same operations as point-to-point messaging, and so have many of the same performance characteristics.

In this section, we shall look at scenarios in which there are publishers, but no subscribers to any topics. This eliminates all queuing, retrieval, and message retention, either in memory or on disk; the only work the queue manager has to do is establish that the topic being published on has no subscribers, and discard the message. In all scenarios, we'll start with a single publishing application, and incrementally add additional applications to see the effect this has on the system. Each application will publish all its messages to the same topic, and all applications will publish messages as fast as they can, with no sleep or "think time" between publishes.

Due to the presence of subscribers, most real-life publish-subscribe scenarios will not be able to achieve the message rates illustrated in this section. However, these rates can be viewed as absolute limits to publication rates, irrespective of the situation with regards to subscribers, and also serve as an indication of the cost of publish-subscribe in MQ, compared with attempting to achieve the same messaging architecture using point-to-point (which would normally be very complicated, and as we shall see, unlikely to perform better).

2.1 Sharing a single topic

To start with, we'll consider the case where every application publishes to the same topic:

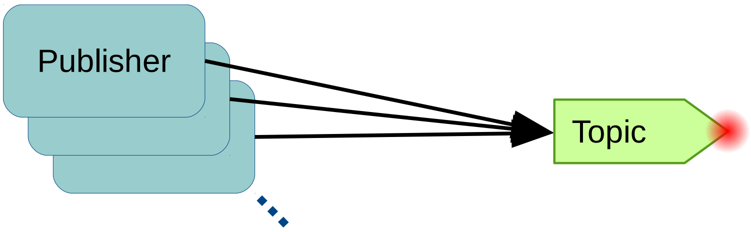

Figure 4: Null publish to a single topic

While most real-world scenarios will have more than one topic being published on, from a performance perspective, the single-topic case is a good representation of most typical topic spaces (see section 2.2.1).

2.1.1 Remote client-bound

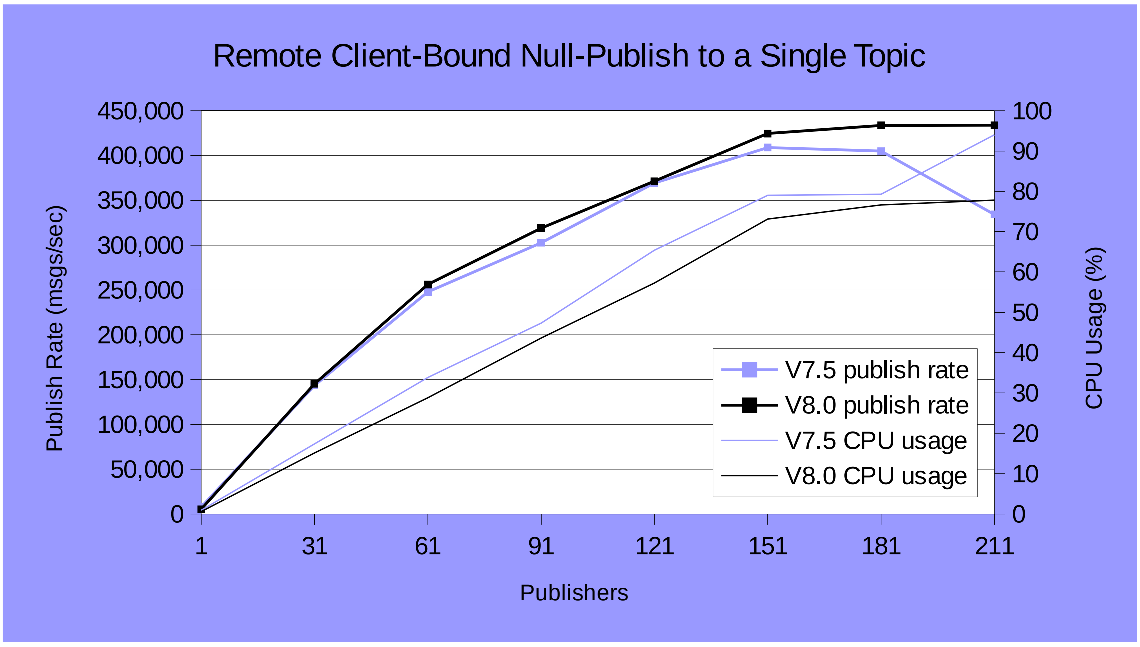

Chart 1: Remote Client-Bound Null-Publish to a Single Topic

Here, the publishing applications are located on a different machine to the queue manager, and send messages over the network.

Chart 1 shows that IBM MQ V8 has improved marginally compared with V7.5. At it's peak, V8 achieves a publishing rate of 433,851 messages per second, spread over 211 publishing applications. This rate equates to an approximate data-rate across the network of 8.3Gb/s, which is about the maximum one could expect to realistically achieve across a 10Gb/s network.

A large number of publishers is needed to achieve the peak rate, and the concave curve of the graph indicates that adding more publishers lowers the mean message rate per publisher. This is because each additional publisher brings more traffic (and hence congestion) to the network, increasing the chance that a publish operation will have to queue on the network subsystem (and the average length of time spent queueing) before being handled.

2.1.2 Local standard-bound

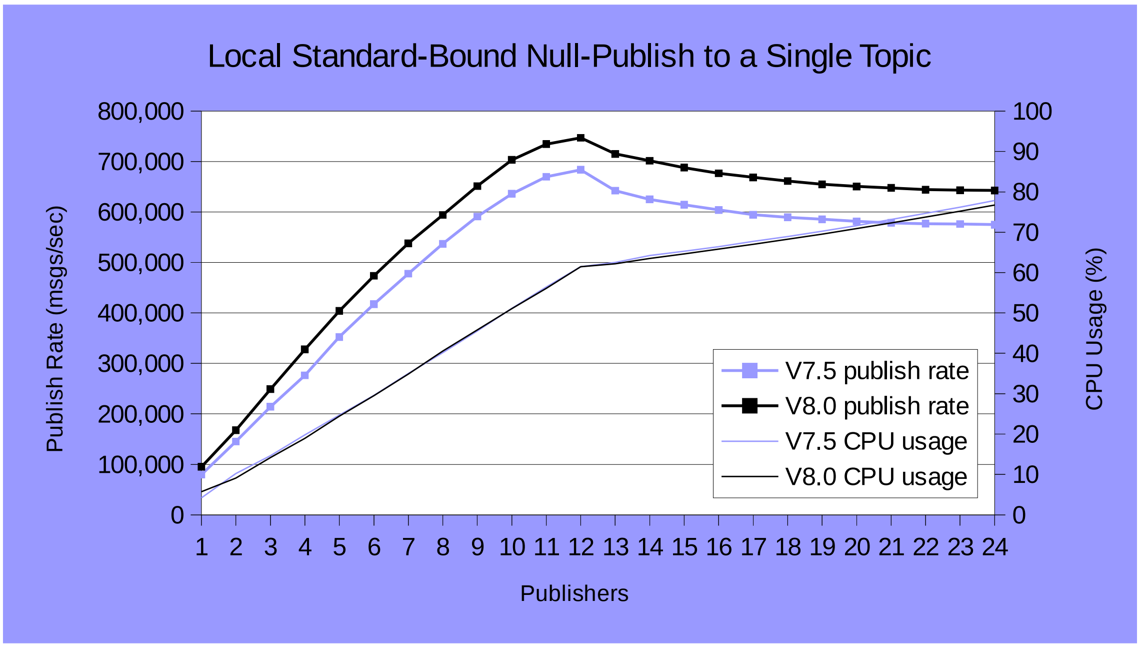

Chart 2: Local Standard-Bound Null-Publish to a Single Topic

In this scenario, all publishers are running on the same machine as the queue manager, and connect using “standard” bindings.

Once again, we see that IBM MQ V8 is improved over V7.5, with an increased publish rate for virtually identical CPU cost. As one might expect, the rates achieved here are considerably higher than those in the client-bound scenario above.

2.1.2.1 Non-uniform memory access

We can see that above 12 publishers, the total rate achieved starts to drop, while the CPU usage continues to rise (albeit at a lower rate of increase). This is due to a combination of the way standard-bound applications communicate with the queue manager, and the hardware configuration of the test machine. Modern servers with large numbers of CPU cores cannot provide sufficiently fast access to all of the RAM for all of the CPUs. Instead the RAM is split into a number of zones that can each be accessed quickly by a subset of the CPUs. Accessing RAM that is not in a CPU's local zone takes longer. This concept is called *non-uniform memory access*, or *NUMA* for short. It is expected that typically, a server will be performing several fairly distinct tasks in parallel, each containable within a single NUMA zone, so this architecture should not be a problem.

The machine being used in this test has 2 NUMA zones, each containing half the RAM and 12 CPU cores (see appendix A.1.1). We can see from Chart 2 that with 12 publishers, just over half the CPU resource has been used, and the message rate has increased almost linearly up to this point. This indicates that the CPU resource of one NUMA zone has been saturated, with each publisher, along with its associated work done by the queue manager, efficiently making use of a single core.

With more than 12 publishers, there is no more CPU available in a single NUMA zone, so the scenario must be split across two. Standard-bound applications communicate with agent processes in the queue manager using shared memory, which must necessarily reside in one NUMA zone or the other. Some processes now take longer to access this memory, and the operating system tries to correct this by moving them to different cores, pushing out other processes which then suffer the same issue. This additional context switching leads to the drop in message rate and further increase in CPU usage seen.

It is not known whether operating systems will become better at handling this sort of situation in the future. It is certainly exceptional, as it involves long-running, sequential threads of execution split over multiple processes, with each process operating for a very short period before handing execution to another. If a scenario suffering from this issue is very predictable and constant, it may be possible to carefully arrange processes on the machine to alleviate it. However, we can avoid it entirely by switching to *fastpath* application bindings, provided that the applications are robust and stable.

2.1.3 Local fastpath-bound

Applications that connect to a queue manager using fastpath bindings (by setting *MQCNO_FASTPATH_BINDING* on the *MQCNO.Options* field) don't have an associated agent process in the queue manager that needs to be communicated with via shared memory. Instead, they have direct access to the queue manager's internal functions, so all messaging operations are performed in the application's own thread. As well as eliminating the communication barrier described in 2.1.2.1, this leads to a much reduced overall path length for messaging operations, including fewer memory copies.

The disadvantage of fastpath bindings is that direct addressability of the queue manager's memory objects brings with it the potential to corrupt the queue manager's state, thus affecting other connected applications. For this reason, it's imperative to ensure that applications that connect using fastpath bindings are robust, handle errors and exceptions well, and always disconnect properly from the queue manager before terminating. This is why by default, applications use standard bindings.

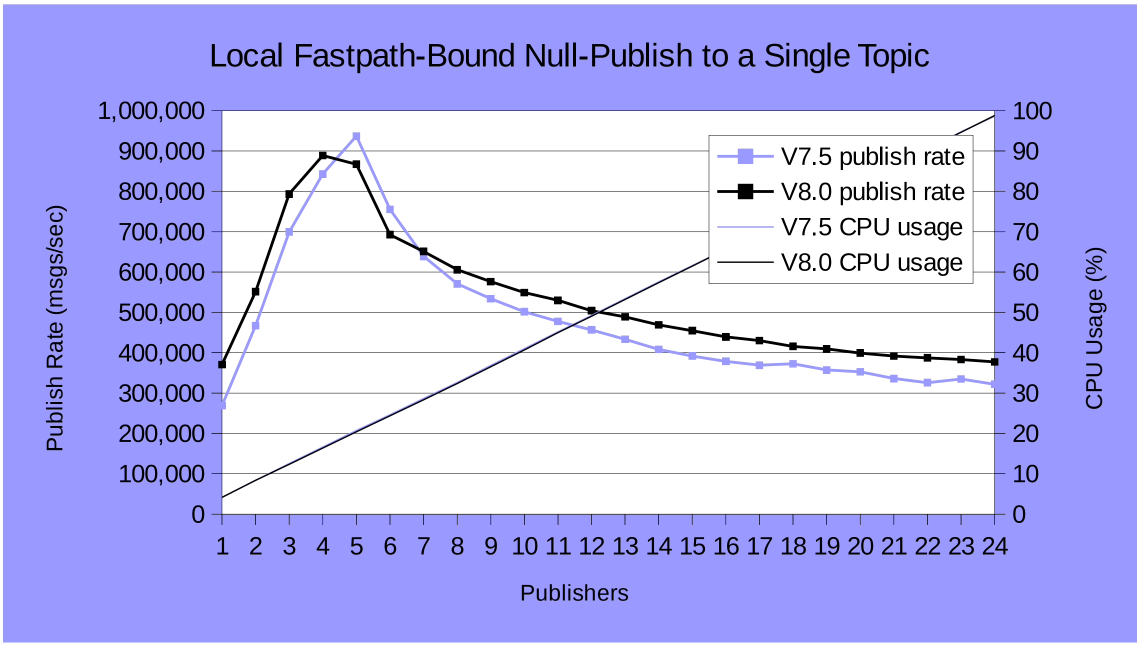

Chart 3: Local Fastpath-Bound Null-Publish to a Single Topic (Publish Rate & CPU)

Chart 3 shows that with fastpath bindings, the peak publish rate is once again improved compared with standard bindings. It also occurs with fewer publishers (4 or 5, as opposed to 12). However, after the peak has been reached, the throughput goes into an even greater decline than in Chart 2, and CPU usage continues to rise at the same rate, with each publisher apparently saturating 1 core.

2.1.3.1 Topic lock

Every time a message is published on a topic, the publisher (or its queue manager agent, for standard-bound applications) has to look up certain properties about the topic, in order to establish what subscriptions to deliver the message to, and how they should be delivered. This must be done with exclusive access, in order to maintain the integrity of certain accounting data. This exclusivity is provided by the *topic lock*, and one such lock exists for each administratively defined topic in the hierarchy.

Topic lock contention is the cause of drop the in publish rate seen in Chart 3. As the lock becomes contended, publishers are not only having to wait for it to become free, but are also spending additional time negotiating, obtaining, and releasing the lock itself. CPU usage continues to increase because, as these locks are in reality waited on for very short periods of time, they are spin-locks.

These differences in time spent holding the lock can be seen more clearly if we invert the data in Chart 3, and look at the mean publish time instead.

