WebSphere®MQ for JMS V7 - Performance Evaluations

Version 1.1

September 2009

Peter Toghill , Pete Hickson .

WebSphere MQ Performance
IBM UK Laboratories
Hursley Park
Winchester
Hampshire
SO21 2JN

Property of IBM

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.

Second Edition, September 2009.

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

© Copyright International Business Machines Corporation 2009. All rights reserved.

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 were measured in a controlled environment. Results obtained in other environments may vary significantly.

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

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

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

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

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

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

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

INTENDED AUDIENCE
This report is intended for architects, systems programmers, analysts and programmers

wanting to understand the performance characteristics of *WebSphere MQ JMS V7*. 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.

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™
-

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 of the JMS clients supplied with WebSphere MQ for Windows V7.0, Linux V7.0, AIX V7.0, and zOS and is intended to assist with programming and capacity planning.

Target audience

This SupportPac is designed for people who:

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

The reader should have a general awareness of the Java programming language, the Java Message Service API, the Windows 2003, Linux, zOS ,and/or AIX operating systems and of WebSphere MQ 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 JMS using WebSphere MQ V7.0
- Performance characteristics of options within JMS
- WebSphere MQ JMS messaging comparisons between Windows, Linux and AIX
- zOS WebSphere MQ JMS messaging comparison between Client and local bindings
- Advice on programming with WebSphere MQ JMS for performance

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.

Introduction

This report uses the JMS Performance Harness (available from Alphaworks) to produce message throughput and CPU information on AIX, Linux, Windows, and zOS.

Some specific Version 7 performance improvements are assessed together with Tuning advice for MQ/JMS

CONTENTS

1 Overview 1
2 Local Bindings 2
2.1 Point to Point Put Get 4Q Scenario 2
2.1.1 Put/Get 4Q Non persistent Messages – Local 3
2.1.2 Put/Get 4Q Persistent Messages – Local 4
2.2 Point to Point , Multiple (Producer, Consumer, Queue) Scenario 5
2.2.1 Producer Consumer, Non Persistent Messages, Local 6
2.2.2 Producer Consumer, Persistent Messages, Local 7
2.3 Publish/Subscribe Single Publisher, Many Subscribers Scenario(1:N) 8
2.3.1 Publish Subscribe 1:N, Non Persistent messages, local 8
2.3.2 Publish Subscribe 1:N, Persistent messages, local 9
2.4 Publish Subscribe multiple (Publisher, Topic, Subscriber) scenario 10
2.4.1 Publish Subscribe (Multiple P/T/S), Non Persistent messages, local 11
2.4.2 Publish Subscribe (Multiple P/T/S), Persistent messages, local 12
3 Client Channels Test Scenario 13
3.1 Point to Point Put Get 4Q Scenario 14
3.1.1 Put/Get 4Q Nonpersistent Messages – Client 14
3.1.2 Put/Get 4Q Persistent Messages – Client 15
3.2 Point to Point , Multiple (Producer, Consumer, Queue) Scenario 16
3.2.1 Producer Consumer, Non Persistent Messages, Client 16
3.2.2 Producer Consumer, Persistent Messages, Client 17
3.3 Publish/Subscribe Single Publisher, Many Subscribers Scenario(1:N) 18
3.3.1 Publish Subscribe 1:N, Non Persistent messages, Client 18
3.3.2 Publish Subscribe 1:N, Persistent messages, Client 19
3.4 Publish Subscribe multiple (Publisher, Topic, Subscriber) scenario 20
3.4.1 Publish Subscribe (Multiple P/T/S), Non Persistent messages, Client 20
3.4.2 Publish Subscribe (Multiple P/T/S), Persistent messages, Client 21
4 z/OS – Local Binding & Client 22
4.1 Point to Point Put Get 4Q Scenario 22
4.1.1 Put/Get 4Q Non persistent Messages 22
4.1.2 Put/Get 4Q Persistent Messages 23
4.1.3 Put/Get 1Q Persistent Messages – Client 24
4.2 Point to Point , Multiple (Producer, Consumer, Queue) Scenario 24
4.2.1 Producer Consumer, Non Persistent Messages 24
4.2.2 Producer Consumer, Persistent Messages 25
4.3 Publish/Subscribe Single Publisher, Many Subscribers Scenario(1:N) 26
4.3.1 Publish Subscribe 1:N, Non Persistent messages 26
4.3.2 Publish Subscribe 1:N, Persistent messages 27
4.4 Publish Subscribe multiple (Publisher, Topic, Subscriber) scenario 28
4.4.1 Publish Subscribe (Multiple P/T/S), Non Persistent messages 28
4.4.2 Publish Subscribe (Multiple P/T/S), Persistent messages 29
4.5 JMS Selectors and Correlation Identifiers on z/OS 29
4.5.1 JMS Selectors and Correlation Identifiers on z/OS – Client Bindings 30
4.5.2 JMS Selectors and Correlation Identifiers, z/OS - Local Bindings 30
5 Message Driven Beans 32
5.1 Overview 32
5.2 Scalability Improvements in WMQ V7.0 32
5.2.1 Queue with message rate gradually increasing 32
5.2.2 Queue with maximum sustainable message rate manager 35
6 Other Performance Enhancements in V7.0 37
6.1 JMS Selectors 37
6.1.1 Recommendations 37
6.2 Asynchronous Put and Read Ahead 38
6.2.1 Asynchronous Put 38
6.2.2 Read Ahead 38
6.2.3 Performance Notes 39
6.3 Asynchronous Consume 39
6.4 Multiplexed Sockets and Conversation Sharing 40
6.5 Large Messages 41
7 Tuning/programming guidelines 42

7.1 Tuning the queue manager 42
7.2 Tuning the heap size for Java 42
7.3 Shared Conversations 43
7.4 Avoiding running in Migration/Compatibility Mode 43
7.5 Use of correlation identifiers 43
7.6 JVM Warmup 44
7.7 Other Programming Recommendations 44
7.8 JMS Persistence 44
JMS delivery mode 45
8 Machine and Test Configurations 47
8.1 Linux, Windows and AIX 47
8.1.1 SAN disk subsystem 47
8.2 zOS 47
8.3 MDB Test Configuration 48
8.4 Other V7 Enhancements 48

TABLES

Table 1 – Put/Get 4Q, non persistent messages, local queue manager 3
Table 2 – Put/Get 4Q, Persistent messages, local queue manager 4
Table 3 – Producer/Consumer, non persistent messages, local queue manager 6
Table 4 – Producer/Consumer, persistent messages, local queue manager 7
Table 5 – Publish Subscribe 1:N, non Persistent messages, local queue manager 9
Table 6 – Publish Subscribe 1:N, Persistent messages, local queue manager 9
Table 7 – Publish Subscribe Multiple, non Persistent messages, local queue manager 11
Table 8 – Publish Subscribe Multiple, Persistent messages, local queue manager 12
Table 9 – Put/Get 4Q, non Persistent messages, Client connection 14
Table 10 – Put/Get 4Q, Persistent messages, Client connection 15
Table 11 – Producer/Consumer, non Persistent messages, Client connection 16
Table 12 – Producer/Consumer, Persistent messages, Client connection 17
Table 13 – Publish/Subscribe 1:N, non Persistent messages, Client connection 18
Table 14 – Publish/Subscribe 1:N, Persistent messages, Client connection 19
Table 15 – Publish/Subscribe Multiple, non Persistent messages, Client connection 20
Table 16 – Publish/Subscribe Multiple, Persistent messages, Client connection 21
Table 17 – Put/Get 4Q, non persistent messages 22
Table 18 – Put/Get 4Q, Persistent messages 23
Table 19 – Producer/Consumer, non Persistent messages 25
Table 20 – Producer/Consumer, Persistent messages 25
Table 21 – Publish/Subscribe 1:N, non Persistent messages 26
Table 22 – Publish/Subscribe 1:N, Persistent messages 27
Table 23 – Publish/Subscribe Multiple, non Persistent messages 28
Table 24 – Publish/Subscribe Multiple, Persistent messages 29

FIGURES

Figure 1 – Applications and Queues in a local queue manager 2
Figure 2 – Put Get 4Q, non persistent messages, local queue manager 3
Figure 3 – Put/Get 4Q, Persistent messages, local queue manager 4
Figure 4 – Producer/Consumer, non persistent 5
Figure 5 – Producer/Consumer, non persistent messages, local queue manager 6
Figure 6 – Producer/Consumer, persistent messages, local queue manager 7
Figure 7 – Publish Subscribe 1:N 8
Figure 8 – Pub/Sub 1:N, non-persistent messages, local queue manager 8
Figure 9 – Pub/Sub 1:N, Persistent messages, local queue manager 9
Figure 10 – Publish Subscribe 10
Figure 11 – Publish Subscribe Multiple, Non persistent messages ,local 11
Figure 12 – Publish Subscribe Multiple, Persistent messages, local 12
Figure 13 – MQI-client channels into a remote queue manager 13
Figure 14 – Put/Get 4Q , non persistent, client 14
Figure 15 – Put/Get 4Q, persistent, client 15
Figure 16 – Producer/Consumer, non persistent, client 16
Figure 17 – producer/Consumer, persistent, client 17
Figure 18 – Publish Subscribe 1:N, non persistent, client 18
Figure 19 – Publish Subscribe 1:N, persistent, client 19
Figure 20 – Publish Subscribe Multiple, non persistent, client 20
Figure 21 – Publish Subscribe multiple, persistent, client 21
Figure 22 – Put/Get 4Q , non persistent 22
Figure 23 – Put/Get 4Q, persistent 23
Figure 24 – Put/Get 1Q, persistent messages, Client connection 24
Figure 25 – Producer/Consumer, non persistent 24
Figure 26 – producer/Consumer, persistent, client 25
Figure 27 – Publish Subscribe 1:N, non persistent 26
Figure 28 – Publish Subscribe 1:N, persistent 27
Figure 29 – Publish Subscribe Multiple, non persistent 28
Figure 30 – Publish Subscribe multiple, persistent, ... 29
Figure 31 – JMS Selector and Correlid Performance on z/OS, client bindings. 30
Figure 32 – JMS Selector and Correlid Performance on z/OS, Local Bindings. 31
Figure 33 – MDB Scenario, Single Consumer 32
Figure 34 – MDB Scenario, Multiple Consumers 33
Figure 35 – MDB Scalability and Performance WMQ V6 34
Figure 36 – MDB Scalability and Performance WMQ V7 35
Figure 37 – JMS Selector Performance. Cpu figures are for QM only. 37
Figure 38 – New Asynchronous Put and Read Ahead features. 38
Figure 39 – Asynchronous Consumer Performance. 40
Figure 40 – Large Message Performance – 4 Streams. 41
Figure 41 – Message Performance – 10 Streams. 41

1 Overview

The four scenarios used in Chapter 2 3 and 4 in this report consist of two Point to Point and two Publish_Subscribe. These are measured and reported with Persistent and non Persistent messages on Windows, Linux, AIX, and zOS systems.

