WebSphere® MQ Telemetry V7.5
Performance Evaluations

Version 1.0

January 2013

Paul Harris

WebSphere MQ Performance
IBM Hursley

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.

First Edition, January 2013.

This edition applies to *WebSphere MQ V7.5* (and to all subsequent releases and modifications until otherwise indicated in new editions).

Notices

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

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

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

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

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

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

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

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

INTENDED AUDIENCE
This report is intended for architects, systems programmers, analysts and programmers wanting to understand the performance characteristics of *WebSphere MQ Telemetry V7.5*.

The information is not intended as the specification of any programming interface that is provided by WebSphere. It is assumed that the reader is familiar with the concepts and operation of WebSphere MQ V7.5.

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 International Business Machines Corporation in the United States, other countries or both:

- IBM
- WebSphere

Java and all Java-based trademarks and logos are trademarks of Oracle, Inc. in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

This report presents the results of performance evaluations using the MQTT clients supplied with WebSphere MQ Telemetry 7.5 for AIX, Linux on System x and Linux on System z and is intended to assist with programming and capacity planning.

Target audience

This SupportPac is designed for people who:

- Will be designing and implementing solutions using WebSphere MQ Telemetry.
- Want to understand the performance limits of WebSphere MQ Telemetry.
- Want to understand what actions may be taken to tune WebSphere MQ Telemetry.

The reader should have a general awareness of the Java programming language, supported operating systems, WebSphere MQ and of WebSphere MQ Telemetry in order to make best use of this SupportPac.

The contents of this SupportPac

This SupportPac includes:

- Charts and tables describing the performance headlines of WebSphere MQ Telemetry V7.5 using IBM MQTTv3 clients.
- WebSphere MQ Telemetry messaging using AIX, Linux on System x and Linux on System z.

Feedback on this SupportPac

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?

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

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

CONTENTS

1 Overview 1
2 Test Scenarios 2
2.1 Multi-publisher, single-subscriber 2
2.2 Few-publishers, multi-subscriber 3
2.3 Test Calibration 3
3 AIX Results 5
3.1 Multi-publisher, single-subscriber 5
Few-publishers, multi-subscriber 8
4 Linux on System x Results 11
4.1 Multi-publisher, single-subscriber 11
4.2 Few-publishers, multi-subscriber 14
5 Linux on System z Results 17
5.1 Multi-publisher, single-subscriber 17
5.2 Few-publishers, multi-subscriber 20
6 Test Calibration Example 23
7 Tuning Parameters and Considerations 25
7.1 Tuning the queue manager 25
7.2 Tuning the heap size for Java 25
7.2.1 JVM Warmup 26
7.3 7.3 The Telemetry Server and Java Considerations 26
7.4 Shared Conversations 26
7.5 Avoiding running in Migration/Compatibility Mode 26
7.6 Use of Correlation Identifiers 27
7.7 Other Programming Recommendations 27
7.8 JMS Persistence 28
JMS delivery mode 28
8 Machine and Test Configurations 30
8.1 AIX 30
8.2 Linux on System x 30
8.3 Linux on System z 30
8.4 SAN disk subsystem 31

FIGURES

Figure 1: Multi-publisher, single-subscriber setup 2
Figure 2: Few-publishers, multi-subscriber 3
Figure 3: AIX/QoS0 Multi-publisher, single-subscriber Results 5
Figure 4: AIX/QoS1 Multi-publisher, single-subscriber Results 6
Figure 5: AIX/QoS2 Multi-publisher, single-subscriber Results 7
Figure 6: AIX/QoS0 Few-publishers, multi-subscribers Results 8
Figure 7: AIX/QoS1 Few-publishers, multi-subscribers Results 9
Figure 8: AIX/QoS2 Few-publishers, multi-subscribers Results 10
Figure 9: Linux on System x/QoS0 Multi-publisher, single-subscriber Results 11
Figure 10: Linux on System x/QoS1 Multi-publisher, single-subscriber Results 12
Figure 11: Linux on System x/QoS2 Multi-publisher, single-subscriber Results 13
Figure 12: Linux on System x/QoS0 Few-publishers, multi-subscribers Results 14
Figure 13: Linux on System x/QoS1 Few-publishers, multi-subscribers Results 15
Figure 14: Linux on System x/QoS2 Few-publishers, multi-subscribers Results 16
Figure 15: Linux on System z/QoS0 Multi-publisher, single-subscriber Results 17
Figure 16: Linux on System z/QoS1 Multi-publisher, single-subscriber Results 18
Figure 17: Linux on System z/QoS2 Multi-publisher, single-subscriber Results 19
Figure 18: Linux on System z/QoS0 Few-publishers, multi-subscribers Results 20
Figure 19: Linux on System z/QoS1 Few-publishers, multi-subscribers Results 21
Figure 20: Linux on System z/QoS2 Few-publishers, multi-subscribers Results 22
Figure 21: Overloaded Subscriber – Thorughput/CPU Utilisation 23
Figure 22: QDepth and Disk I/O Metrics 24

1 Overview

The two MQTT Publish Subscribe scenarios in this report are:

1) Multi-publisher, single-subscriber.
2) Few-publishers, multi-subscriber.

These are measured on three operating systems (AIX, Linux on System x and Linux on System z) and reported with three qualities of service (from here on referred to as QoS) as defined by the MQTT V3:

1) QoS level 0 - at most once delivery. This is the fastest method of messaging using the protocol, but is trading assured delivery for performance.
2) QoS level 1 - at least once delivery. Ensures a message is received by the receiver at least once (i.e. duplicates are allowed).
3) QoS level 2 - exactly once delivery. This is the best level of service achievable using the MQTT protocol. Using this method with WebSphere MQ Telemetry will ensure a message will always reach its destination exactly once.

All client were connected with

Due to the differences between the hardware used for each operating system, it is not possible to compare throughput across operating systems.

- The message size used in all tests is 256 Bytes.
- Depending on platform, message rates for each scenario are recorded with up to 100,000 connected MQTT clients.
- Each sample point reported is the average of two minutes of data collection.

The graphs in this report show the number of messages per second that are processed by *all* the connected applications or clients. So for example, a Publisher publishing at 100 msgs/sec with two Subscribers would result in a throughput of 300 msgs/sec.
The tables in the report show the peak throughput achieved and the number of connected applications and CPU usage at the peak throughput.

2 Test Scenarios

2.1 Multi-publisher, single-subscriber

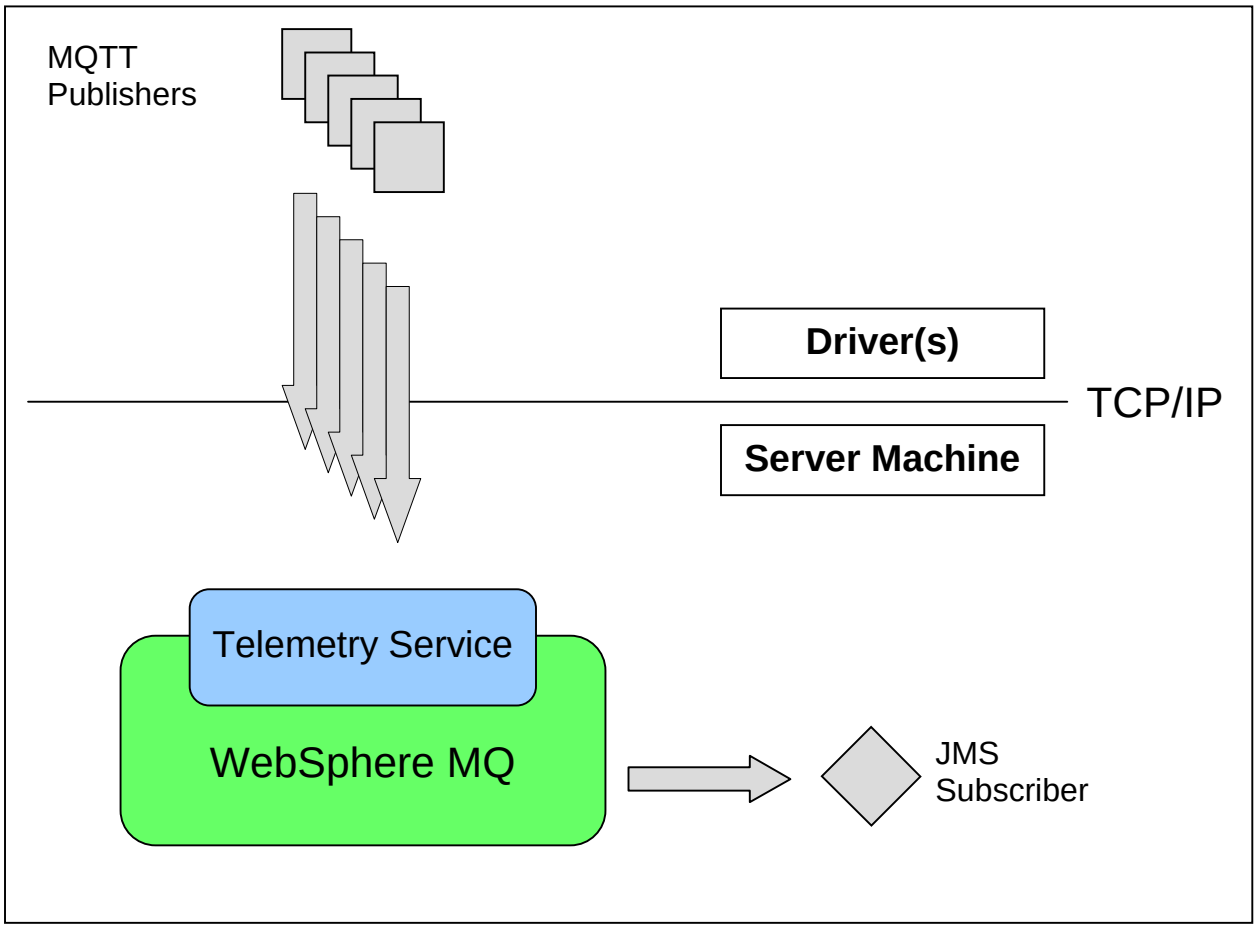

Figure 1: Multi-publisher, single-subscriber setup

In this scenario, messages are published to the telemetry server by MQTT clients from a driver machines at a fixed rate. They publish to a single topic (called TOPIC1).

A single JMS consumer local to the telemetry server is used to take messages off a managed subscription queue for that topic. The JMS consumer is connected to the queue manager using client bindings.