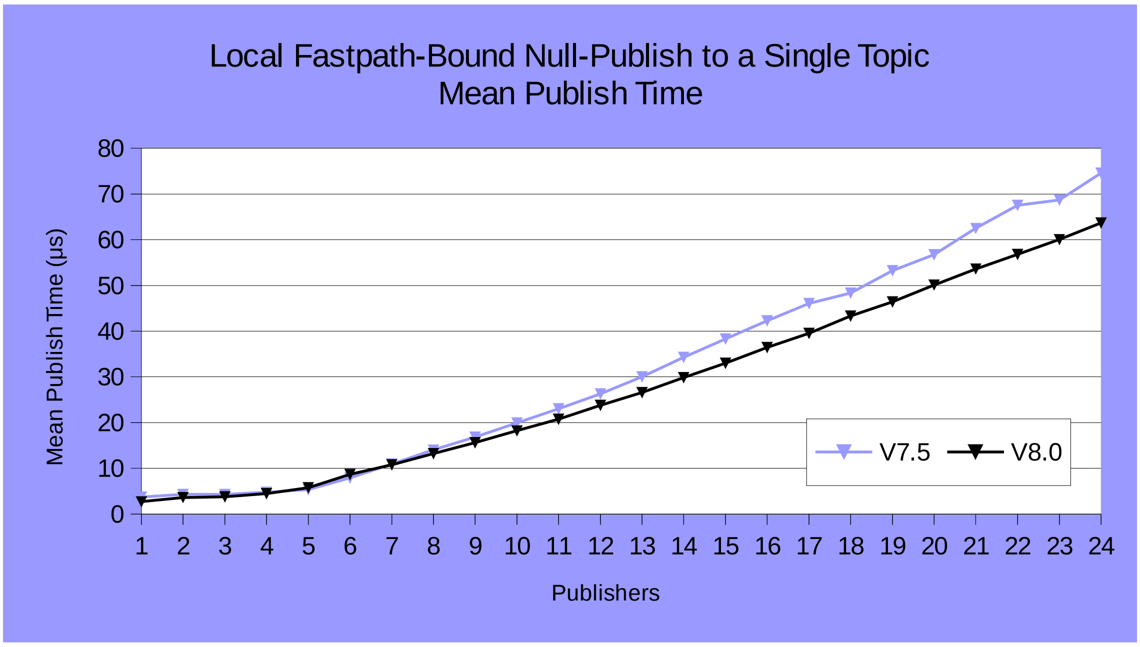

Chart 4: Local Fastpath-Bound Null-Publish to a Single Topic (Mean Publish Time)

At the throughput peak of 4 publishers, publishes in V8 take on average 4.5 microseconds, suggesting that topic lock is held for 4.5/4 = 1.125μs. After this, the gradient of the graph in Chart 4 indicates that each additional publisher increases the mean publish time by approximately 3μs (3.6μs for V7.5), suggesting that each publish now prevents other threads from acquiring the lock for this longer period of time.

It's worth noting that this phenomenon only occurs because each publish isn't really doing much. Null-publishes are so cheap that negotiating a contended lock becomes the dominant factor in performance. Still, it's useful understanding that this limit exists as a theoretical maximum to the rate of publishes on a topic (or tree of topics as explained in section 2.2.1).

2.2 Individual isolated topics

To alleviate the topic lock described in 2.1.3.1, we'll have each publisher publish to its own topic (Figure 5). This should show us whether anything else constrains the performance of publishing, or whether the full hardware resource of the machine can now be utilised.

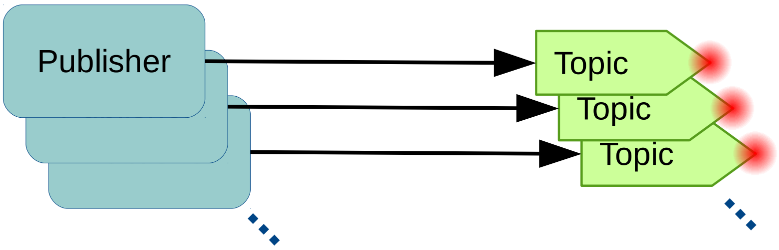

Figure 5: Null-publish to individual topics

However, eliminating topic lock is not a simple case of giving each publisher a different topic-string to publish on. Section 2.2.1 explains why this is the case, and how to properly achieve isolated topics.

2.2.1 Properties inheritance in the topic hierarchy

As you're probably aware, topics and topic-strings in MQ come together to form a hierarchy. This allows for a lot of flexibility in controlling what messages subscriptions receive. The hierarchy also simplifies the task of administering topics: properties set on a topic are automatically inherited by any child topic that doesn't explicitly have the property set itself. This include administratively created child topic objects, as well as implicitly created child topic strings. Subsequent changes to a topic's properties are automatically propagated to inheriting topics.

A side-effect of this administrative simplicity is that when a topic's property is looked up, it must be done so from the ancestor topic that explicitly defines it. As mentioned in section 2.1.3.1, looking up the property of a topic requires taking the topic's lock. So, if property *X* is defined in topic *A*, and not in topics *A/B* or *A/C*, then publishes on both *A/B* and *A/C* will require *A*'s lock to be obtained.

The topic tree has an automatically-created root topic, sitting at the top of the tree, that provides default definitions for any properties not explicitly set on topics. Thus, in order to achieve a truly isolated topic, that wont require the root topic's lock to be taken during publishes, every property of the topic must be explicitly defined. Figure 6 below gives an example of how to define such a topic using MQSC.

```
DEFINE TOPIC(TOPIC1) TOPICSTR(TOPIC1) TYPE(LOCAL) +
       DESCR('Performance test topic') CLUSTER(FAKE) +
       COMMINFO(SYSTEM.DEFAULT.COMMINFO.MULTICAST) CUSTOM(' ') +
       DEFPRTY(0) DEFPSIST(NO) DEFPRESP(SYNC) DURSUB(YES) +
       MCAST(DISABLED) MDURMDL(SYSTEM.DURABLE.MODEL.QUEUE) +
       MNDURMDL(SYSTEM.NDURABLE.MODEL.QUEUE) +
       NPMSGDLV(ALLAVAIL) PMSGDLV(ALLDUR) PROXYSUB(FIRSTUSE) +
       PUB(ENABLED) PUBSCOPE(QMGR) SUB(ENABLED) SUBSCOPE(QMGR) +
       USEDLQ(YES) WILDCARD(PASSTHRU)
```

Figure 6: Defining an isolated topic in MQSC

2.2.2 Local fastpath-bound test

By assigning each publisher its own topic, defined as in Figure 6, we get the following results:

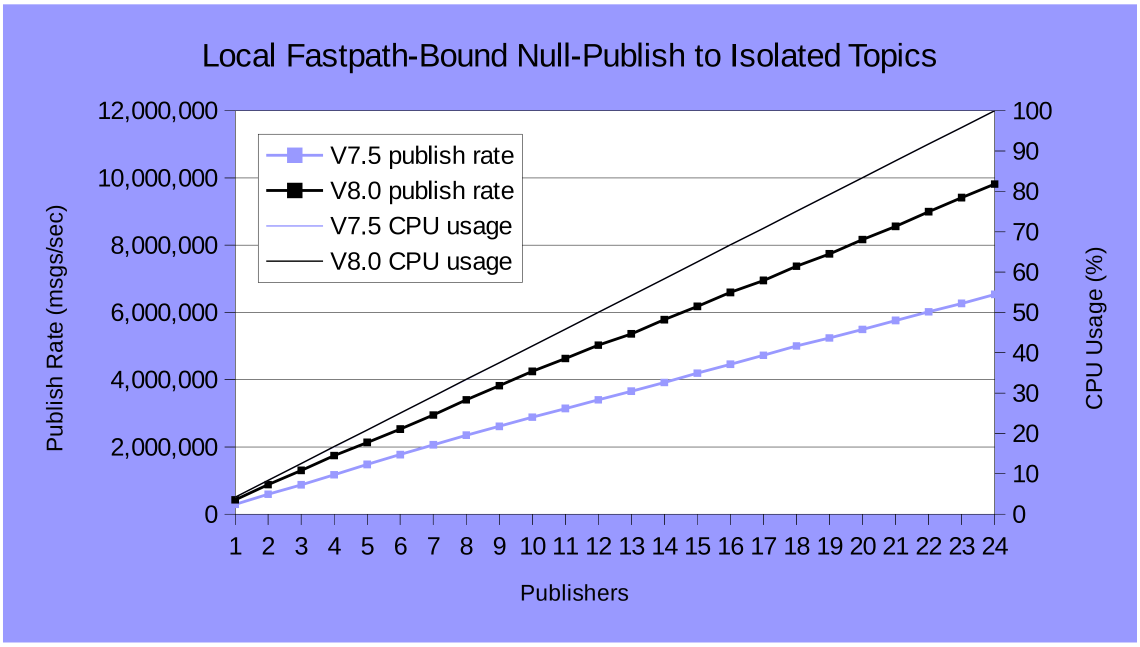

Chart 5: Local Fastpath-Bound Null-Publish to Isolated Topics

As you can see from Chart 5, the publish rate now increases linearly with the number of publishers for both V7.5 and V8, with each publisher saturating a single core of CPU (see appendix A.1.1), such that with 24 publishers, the entire machine is being used. However, V8 out-performs V7.5 considerably, with over 409,000 messages per second per publisher, compared with 272,000 for V7.5, an increase of over 50%.

Note also that these 'per publisher' rates exceed those of even a single publisher in the shared topic case (Chart 3). This reflects the shorter path-length of only having to look up properties of a single topic, as opposed to 2 in the shared topic case (the topic and its parent, the root topic).

2.3 Conclusions

2.3.1 Summary

In this section, we've looked at the pure cost of publication, without any subscribers. This is useful for three reasons:

1. It represents the most significant part of the publish-subscribe message flow that differs from point-to-point messaging.

2. It shows the additional strain placed on an infrastructure by publishes on non-subscribed topics.

3. It represents an absolute upper limit to the performance obtainable with publish-subscribe messaging.

Depending on the situation, there are a number of hardware factors that can constrain the rate of such publications, including network bandwidth, CPU capacity, and the CPU capacity of a NUMA zone (section 2.1.2.1). For sub-trees of the topic hierarchy inheriting properties from a single topic (see section 2.2.1), there is also the issue of *topic lock* to be contended with (section 2.1.3.1).

Topic Structure	Application Bindings	V7.5 Peak Rate	V8 Peak Rate	Contending Factor	Section
Shared	Client	408,891	433,851 (+6%)	Network Bandwidth	2.1.1 (page 12)
Shared	Standard	683,862	747,010 (+10%)	CPU (NUMA zone)	2.1.2 (page 13)
Shared	Fastpath	936,878	888,744 (-5%)	Topic Lock	2.1.3 (page 14)
Isolated	Fastpath	6,537,441	9,816,346 (+50%)	CPU (Machine)	2.2.2 (page 17)

Chart 6: Peak Rates for Null-Publish Scenarios

Chart 6 summarises the results of performance tests from this section.[2] In almost all situations, the performance of IBM MQ V8 exceeds that of V7.5.

2.3.2 Determining the maximum publish rate through a topic

Section 2.1.3 established that there is a maximum publish rate achievable to each isolated topic, and that attempting to exceed this rate can result in a serious performance penalty. In section 2.2.1, it was explained that this limit in fact applies to any sub-tree of the topic hierarchy that inherits at least one property from a common root topic. On the machine used for the tests in this report, this limit is around 900,000 publications per second.

On different hardware, the maximum publication rate through a single topic (or sub-tree) could be less or more than this, as the mean publish time will depend on the speed of the processor cores. However, the *proportion* of that time spent holding the topic lock should be about the same, and hence so should the number of concurrent threads that can run flat-out while not putting excessive strain on the lock.

[2] **Note** that the rates and contending factors in Chart 6 are specific to the hardware used for these tests. In each scenario, the contending factor is likely to remain the same unless hardware configuration is significantly different. In particular, on machines with much lower CPU capacity, CPU may be exhausted before network bandwidth or topic lock come into play.

So, in order to establish the maximum throughput of a topic, that you should not attempt to exceed, have 3, 4, and 5 application threads connect to the queue manager using *fastpath* bindings, and publish as fast as they can. Whichever total publication rate is highest will be a good indication of the limit for a single isolated topic sub-tree.

3 — How the Number of Subscriptions Affects Performance

One of the key differences between publish-subscribe messaging and point-to-point is that a published message may be delivered to multiple subscribers, rather than just a single queue. In MQ, this distributed delivery is implemented by putting a copy of the published message on the subscriber queue of each subscription to the topic (see section 1.1). In this section we'll investigate how the number of subscriber queues that a publication must be delivered to affects the publication rate, publication time, and overall message rate in MQ.

To do this, we'll use an experiment whereby a single application publishes messages to a single topic, and we'll gradually increase the number of subscribers to that topic to see the effect. The publisher will always publish repeatedly as fast as it can, with no pause or “think time” between publications.

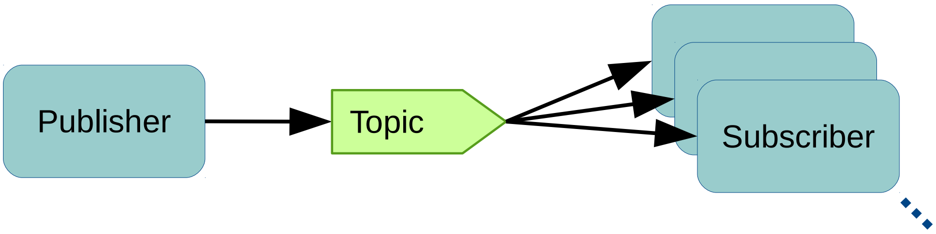

Figure 7: Increasing subscriptions to a single topic

3.1 Note on subscriber bindings

In all of the tests in this section, locally-bound subscribers will be used. This is because the lack of network-communication on the subscriber side makes results more consistent and easier to interpret. The design of MQ means that in these high fan-out scenarios, message throughput is bottle-necked by the publisher, so the exact nature of the subscribing applications normally makes little difference.

As a rule of thumb, using remote, client-bound subscribers may lead to an up to 15% degradation at times compared with locally-bound subscribers, but the occurrence of this is intermittent, and overall, the same pattern is seen as the number of subscriptions scales. This should remain the case unless network bandwidth is restricted. Chart 7, below, illustrates a typical example using a local publisher.

Chart 7 also shows that CPU usage is a little greater per message with remote subscribers. This should not come as a surprise, as the subscriber applications have less work to do than the message channel agents that send the messages to remote subscribers. Both must retrieve messages from the queue, but only the channel agents need to then forward them over the network. A 'real' subscribing application will do more work than just retrieve each message and discard it, and as such is likely to end up costing more CPU than the channel agents. As such it may be advantageous to offload subscribers to a separate machine, freeing up CPU capacity for the queue manager.

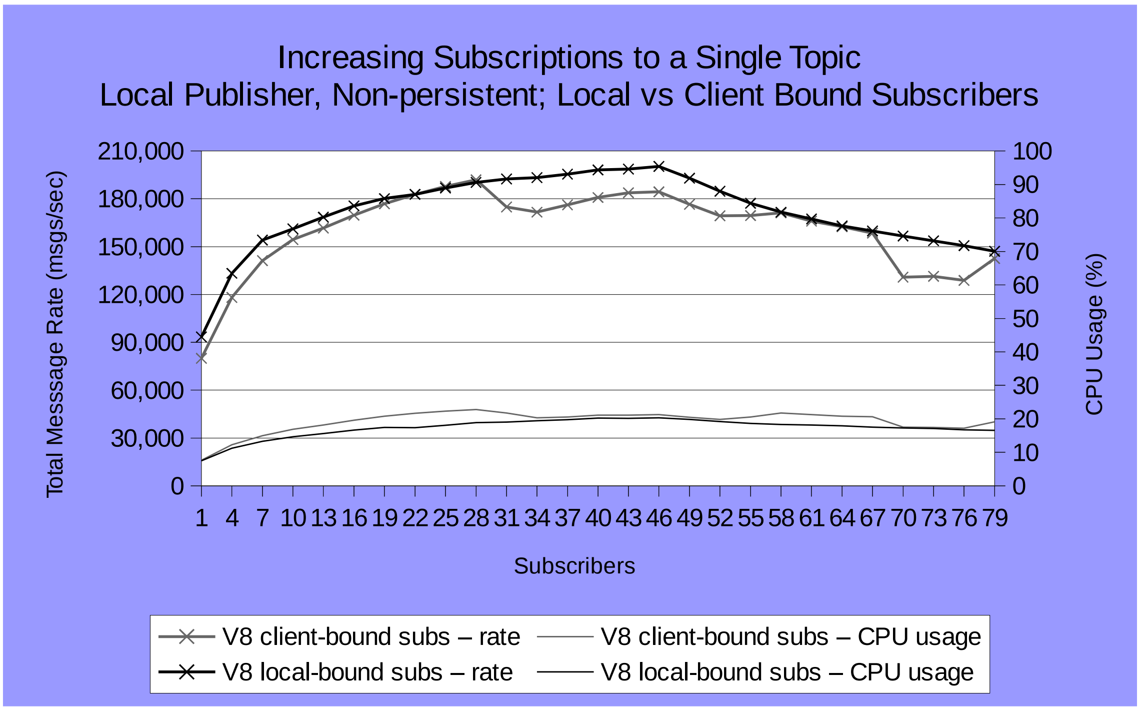

Chart 7: Increasing Subscriptions to a Single Topic; Local Publisher, Non-persistent; Local vs Remote Subscribers

3.2 Note on CPU usage

As with Chart 7, CPU usage in all of the tests in this section is low. Although the work done is not limited to a single thread, the maximum achievable rate is in practice limited by the rate at which the publisher thread can operate. Since the machines used in these experiments have 24 CPU cores, all the work required of subscriber threads can be easily handled by the remaining 23 cores, once one has been taken for the publisher. For this reason, CPU usage data will be omitted from results in the rest of this section.