1) Sender/Receiver – Four Queues
2) Multiple sets of (Producer ,Queue, Consumer)
3) Publish Subscribe (single publisher, single topic, multiple subscribers)
4) Multiple sets of Publisher Topic Subscriber (single publisher, single topic, single subscribers)

- The message format used is a 2048 byte JMSTextMessage.
- Persistent messages are transactional. (session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE); This also significantly improves throughput when multiple threads are processing messages on the same queue especially when using non cached disks for the MQ Log.
- The 'multiple sets' message producers insert messages at a fixed rate.
- "IBM Performance Harness for Java Message Service".
- Messages Producer/Consumer are co-located on the Queue manager system for 'Local Bindings' Measurements and on Linux driver systems for 'Client' measurements.
- Each sample point reported is the average of two minutes of data collection.
- Clients run on Linux and the bottleneck with Client measurements is the server because adequate power is available in the driving machines.

2 Local Bindings

.

2.1 Point to Point Put Get 4Q Scenario

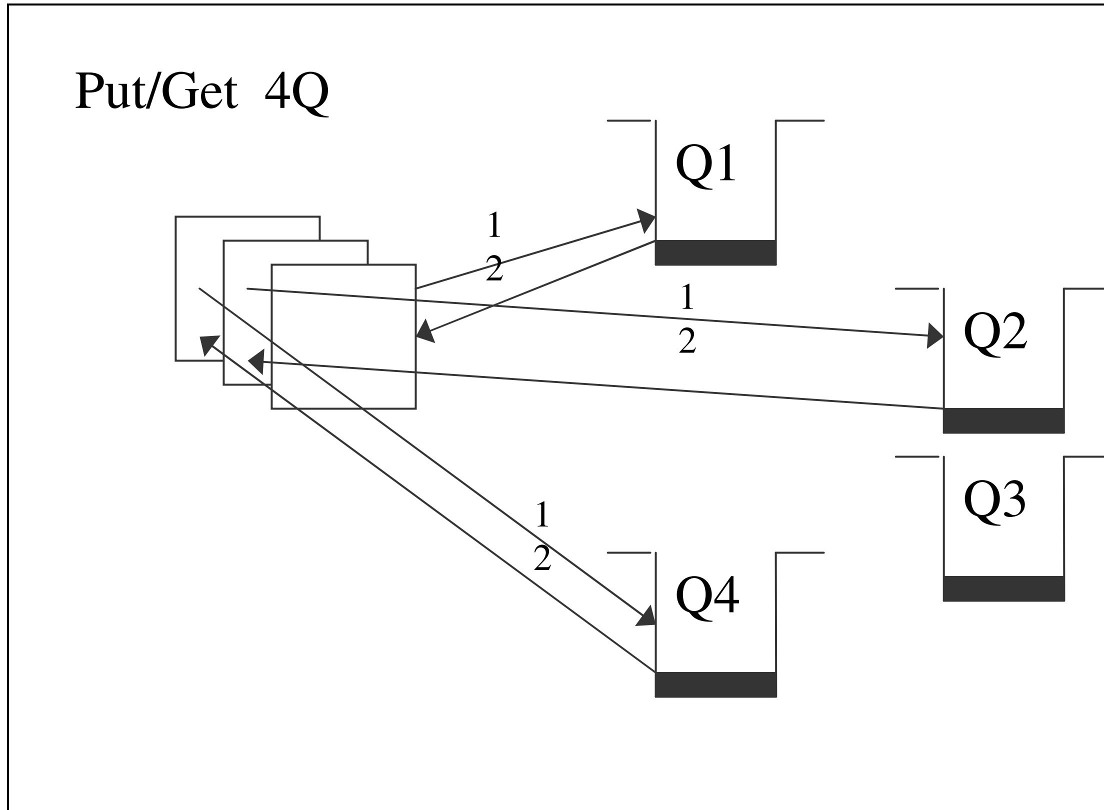


Figure 1 – Applications and Queues in a local queue manager

1) Each Put/Get application thread has been allocated to a particular Queue.

2) A message is put to the particular queue on the local queue manager, and stores the message identifier returned in the message descriptor. The Get specifies the message identifier and retrieves the message from the queue.

3) Only one message per Put/Get application exists at any point in time. This is synchronous messaging

Non-persistent and persistent messages were used in the local queue manager tests, with a message size of 2K. Messages are counted as they are retrieved from the queue hence a Put/Get round trip is counted as one message.

2.1.1 Put/Get 4Q Non persistent Messages – Local

Figure 2 and **Figure 3** show the non-persistent and persistent message throughput achieved using an increasing number of driving applications in the local queue manager scenario (see Figure 1 on the previous page), for Linux, Windows, and AIX.

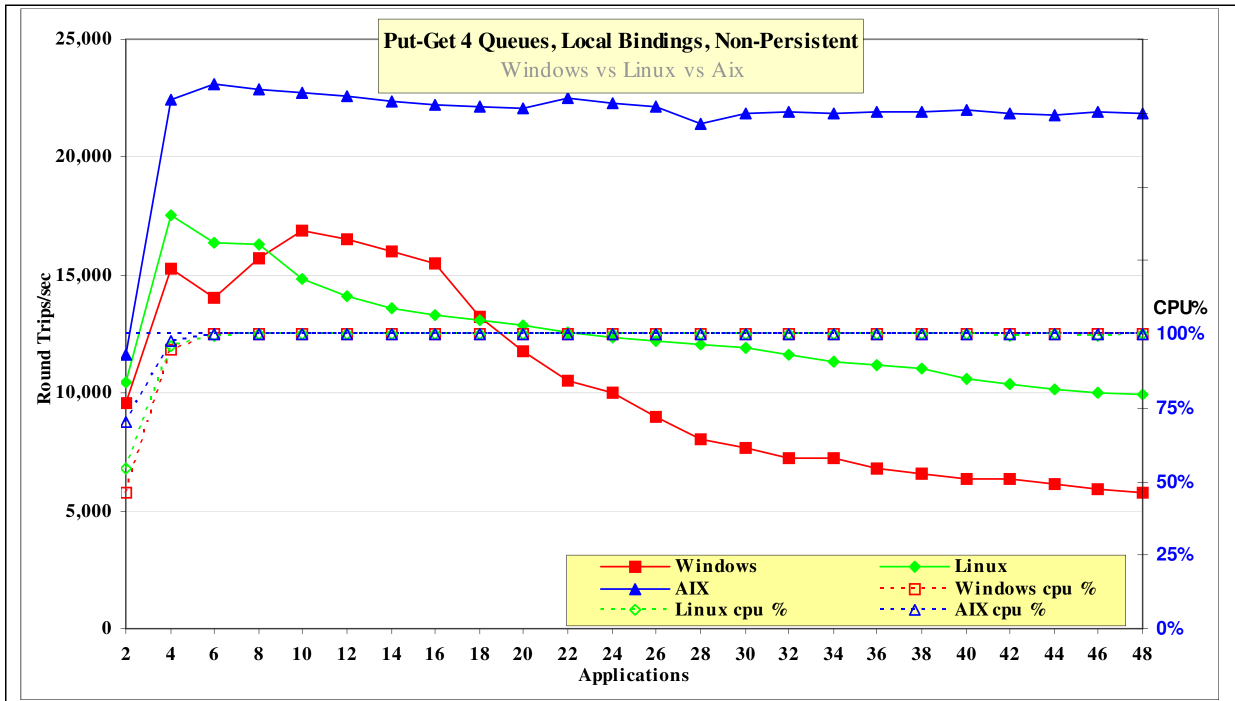


Figure 2 – Put Get 4Q, non persistent messages, local queue manager

Figure 2 and **Table 1** shows the throughput of non persistent messages.

Test name: PG4QLN	Apps	Round Trips/sec	CPU
Linux	**4**	**17571**	**96%**
Windows	**10**	**16856**	**100%**
Aix	**6**	**23088**	**100%**

Table 1 – Put/Get 4Q, non persistent messages, local queue manager

Note: These tables (like **Table 1** *) show the peak number of round trip or messages per second that are processed by all the connected applications or clients together with the number of driving application(or clients)s used to achieve the peak throughput.*

The AIX operating system continues to process the peak message load as additional work requests are submitted while Linux and Windows gradually process less work as additional work requests are submitted.