2.2 Few-publishers, multi-subscriber

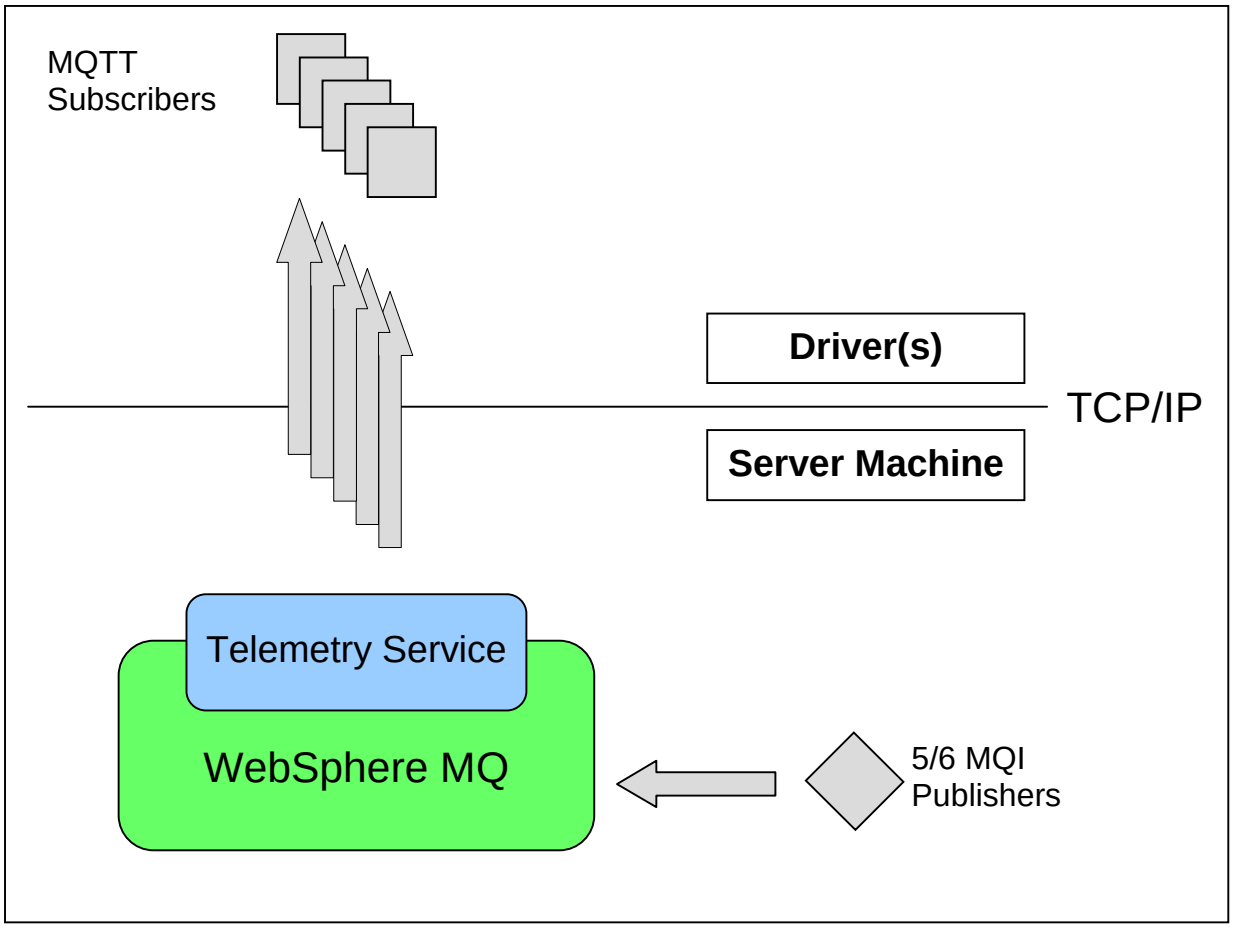

Figure 2: Few-publishers, multi-subscriber

In this scenario MQTT clients are located on remote driver machines and subscribe to unique topics. A small number of MQI publishers (5 or 6) loop though the topics (in the range TOPIC1 - TOPIC100000) at a fixed rate and attempt to publish a message to each of the topics one at a time. Because clients are being connected gradually, publications to the topics that do not have matching subscriptions are discarded. As the test progresses the number of discarded publications decreases resulting in a higher total message rate.

The MQI publisher is connected to the queue manager using client bindings. Whilst the mirror of the multi-publisher, single-subscriber test would be a single-publisher, multi-subscriber test, in practice it was found that more than one publisher is required before the limit of the test is met (the rate at which the subscribers can consume messages).

Note that for these tests an additional queue manager tuning parameter was set:

TuningParameters / DurableSubMgrFrequency = 60

Some housekeeping is periodically performed on durable subscriptions. This is carried out every 30 minutes by default. With the nature of this test (lots of subscriptions) this can cause a CPU spike at 30 minutes. Reducing this time to 1 minute (60 seconds) spreads the work out more evenly.

2.3 Test Calibration

Note that all tests in this report are rated tests. This means that a calibrated rate is set taking the overall throughput close to the limit for the test. The limiting factor in the Multi-publisher, Single subscriber tests is the rate at which a single subscriber can read from its queue: SYSTEM.MANAGED.NDURABLE.xxxxxxxxx or SYSTEM.MANAGED.DURABLE.xxxxxxxxx depending on the test, where xxxxxxxxx is an internally

generated value associated with the subscriber. Once this queue starts to build up performance degrades. See section 6 for a description of the calibration for one of the AIX scenarios.

Many of the tests do not come close to exhausting CPU on the server machine. In this case, the solution would be scaled up by having multiple topics or queue managers in the case of the Multiple-Producer, Single Subscriber tests for example. The main purpose of the measurements below are to show that the product scales up to a large number of client connections without issue.

3 AIX Results

All AIX measurements used a 64-bit P7 server (see section 8.1)

3.1 Multi-publisher, single-subscriber

See section 2.1 for an overview of these scenarios.

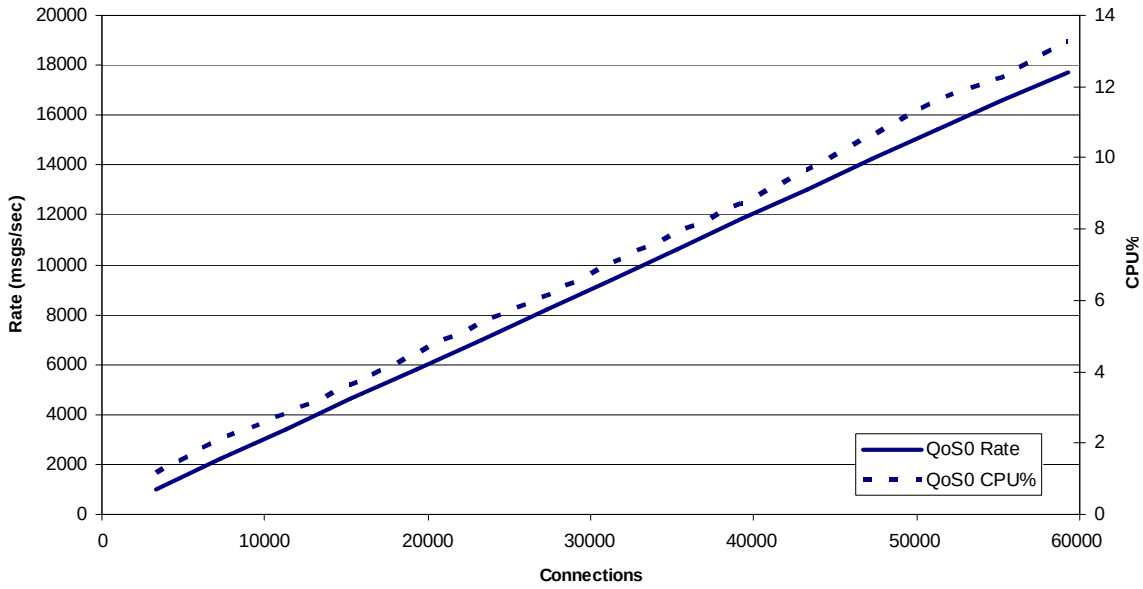

Figure 3: AIX/QoS0 Multi-publisher, single-subscriber Results

Other Platforms:
Figure 9: Linux on System x/QoS0 Multi-publisher, single-subscriber Results
Figure 15: Linux on System z/QoS0 Multi-publisher, single-subscriber Results

Publishers were added in batches of 4000 with every 1000 additional publishers running in a new client JVM. The JVMs were spread across two client machines (see section 8). For AIX the tests were limited to 60,000 clients. A single subscriber consumed the messages.

The publish rate for each publisher was 0.15 messages/sec so for every 4000 publishers there were potentially 4000 * 2 * 0.15 msgs/sec through the system = 1200/sec (18,000/sec at 60K publishers).

For QoS0 the test scaled cleanly to 60K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

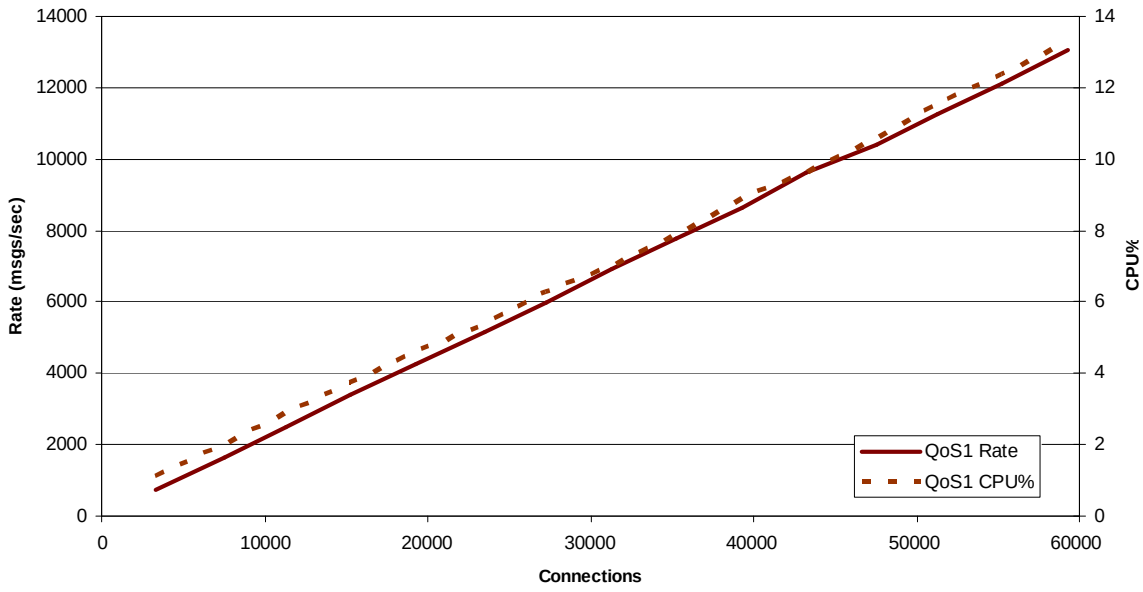

Figure 4: AIX/QoS1 Multi-publisher, single-subscriber Results

Other Platforms:
Figure 10: Linux on System x/QoS1 Multi-publisher, single-subscriber Results
Figure 16: Linux on System z/QoS1 Multi-publisher, single-subscriber Results

The QoS1 test was setup identically to the QoS0 test but the publish rate for each publisher was 0.11 messages/sec so for every 4000 publishers there were potentially 4000 * 2 * 0.11 msgs/sec through the system = 880/sec (13,200/sec at 60K publishers).