Note, however, that a queue manager running on a machine with much fewer cores could potentially become CPU-contended under these scenarios. For guidance on where and how these limits might be reached, see section 4, and also the main IBM MQ V8 performance report, MPLB[3].

[3] www-01.ibm.com/support/docview.wss?uid=swg24038996

3.3 Local publisher, non-persistent

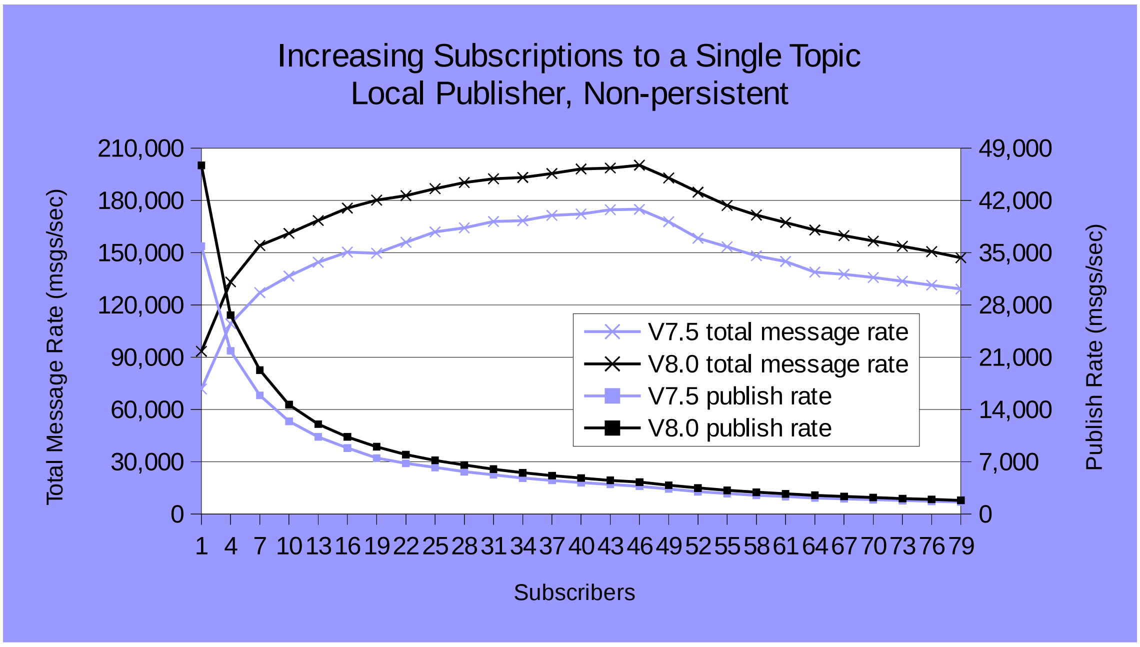

Chart 8: Increasing Subscriptions to a Single Topic; Local Publisher, Non-persistent (Rates)

Chart 8 shows the total message rate increases steadily but concavely up to a peak between 46 and 49 subscribers, with V8 reaching a total throughput of just over 200,000 messages per second, compared with just under 175,000 for V7.5. While the total rate increases, it can be seen that the publication rate falls as the number of subscribers increases, apparently asymptotically approaching zero. This decrease in publication rate is due to the fact that the publisher thread must put a copy of each message on each subscriber queue, thus taking longer to publish each message as the number of subscribers grows.

The total message rate also appears to suddenly start decreasing beyond the peak, rather than smoothly and gradually beginning to decline, or reaching an asymptotic plateau as one might expect. This change in behaviour is not visible in the publication rates, but can be seen if we invert them and look at mean publish times (Chart 9).

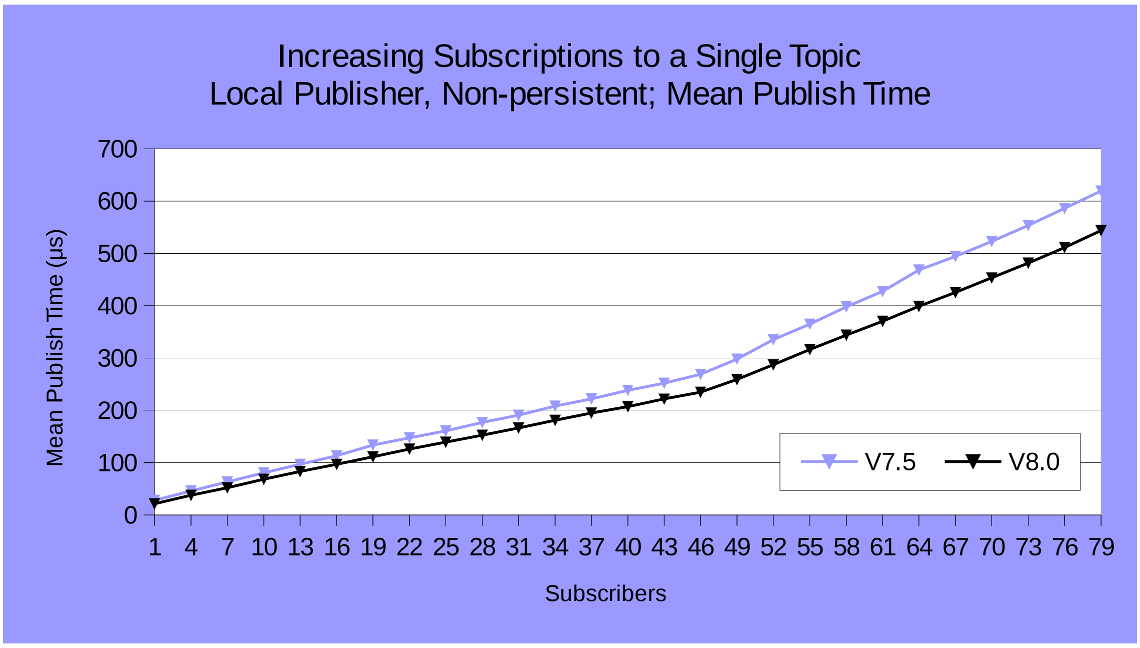

Chart 9: Increasing Subscriptions to a Single Topic; Local Publisher, Non-persistent (Mean Publish Time)

The linearity of Chart 9 at fewer than 46 clients clearly demonstrates that each additional subscriber adds a fixed cost to the publish time. For V7.5 each subscriber increases the publish time by about 5.4µs, whereas for V8 the cost is only 4.7µs.

However, above 49 subscribers, the cost per *additional* subscriber doubles for both versions (to 9.5µs for V8, and 10.7µs for V7.5). This is what leads to the sudden downward trend in the total message rates seen in Chart 8.

3.3.1 PubSubCacheSize

The cause of this increased cost at higher numbers of subscribers – or more accurately, the cause of the *reduced* cost at low numbers of subscribers – is the *publish-subscribe cache*. Each publisher in MQ is assigned a cache in which it can store open handles to subscriber queues that it regularly delivers to. By default, this cache can hold up to 48 such handles. Once the number of subscribers that a publisher must deliver to exceeds this number, then additional subscriber queues must be looked-up, opened, and closed on each publish. Hence, above 48 subscribers, additional subscribers require more work, and more time to deliver messages to.

The size of the cache can be changed using a queue-manager-wide configuration parameter in the *qm.ini* file, by setting the *PubSubCacheSize* value under the *TuningParameters* stanza. The results of this are demonstrated in Chart 10, where one can clearly see both the total message rate and mean publish time diverging at the point where the *PubSubCacheSize* is reached, for all three values tested. It can also be seen that the cost for each additional subscriber beyond the *PubSubCacheSize* is the same for all tests.

It is expected that the default of size of 48 would be sufficient for the majority of scenarios. If, however, you expect a significant proportion of your publishers to be regularly delivering to more than 48 subscriber queues, provided the machine is not running short of available RAM, it should be perfectly safe to increase this limit in order to boost publication rates. Note that changes to the *qm.ini* file won't take effect until the queue manager is restarted.

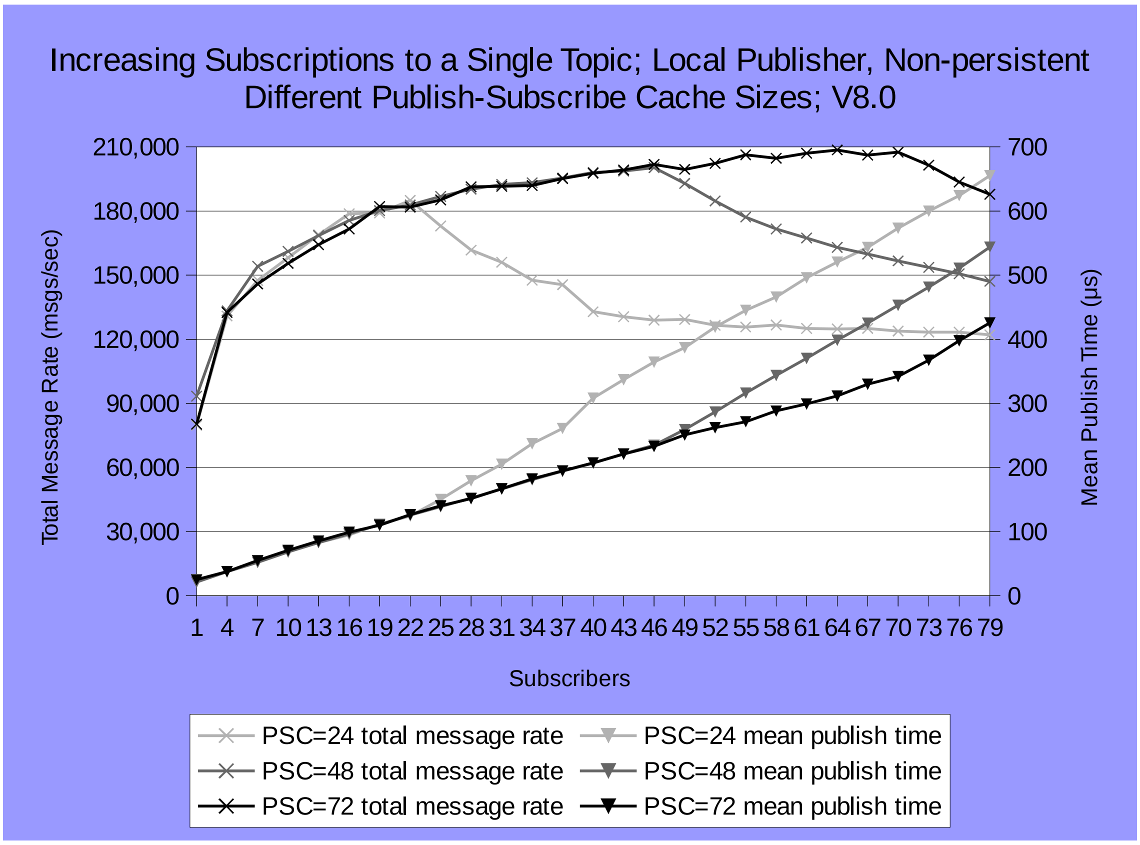

Chart 10: Increasing Subscriptions to a Single Topic; Local Publisher, Non-persistent; Different Publish-Subscribe Cache Sizes; V8.0

3.4 Local publisher, persistent

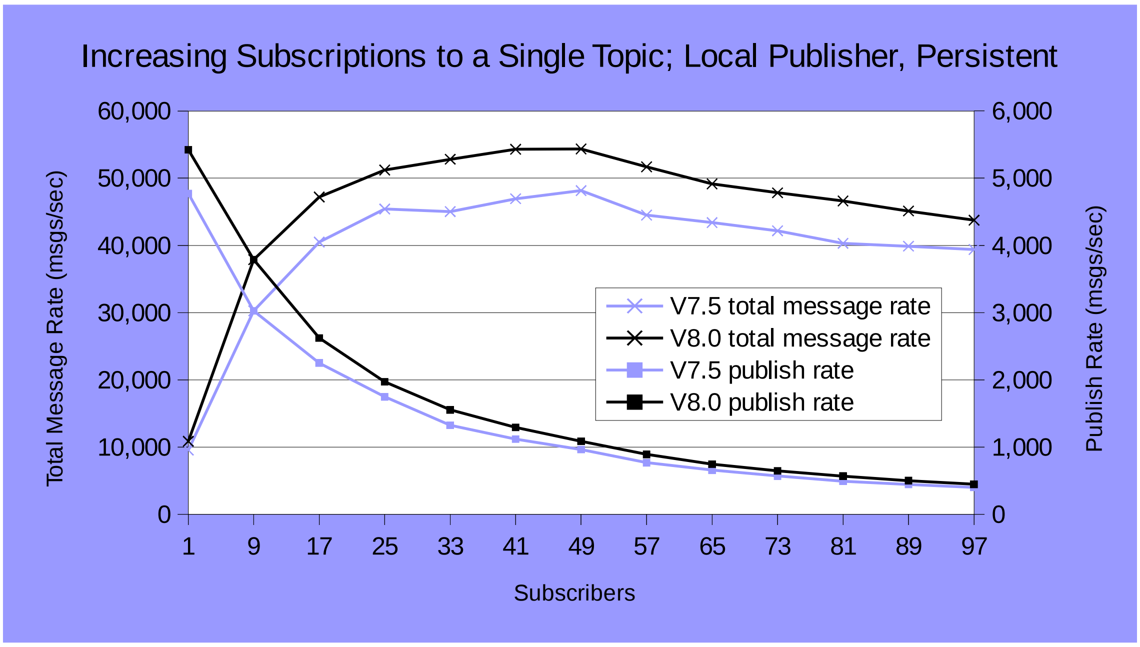

Chart 11: Increasing Subscriptions to a Single Topic; Local Publisher, Persistent (Rates)