2.1.2 Put/Get 4Q Persistent Messages – Local

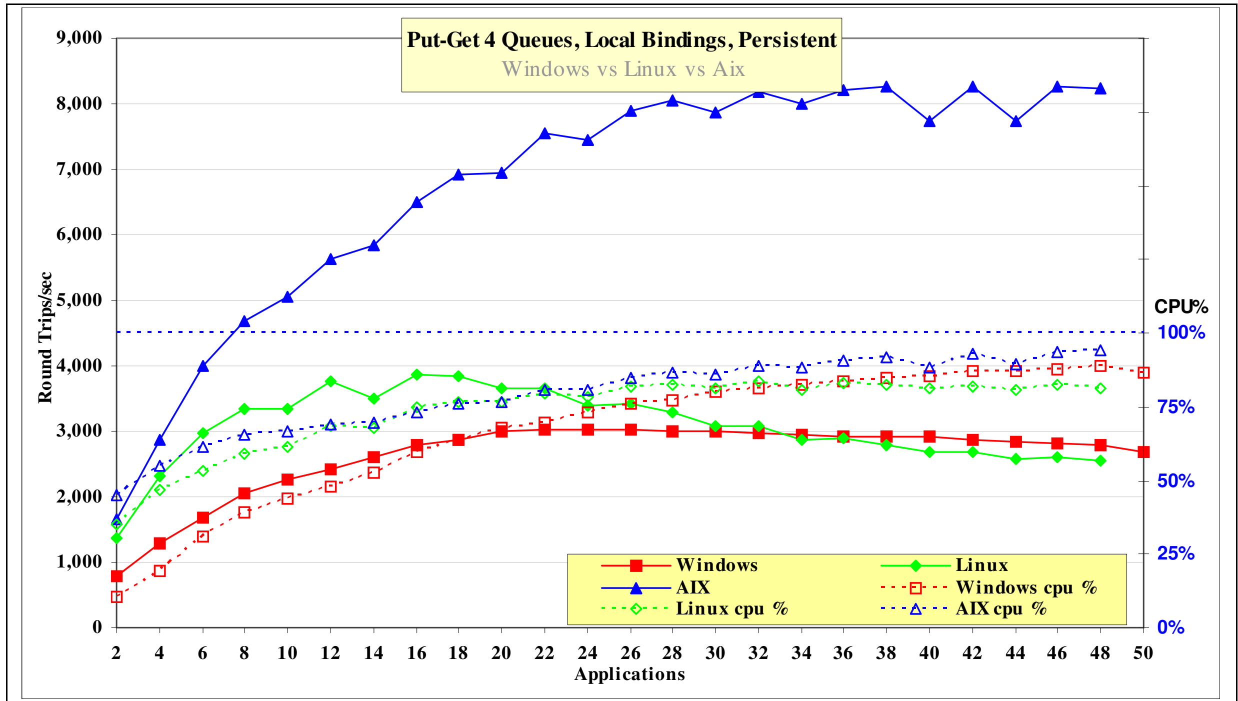


Figure 3 – Put/Get 4Q, Persistent messages, local queue manager

Test name: PG4QLN	Apps	Round Trips/sec	CPU
Linux	**16**	**3877**	**75%**
Windows	**26**	**3036**	**76%**
Aix	**42**	**8264**	**93%**

Table 2 – Put/Get 4Q, Persistent messages, local queue manager

Figure 3 and **Table 2** show the throughput of persistent messages.

2.2 Point to Point , Multiple (Producer, Consumer, Queue) Scenario

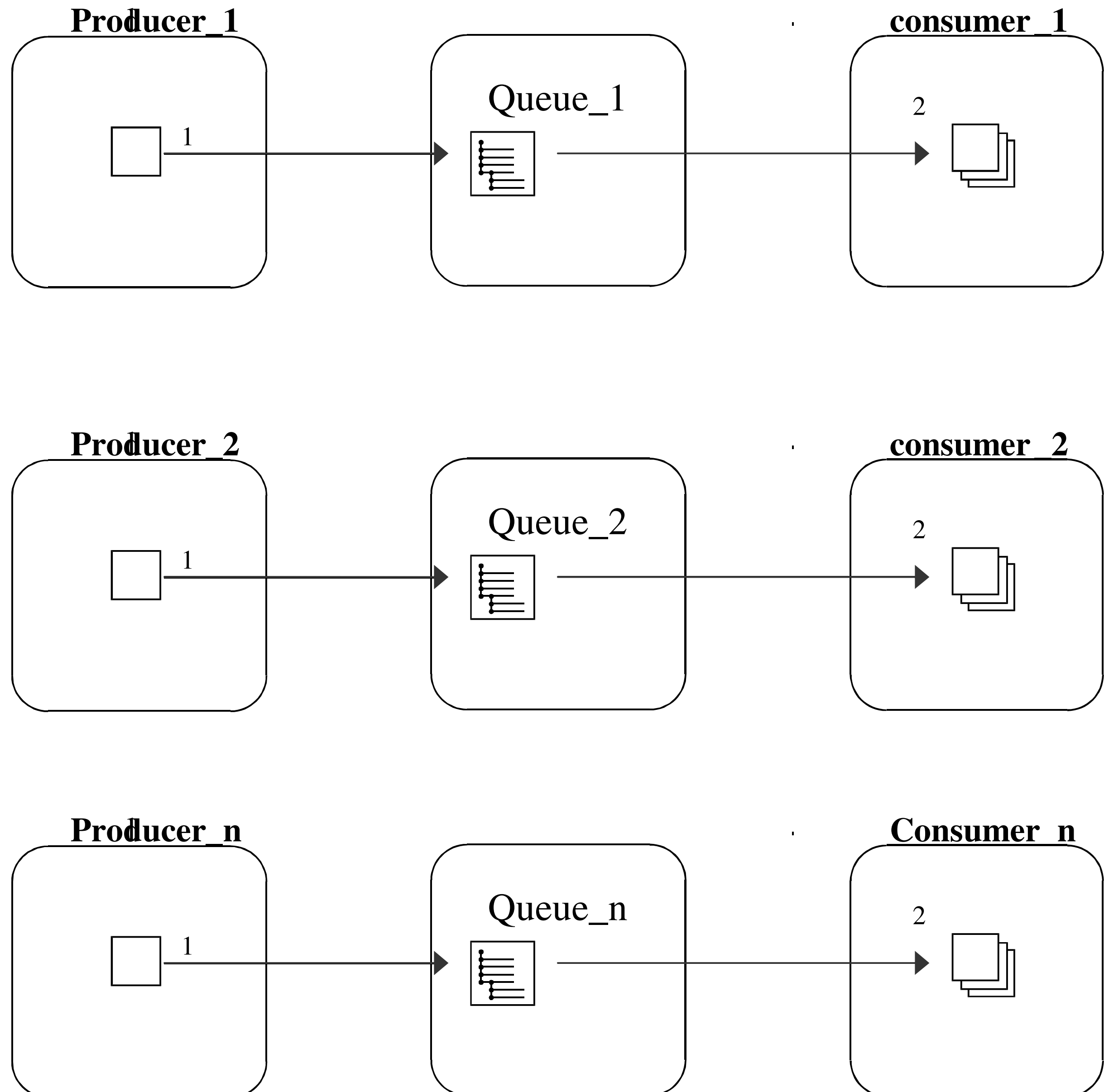


Figure 4 – Producer/Consumer, non persistent

Each Queue is used by only one Producer and one Consumer. The message Producer inserts messages at a predefined rate. The message production rate is 1600 per second for non-persistent and 400 per second for persistent messages. The number of (Producer , Consumer, Queue) triplets is gradually increased and the maximum rate occurs when the consumers prevent the Queue from exceeding a queue depth of one. This testcase provides asynchronous messaging since there is no connection between the number of messages in the system and the number of publishers or subscribers. Messages are counted as put to queue by the producer and as they are retrieved by the consumer hence a particular message created by the producer and got by the consumer is counted as 2 messages.

2.2.1 Producer Consumer, Non Persistent Messages, Local

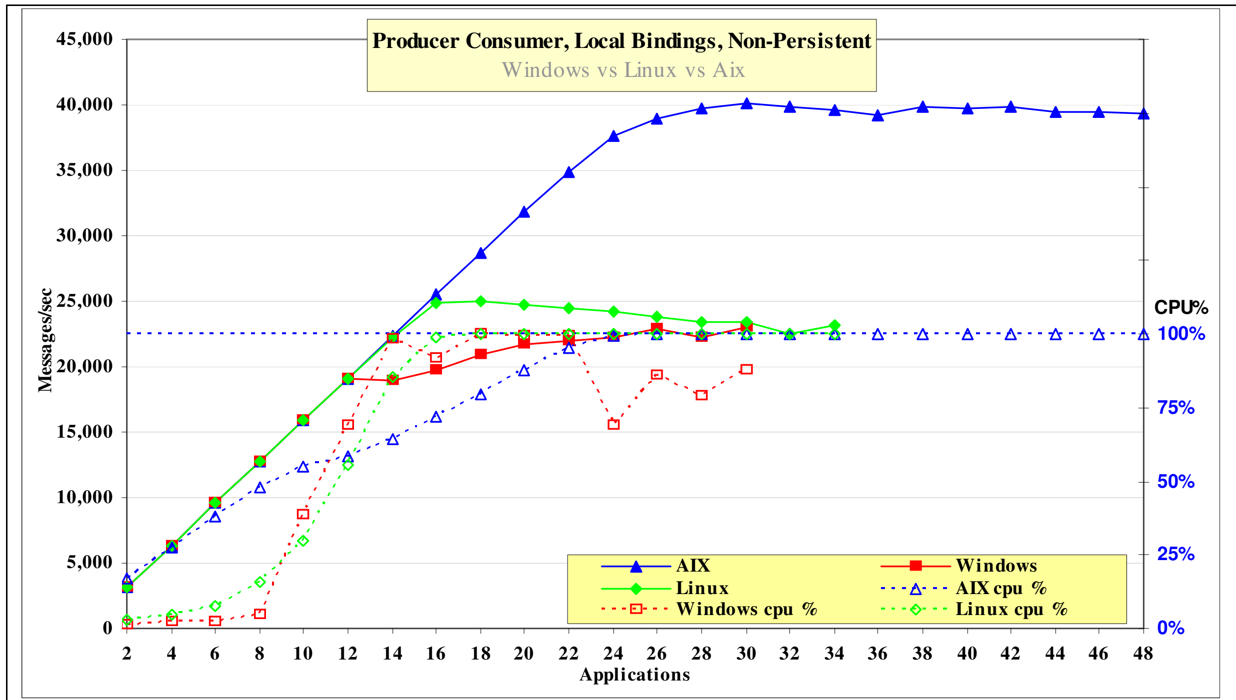


Figure 5 – Producer/Consumer, non persistent messages, local queue manager

Figure 5 and Table 3 show that the throughput of non-persistent messages

Test name: PCLN	Apps	Messages Per second	CPU
Linux	18	24971	100%
Windows	30	22971	88%
Aix	30	40117	100%