The QoS1 measurements scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

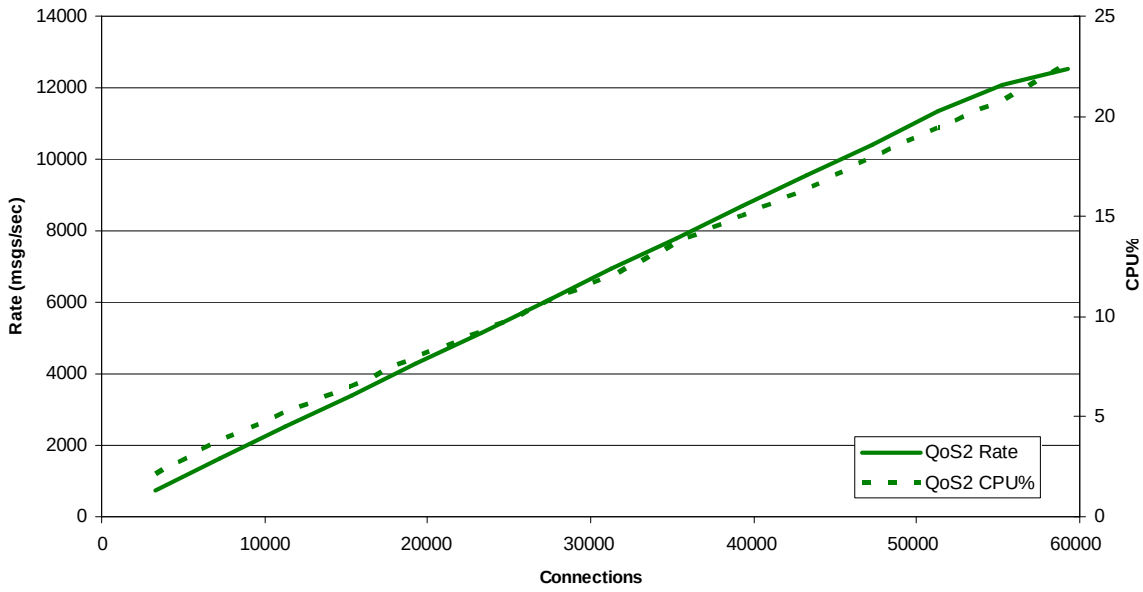

Figure 5: AIX/QoS2 Multi-publisher, single-subscriber Results

Other Platforms:
Figure 11: Linux on System x/QoS2 Multi-publisher, single-subscriber Results
Figure 17: Linux on System z/QoS2 Multi-publisher, single-subscriber Results

The QoS2 test was setup identically to the QoS0 test but the publish rate for each publisher was 0.11 messages/sec so for every 4000 publishers there were potentially 4000 * 2 * 0.11 msgs/sec through the system = 880/sec (13,200/sec at 60K publishers).

The QoS2 measurements scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

Few-publishers, multi-subscriber

See section 2.2 for an overview of these scenarios.

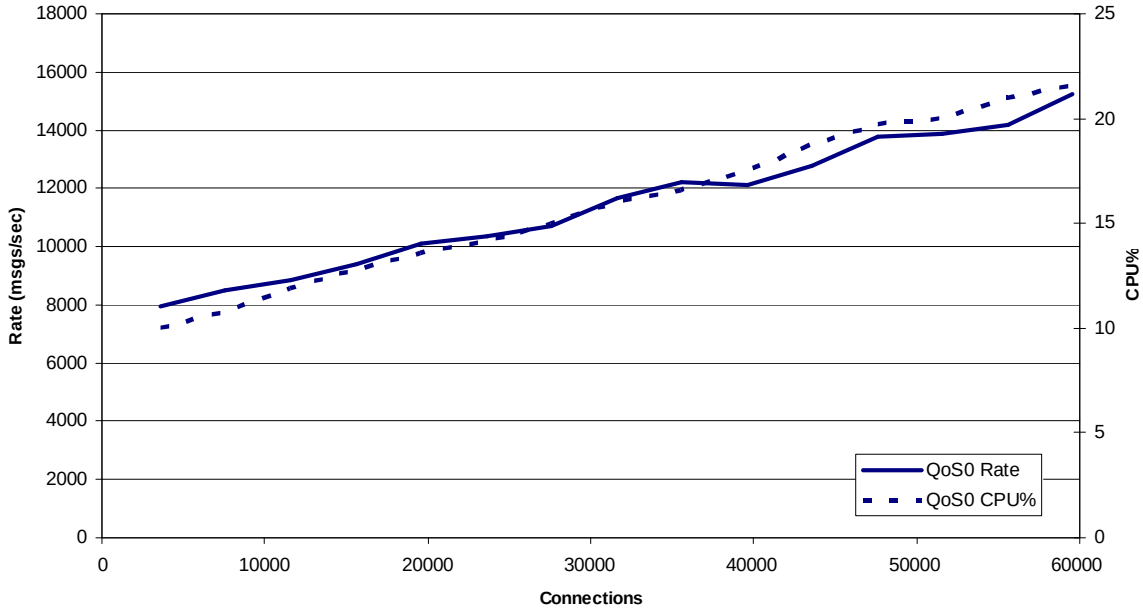

Figure 6: AIX/QoS0 Few-publishers, multi-subscribers Results

Other Platforms:
Figure 12: Linux on System x/QoS0 Few-publishers, multi-subscribers Results
Figure 18: Linux on System z/QoS0 Few-publishers, multi-subscribers Results

For this test 5 publishers were started with a rate of 1500/sec (total 7500/sec). As each publisher publishes onto all eventual topics (60,000) the rate of arrival on each topic is 7500/60,000 = 0.125 msgs/sec.

The theoretical maximum rate of messages in the system at any point in the test is then:

Total message rate of publishers + 0.125 * connected subscribers (messages published to a topic for which there are no subscribers are discarded).

= 7500 + n * 0.125

When all subscribers are connected the theoretical rate will simply be 2 times the publish rate (15,000/sec at 60K subscribers).

The graph above follows the theoretical maximum very closely, exhibiting good scaling.

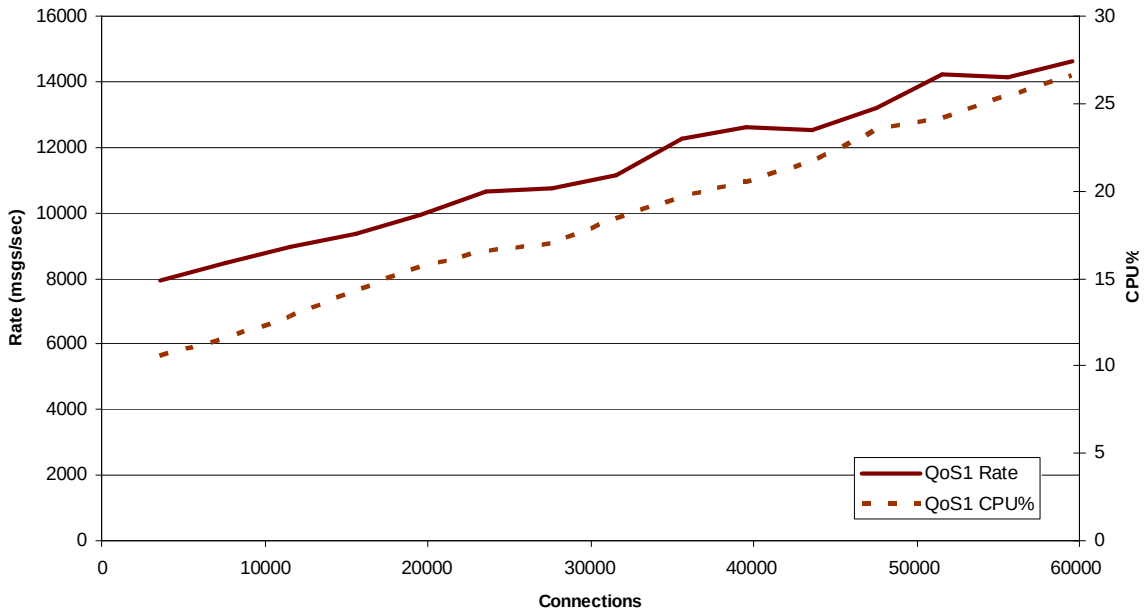

Figure 7: AIX/QoS1 Few-publishers, multi-subscribers Results

Other Platforms:
Figure 13: Linux on System x/QoS1 Few-publishers, multi-subscribers Results
Figure 19: Linux on System z/QoS1 Few-publishers, multi-subscribers Results

The QoS1 test was setup identically to the QoS0 in terms of rates so the theoretical maximum rate of messages in the system at any point in the test is

= 7500 + n * 0.125

When all subscribers are connected the theoretical rate will simply be 2 times the publish rate (15,000/sec at 60K subscribers).

The graph above follows the theoretical maximum very closely, exhibiting good scaling.

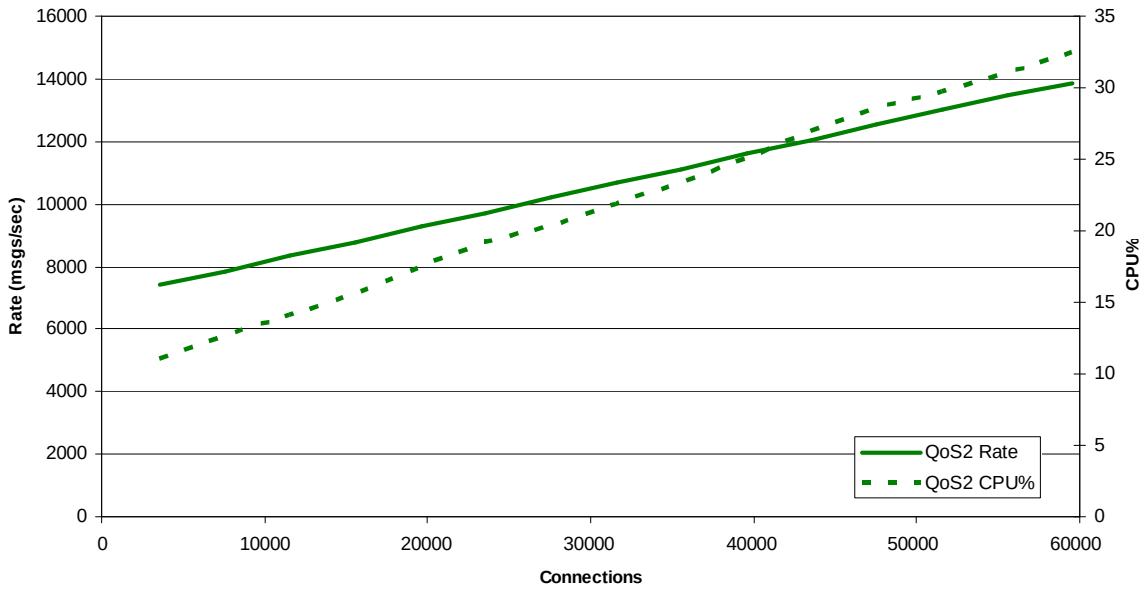

Figure 8: AIX/QoS2 Few-publishers, multi-subscribers Results

Other Platforms:
Figure 14: Linux on System x/QoS2 Few-publishers, multi-subscribers Results
Figure 20: Linux on System z/QoS2 Few-publishers, multi-subscribers Results

The QoS2 test was setup identically to the QoS0 test but the publish rate for each publisher was 1400/sec (total 7000/sec). As each publisher publishes onto all eventual topics (60,000) the rate of arrival on each topic is 7000/60,000 = 0.117 msgs/sec.

The theoretical maximum rate of messages in the system at any point in the test is then:

Total message rate of publishers + 0.117 * connected subscribers (messages published to a topic for which there are no subscribers are discarded).

= 7000 + n * 0.117

When all subscribers are connected the theoretical rate will simply be 2 times the publish rate (14,000/sec at 60K subscribers).

The graph above follows the theoretical maximum very closely, exhibiting good scaling.

4 Linux on System x Results

All Linux measurements used a 64-bit xSeries server (see section 8.2)

4.1 Multi-publisher, single-subscriber

See section 2.1 for an overview of these scenarios.