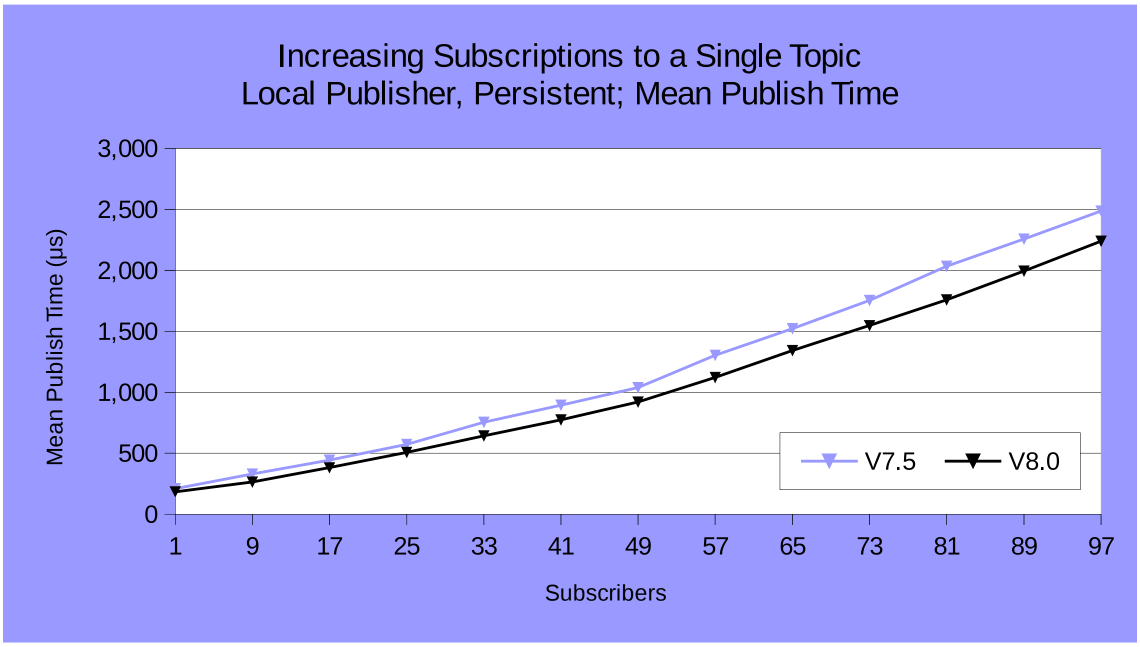

Chart 12: Increasing Subscriptions to a Single Topic; Local Publisher, Persistent (Mean Publish Time)

From Chart 11 and Chart 12, we see that result for the persistent scenario are qualitatively similar to the non-persistent scenario. However, due to the additional latency introduced by the logger and having to write to disk, quantitatively performance is much reduced. The cost per subscriber to mean publish time, before the publish-subscribe cache has been filled, is not 14.7µs for V8 and 17.1µs for V7.5. This increases to 27.5µs and 30.2µs respectively once the cache has been filled.

It is known from other tests on these machines, that the disks (local RAID 0 array with battery-backed cache) are capable of achieving a total message rate approaching 200,000 2KB messages per second. This, combined with the fact that the publish-subscribe cache is still clearly having an effect suggests that the disks are not writing at their full capacity. Indeed, just as section 3.2 discusses for CPU utilisation, the additional cost we're seeing here is due to the *latency* of the disk subsystem, as well as the additional path-length introduced by MQ's logger. It's worth noting that, as with CPU capacity, if disk write speeds were lower, then these may become a contending factor, and one would expect the message rates to be capped at the maximum rate the logger can work at. Section 4.2 gives a good indication of the true write-capacity limits of the hardware used in these tests.

3.5 Remote publisher, non-persistent

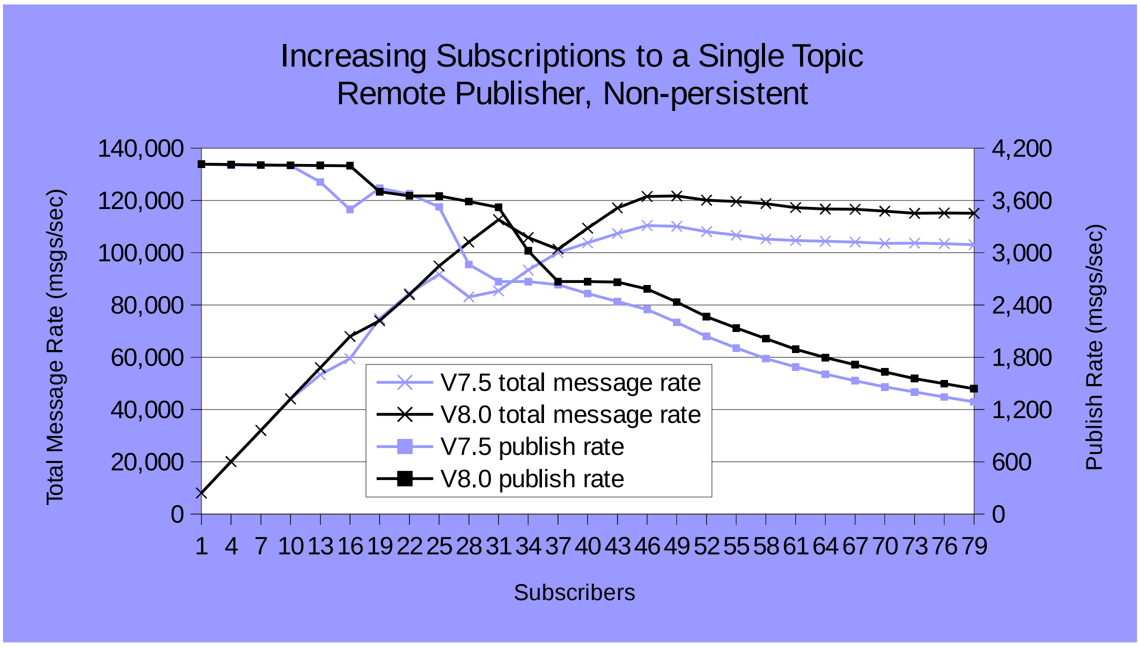

Chart 13: Increasing Subscriptions to a Single Topic; Remote Publisher, Non-persistent

When the publishing application is moved to a different machine from the queue-manager, the performance behaviour changes considerably. Instead of asymptotically decreasing, the publication rate initially stays constant at around 4,000 publications per second, irrespective of the number of subscribers, then “steps” down before briefly reaching another lower plateau of around 2,650 publications per second (at 37-43 subscribers for V8), and finally going into a gentle asymptotic decline.

Furthermore, initially V7.5 matches the performance of V8 almost exactly, but begins stepping down sooner, and reaches a lower maximum total message rate at its peak.

The initial stepped nature of the throughput can also be seen by looking at the mean publish time, as shown in Chart 14, where the V7.5 data has been omitted for clarity, and instead we've made a comparison to the local publisher scenario.

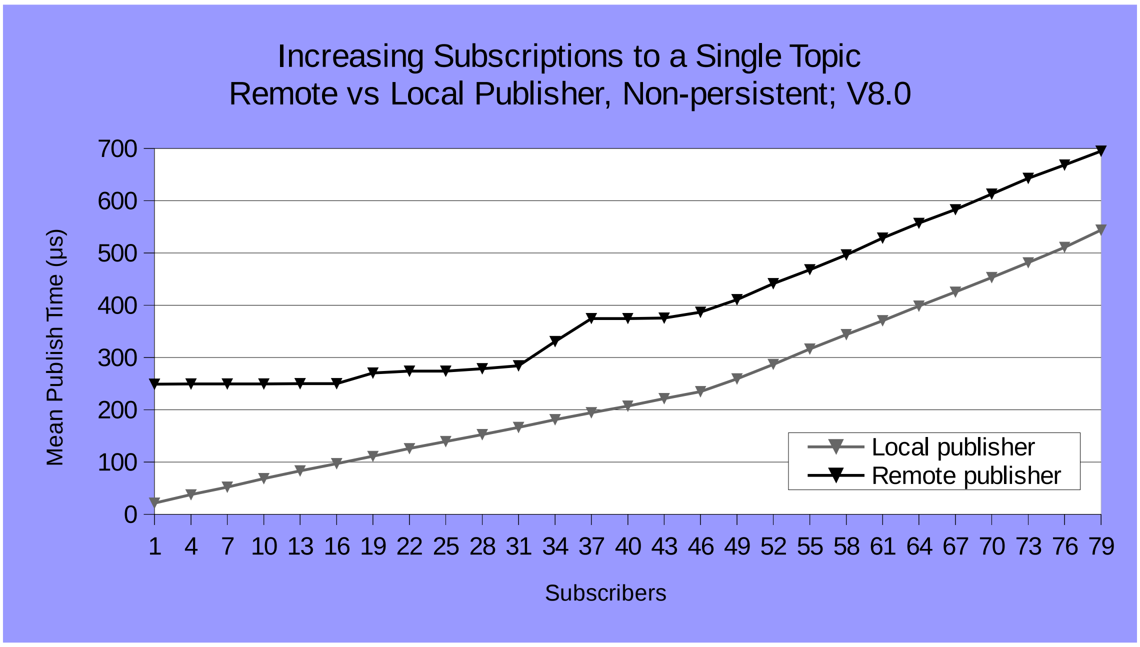

Chart 14: Increasing Subscriptions to a Single Topic; Remote vs Local Publisher, Non-persistent; V8.0

Compared with the local case, the right-hand side of Chart 14 shows a behaviour one might expect: the cost per additional subscriber is the same in both cases (as illustrated by the parallel gradient of the lines), but the remote publisher case has an additional fixed cost per publish, which is the latency introduced in communicating over the network.

On the left-hand-side, we note that the initial plateau in publish times is at almost exactly 250 µs, or 2/8000s. The higher brief plateau, at 37-43 subscribers is at 375 µs, or 3/8000s. This suggests that there is some sort of minimum response time or polling frequency for quick responses, when publishing over the network. No such granularity is known to be in MQ itself, and it is believed that this is due to the underlying design of the Linux network sockets used for communication. This is backed up by the fact that, as can be seen in Chart 13, these exact same minimum response times exist for V7.5, although the slower queue manager causes these minima to be exceeded with fewer clients.

The intermediate, slightly sloped nature of the publish times between these two plateaus is due to the fact that the publish times taken are an *average*, so between 19 and 34 subscribers (for V8), some publishes respond within 250µs, while an increasing number begin to take 375µs instead.

3.6 Conclusions

In this section we looked at how the number of subscriptions to a topic affects publish-subscribe performance, by having a single application publish as fast as it could to a single topic, and gradually increasing the number of subscribers. In all situations, we see that each subscriber adds a fixed amount to the time that a publish takes, but the presence of the publish-subscribe cache reduces this cost at low numbers of subscribers (section 3.3.1). Publishing over the network adds a fixed latency per publish, but at small numbers of subscribers, network socket granularity sets a minimum on the time a publish takes (section 3.5).

Publisher Locality / Persistence	Time Per Publish to 1 Subscriber (µs)		Time Per Additional Subscriber (Cached / Non-cached) (µs)		Peak Total Message Rate (msgs/sec)		Section
	V7.5	V8	V7.5	V8	V7.5	V8	
Local / Non-persistent	27.9	21.4 (-23%)	5.4 / 10.7	4.7 (-11%) / 9.5 (-11%)	174,929	200,284 (+14%)	3.3 (page 22)
Local / Persistent	209.5	184.4 (-12%)	17.1 / 30.2	14.7 (-14%) / 27.5 (-9%)	48,145	54,342 (+13%)	3.4 (page 24)
Remote / Non-persistent	250	250	? / 10.7	? / 9.5 (-11%)	110,415	121,695 (+10%)	3.5 (page 26)

Chart 15: Peak Rates and Costs for Publisher Fan-Out Scenarios

4 — Comparison With Point-To-Point

In this section, we will look at a scenario where publish-subscribe messaging in IBM MQ can be compared directly with point-to-point. Specifically, we'll consider the case where each topic has a single publisher and a single subscriber, calling this trio of publisher-topic-subscriber a "triplet". We'll then increase the number of triplets to ramp up the load on the queue manager.

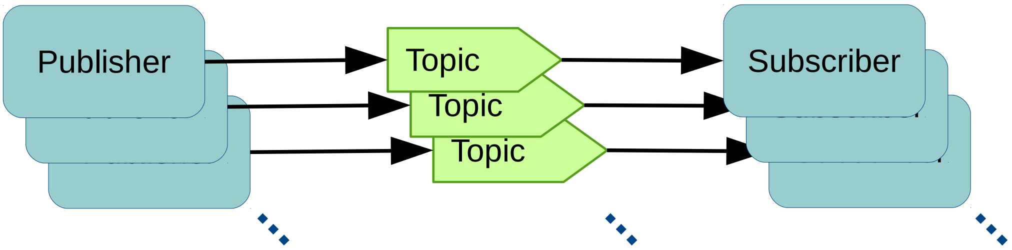

Figure 8: Publish-subscribe triplets

This same scenario can be replicated in point-to-point messaging by simply replacing the topic in each triplet by a queue, and the publisher/subscriber applications by suitable basic producers/consumers.

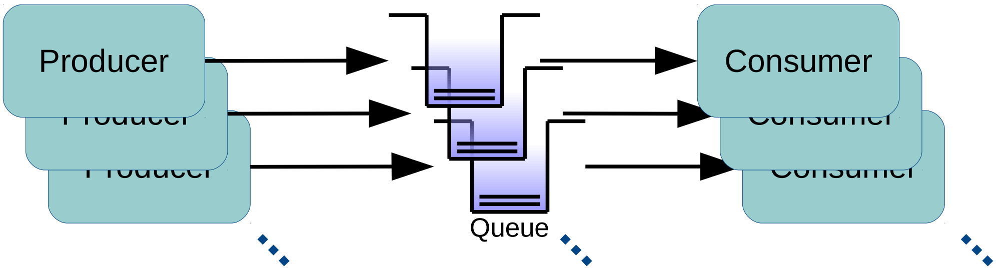

Figure 9: Point-to-point triplets

The publisher/producer of each triplet will attempt to produce messages at a fixed rate, rather than sending messages as fast as it can. This is because cases can occur whereby the producing application is able to achieve a higher rate than the subscriber/consumer, leading to a build-up of messages on the associated queue. This ends up having an overall negative impact on performance, particularly is the queue's in-memory buffers are exceeded, and messages must be written to disk. The main IBM MQ performance report discusses this phenomenon in more detail.

This type of experiment is also useful for determining the overall hardware resource usage for publish-subscribe messaging, because (as we shall see) there is very little locking and serialisation present and so the tests scale quite linearly until some hardware aspect is exhausted.

4.1 Local publishers & subscribers, non-persistent

In this section, the producing application in each triplet is rated at 4000 messages per second.