Table 3 – Producer/Consumer, non persistent messages, local queue manager

Each message producer creates 1600 non persistent messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 12 producers and 12 consumers (24 Applications) on AIX, the expected throughput is 1600*12*2=38400 whereas the measured throughput is 37590 messages per second

2.2.2 Producer Consumer, Persistent Messages, Local

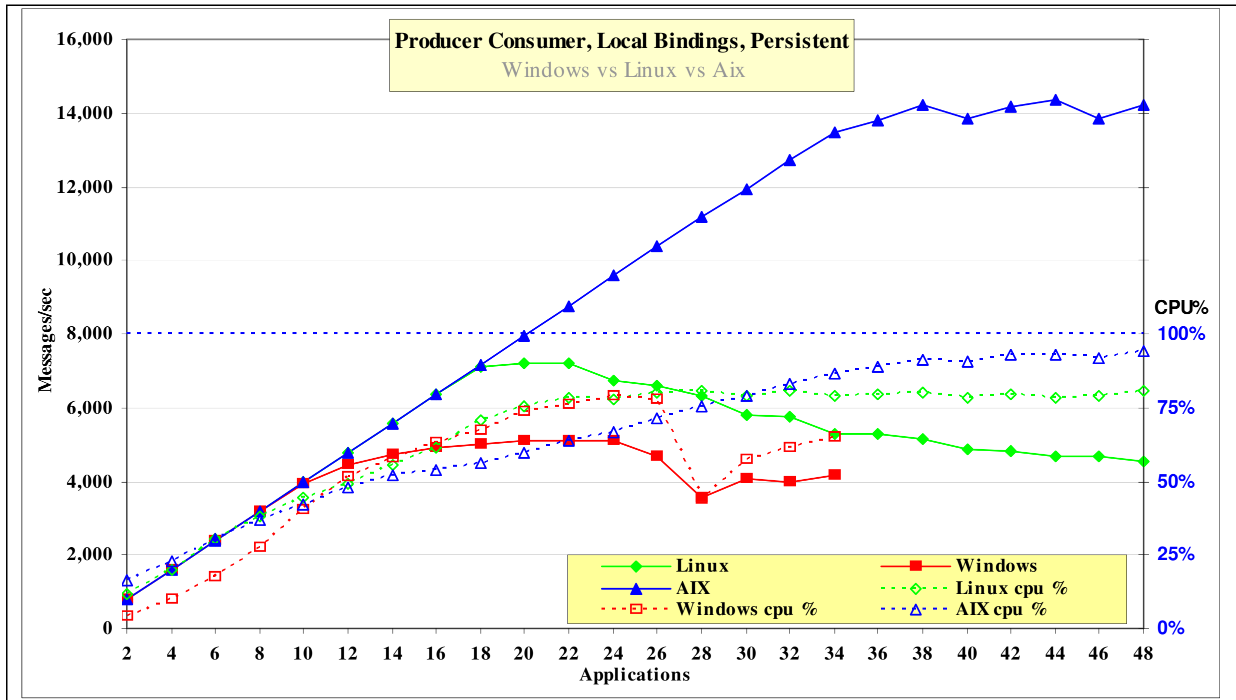


Figure 6 – Producer/Consumer, persistent messages, local queue manager

Test name: PCLP	Apps	Messages Per second	CPU
Linux	20	7191	75%
Windows	20	5084	73%
Aix	44	14341	94%

Table 4 – Producer/Consumer, persistent messages, local queue manager

Each message producer creates 400 messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 16 producers and 16 consumers (32 Applications) on AIX, the expected throughput is 400*16*2=12800 whereas the measured throughput is 12724 messages per second.

2.3 Publish/Subscribe Single Publisher, Many Subscribers Scenario(1:N)

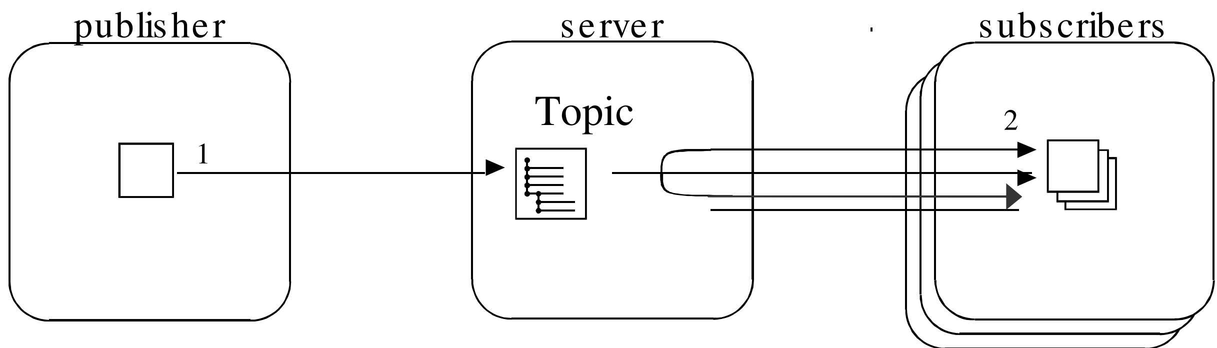


Figure 7 – Publish Subscribe 1:N

All subscribers used unique subscriber queues. Persistent subscribers received five messages in each transaction.

1 A publisher publishes a message to the single topic.

2 Each subscriber then receives the message.

This testcase provides asynchronous messaging since there is no connection between the number of messages in the system and the number of publishers or subscribers. The publisher publishes the next message without any 'think' time. Message count is the number of published messages plus those consumed by the subscribers.

2.3.1 Publish Subscribe 1:N, Non Persistent messages, local

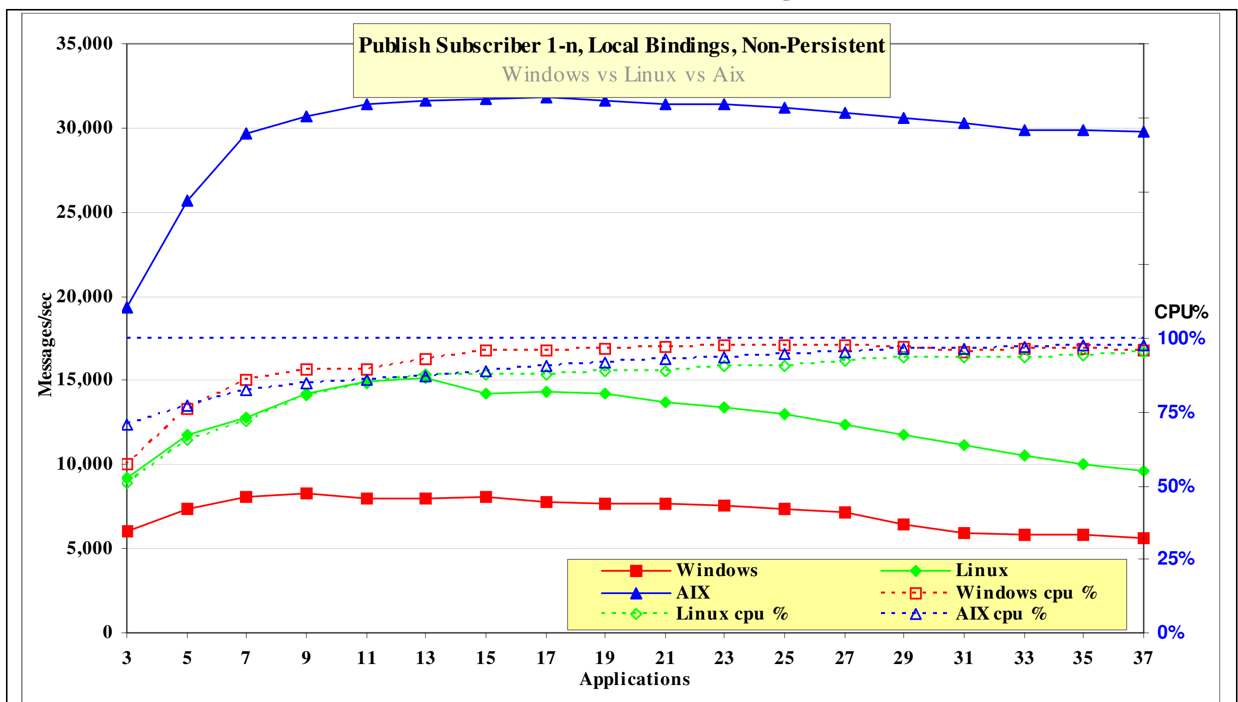


Figure 8 – Pub/Sub 1:N, non-persistent messages, local queue manager

Test name:	Apps	Messages	Publications	CPU	Pubs per second

PS1NLN		Per second	per second		With 2 subscribers Per publication
Linux	13	15120	1163	88%	3058
Windows	9	8281	920	90%	2022
Aix	17	31842	1873	91%	6463

Table 5 – Publish Subscribe 1:N, non Persistent messages, local queue manager

The publisher produces messages as fast as possible. Initially there are 2 subscribers and one publisher when 6463 publications per second can be achieved on AIX. The response time for the publish command increases as the number of subscribers increase hence the system message rate plateaus after 10 subscribers. On AIX with 16 subscribers, the publisher creates 1873 messages per second which are all consumed by the subscribers.