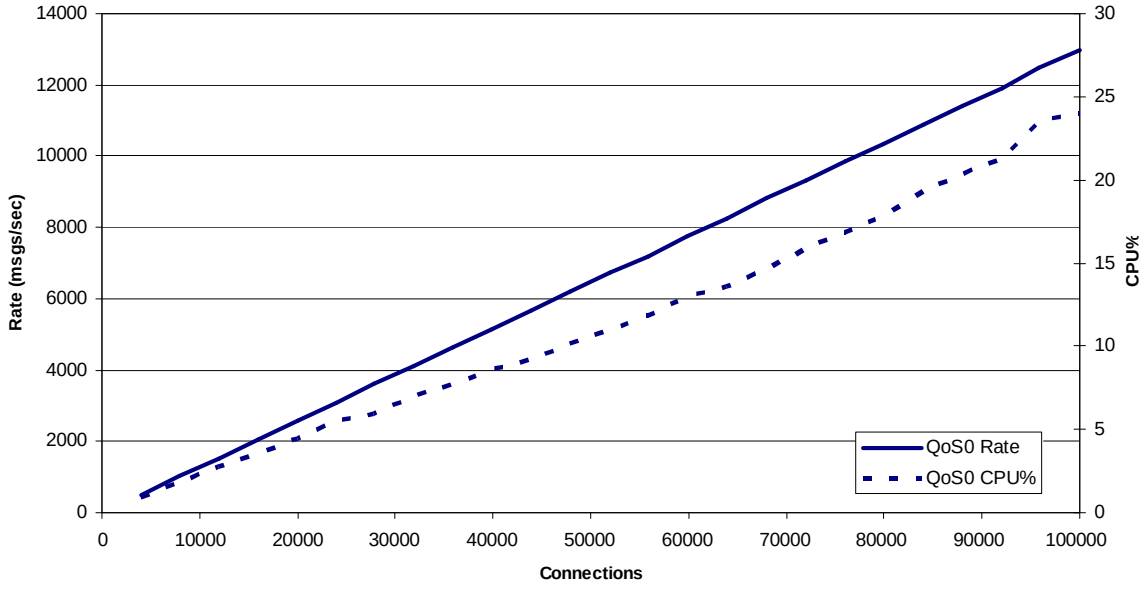

Figure 9: Linux on System x/QoS0 Multi-publisher, single-subscriber Results

Other Platforms:
Figure 3: AIX/QoS0 Multi-publisher, single-subscriber Results
Figure 15: Linux on System z/QoS0 Multi-publisher, single-subscriber Results

Publishers were added in batches of 4000 with every 1000 additional publishers running in a new client JVM. The JVMs were spread across two client machines (see section 8). For Linux on System x the tests were limited to 100,000 clients. A single subscriber consumed the messages.

The publish rate for each publisher was 0.065 messages/sec so for every 4000 publishers there were potentially 4000 * 2 * 0.065 msgs/sec through the system = 520/sec (13,000/sec at 100K publishers)

For QoS0 the test scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

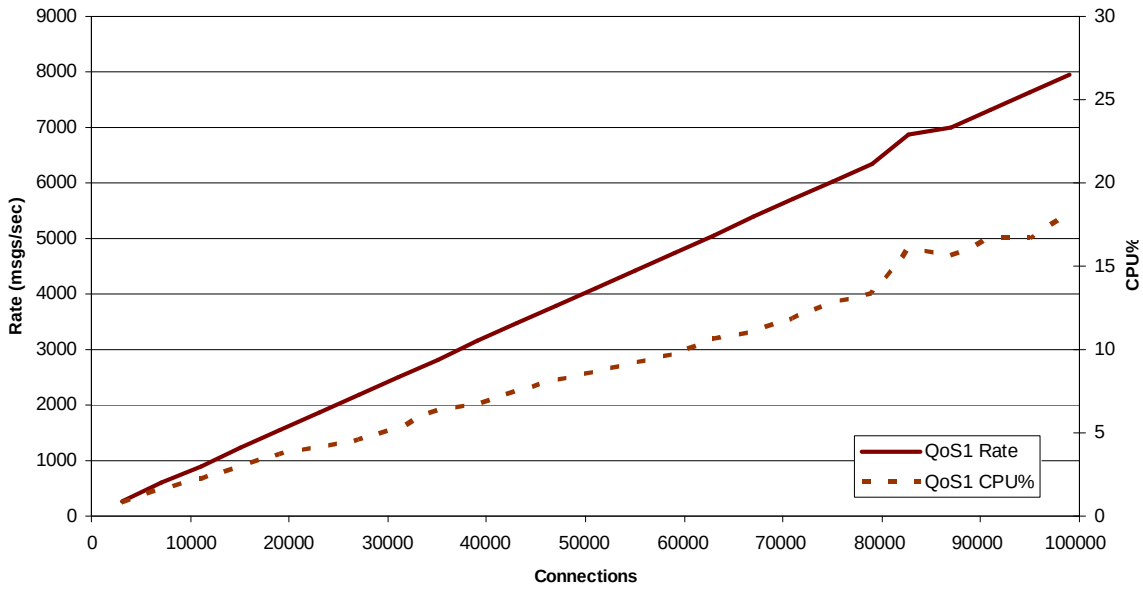

Figure 10: Linux on System x/QoS1 Multi-publisher, single-subscriber Results

Other Platforms:
Figure 4: AIX/QoS1 Multi-publisher, single-subscriber Results
Figure 16: Linux on System z/QoS1 Multi-publisher, single-subscriber Results

The QoS1 test was setup identically to the QoS0 test but the publish rate for each publisher was 0.04 messages/sec so for every 4000 publishers there were potentially 4000 * 2 * 0.04 msgs/sec through the system = 320/sec (8000/sec at 100K publishers).

The QoS1 measurements scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

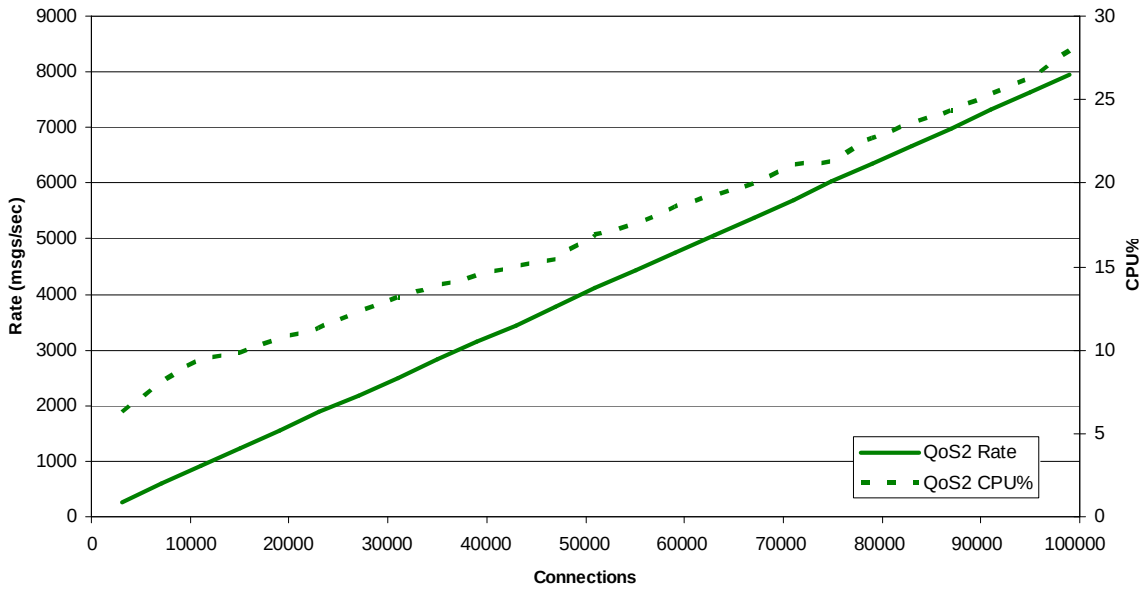

Figure 11: Linux on System x/QoS2 Multi-publisher, single-subscriber Results

Other Platforms:
Figure 5: AIX/QoS2 Multi-publisher, single-subscriber Results
Figure 17: Linux on System z/QoS2 Multi-publisher, single-subscriber Results

The QoS2 test was setup identically to the QoS0 test but the publish rate for each publisher was 0.04 messages/sec so for every 4000 publishers there were potentially 4000 * 2 * 0.04 msgs/sec through the system = 320/sec (8000/sec at 100K publishers).

The QoS2 measurements scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

4.2 Few-publishers, multi-subscriber

See section 2.2 for an overview of these scenarios.

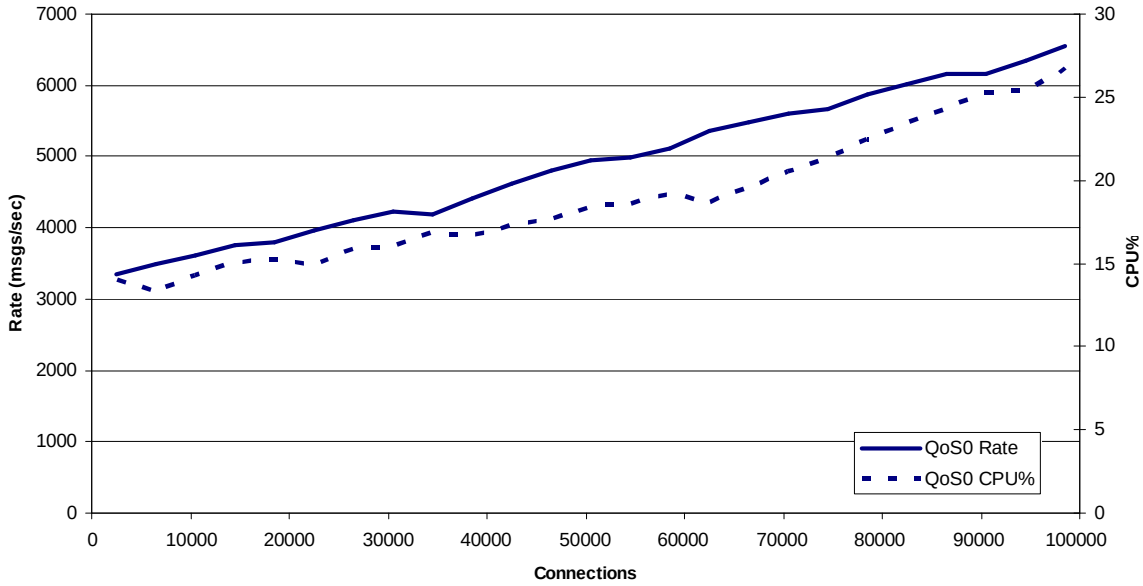

Figure 12: Linux on System x/QoS0 Few-publishers, multi-subscribers Results

Other Platforms:
Figure 6: AIX/QoS0 Few-publishers, multi-subscribers Results
Figure 18: Linux on System z/QoS0 Few-publishers, multi-subscribers Results

For this test 5 publishers were started with a rate of 650/sec (total 3250/sec). As each publisher publishes onto all eventual topics (100,000) the rate of arrival on each topic is 3250/100,000 = 0.033 msgs/sec.

The theoretical maximum rate of messages in the system at any point in the test is then:

Total message rate of publishers + 0.033 * connected subscribers (messages published to a topic for which there are no subscribers are discarded).

= 3250 + n * 0.033

When all subscribers are connected the theoretical rate will simply be 2 times the publish rate (6,500/sec at 100K subscribers).

The graph above follows the theoretical maximum very closely, exhibiting good scaling.