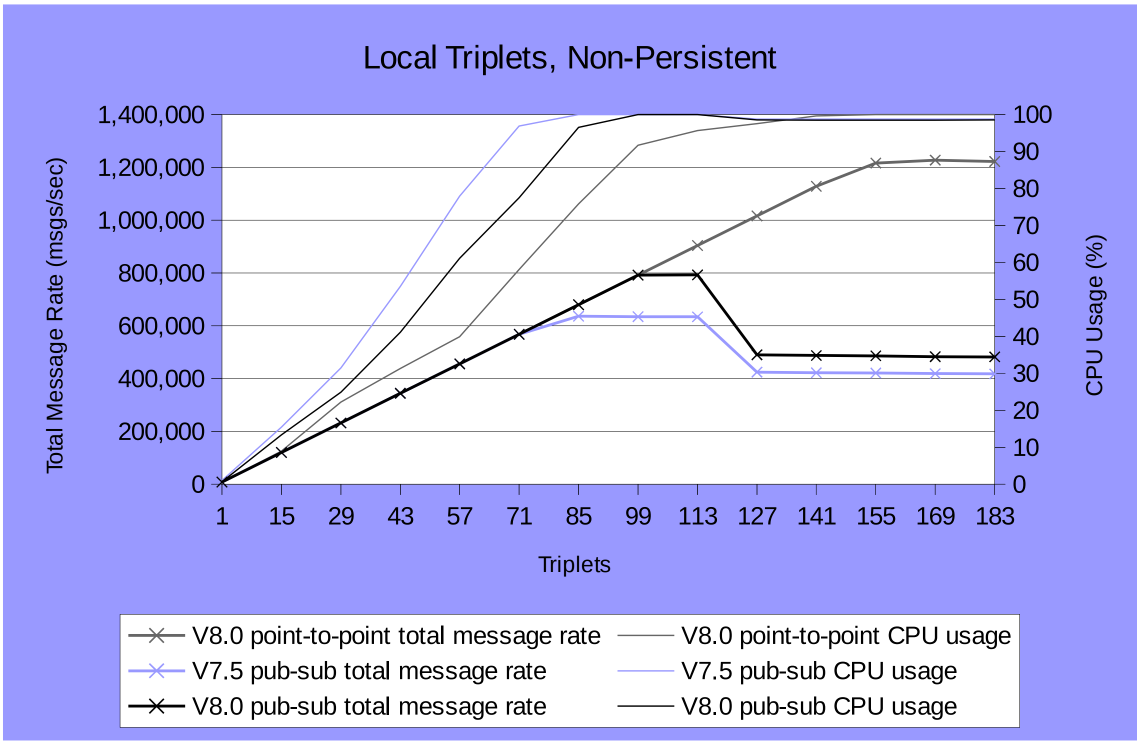

Chart 16: Local Triplets, Non-Persistent

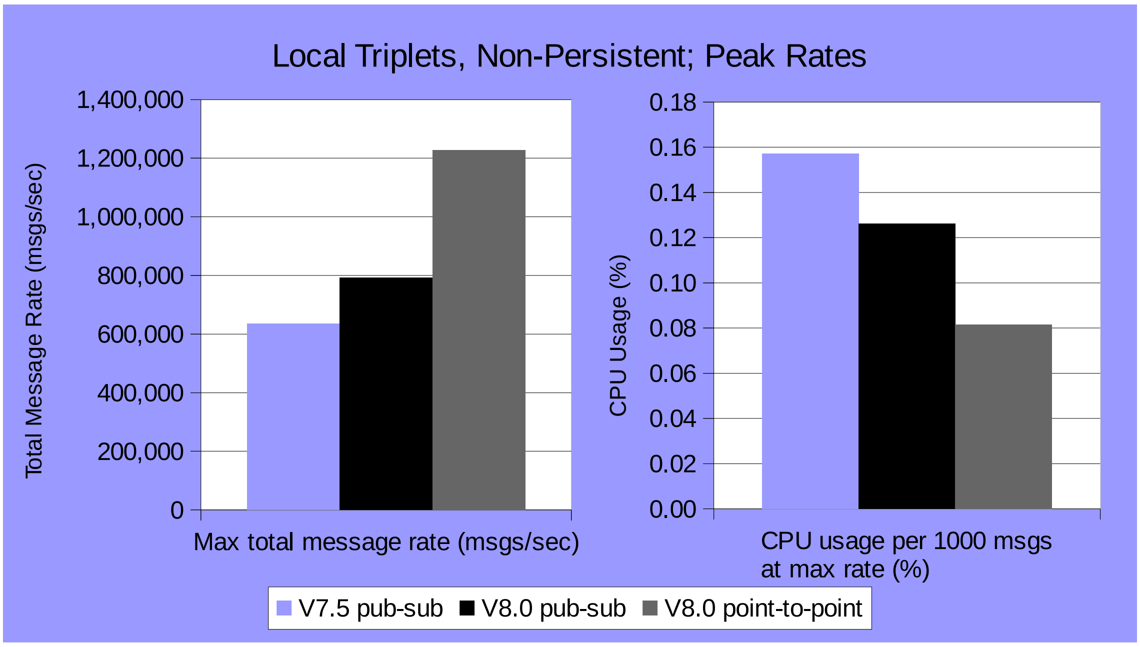

Chart 17: Local Triplets, Non-Persistent; Peak Rates

Chart 16 shows that V8 maintains its attempted total message rate of 8000 messages per second per triplet up until 99 triplets with an overall peak rate of around 800,000 messages per second. At this point, CPU on the machine is exhausted, suggesting that this is the contending factor. V7.5 peaks a little lower, at slightly fewer triplets. The peak rate for point-to-point triplets is 50% faster, again apparently due to CPU exhaustion, suggesting that the publish-subscribe engine adds a 33% CPU cost to each message.

You will notice that, after peaking, the publish-subscribe triplets' total message rate declines, before settling at about 500,000 messages per second (V8; 420,000 for V7.5). It turns out that once again, this is due to topic lock, as discussed in section 2.1.3.1. The topics used for the triplets in Chart 16 are not administratively defined, and are just topic strings given by the publishing and subscribing applications. If instead we define each topic as in Figure 6 (page 17), then we get the results shown in Chart 18.

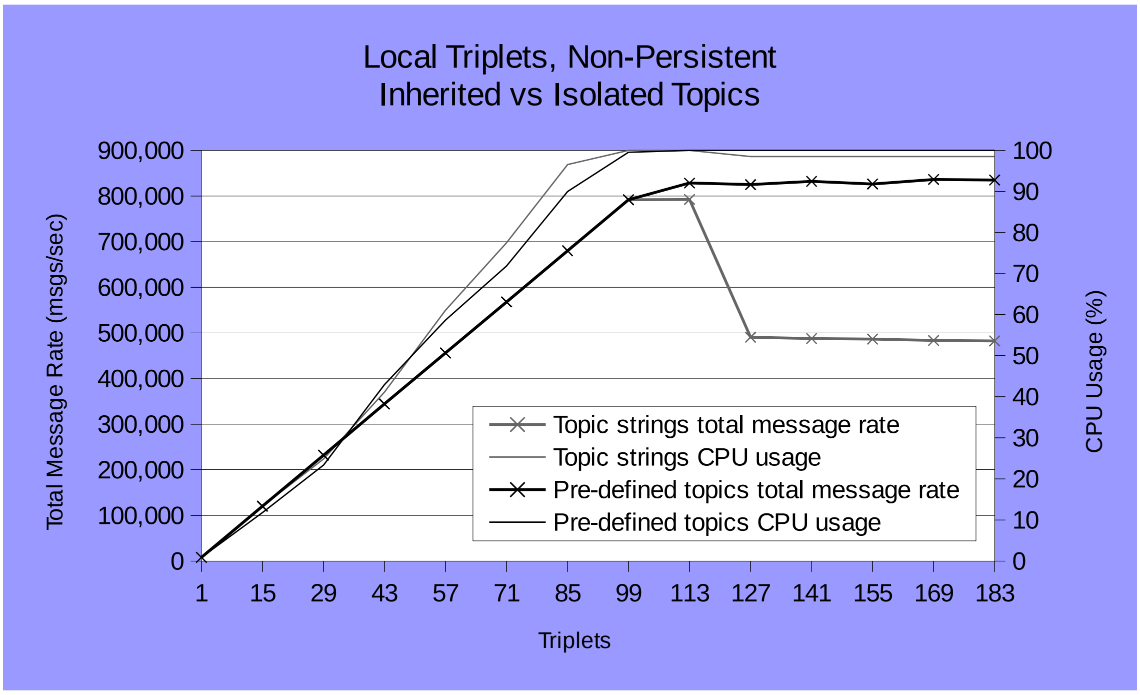

Chart 18: Local Triplets, Non-Persistent; Inherited vs Isolated Topics

Here, we see that, after saturating its CPU capacity, the test with pre-defined, isolated topics maintains its peak total message rate. It also has a slightly higher peak rate, and slightly lower CPU cost per message than the case where non-defined topic strings are used, due to the reduced cost of resolving topic properties.

4.2 Local publishers & subscribers, persistent

In this section, the producing application in each triplet is rated at 1000 messages per second.

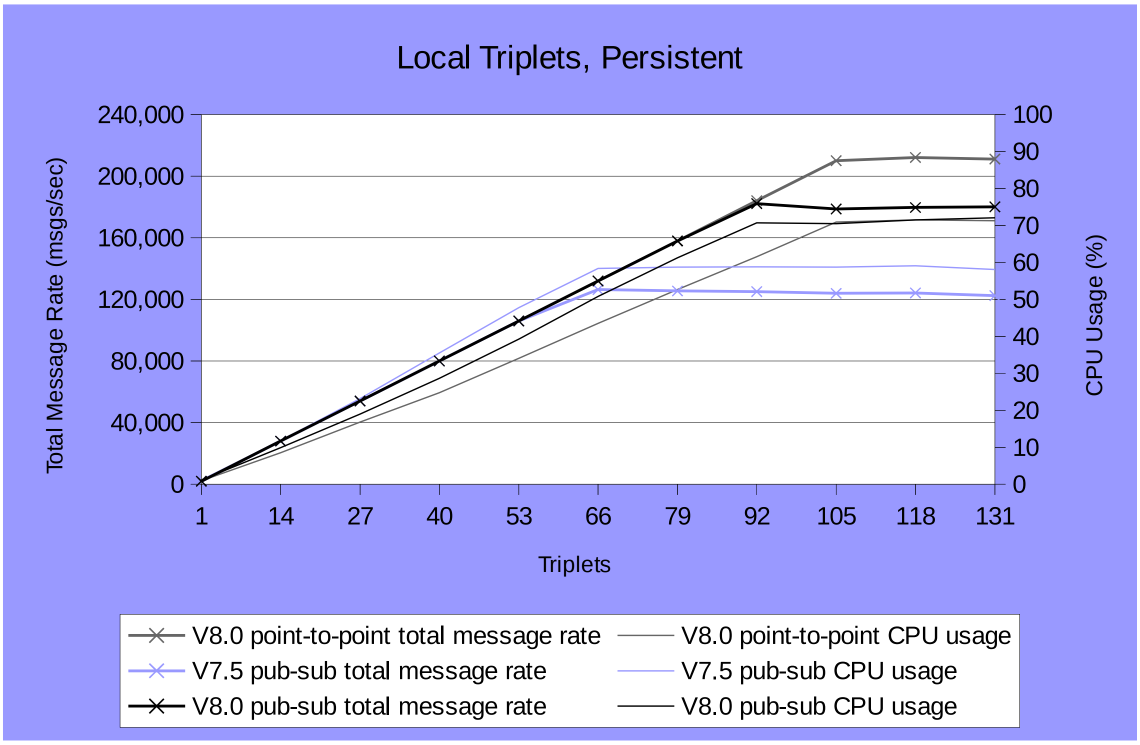

Chart 19: Local Triplets, Persistent

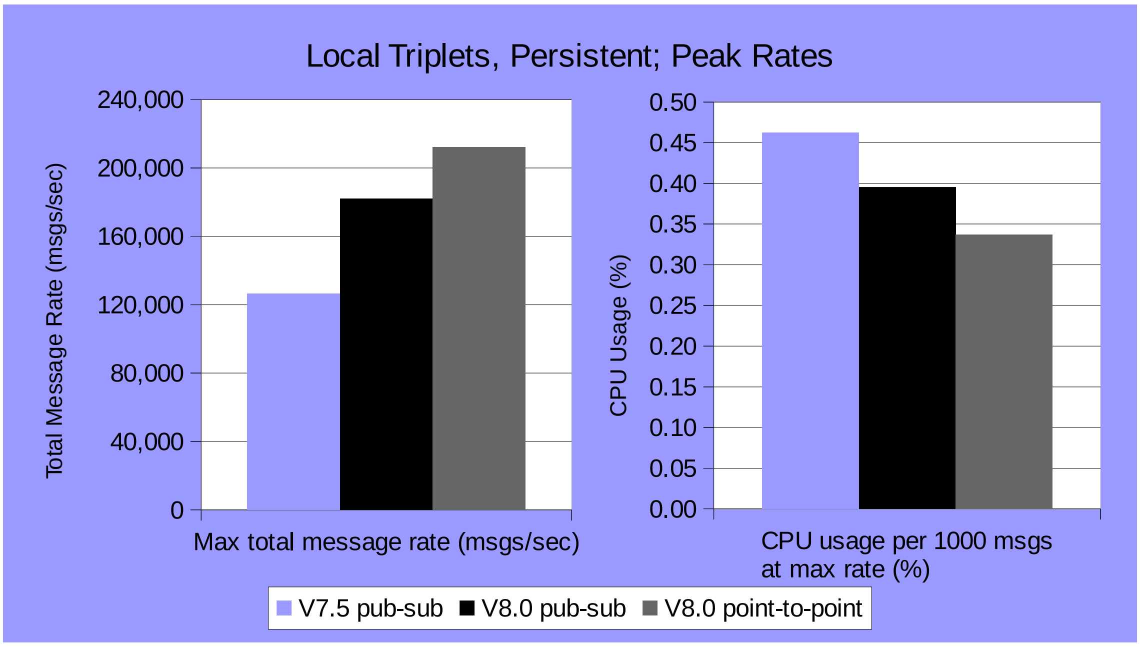

Chart 20: Local Triplets, Persistent; Peak Rates

For persistent messaging, the improvement of V8 over V7.5 is greater, compared to the non-persistent case, with a 44% increase in the peak message rate achieved. The difference between pub-sub and point-to-point is also less, reflecting the fact that the contending factor (the logger) behaves very similarly for both message architectures. Despite using topic strings, that weren't administratively defined before hand, there is no drop-off in rate after reaching the peak. The lower overall message rates involved in the persistent scenario mean that topic lock never becomes contended.

4.3 Remote publishers & subscribers, non-persistent

In this section, the producing application in each triplet is rated at 2500 messages per second.