2.3.2 Publish Subscribe 1:N, Persistent messages, local

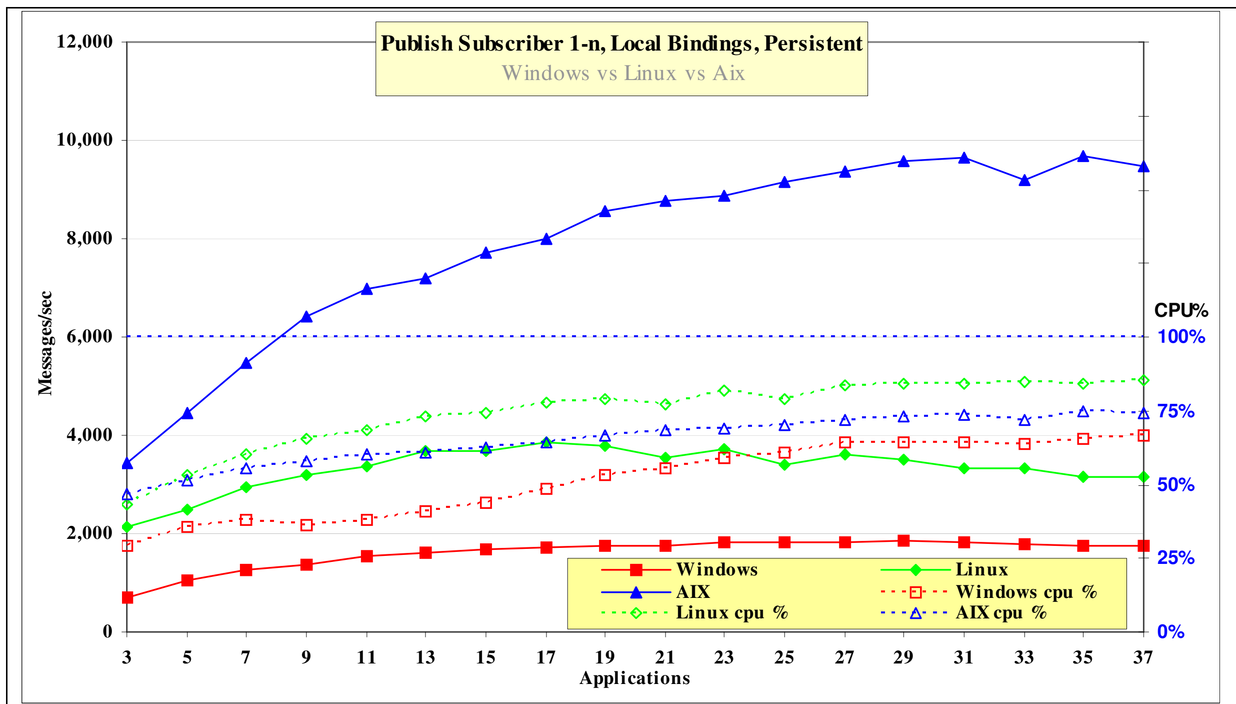


Figure 9 – Pub/Sub 1:N, Persistent messages, local queue manager

Test name: PS1NLP	Apps	Messages Per second	Publications per second	CPU	Pubs per second With 2 subscribers Per publication
Linux	17	3876	228	78%	714
Windows	29	1845	63	64%	234
Aix	35	9667	276	75%	1144

Table 6 – Publish Subscribe 1:N, Persistent messages, local queue manager

The publisher produces messages as fast as possible. Initially there are 2 subscribers and one publisher when 1144 publications per second can be achieved on AIX. The response time for the publish command increases as the number of subscribers increase hence the system message rate plateaus between 16 and 32 subscribers. On AIX with 34 subscribers, the publisher creates 276 messages per second which are all consumed by the subscribers.

2.4 Publish Subscribe multiple (Publisher, Topic, Subscriber) scenario

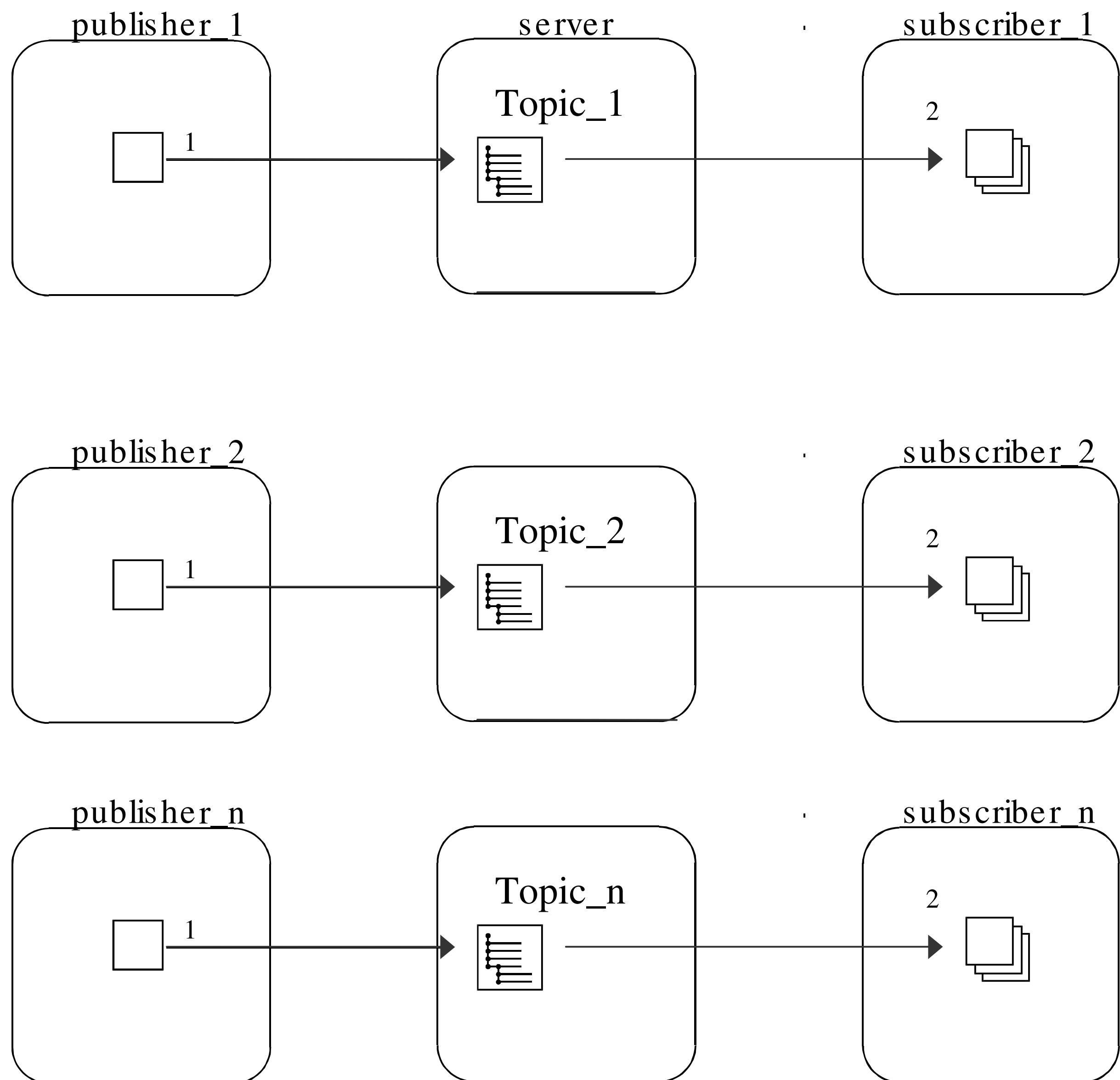


Figure 10 – Publish Subscribe

All subscribers used unique subscriber queues. Persistent subscribers received five messages in each transaction.

1 A publisher publishes a message to the single topic.

2 Only one subscriber had registered for the topic then receives the message.

This testcase provides asynchronous messaging since there is no connection between the number of messages in the system and the number of publishers or subscribers. The publisher publishes message at a predetermined rate which provides for a gradually increasing workload as the number of (Publisher, Topic, Subscriber) triplets is increased. The message production rate is 1600 per second for non-persistent and 400 per second for

persistent messages. Message count is the number of published messages plus those consumed by the subscribers.