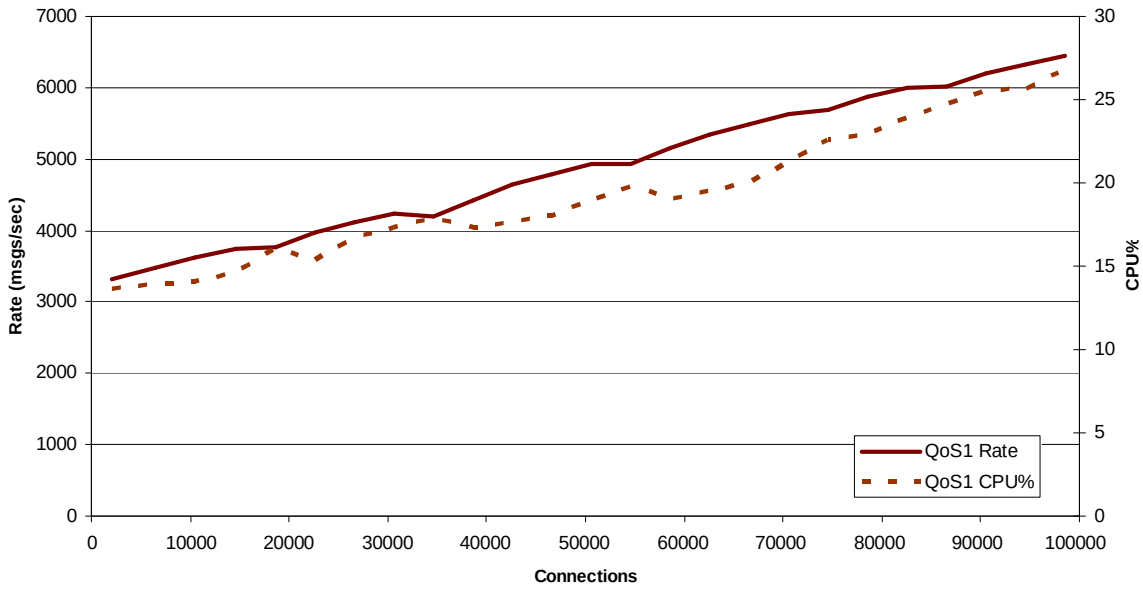

Figure 13: Linux on System x/QoS1 Few-publishers, multi-subscribers Results

Other Platforms:
Figure 7: AIX/QoS1 Few-publishers, multi-subscribers Results
Figure 19: Linux on System z/QoS1 Few-publishers, multi-subscribers Results

The QoS1 test was setup identically to the QoS0 in terms of rates so the theoretical maximum rate of messages in the system at any point in the test is

= 3250 + n * 0.033

When all subscribers are connected the theoretical rate will simply be 2 times the publish rate (6,500/sec at 100K subscribers).

The graph above follows the theoretical maximum very closely, exhibiting good scaling.

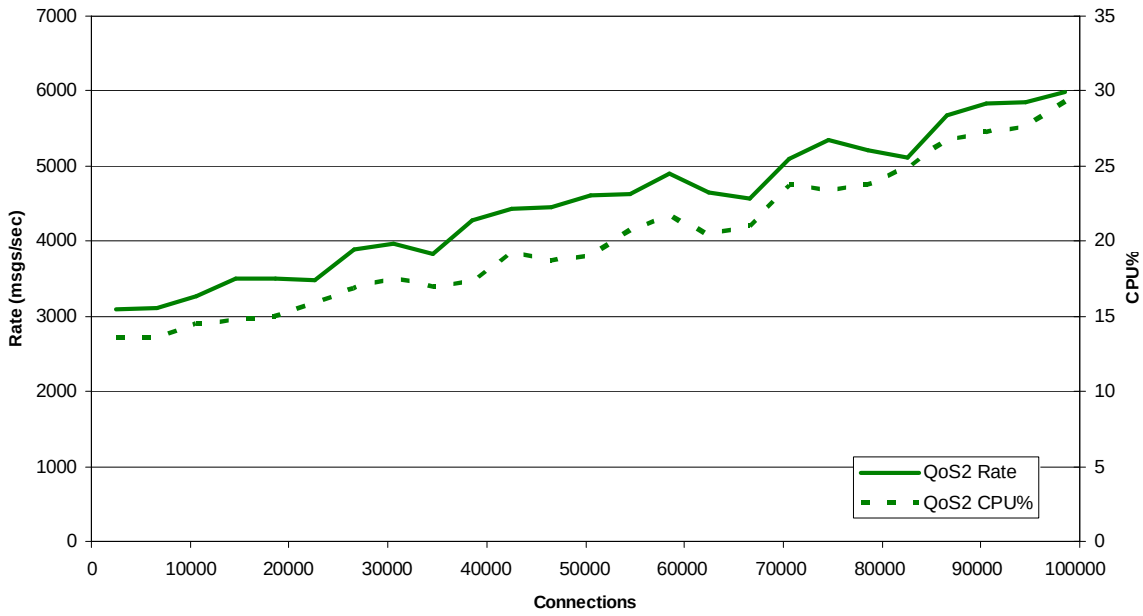

Figure 14: Linux on System x/QoS2 Few-publishers, multi-subscribers Results

Other Platforms:
Figure 8: AIX/QoS2 Few-publishers, multi-subscribers Results
Figure 20: Linux on System z/QoS2 Few-publishers, multi-subscribers Results

The QoS2 test was setup identically to the QoS0 test but the publish rate for each publisher was 600/sec (total 3000/sec). As each publisher publishes onto all eventual topics (100,000) the rate of arrival on each topic is 3000/100,000 = 0.030 msgs/sec.

The theoretical maximum rate of messages in the system at any point in the test is then:

Total message rate of publishers + 0.030 * connected subscribers (messages published to a topic for which there are no subscribers are discarded).

= 3000 + n * 0.030

When all subscribers are connected the theoretical rate will simply be 2 times the publish rate (6,000/sec at 100K subscribers).

The graph above follows the theoretical maximum very closely, exhibiting good scaling.

5 Linux on System z Results

All Linux on System z measurements used a z196 server (see section 8.3)

Note that the Linux on System z results were run on machines connected to a 1Gb LAN rather than the 10Gb LAN used for AIX and Linux on System x.

5.1 Multi-publisher, single-subscriber

See section 2.1 for an overview of these scenarios.

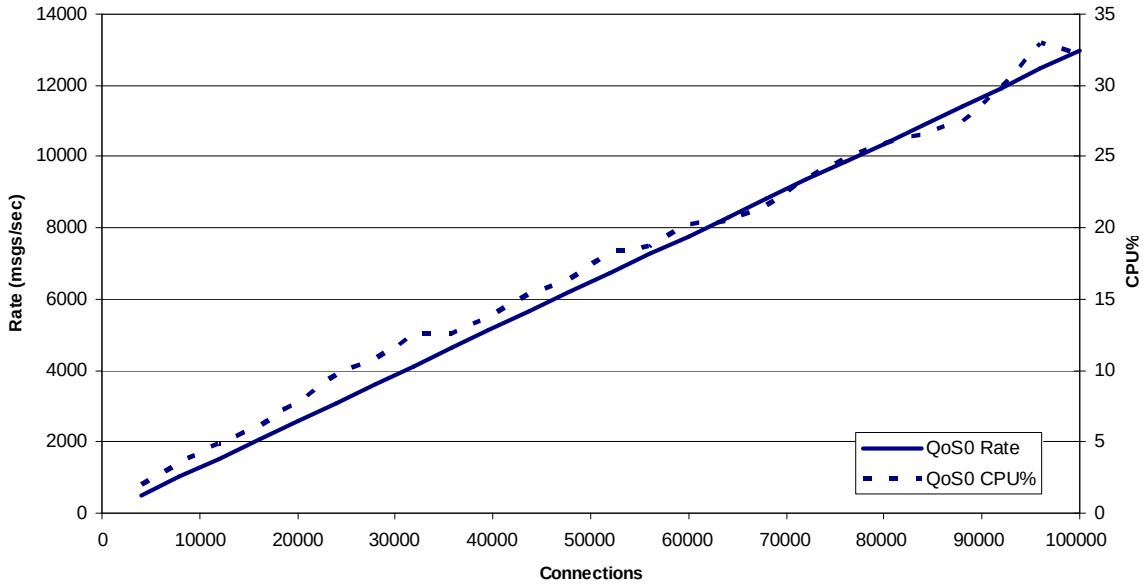

Figure 15: Linux on System z/QoS0 Multi-publisher, single-subscriber Results

Other Platforms:
Figure 3: AIX/QoS0 Multi-publisher, single-subscriber Results
Figure 9: Linux on System x/QoS0 Multi-publisher, single-subscriber Results

Publishers were added in batches of 4000 with every 1000 additional publishers running in a new client JVM. The JVMs were spread across two client machines (see section 8). For Linux on System z the tests were limited to 100,000 clients. A single subscriber consumed the messages.

The publish rate for each publisher was 0.065 messages/sec so for every 4000 publishers there were potentially 4000 * 2 * 0.065 msgs/sec through the system = 520/sec (13,000/sec at 100K publishers)

For QoS0 the test scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

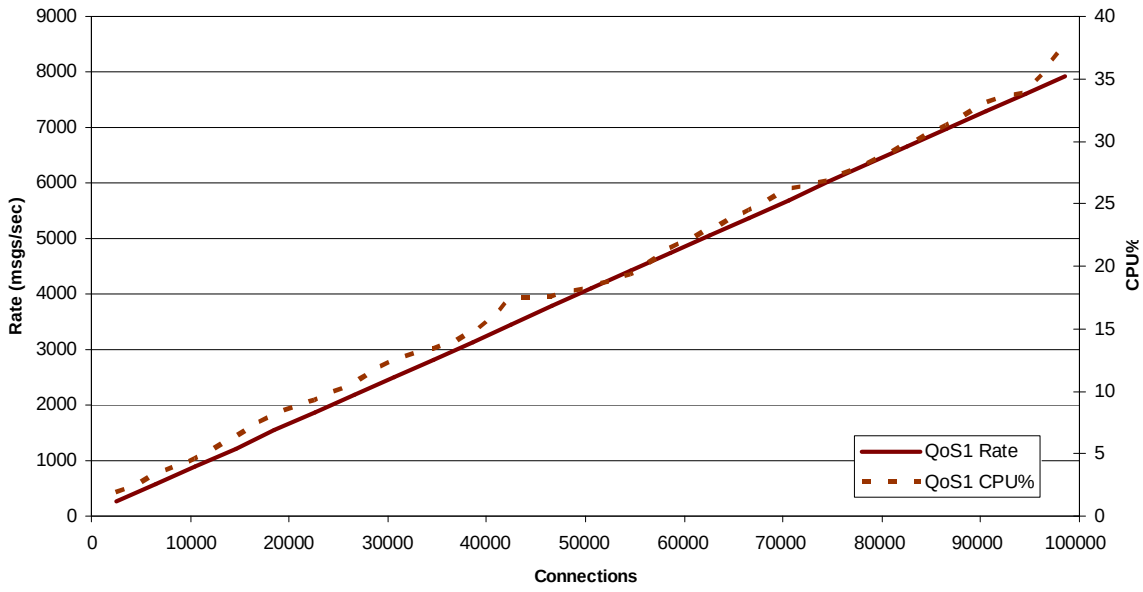

Figure 16: Linux on System z/QoS1 Multi-publisher, single-subscriber Results

Other Platforms:
Figure 4: AIX/QoS1 Multi-publisher, single-subscriber Results
Figure 10: Linux on System x/QoS1 Multi-publisher, single-subscriber Results

The QoS1 test was setup identically to the QoS0 test but the publish rate for each publisher was 0.04 messages/sec so for every 4000 publishers there were potentially 4000 * 2 * 0.04 msgs/sec through the system = 320/sec (8,000/sec at 60K publishers).