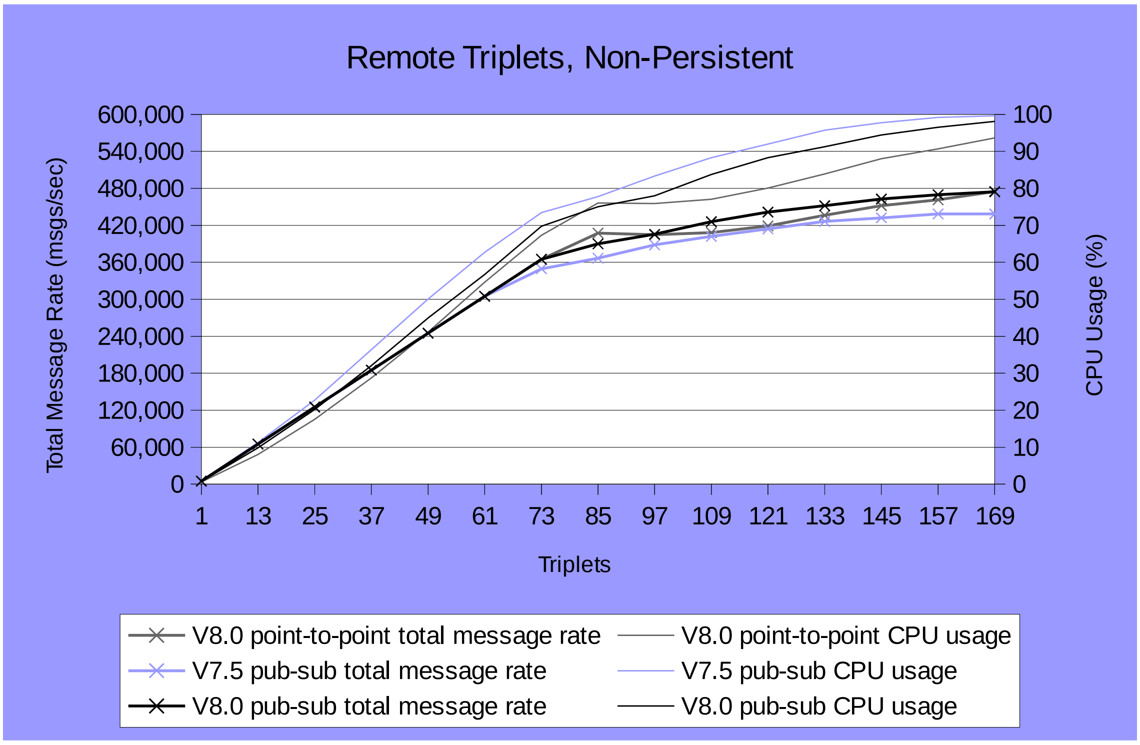

Chart 21: Remote Triplets, Non-Persistent

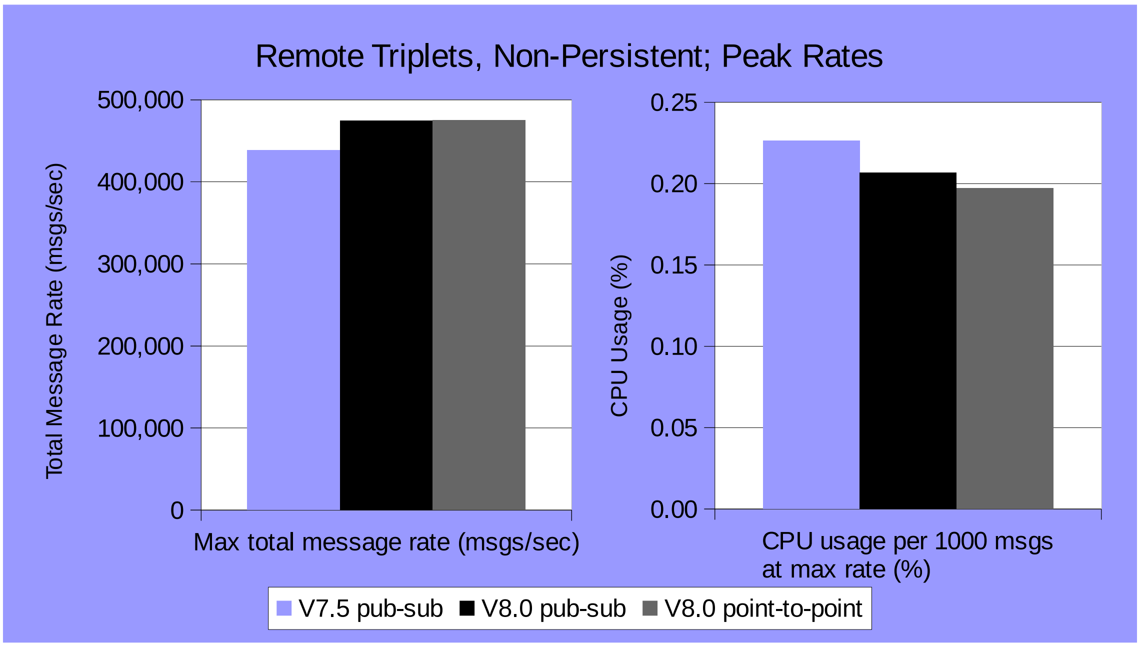

Chart 22: Remote Triplets, Non-Persistent; Peak Rates

In the remote case, there's hardly any difference between publish-subscribe and point-to-point messaging, demonstrating that network communication, which is almost identical between the two scenarios, takes the lions share of the resources. However, in the contending factor is CPU, used by MQ's *Message Channel Agents*, rather than network bandwidth. On a slower network (e.g. 1Gb/s), this might not be the case.

4.4 Conclusions

Publish-subscribe messaging introduces a performance overhead when compared with point-to-point, primarily in the local, non-persistent case. However, in the majority of situations, where a large proportion of the client applications connect to MQ over the network, this overhead is unlikely to be noticed. When very high message rates are involved (most likely in locally-bound scenarios), topic-lock may become an issue. It would be prudent to consider splitting the topic hierarchy into a number of administratively-defined branches in such high-volume scenarios (see section 2.2.1).

	Peak Total Message Rate			**CPU Usage Per 1000 Messages at Peak Rate**			
Application Locality / Persistence	**V8.0**	**vs V7.5**	**vs point-to-point**	**V8.0**	**vs V7.5**	**vs point-to-point**	**Section**
Local / Non-Persistent	792,464	636,322 (+25%)	1,227,511 (-35%)	0.13%	0.25% (-20%)	0.08% (+54%)	4.1 (page 30)
Local / Persistent	182,201	126,333 (+44%)	212,171 (-14%)	0.40%	0.46% (-14%)	0.34% (+17%)	4.2 (page 32)
Remote / Non-Persistent	474,542	438,408 (+8%)	474,714 (-0%)	0.21%	0.23% (-9%)	0.20% (+5%)	4.3 (page 33)

Chart 23: Peak Rates and Costs for Triplets Scenarios

5 — JMS

In this section we'll look briefly at the difference between the standard C MQI interface, and JMS when using publish-subscribe messaging. Architecturally, there shouldn't be any difference between these two protocols in the way publish-subscribe messages are distributed by the queue manager. However, the processing of larger message headers required for JMS, and the use of non-native Java by the client applications, means that there is an additional performance cost when using JMS.

In all of the tests in this section, we'll use remote, client-bound applications for both publishers and subscribers. The reason for this is that the JMS client application uses considerably more CPU capacity than the equivalent C application, or even MQ's *Message Channel* Agents. Hence, locating the JMS client applications on the same machine as the queue manager often causes the tests to be constrained by CPU capacity, most of which is being consumed by the applications. As a result, the queue manager's own limits and capacities aren't tested very well if the JMS applications are running locally.

All tests in this section will be non-persistent.

5.1 Increasing subscriptions to a single topic

Here, we'll use a test scenario similar to those used in section 3, except that, for the reasons discussed above we'll use remote, client-bound applications for both the publisher and the subscribers.

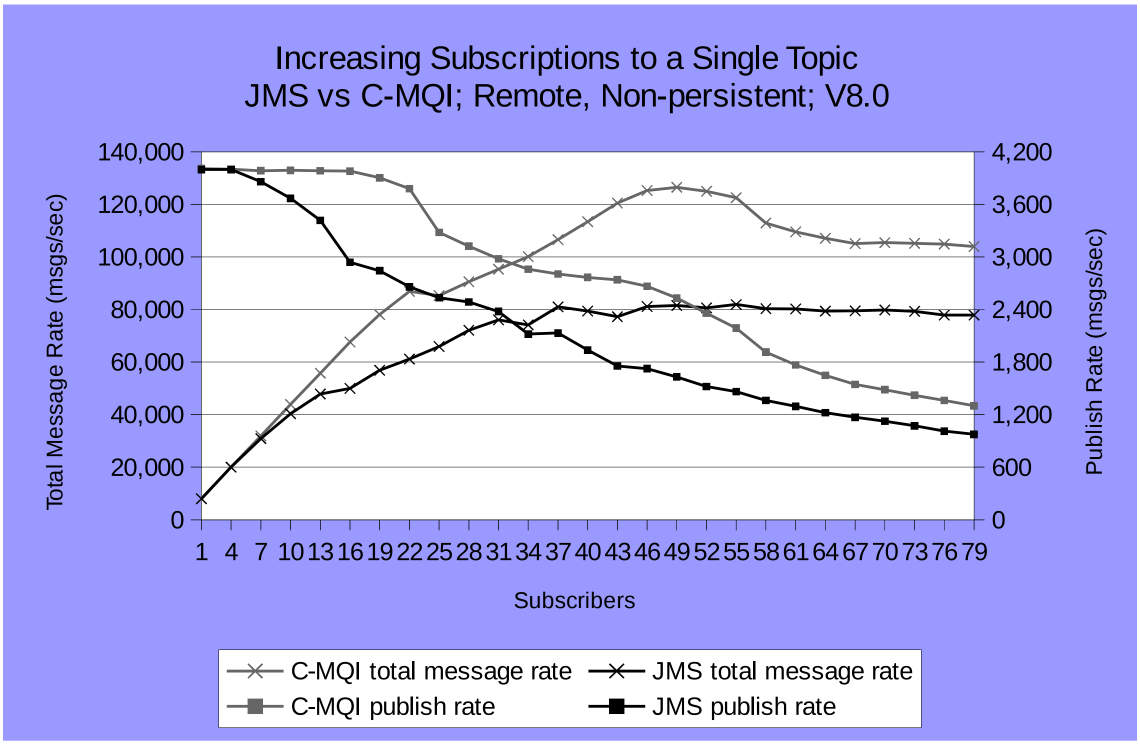

Chart 24: JMS vs C-MQI; Increasing Subscriptions to a Single Topic; Rates

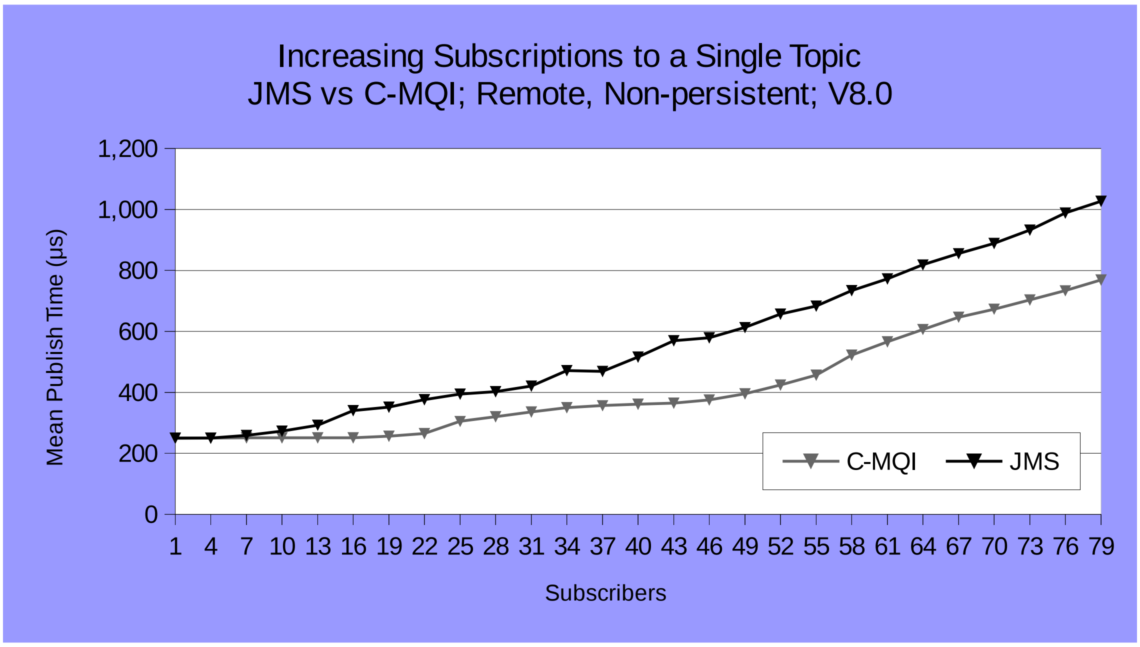

Chart 25: JMS vs C-MQI; Increasing Subscriptions to a Single Topic; Mean Publish Time

As can be seen from Chart 24, the peak total message rate achieved for JMS is about 35% lower than when using C clients, and Chart 25 shows us that there is a 20% increase in the mean publish time per additional subscriber beyond the 48 whose queue handles are cached (see section 3.3.1, page 23). The variance introduced by the network communication, as well as the granularity of the network sockets at low numbers of clients makes determining the additional cost for cached subscribers infeasible, but once again it is clearly higher for JMS.

5.2 Triplets

Here, we'll look at the “triplets” scenario used in section 4, in order to see the impact of using JMS on available CPU on the queue-manager machine. Once again, all publishers and subscribers are client-bound.

The publishers in this section were rated at 2500 messages per second.

We can see from Chart 26, below, that with JMS, the scenario deviates from its target rate of 2500 messages per second per publisher sooner than when using MQI. The peak rate it achieves, with CPU saturated, is also 25% lower. However, a far larger number of triplets are required in order to saturate CPU when using JMS. This is likely to be caused by contention on some part of the networking subsystem (e.g. input or output buffer queues) caused by the fact that JMS messages require a larger header, and so take up more space and bandwidth.