2.4.1 Publish Subscribe (Multiple P/T/S), Non Persistent messages, local

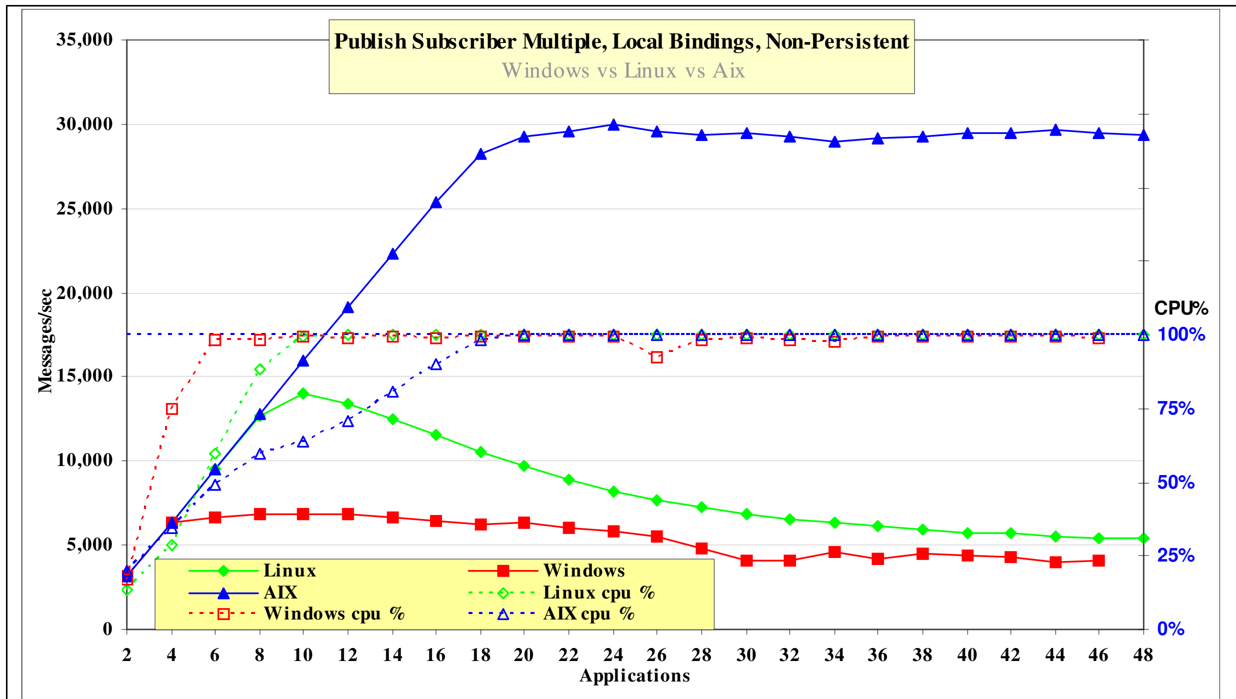


Figure 11 – Publish Subscribe Multiple, Non persistent messages ,local

Test name: PSMLN	Apps	Messages Per second	CPU
Linux	10	14053	100%
Windows	8	6845	99%
Aix	24	29937	100%

Table 7 – Publish Subscribe Multiple, non Persistent messages, local queue manager

Each publisher creates 1600 non persistent messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 9 producers and 9 consumers (18 Applications) on AIX, the expected throughput is 1600*9*2=28800 whereas the measured throughput is 28252 messages per second

2.4.2 Publish Subscribe (Multiple P/T/S), Persistent messages, local

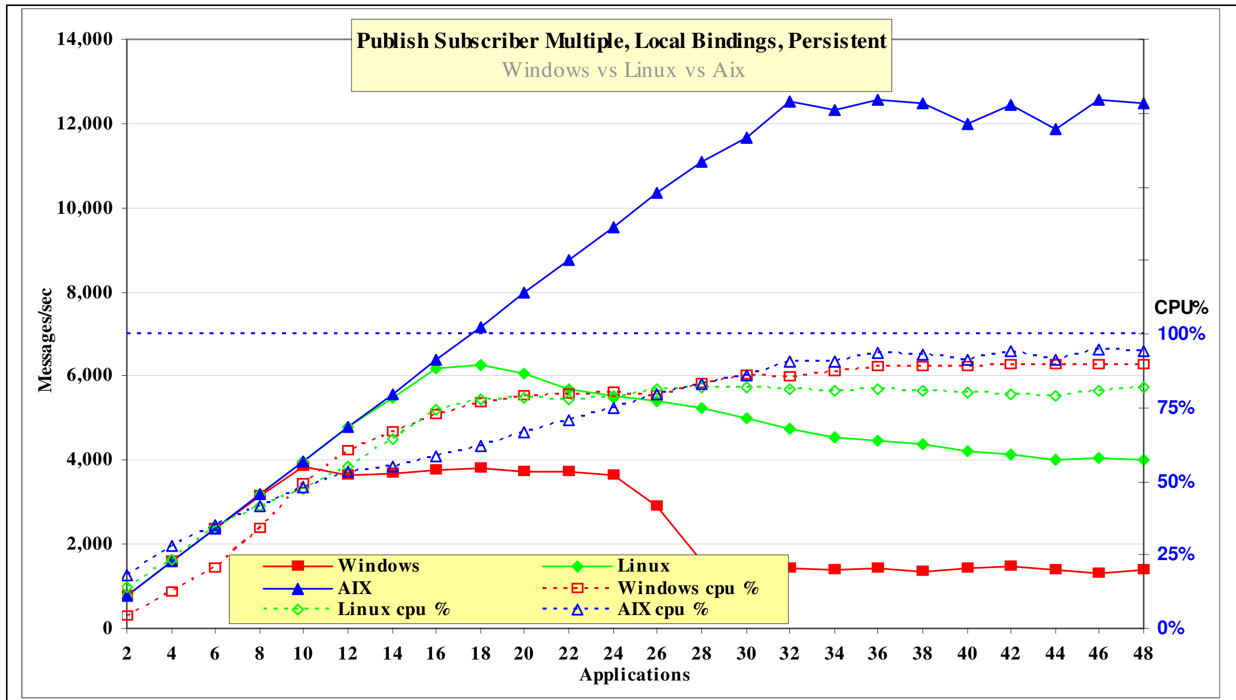


Figure 12 – Publish Subscribe Multiple, Persistent messages, local

Test name: PSMLP	Apps	Messages Per second	CPU
Linux	**18**	**6252**	**78%**
Windows	**10**	**3845**	**50%**
Aix	**36**	**12564**	**94%**

Table 8 – Publish Subscribe Multiple, Persistent messages, local queue manager

Each message producer creates 400 messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 16 producers and 16 consumers (32 Applications) on AIX, the expected throughput is 400*16*2=12800 whereas the measured throughput is 12564 messages per second.

3 Client Channels Test Scenario

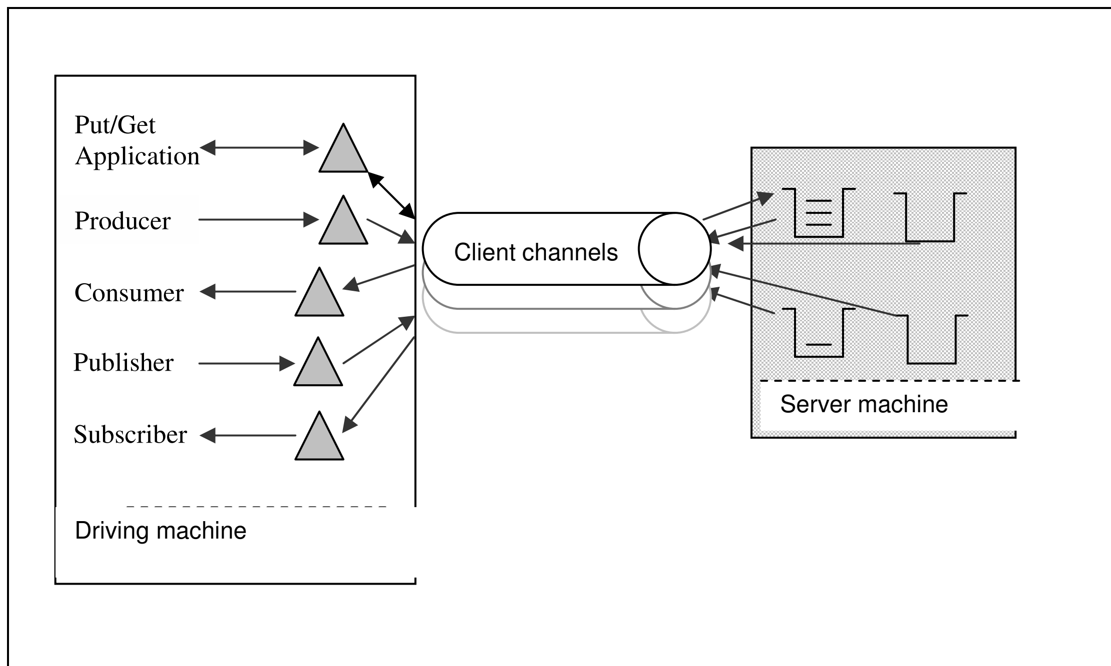


Figure 13 – MQI-client channels into a remote queue manager

The various message producers put a message (over a client channel), to the relevant queue on the server. The consumer application then waits indefinitely for messages to arrive on its input queue.

All of the JMS code is executed on the Client (driver) machine and the individual MQ verbs (Put, Get, Commit are sent to the server to drive the Queue Manager.

The Client Channel is set to 'MQIBindType = FASTPATH' . The major benefit is for non persistent messages because it eliminates the AGENT process (AMQZLAA) and reduces CPU cost. Environments using Channel exits should be aware that the exit code would run inside the Queue Manager.

Version 7 will multiplex multiple clients from the same process over one TCP socket but these measurements create clients on separate processes and avoid the multiplex protocol. This is equivalent to defining the svr-con channel with SHARECNV=1

Higher volumes of Persistent messages are achieved with Client connections than Local bindings.