The QoS1 measurements scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager.

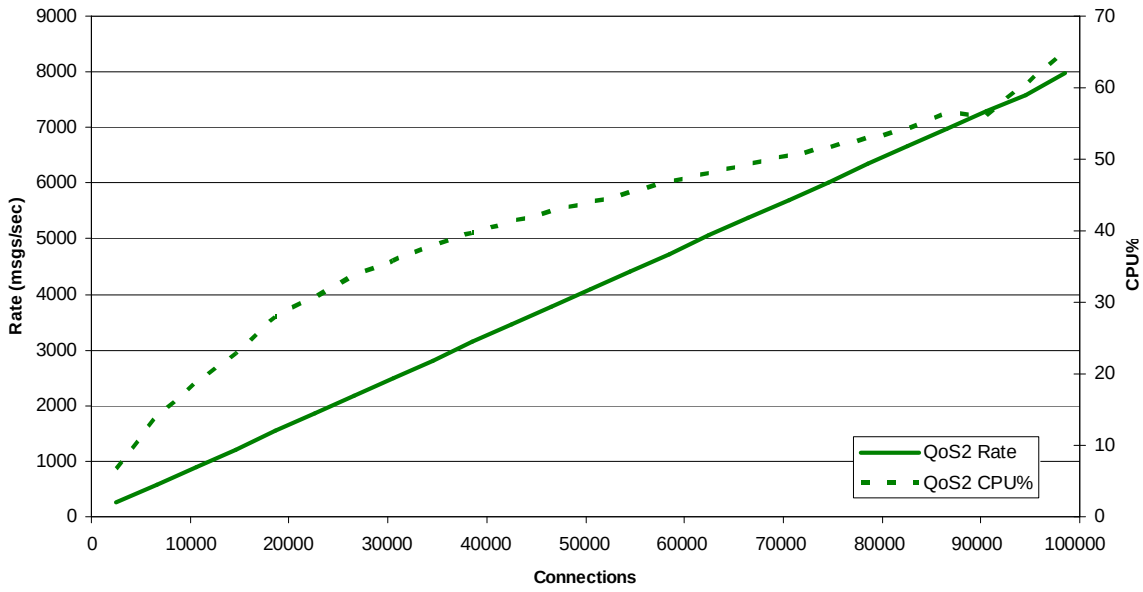

Figure 17: Linux on System z/QoS2 Multi-publisher, single-subscriber Results

Other Platforms:
Figure 5: AIX/QoS2 Multi-publisher, single-subscriber Results
Figure 11: Linux on System x/QoS2 Multi-publisher, single-subscriber Results

The QoS2 test was setup identically to the QoS0 test but the publish rate for each publisher was 0.04 messages/sec so for every 4000 publishers there were potentially 4000 * 2 * 0.04 msgs/sec through the system = 320/sec (8,000/sec at 60K publishers).

The QoS2 measurements scaled cleanly to 100K subscribers, demonstrating a linear increase in the overall messaging rate of the queue manager though the CPU costs increased at a greater rate during the first part of the test.

5.2 Few-publishers, multi-subscriber

See section 2.2 for an overview of these scenarios.

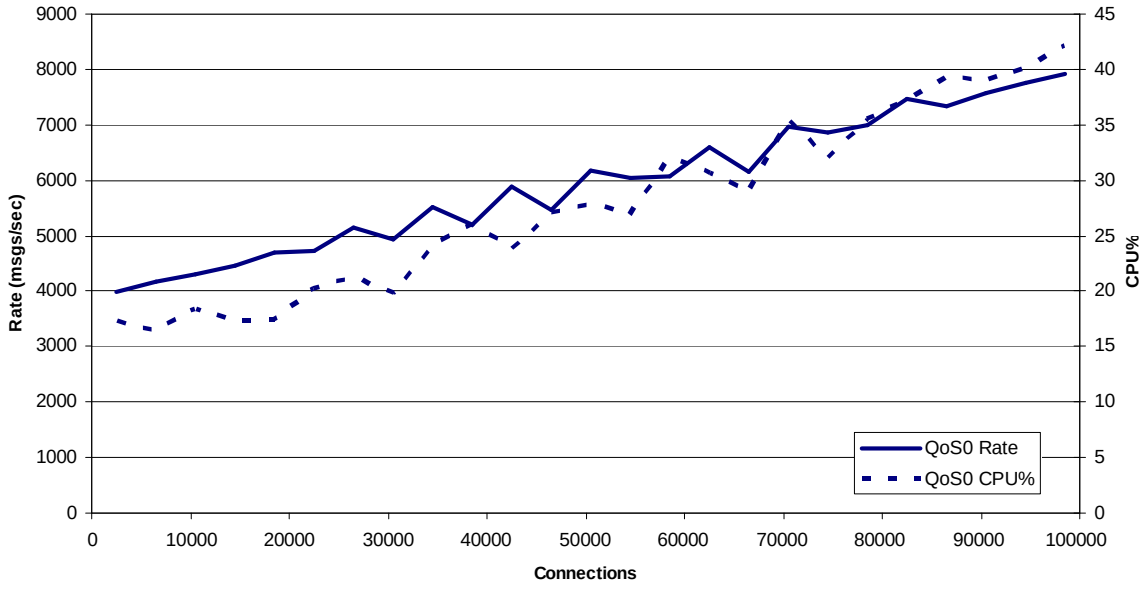

Figure 18: Linux on System z/QoS0 Few-publishers, multi-subscribers Results

Other Platforms:
Figure 6: AIX/QoS0 Few-publishers, multi-subscribers Results
Figure 12: Linux on System x/QoS0 Few-publishers, multi-subscribers Results

For this test 6 publishers were started with a rate of 650/sec (total 3900/sec). As each publisher publishes onto all eventual topics (100,000) the rate of arrival on each topic is 3900/100,000 = 0.039 msgs/sec.

The theoretical maximum rate of messages in the system at any point in the test is then:

Total message rate of publishers + 0.039 * connected subscribers (messages published to a topic for which there are no subscribers are discarded).

= 3900 + n * 0.039

When all subscribers are connected the theoretical rate will simply be 2 times the publish rate (7,800/sec at 100K subscribers).

The graph above follows the theoretical maximum very closely, exhibiting good scaling.

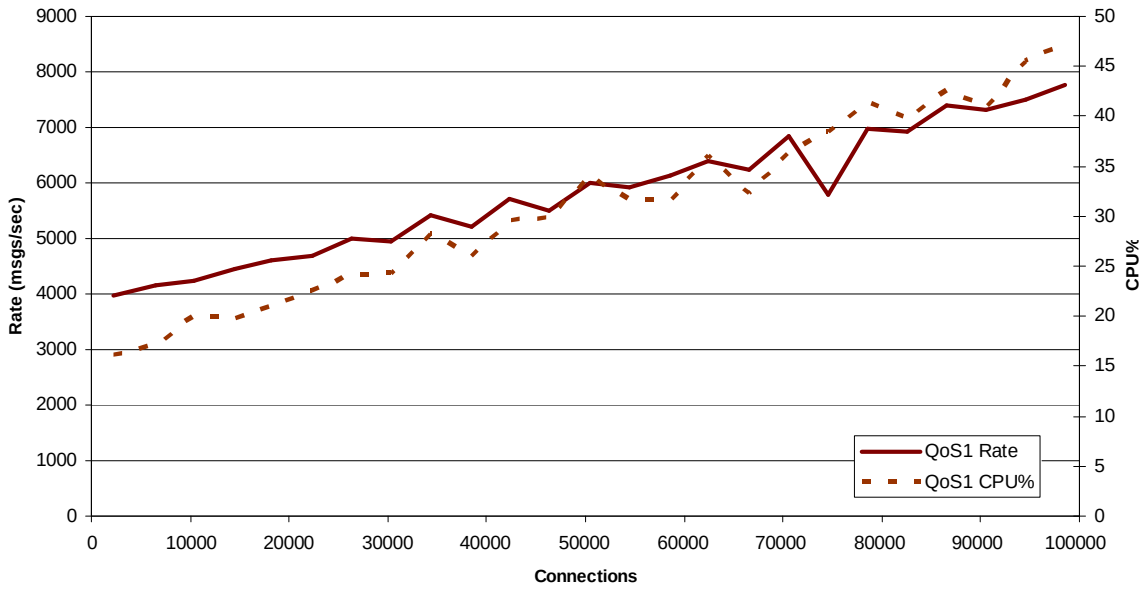

Figure 19: Linux on System z/QoS1 Few-publishers, multi-subscribers Results

Other Platforms:
Figure 7: AIX/QoS1 Few-publishers, multi-subscribers Results
Figure 19: Linux on System z/QoS1 Few-publishers, multi-subscribers Results

The QoS1 test was setup identically to the QoS0 in terms of rates so the theoretical maximum rate of messages in the system at any point in the test is

= 3900 + n * 0.039

When all subscribers are connected the theoretical rate will simply be 2 times the publish rate (7,800/sec at 100K subscribers).

The graph above follows the theoretical maximum very closely, exhibiting good scaling.

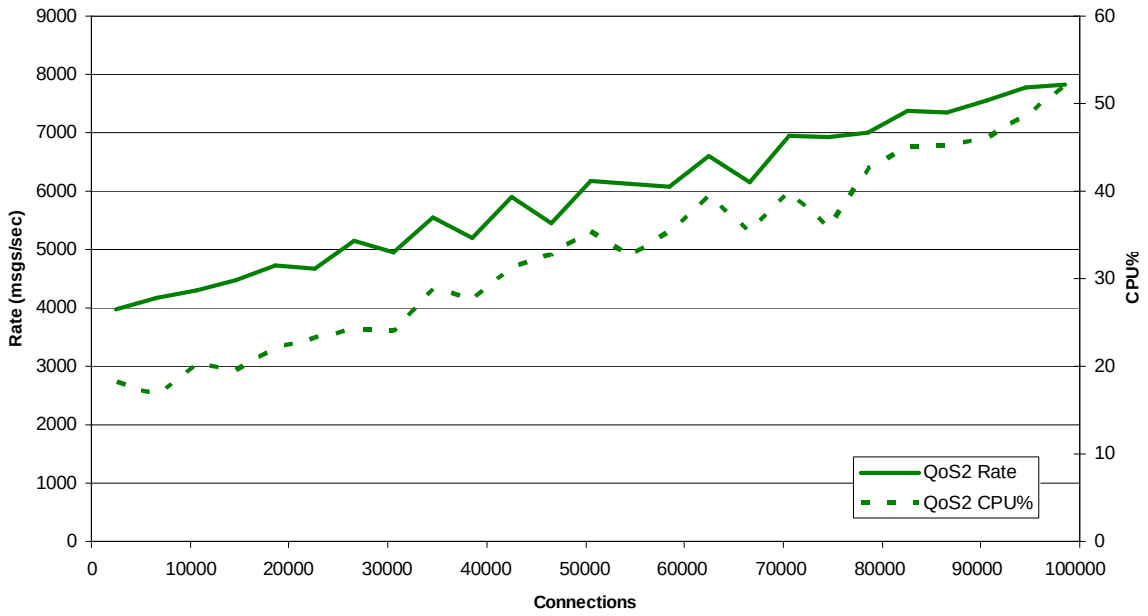

Figure 20: Linux on System z/QoS2 Few-publishers, multi-subscribers Results

Other Platforms:
Figure 8: AIX/QoS2 Few-publishers, multi-subscribers Results
Figure 14: Linux on System x/QoS2 Few-publishers, multi-subscribers Results

The QoS2 test was setup identically to the QoS0 in terms of rates so the theoretical maximum rate of messages in the system at any point in the test is

= 3900 + n * 0.039

When all subscribers are connected the theoretical rate will simply be 2 times the publish rate (7,800/sec at 100K subscribers).