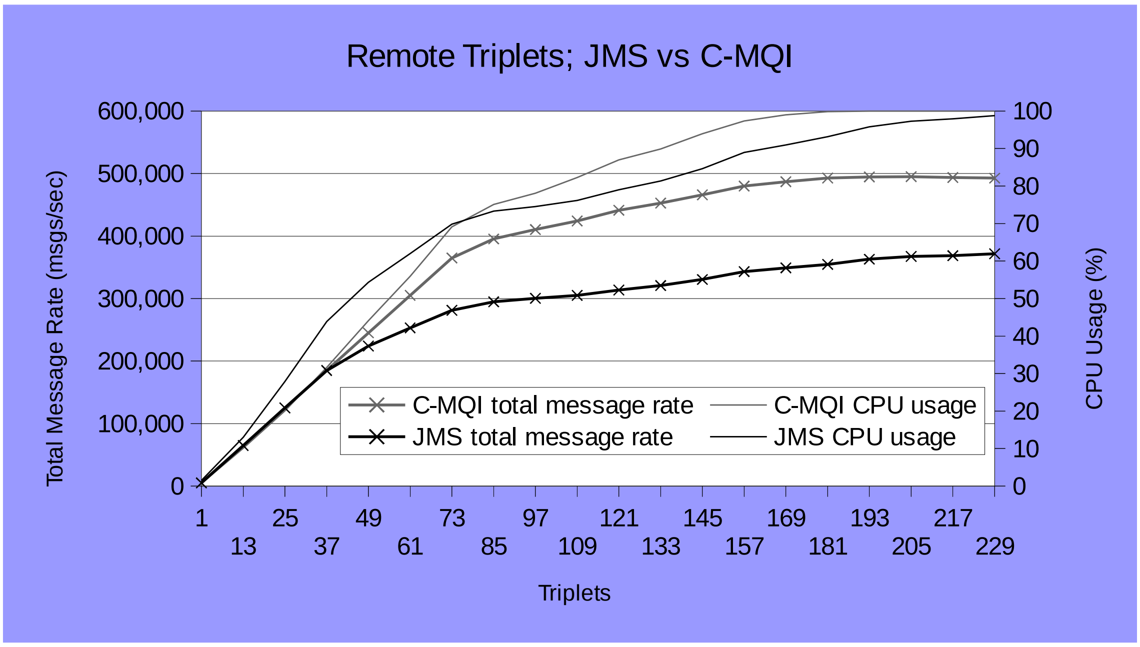

Chart 26: JMS vs C-MQI; Remote Triplets, Non-Persistent

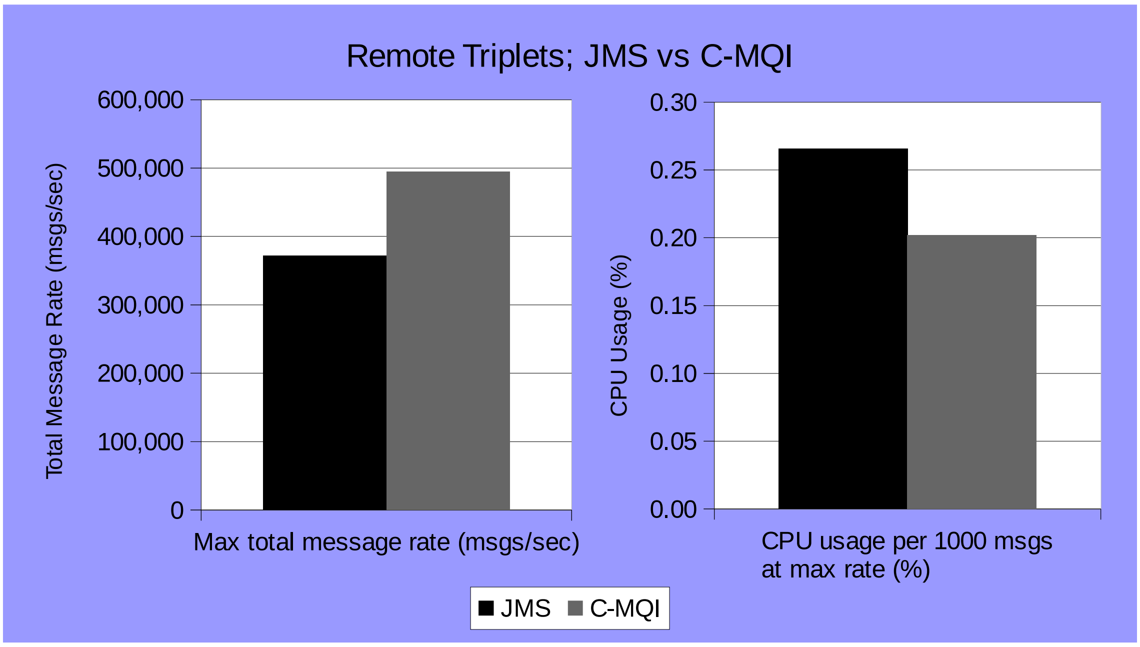

Chart 27: JMS vs C-MQI; Remote Triplets, Non-Persistent; Peak Rates

5.3 Conclusion

In conclusion, it appears that the JMS protocol adds an additional 20-35% cost to publish-subscribe messaging in IBM MQ, when compared with the MQI interface.

6 — Publish-Subscribe Clustering

There are a number of reasons to perform publish-subscribe messaging across a cluster of queue managers: either to provide messaging across a physically distributed MQ infrastructure or to improve availability by providing redundancy across queue managers. Alternatively it may be advantageous to use a clustered topology to spread the load of a large publish-subscribe system, even within a single machine. For example, if you have a very high number of subscribers (greater than 60,000) then it may be preferable to split these across a number of queue managers. This will reduce the impact of a queue manager restart by reducing the time taken to restart and by only impacting a subset of the subscriptions in the system. It will also simplify the administration of each individual queue manager.

It is important to understand that there is a performance overhead associated with publish-subscribe clustering in IBM MQ due to the requirement to send messages between the queue managers in order for them to reach their target. This is the case whenever publishers are connected to different queue managers to subscribers of the same topic.

When creating a publish-subscribe cluster, administrative topic objects are configured with the *CLUSTER* attribute defined[4]. Information about each clustered topic object is sent to all queue managers in a cluster. By default, when the first application subscribes to a topic string at or below a clustered topic, a proxy subscription for that topic string is sent from the queue manager where the subscription is created to all the other members of the cluster. This means that there are additional messages flowing between queue managers whenever subscriptions are created and deleted. This can affect the maximum message rate which can be sustained, especially if there is a high rate of change of subscriptions across many different topic strings.

Messages published on a topic are sent to every matching subscription known to the queue manager that the publisher is connected to. For any proxy subscriptions, a single copy of the published message is sent to the queue manager that originated the proxy subscription. The receiving queue manager then delivers a copy of the message to every local subscription matching that topic. This ensures that the subscriber to a clustered topic receives publications from publishers connected to any of the queue managers in the cluster.

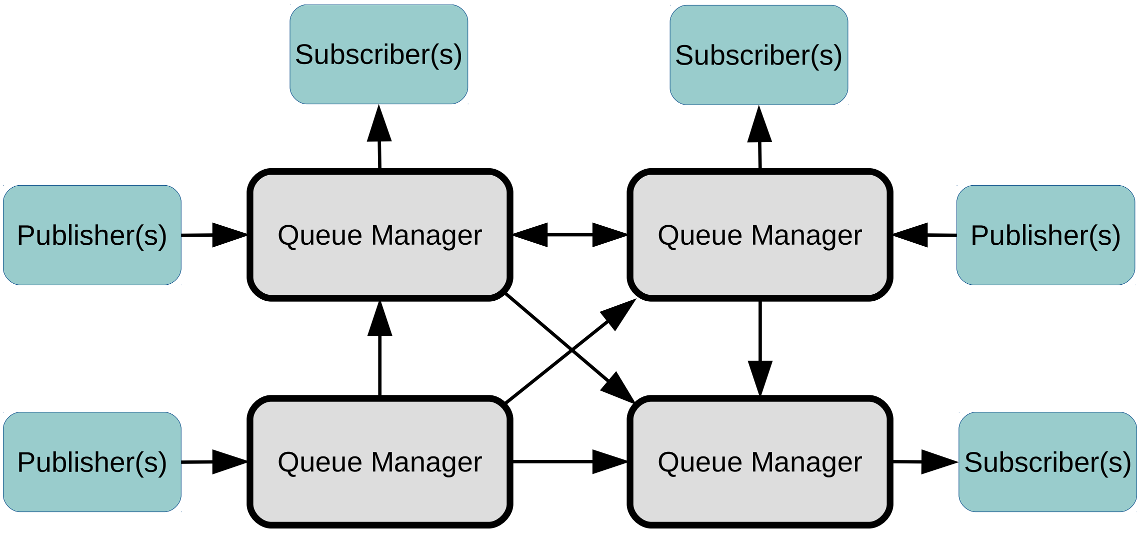

Figure 10: Message flows for an unhosted clustered topic

[4] www-01.ibm.com/support/knowledgecenter/SSFKSJ_8.0.0/com.ibm.mq.pro.doc/q005221_.htm

6.1 Routed Topics

In IBM MQ V8, a new feature has been introduced which allows finer control over the architecture of a publish-subscribe cluster, and can in some cases reduce the total number of messages flowing through the cluster.

When a topic is defined administratively, the new *CLROUTE* property can be set to one of two values[5]:

- *DIRECT* – Clustered topic is defined as described above, with messages flowing directly from the publishing QM to all those with subscriptions defined. This is the default, and the behaviour of publish-subscribe clustering in IBM MQ versions prior to V8.

- *TOPICHOST* – The queue manager on which the topic is defined acts as the 'host' for this topic. All messages on the branch of the topic tree below this topic that need to be forwarded to additional queue managers are forwarded *only* to the host, which then passes them on to any queue managers with subscriptions defined. Multiple queue managers can host the same clustered topic, which will result in workload balancing of publications across all topic hosts, reducing the publication overhead to each.

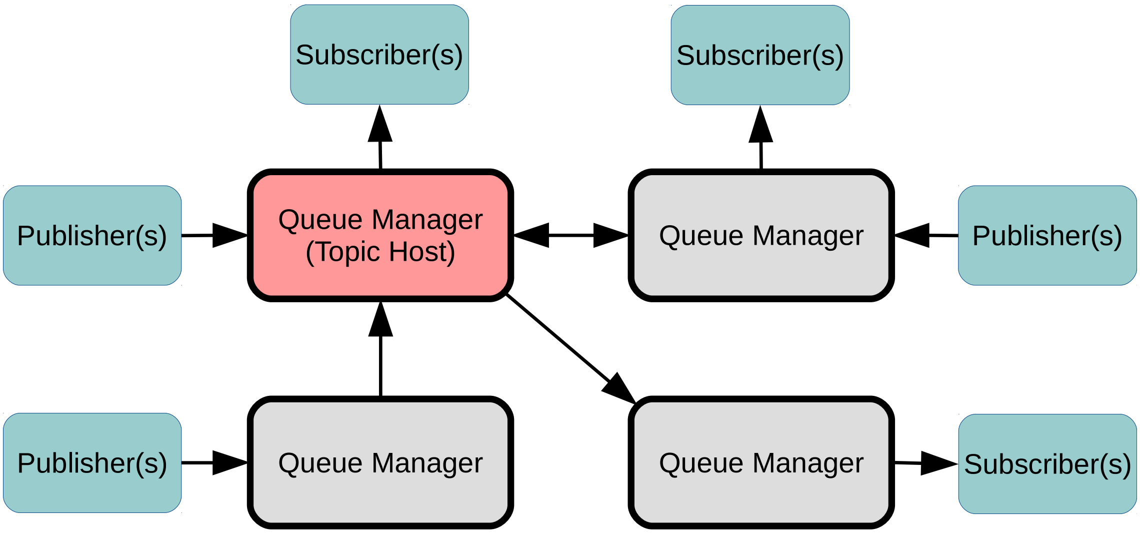

Figure 11: Message flows for a hosted clustered topic

Figures 10 & 11 illustrate the difference in message flow topologies between a traditional “direct” clustered topic, and a hosted or “routed” one, for an example scenario involving a single topic. Note that these diagrams do *not* illustrate the administrative messages sent between queue managers in the cluster when subscriptions are created and destroyed.

There are various reasons why one might choose to use routed topics; for instance:

- To reduce requirements for queue-manager to queue-manager connectivity.

[5] www-01.ibm.com/support/knowledgecenter/SSFKSJ_8.0.0/com.ibm.mq.con.doc/q017435_.htm

- To reduce the volume of administrative messages in situations where subscribers frequently come and go.
- To offload bookkeeping and forwarding work to the most capable machines in the cluster.
- To simplify administration and auditing of messages and subscribers on a topic.

Note, however, that routed topics are not designed to improve performance of messaging in a cluster. In fact, as we shall see, in situations where subscribers don't come and go frequently, using routed topics can introduce a performance overhead, particularly in the cases where messages are now passed through a 3rd intermediary queue manager.

6.2 Publish-subscribe cluster performance

To look at the performance of publish-subscribe clustering, we'll compare 3 simple scenarios, each with unidirectional message flows, as described in Figure 12.

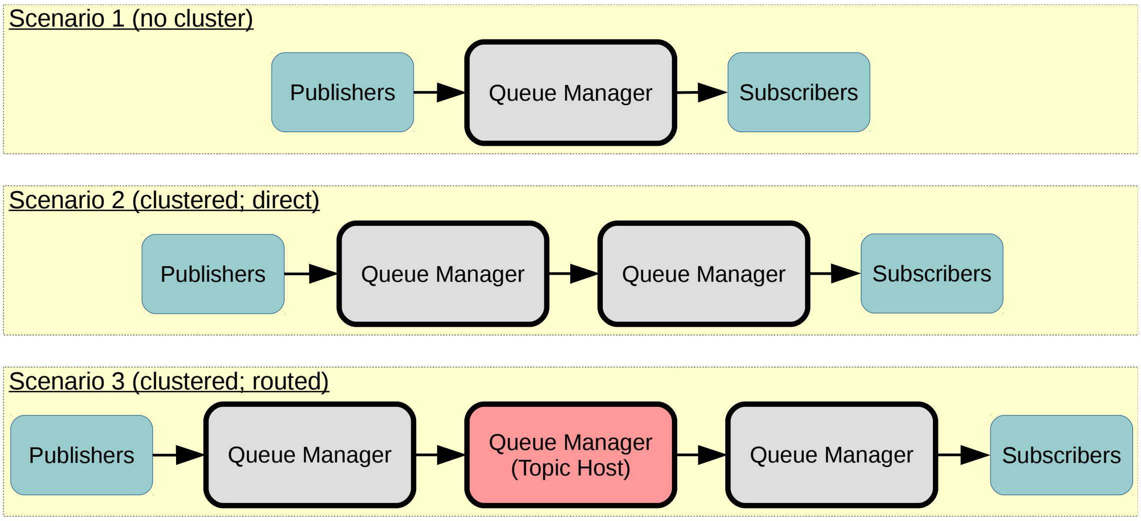

Figure 12: Publish-subscribe cluster test scenarios

In each scenario, we'll employ 4 'triplets', as described in section 4 (page 29). However, for each scenario we'll compare the highest total message rate achievable across all 4 triplets without a build-up or bottle-neck occurring. All topics used will be administratively defined, so as to be 'isolated' from the rest of the hierarchy (see section 2.2.1).

In scenarios 2 and 3, both the publishers and subscribers will be local to the queue manager they connect to. In scenario 1, however, the publishing applications will connect using client bindings from a different machine. This is to introduce an element of networked communication to scenario 1, to make it a fairer comparison with scenarios 2 and 3, which both have this bottle-neck.

All messages in this scenario will be non-persistent.

Note that the machines used in this section are different to those used for tests in the rest of the report. In the results that follow, all machines involved are small, 4-core devices, with a single NUMA zone (see appendix A.1.2).

6.2.1 Initial Comparison

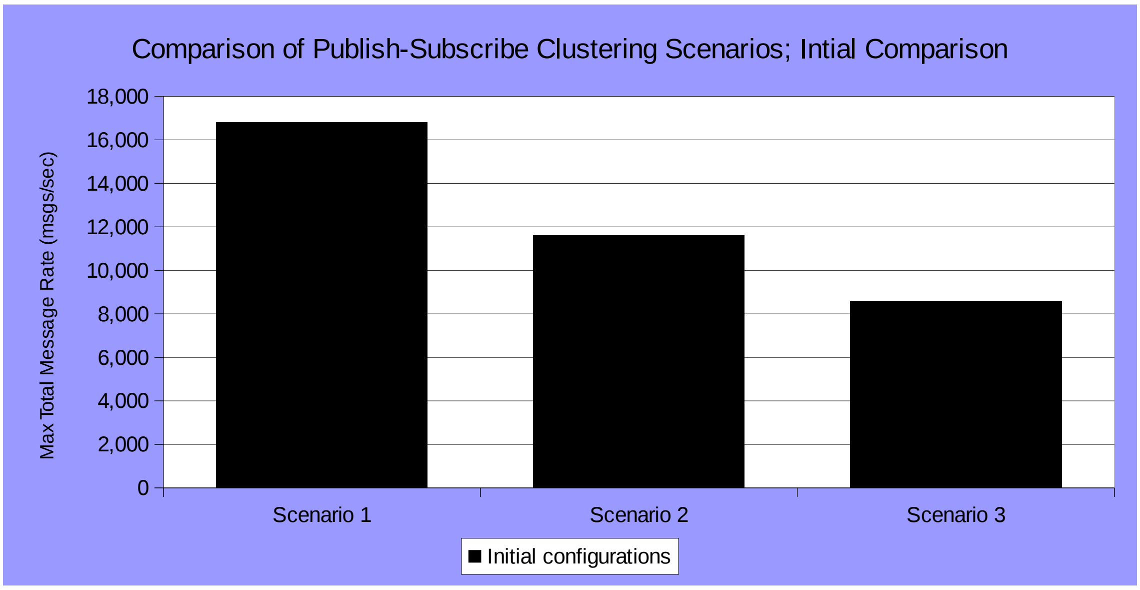

Chart 28: Comparison of Publish-Subscribe Clustering Scenarios; Initial Comparison

Using default configurations, we can see from Chart 28 that sending messages via a clustered topic between two machines (scenario 2) achieves a 31% lower throughput when compared with publishing messages over remote, client-bindings to achieve the same delivery (scenario 1). Routing the messages via an intermediate topic host (scenario 3) results in a further 26% drop in throughput.