3.1 Point to Point Put Get 4Q Scenario

3.1.1 Put/Get 4Q Nonpersistent Messages – Client

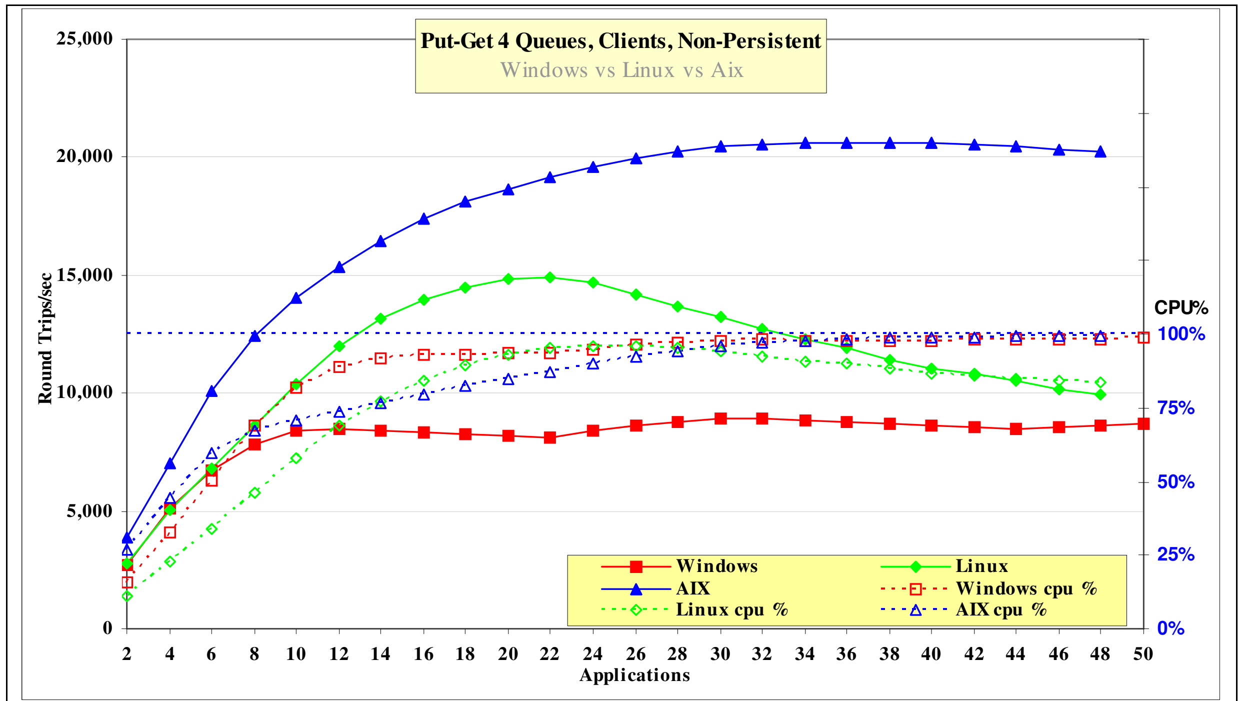


Figure 14 – Put/Get 4Q , non persistent, client

Test name: PG4QCN	Apps	Round Trips/sec	CPU
Linux	22	14903	96%
Windows	32	8952	99%
Aix	34	20626	98%

Table 9 – Put/Get 4Q, non Persistent messages, Client connection

3.1.2 Put/Get 4Q Persistent Messages – Client

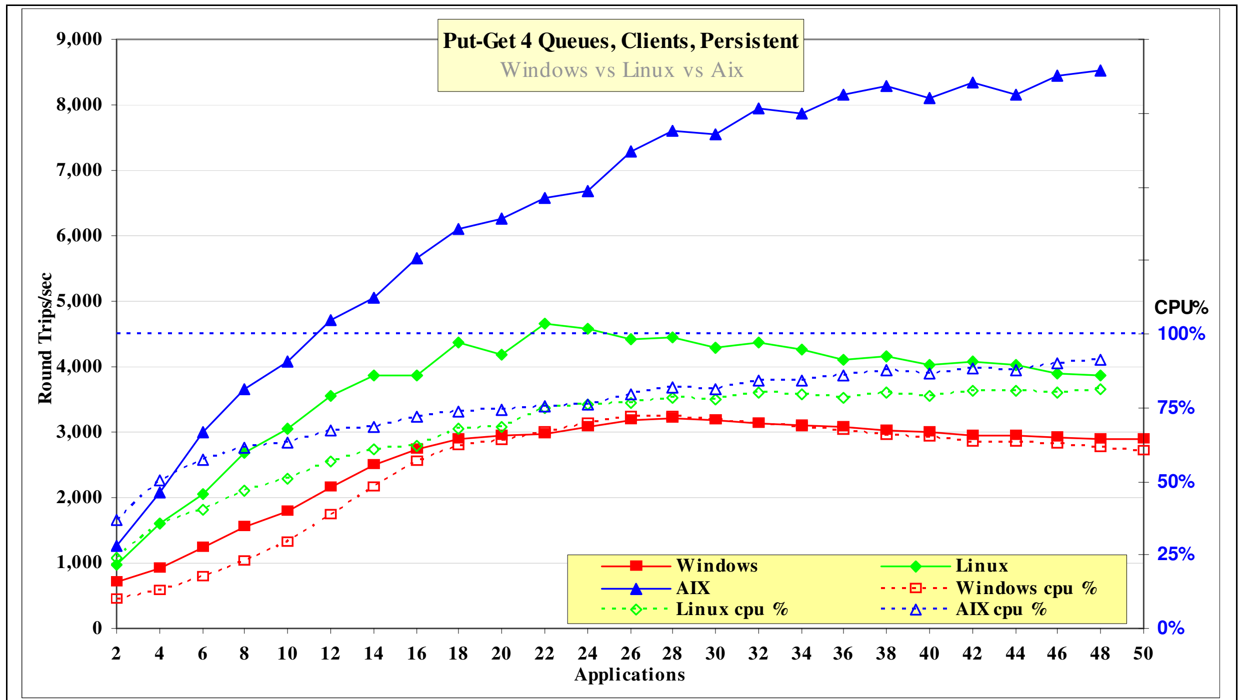


Figure 15 – Put/Get 4Q, persistent, client

Test name: **PG4QCP**	**Apps**	**Round Trips/sec**	**Server CPU**
Linux	**22**	**4664**	**75%**
Windows	**28**	**3219**	**73%**
Aix	**48**	**8529**	**92%**

Table 10 – Put/Get 4Q, Persistent messages, Client connection

.

3.2 Point to Point , Multiple (Producer, Consumer, Queue) Scenario

3.2.1 Producer Consumer, Non Persistent Messages, Client

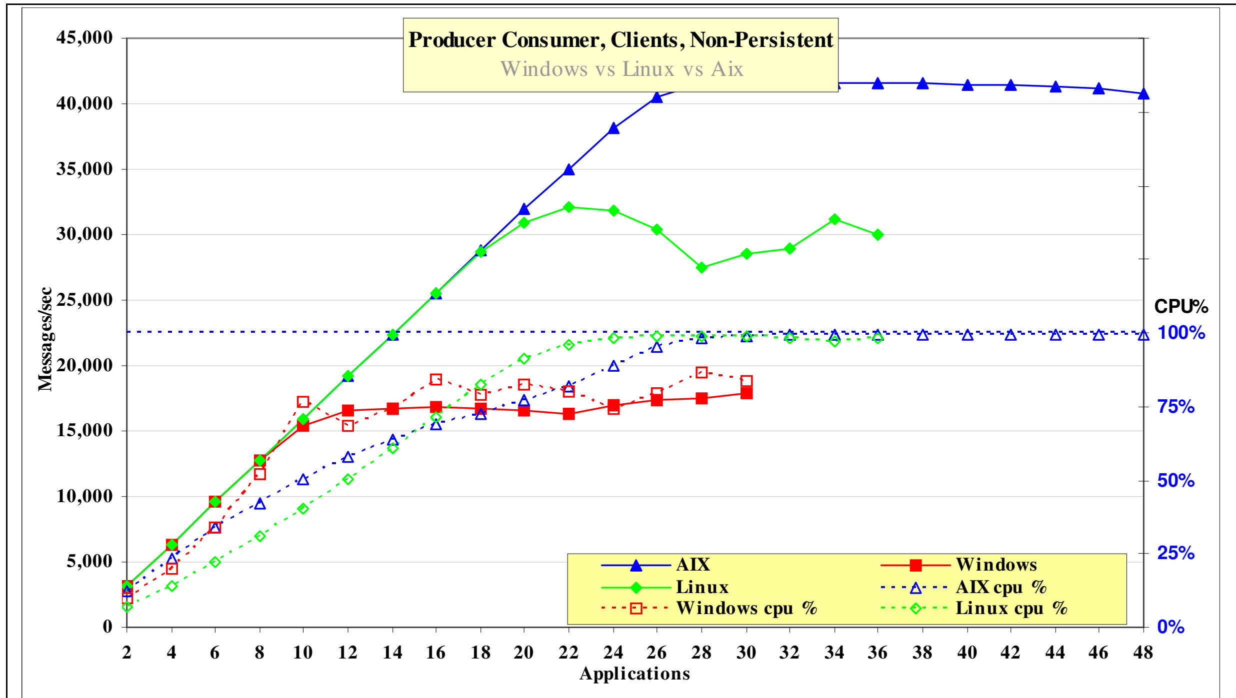


Figure 16 – Producer/Consumer, non persistent, client

Test name: PCCN	Apps	Messages Per second	Server CPU
Linux	22	32077	96%
Windows	30	17876	84%
Aix	30	41842	100%

Table 11 – Producer/Consumer, non Persistent messages, Client connection

Each message producer creates 1600 non persistent messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 12 producers and 12 consumers (24 Applications) on AIX, the expected throughput is 1600*12*2=38400 whereas the measured throughput is 38183 messages per second