The graph above follows the theoretical maximum very closely, exhibiting good scaling.

6 Test Calibration Example

AIX: Over-committed Subscriber

During the setup and measuring for the AIX QoS1 multi-producer, single subscriber performance test, limiting factors were investigated. Initial tests where the producer rate was set higher resulted in a drop of in total throughput towards the end of the test. This case demonstrates a few useful things to consider when designing and testing a scenario of this type.

Publishers were added in batches of 4000 with every 1000 additional publishers running in a new client JVM. The JVMs were spread across two client machines (see section 8). A single subscriber consumed the messages.

The publish rate for each publisher was 0.15 messages/sec so for every 4000 publishers there were potentially 4000 * 2 * 0.15 msgs/sec through the system = 1200/sec

It can be seen from the graph below that once ~47,000 publishers were connected the rate did not continue to increase in a linear fashion. In fact, the message rate began to drop at around 53,000 publishers.

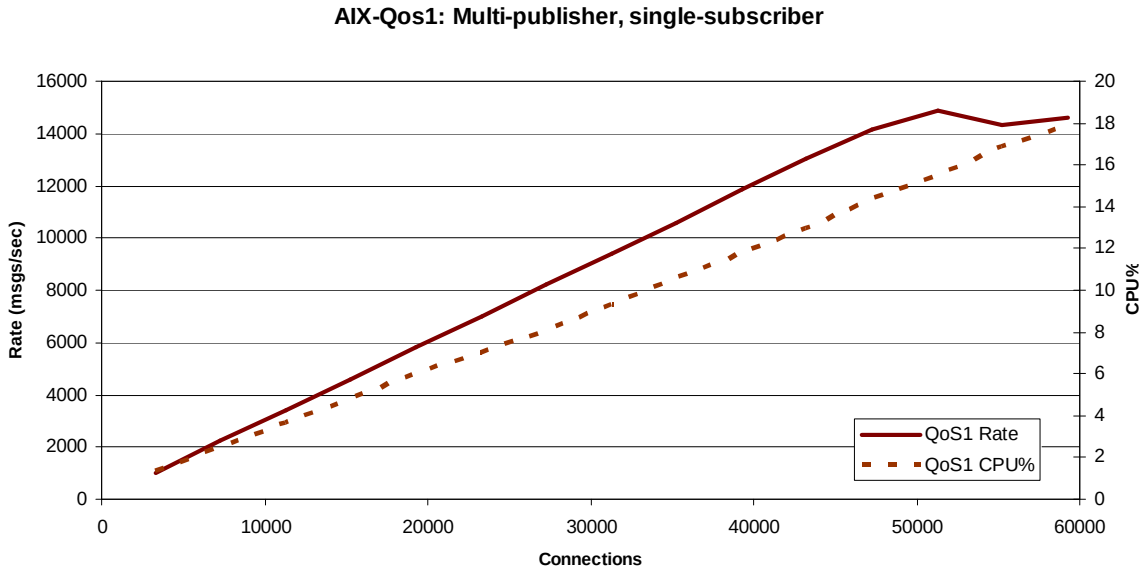

Figure 21: Overloaded Subscriber – Thorughput/CPU Utilisation

In this case, metrics collected using SAR showed that the disk associated with the queue files for the test became very active at the point the throughput dropped. Monitoring the queue depth of the SYSTEM.MANAGED.NDURABLE.xxxxxx queue associated with the single subscriber showed a correlating increase (see Figure 22). The limiting factor for the test was basically the rate at which the subscriber could read the ever increasing load on its queue.

The DefaultQBufferSize was set to 1MB in these tests. Once this buffer could no longer hold the messages on the queue WMQ started 'spilling' to disk. The point at which this occurs can be tuned of course by increasing the buffer but the nature of the test means that we only delay the inevitable as the load continues to increase.

In a production environment, DefaultQBufferSize is a useful tool to accommodate spikes in load but if the rate at which messages arrive continues to outpace the rate at which messages can be read off the queue then disk I/O and degradation in performance will occur at some point.

In the case of this test, the buffer was left at 1MB which was adequate for the concurrent load and rate at which the subscriber could read. The producer rate was then decreased so that at 100K connections the rate could still be processed by the subscriber.

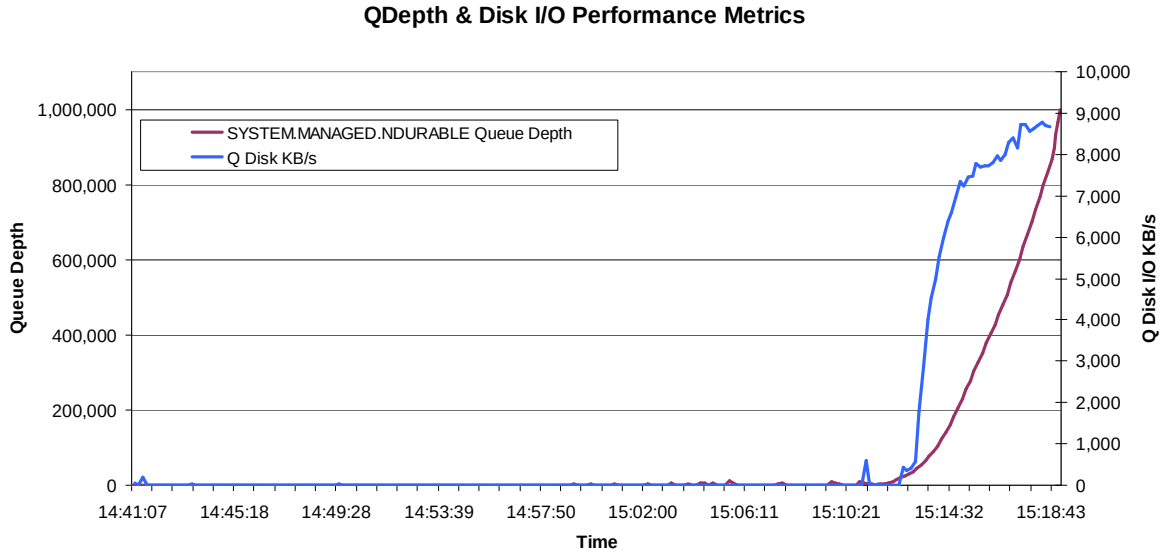

Figure 22: QDepth and Disk I/O Metrics

7 Tuning Parameters and Considerations

This section details the tuning parameters used for the measurements in this report and recommendations in general. Some of these recommendations may not apply directly to the telemetry tests but are left here for your information.

7.1 Tuning the queue manager

Performance reports with tuning information for WebSphere MQ v7.5 on all supported operating systems can be found on the IBM SupportPac webpage at the following URL:
http://www.ibm.com/software/integration/wmq/support/

The main tuning actions taken for the tests in this report were:

- Log / LogBufferPages = 4096 (size of memory used to build log I/O records)
- Log / LogFilePages = 16348 (size of Log disk file extent)
- Log / LogPrimaryFiles = 16 (number of disks extents in log cycle)
- LogWriteIntegrity=TripleWrite
- Channels / MQIBindType = FASTPATH (channels are an extension to QM address space)
- Channels / SHARECNV = 1
- TuningParameters / DefaultQBufferSize = 1MB (use 1MB of main memory per Q to hold non persistent messages before spilling to the file system
- TuningParameters / DefaultPQBufferSize = 1MB (use 1MB of main memory per Q to hold persistent messages)
- For Few publishers, Multi Subscriber tests:
    - TuningParameters / DurableSubMgrFrequency = 1
    - MAXHANDS = 999999999 This is a queue manager parameter that sets the maximum number of open handles that any one connection can have at the same time. To enable connections of thousands of telemetry clients, this number should be set appropriately to cope. Default is 256.

MQTT messages using the telemetry server are placed on the WebSphere MQ queue 'SYSTEM.MQTT.TRANSMIT.QUEUE'. Messages may be placed on the queue for each subscriber to receive a message so, for ten subscribers receiving a single message, up to ten 'transmit' messages will be placed on the transmit queue. Ensure the transmit queue size is sufficiently large to cope with the expected message rates. See the MAXDEPTH parameter on the queue description.

7.2 Tuning the heap size for Java

During operation, current garbage collectors (GC) will normally interrupt the execution of all other threads in a JVM to some extent. The level of interruption depends on the amount and the type of work the GC is doing. This is largely dependant on how the memory is being used by the application and the GC settings currently in operation.

JMS has characteristics such that fixed memory requirements are low but transient memory requirements can be high, depending on message size and application design. Without tuning, or with incorrect tuning, the automatic garbage collection policies of Java can adversely affect messaging performance.
The most common GC settings are:

`-Xms`	Minimum heap size.
`-Xmx`	Maximum heap size.
`-verbose:gc`	Display garbage collection events.

As an example, the following line fixes the heap size at 512MB and enables verbose garbage collection.

```
java -Xms512M -Xmx512M -verbose:gc
```

Recommendations

- Use –verbose:gc to monitor the frequency of your application's garbage collection under different loads and adjust the minimum and maximum heap sizes accordingly.
- A garbage collection interval of less than one second is detrimental to performance. A sensible minimum GC interval is 1-2 seconds, but consideration should also be given to the GC pause time.
- If the machine has sufficient memory then setting –Xms equal to –Xmx will allocate the specified maximum heap size at jvm startup. This avoids any costs involved in dynamically resizing the heap.
- GC implementations offer a variety of GC policies including concurrent and generational modes and you should consult your JVM documentation to determine the best option for your workload, then experiment with -verbose:gc to tune the settings.

7.2.1 JVM Warmup

- JVMs employ sophisticated Just-In-Time (JIT) compilers to optimise the executable code. These JITs can continue to recompile selected java methods for many minutes or even hours after the jvm has initialised. Full performance may not be achieved until this is completed, and indeed the cost of compilation can slow down performance in the early stages of execution. In most cases the default JIT settings will give best overall performance but in situations where a faster startup is desirable the JIT activity can be reduced at the expense of absolute performance.
- Performance Measurements on JMS workloads should only be done after a warmup period to ensure JIT activity has largely completed. You may need to experiment to find this point.

7.3 7.3 The Telemetry Server and Java Considerations

The telemetry server is a Java application that extends WebSphere MQ. A Java Runtime Environment (JRE) is supplied with the Service and may need to be optimized for a large number of connections.

To manage large numbers of connections, it is advisable to increase the initial Java heap size beyond the default settings. Xmx sets the maximum allowable heap size for the JRE. Xms sets the initial heap size for the JRE. For 50,000 connections, this document advises setting Xmx and Xms to 1024MB. For the tests in this document that connected 100,000 clients Xmx and Xms were set to 2048MB.

All the general advice above on JVM heap sizes still applies.

Set Xmx and Xms using the java.properties file in the mqxr configuration directory e.g.