The primary reason for the relatively poor performance of clustered topics, when compared to client application bindings, is that by default, transmission of messages between queue managers, and the subsequent republication of messages on the destination queue manager, occurs in a single thread. Some tuning of the queue manager configuration, to suit the hardware its running on, can lead to significant improvements, as the sections below will demonstrate.

6.2.2 pscNumPubThreads

In clustered publish-subscribe, when a publication is sent between queue managers by the cluster send/receive channels, it is placed on the *SYSTEM.INTER.QMGR.PUBS* queue. From there, the publish-subscribe engine retrieves the messages and republishes them locally, and to any additional proxy subscriptions (if the queue manager is the topic host). By default, a single thread is created to perform this task, this is to ensure that messages are republished in the same order in which they arrive on the queue manager.

However, if strict message-ordering is not essential, the number of threads servicing the *SYSTEM.INTER.QMGR.PUBS* queue can safely be increased, in order to process the publications on this queue faster. The number of such threads is controlled by setting the *pscNumPubThreads* parameter in the *TuningParameters* stanza of the queue manager's *qm.ini* file.

In Chart 29, below, we show the additional throughput attained by setting *pscNumPubThreads* to 5, one more than the number of CPU cores on each machine. The result is a 9% improvement to the throughput in scenario 2, and a 27% improvement in scenario 3.

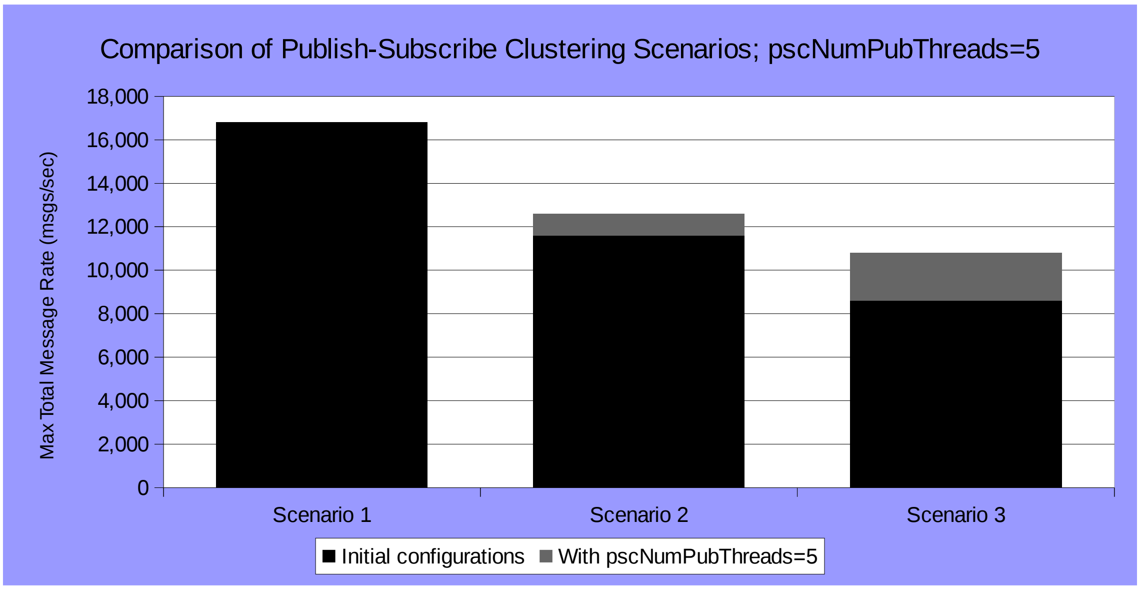

Chart 29: Comparison of Publish-Subscribe Clustering Scenarios; pscNumPubThreads=5

While this improvement is not insignificant, it still leaves both clustered scenarios wanting in comparison to scenario 1, with scenario 2 achieving 25% fewer messages, and scenario 3 achieving 36% fewer. Increasing *pscNumPubThreads* further does not yield any further improvements, and setting it too high actually leads to a degradation in throughput, due to increased lock negotiation between the threads. Typically, it would be expected that the optimal number of threads would be close to the number of available CPU cores on the machine.

In fact, setting *pscNumPubThreads* to 5 in these scenarios has lead to a greater widening of the bottle-neck caused by the republishing thread than the results suggest. The problem is that an additional bottle-neck exists: in the message channels transmitting messages between queue managers.

6.2.3 Multiple Cluster Channels

When a message must be transmitted between queue managers in a cluster, by default it is placed on the *SYSTEM.CLUSTER.TRANSMIT.QUEUE*. From there messages are retrieved by *CLUSSDR* message channel agents and transmitted across the network to other queue managers in the cluster. The number of *CLUSSDR* channel agent threads sending to each destination queue manager is determined by the number of corresponding *CLUSRCVR* channel objects defined on that destination. The minimum number required to achieve connectivity to a queue manager in a cluster is 1, and as with *pscNumPubThreads*, only 1 must be used if messages are to arrive at the destination QM in the same order that they arrived at the source QM.

However, if strict message ordering is not required, we are free to define additional *CLUSRCVR* channels so that, if there is spare CPU capacity on both the source and destination machines, and additional available network bandwidth, the transmission rate of messages between queue managers may be increased.

Since we have already broken the message-ordering guarantee by increasing *pscNumPubThreads*, in Chart 30 below we also increase the number of *CLUSRCVR* channels on each QM to 4, the number of cores on the machine.

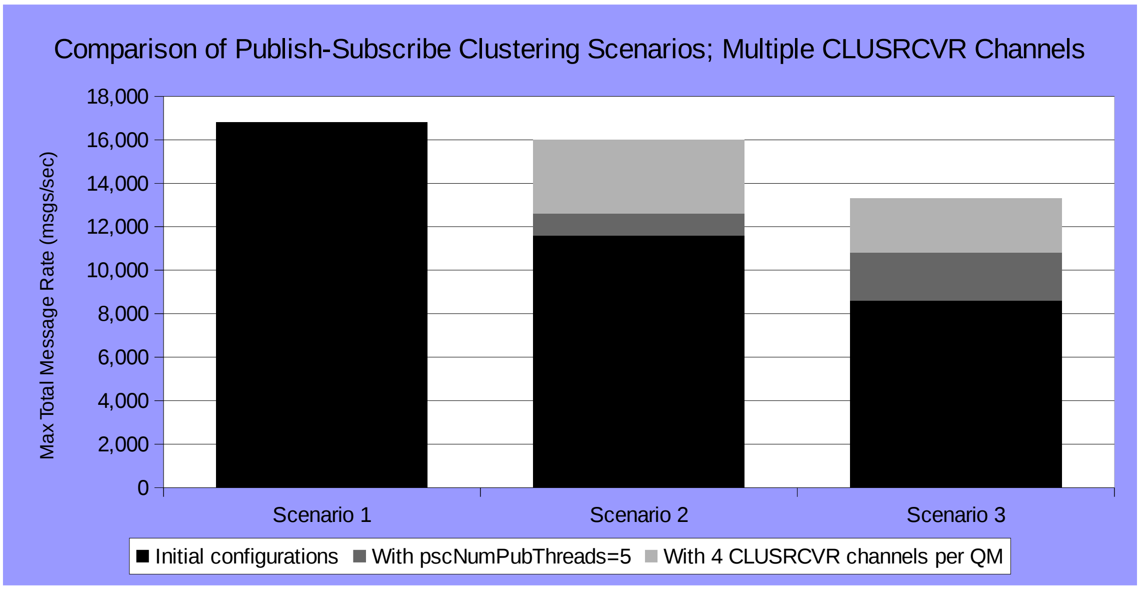

Chart 30: Comparison of Publish-Subscribe Clustering Scenarios; Multiple CLUSRCVR Channels

The result is that scenario 2 now performs almost as well as scenario 1, with only a 5% loss in peak throughput. The throughput of scenario 3 is still a further 17% lower than that of scenario 2, and during the test, CPU usage information suggested that the bottle-neck now lies in available CPU on the topic host, whose channels must both send and receive receive messages, and as such has significantly more work to do than any of the other machines used in any of the scenarios.

It is possible to have multiple queue managers act as hosts to the same topic, in which case, one could alter scenario 3 to include 2 intermediate, topic-host queue managers in parallel. As a result, the workload would be balanced across both of them, thus increasing the total CPU and networking capacity of the infrastructure. Although this has not been tested in this report, one should expect that if the circumstances were right, this should lead to an increase in the total message rate that a publish-subscribe cluster using hosted topics can achieve. Note, however, that in the scenarios tested here, even with a second topic-host queue manager, we would not expect the performance of scenario 3 to exceed that of scenario 2, as the bottle-necks constraining scenario 2 are also present in scenario 3, in the end-point queue managers that the applications are connected to.

In these tests, the hardware used has meant that CPU limits, either of a single thread or the entire machine, were reached before network bandwidth was exhausted. If larger machines or slower networks were used, the result could have been the other way round, in which case we may have seen the peak throughput of scenario 3 being just as high as that of scenario 2. Also, had the number of CPU cores on the machines been much larger, allowing us to effectively employ more cluster channels per QM, we may have encountered a further bottle-neck: locking on the *SYSTEM.CLUSTER.TRANSMIT.QUEUE*. This could have been relieved by making use of multiple cluster transmit queues, a feature introduced in IBM MQ V7.5[6]. In this particular scenario, however, no additional gains were observed by using multiple cluster transmit queues.

[6] *www-01.ibm.com/support/knowledgecenter/SSFKSJ_8.0.0/com.ibm.mq.pla.doc/q004790_.htm*

Appendix A — Machine Configurations

A.1 Hardware

A.1.1 Tests in sections 2–5

Depending on the scenario, up to 3 machines were used:

- 2 x IBM System x3550 M4, with:
  - Intel Xeon E5-2679 v2 (24 cores @ 2.70GHz; hyper-threading off)
  - 96GB RAM (split over 2 NUMA zones)
  - 10Gb/s network link to private LAN
- 1 x IBM System x3850, with:
  - 4 x Intel Xeon E7-8870 (in total, 40 cores @ 2.4GHz; hyper-threading off)
  - 64GB RAM (split over 4 NUMA zones)
  - 10Gb/s network link to private LAN

In each scenario, the queue manager was located on one of the IBM System x3550 M4 machines, with the remaining two machines being used to host client applications whn they were remote (client-bound).

A.1.2 Tests in section 6

In all 3 scenarios, each queue manager was hosted on a separate, identical machine of the following specification:

- IBM System x3850, with:
  - Intel Xeon CPU (4 cores @ 3.33GHz; hyper-threading off)
  - 24GB RAM (single NUMA zone)
  - 10Gb/s network link to private LAN

In scenario 1, publisher applications were located on a separate one of these machines from the queue manager. In all other applications in all scenarios were located on the same machine as the queue manager they connected to.

A.2 OS (all tests)

On all machines used, the operating system was Red Hat Enterprise Linux, version 6.6. The following customizations were made to the OS configuration:

- In the *grub.conf* boot-loader configuration file, the kernel command line was appended with the arguments "`intel_idle.max_cstate=0 idle=poll`". This prevents the CPU cores from dropping into less responsive, energy-saving "*c-states*", which have been shown to degrade performance at low numbers of clients.

- Similarly, the *cpuspeed* CPU throttling daemon was disabled for all run-levels, by renaming any files named *S<number>cpuspeed* to *K99cpuspeed* in any of the */etc/rc.d/rc<number>.d* directories.

- The maximum number of open file handles and running processes was increased for users in the *mqm* group[7], by creating the file */etc/security/limits.d/mqm.conf* with the following contents:

  ```
  @mqm soft nofile 1048576
  @mqm hard nofile 1048576
  @mqm soft nproc  1048576
  @mqm hard nproc  1048576
  ```

  Note, however, that in all tests, all applications were run by a user who is a member of the *mqm* group. If, as is recommended, the *mqm* group is reserved for MQ administrators, and you will be running very large numbers of applications as a user not in this group, then you may also need to increase the security limits for this user.

- The following changes were made to */etc/sysctl.conf* to increase som OS-wide limits, and to tune the behaviour of the OS task scheduler to suit MQ:

  ```
  net.ipv4.ip_local_port_range = 1024 65535
  vm.max_map_count = 1966080
  kernel.pid_max = 4194303
  kernel.sem = 1000 1024000 500 8192
  kernel.msgmnb = 131072
  kernel.msgmax = 131072
  kernel.msgmni = 32768
  kernel.shmmni = 8192
  kernel.shmall = 4294967296
  kernel.shmmax = 137438953472
  kernel.sched_latency_ns = 2000000
  kernel.sched_min_granularity_ns = 1000000
  kernel.sched_wakeup_granularity_ns = 400000
  ```

[7] www-01.ibm.com/support/knowledgecenter/SSFKSJ_8.0.0/com.ibm.mq.sec.doc/q014050_.htm

A.3 MQ Configuration

- In all tests, the following parameters were set in any queue manager's *qm.ini* file:

```
Channels:
   MQIBindType=FASTPATH
   MaxActiveChannels=5000
   MaxChannels=5000
ExitPath:
   ExitsDefaultPath=/var/mqm/exits
   ExitsDefaultPath64=/var/mqm/exits64
Log:
   LogBufferPages=16
   LogFilePages=16384
   LogPath=/var/mqm/log/<QM_NAME>/
   LogPrimaryFiles=16
   LogSecondaryFiles=2
   LogType=CIRCULAR
   LogWriteIntegrity=TripleWrite
Service:
   EntryPoints=14
   Name=AuthorizationService
ServiceComponent:
   ComponentDataSize=0
   Module=amqzfu
   Name=MQSeries.UNIX.auth.service
   Service=AuthorizationService
TCP:
   ClntRcvBuffSize=0
   ClntSndBuffSize=0
   RcvBuffSize=0
   RcvRcvBuffSize=0
   RcvSndBuffSize=0
   SndBuffSize=0
   SvrRcvBuffSize=0
   SvrSndBuffSize=0
TuningParameters:
   DefaultPQBufferSize=1048576
   DefaultQBufferSize=1048576
   pscNumPubThreads=5
```

- All applications that connected using client-bindings connected via the *SYSTEM.DEFAULT.SVRCONN* channel, which had *SHARECNV(1)* set on it in *MQSC*[8].

[8] www-01.ibm.com/support/knowledgecenter/SSFKSJ_8.0.0/com.ibm.mq.mon.doc/q036143_.htm