Heap sizing options - uncomment the following lines to set the heap to 1G

```
-Xmx1024m
-Xms1024m
```

7.4 Shared Conversations

Clients producing or consuming a small number of messages per second can usefully share the TCP socket with other threads in the same process. For MQ v7.5 the default is for 10 applications to share a channel and hence a TCP socket. Benchmarks that produce or consume multiple hundreds of messages per second will bottleneck on the shared socket and should use a single socket per application by setting SHARECNV=1.

7.5 Avoiding running in Migration/Compatibility Mode

An MQ JMS 7.1 client can connect to V7.1, V6 and V7 Queue Managers. When connected to a V6 Queue Manager a less optimised codepath is used. This facilitates migration from V6 to V7 but should not be considered as a long-term solution if performance is important.
It is also possible to connect a V7.1 JMS client to a V7.1 Queue Manager in migration mode by setting WMQ_PROVIDER_VERSION to "6.0.0.0" on the ConnectionFactory, but for best performance the default V7.1 value should be used.

7.6 Use of Correlation Identifiers

- Selecting against *correlationId* or *messageId* follows an optimised path through WebSphere MQ 7.1 and the selection occurs on the server-side (in the queue manager). This gives better performance than when using arbitrary JMS selectors.
- Use of the provider-specific "ID:" tag is applicable to these two fields only and is of practical use only with correlation identifiers.
- To use the optimised path, the *correlationId* must be prefixed with "ID:" and must be formatted correctly as 24 bytes represented as a hex-string (of 48 characters). Failure to adhere to this means the selection will revert to expensive client-side methods.

  Example:
  ```
  Session.createConsumer(
      destination,
      "JMSCorrelationID='ID:574d51373053616d706c65436f7272656c6174696f6e4944'");
  ```
  In this case, the hexadecimal represents a 24-byte ASCII string "WMQ70SampleCorrelationID"
- The safest way of generating a correct identifier is to use *JMSMessage.setJMSCorrelationIDAsBytes*. This allows the formatted version to be returned by *getJMSCorrelationID*. The number of bytes input should not be more than 24 or the identifier will be truncated.

  Example:
  ```
  Message.setJMSCorrelationIDAsBytes( "WMQ70SampleCorrelationID".getBytes("UTF8") );
  Session.createConsumer(
      destination,
      "JMSCorrelationID='" + message.getJMSCorrelationID() + "'" );
  ```
- A change to the *correlationId* (or indeed any selector) that you are matching against requires opening a new MessageConsumer and discarding the old one. This is an expensive operation if it is done for every message that is processed since it involves closing and re-opening the underlying queue. For this reason you should consider generating your own *correlationId* for each **client** rather than the common design pattern of using the *messageId* of a sent message as the *correlationId* of its reply. Another alternative is to use a temporary queue per client.

7.7 Other Programming Recommendations

- Use Non-Persistent, Non-Transactional messages whenever possible
- Take performance into account when choosing which message type to use. The relative performance of the different JMS message types running a typical workload his as follows (fastest first)
  1. JmsTextMessage
  2. JmsBytesMessage (typically 5% slower than JmsTextMessage for a 2k message size)
  3. JmsObjectMessage (+10%)
  4. JmsStreamMessage (+15%)
  5. JmsMapMessage (+20%)
- If your application uses both transactional and non-transactional messages, consider creating separate transactional and non-transactional sessions for the different message types.
- Always call the *close()* method on JMS connection and session objects when they are no longer needed. This releases the underlying resource handle. This is especially important for publish-subscribe, where clients need to deregister from their subscriptions. Closing the objects allows the queue manager to release the corresponding resources in a timely fashion; failure to do so can affect the capacity of the queue manager for large numbers of applications or threads.
- Do not lose references to connection and session objects (e.g. after registering an asynchronous listener) as this precludes being able to call their *close()* methods.
- To ensure an application or internal object will always tidy up correctly, including if it should fail, these *close()* calls should be made in the *final* part of a try-catch-finally control structure.

- Do not create sender or receiver objects regularly if you can reuse them instead. This avoids releasing then re-acquiring the same queue manager resource.
- Always call delete() on temporary queues and topics when they are no longer needed. Otherwise, they will not be deleted until the connection is closed. For long running applications this will cause performance and administration problems.

7.8 JMS Persistence

Several JMS settings control the effective QoS of a JMS client's communication. The delivery and acknowledgement modes indicate how many times a given message can be delivered to an application: at-most-once or once-and-only-once. The customer solution relies on a certain level of resilience from the messaging provider.

If the messages are carrying 'inquiry' questions and answers, then it is likely that speed is far more important than resilience, so the architects can make this trade-off and use non-persistent messages.

JMS delivery mode

The JMS API supports two delivery modes to specify what should happen to messages if the JMS provider fails.

The PERSISTENT delivery mode, which is the default, instructs the JMS provider that a message should not be lost in transit in case of a JMS provider failure. A message sent with this delivery mode is logged to stable storage when it is sent. Only a hard media failure should cause a PERSISTENT message to be lost. PERSISTENT has the caveat that it does not cover message destruction due to message expiration (which would be considered a normal event), or loss due to "resource restrictions" (which the JMS specification does not define further). PERSISTENT messages should not be lost during a controlled restart of a JMS provider but there are no guarantees of protection across an unexpected failure.

The NON_PERSISTENT delivery mode does not require the JMS provider to store the message or otherwise guarantee that it is not lost if the provider fails or is restarted - in fact NON_PERSISTENT messages should NOT be kept across a restart of a JMS provider.

JMS acknowledgement mode

The JMS API also supports the ACKNOWLEDGE_MODE property that controls message duplication on non-persistent messaging.

Auto acknowledgement (default) means messages will not be delivered more than once

DUPS_OK acknowledgement means messages may be delivered more than once in certain circumstances and the client application must be prepared to deal with seeing the same message twice.

Client acknowledgement leaves control of this feature entirely to the user.

WebSphere MQ Quality-of-Service

The JMS definition of persistence allows considerable scope for different quality of service (QoS).

WebSphere MQ provides QoS that have been appreciated by customers over the last 18 years. Many of the installation defaults provide robustness and small memory footprint rather than maximising good performance.

WebSphere MQ has traditionally provided two QoS: persistent and non-persistent. The WebSphere MQ definitions are similar, but not identical to the JMS requirements. In particular,

WebSphere MQ does not discard non-persistent messages while the queue manager is running, even in the event of a memory buffer shortage.

WebSphere MQ provides a persistence and transaction integrity, above and beyond the specification of JMS, which has been industry-proven for a decade.

These QoS are usually paired together with transactionality. If messages are persistent it is expected, though not required, that they should be transactional and if they are non-persistent, they should be non-transactional. Messages carrying 'valuables' should normally be persistent and transactional since that eliminates most causes of failure. The application and system designer needs to consider the levels of resilience and recovery needed in different places, and the complexity needed in each component - the application, the messaging provider, a database, and so on. Using persistent, transactional messaging can remove a lot of complexity from application code.

Non-persistent messages are discarded by WebSphere MQ in the event of a queue manager restart but otherwise are not lost. The discarding of non-persistent messages can be altered on an individual queue basis by specifying NPMCLASS=HIGH which tells the Queue manager to preserve non-persistent messages when a controlled shutdown and restart of the queue manager is undertaken. During a failure (hardware or software) messages may have been lost and because there is no message log, we cannot rebuild the queue with integrity. These uncontrolled failures are outside of the JMS definition of persistent messages. Consequently, because MQ does not discard non-persistent messages during resource shortages, MQ non-persistent messages qualify as JMS Persistent messages when they are stored on a queue marked with NPMCLASS=HIGH. This code fragment shows how Persistence is taken from the JMS destination/queue using the WMQ_PER_NPHIGH string, which tells WebSphere MQ that it should treat messages sent to that destination as JMS PERSISTENT messages, but that it can use its knowledge of the underlying Websphere MQ queue configuration to optimise performance by using WebSphere MQ non-persistent messaging where possible

```
// Create a connection factory
    JmsFactoryFactory ff = JmsFactoryFactory.getInstance(WMQConstants.WMQ_PROVIDER);
    JmsConnectionFactory cf = ff.createConnectionFactory();
//Add some connection factory configuration here to tell the application how to connect to WebSphere MQ
    connection = cf.createConnection();
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    destination = (JmsDestination) session.createQueue("queue:///Q2");
    destination.setIntProperty(WMQConstants.WMQ_PERSISTENCE, WMQConstants.WMQ_PER_NPHIGH);
```

WebSphere MQ's use of transactional recovery logs in combination with secondary storage of queues results in resilience against individual failures can be used for high availability and disaster recovery scenarios.

The JMS definition of a persistent message is not precise so application solutions must decide how much dependence is put on the message provider.

- Does the message have to survive if various resource shortages are encountered on the journey?
- Does the message survive if various application, software, or hardware failures are encountered on the journey?
- Greater reliability inevitably means lower run-time performance because of the extra work needed to provide the information needed during recovery.

8 Machine and Test Configurations

MQTT clients used to drive tests in this report were located on 2 Linux on System x machines, grouped in a JVMs of 1000 clients per JVM. Driver machines were connected to Linux on System x servers using 10Gb LAN, but used 1Gb LAN to connect to AIX and Linux on System z servers.

Driver Machine 1:

IBM System X3755 M3 -[716472G]
4 x AMD Opteron(tm) Processor 6174, 2.2GHz 12-core. (48 physical cores)
Hyper-Threading not supported
64 GB RAM
10Gbit Ethernet
SAN Storage
64bit Red Hat Enterprise Linux Server release 5.8 (Tikanga)

Driver Machine 2:

IBM System X3755 M3 -[716472G]
4 x AMD Opteron(tm) Processor 6174, 2.2GHz 12-core. (48 physical cores)
Hyper-Threading not supported
64 GB RAM
10Gbit Ethernet
SAN Storage
64bit Red Hat Enterprise Linux Server release 5.8 (Tikanga)

Due to the differences between the hardware used for each operating system, it is not possible to compare throughput across operating systems.

8.1 AIX

IBM Power7 P750 8233-E8B
16 core x 3.55GHz SMT x4 enabled
32 GB of RAM
AIX 7.1.0.0 TL05 SP2
MQ Log and Queues on SAN disks on DS8700
10Gbit Ethernet Adapter

8.2 Linux on System x

An xSeries 3850 4 x 4-core 2.93GHz Intel Xeon with 32GB of RAM and 10Gbit LAN was used as the Linux on System x machine under test.

Linux on System x Redhat 5.4 (kernel 2.6.18) with MQ Log and Queues on 2 local cached disks. Hosted on z/VM.

8.3 Linux on System z

CPU: 4 way SMP LPAR on a 2817-779 (z196)
2GB of virtual memory. (VM system has 5GB of main storage + 2GB of expanded)
DASD: DS8800's with dedicated links.
Network: 1Gbit LAN

SUSE Linux Enterprise Server 10 (s390x)

8.4 SAN disk subsystem

The machines under test are connected to a SAN via a dedicated SVC. The SVC provides a transparent buffer between the server and SAN that will smooth any fluctuations in the response of the SAN due to external workloads. The server machines are connected via a fibre channel trunk to an 8Gb Brocade DCX director. The speed of each server is dictated by the server's HBA (typically 2Gb). 5GB generic LUNs are provisioned via SVC. The SVC is a 2145-8G4 which connects to the DCX at 4Gb. The SAN storage is provided by an IBM DS8700 which is connected to the DCX at 4Gb.