3.2.2 Producer Consumer, Persistent Messages, Client

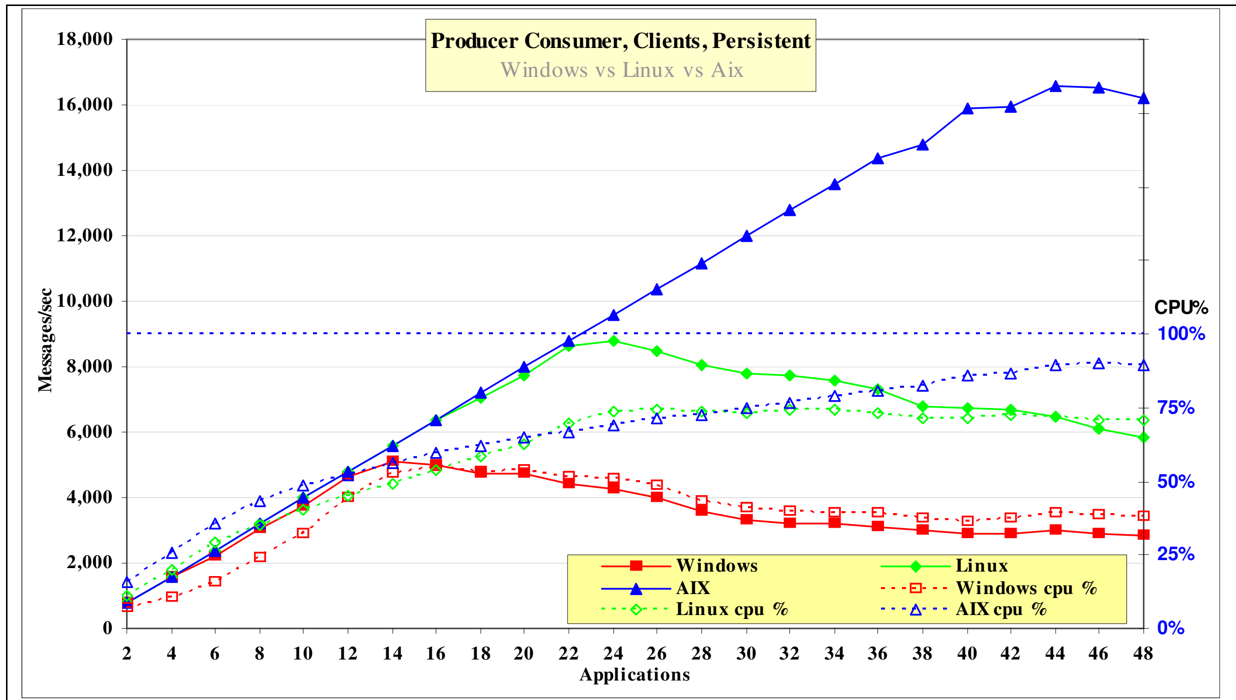


Figure 17 – producer/Consumer, persistent, client

Test name: PCCP	Apps	Messages Per second	Server CPU
Linux	24	8806	74%
Windows	14	5114	53%
Aix	44	16585	90%

Table 12 – Producer/Consumer, Persistent messages, Client connection

Each message producer creates 400 messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 20 producers and 20 consumers (40 Applications) on AIX, the expected throughput is 400*20*2=16000 whereas the measured throughput is 15916 messages per second

3.3 Publish/Subscribe Single Publisher, Many Subscribers Scenario(1:N)

3.3.1 Publish Subscribe 1:N, Non Persistent messages, Client

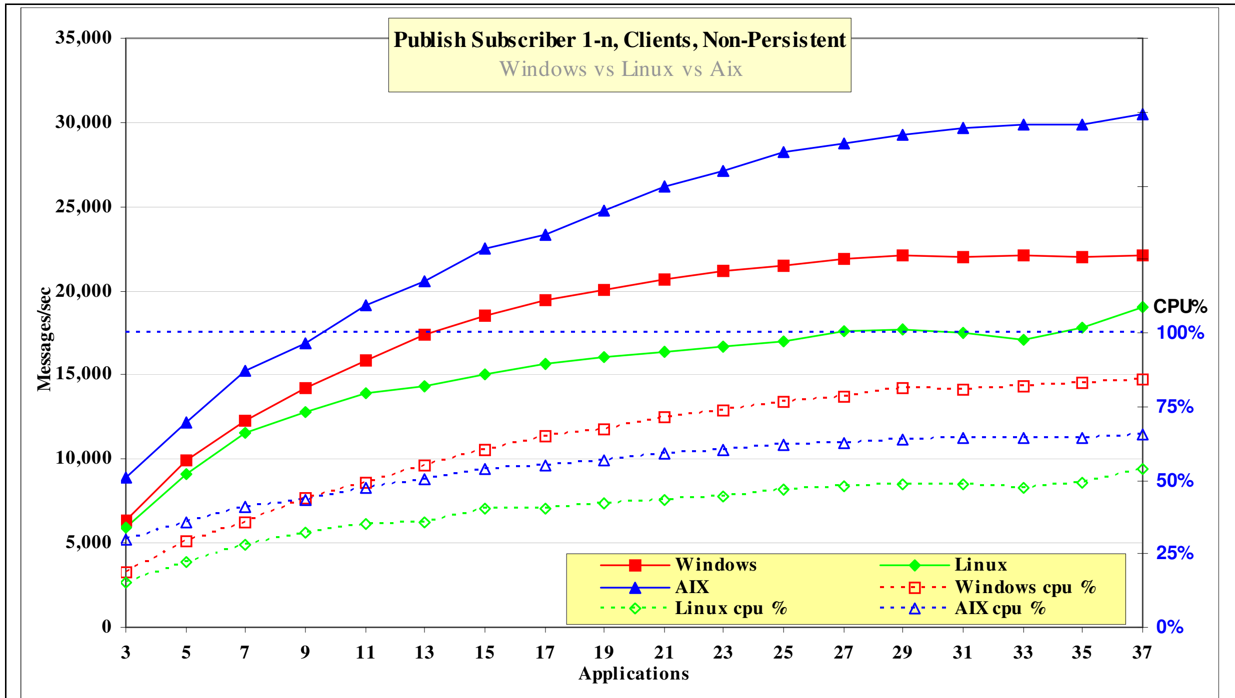


Figure 18 – Publish Subscribe 1:N, non persistent, client

Test name: PS1NCN	Apps	Messages Per second	Publications per second	Server CPU	Pubs per second With 2 subscribers Per publication
Linux	37	19065	515	53%	1980
Windows	29	22073	761	82%	2127
Aix	37	30547	825	66%	2972

Table 13 – Publish/Subscribe 1:N, non Persistent messages, Client connection

The publisher produces messages as fast as possible. Initially there are 2 subscribers and one publisher when 2972 publications per second can be achieved on AIX. The response time for the publish command increases as the number of subscribers increase. On AIX with 36 subscribers, the publisher creates 825 messages per second which are all consumed by the subscribers.

3.3.2 Publish Subscribe 1:N, Persistent messages, Client

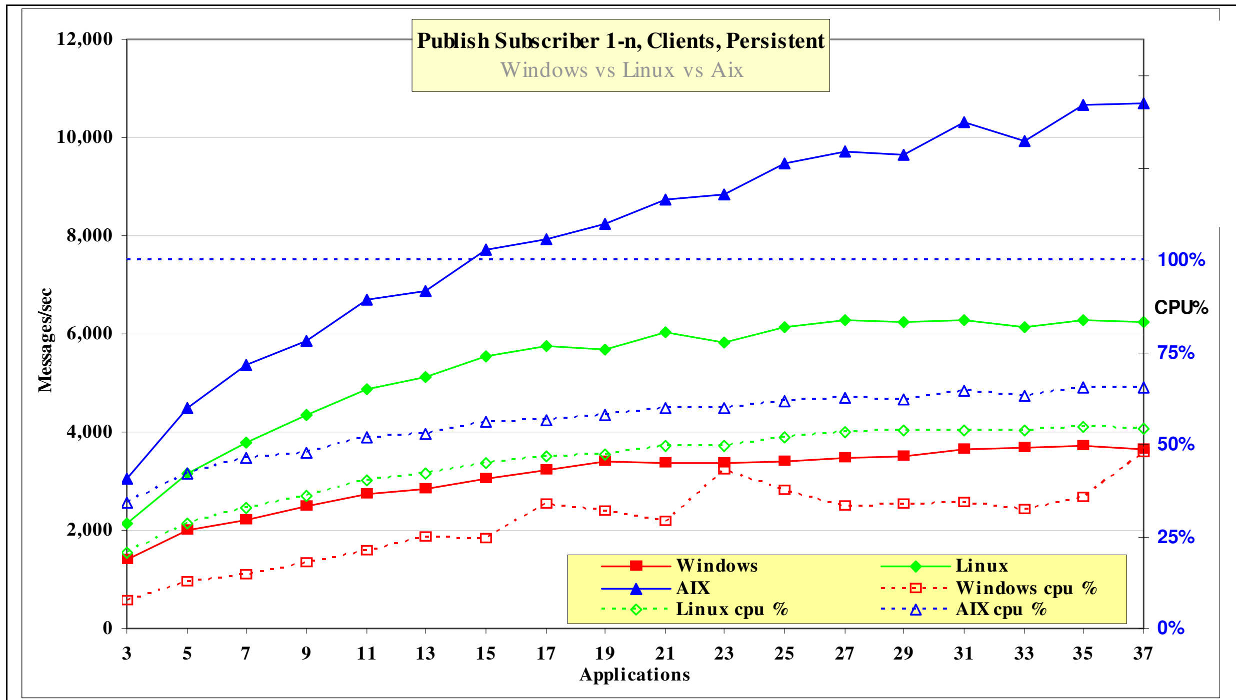


Figure 19 – Publish Subscribe 1:N, persistent, client

Test name: **PS1NCP**	**Apps**	**Messages Per second**	**Publications per second**	**Server CPU**	**Pubs per second With 2 subscribers Per publication**
Linux	**35**	**6292**	**180**	**55%**	**719**
Windows	**35**	**3706**	**106**	**36%**	**469**
Aix	**37**	**10694**	**289**	**66%**	**1013**

Table 14 – Publish/Subscribe 1:N, Persistent messages, Client connection

The publisher produces messages as fast as possible. Initially there are 2 subscribers and one publisher when 1013 publications per second can be achieved on AIX. The response time for the publish command increases as the number of subscribers increase. On AIX with 36 subscribers, the publisher creates 289 messages per second which are all consumed by the subscribers

3.4 Publish Subscribe multiple (Publisher, Topic, Subscriber) scenario

3.4.1 Publish Subscribe (Multiple P/T/S), Non Persistent messages, Client

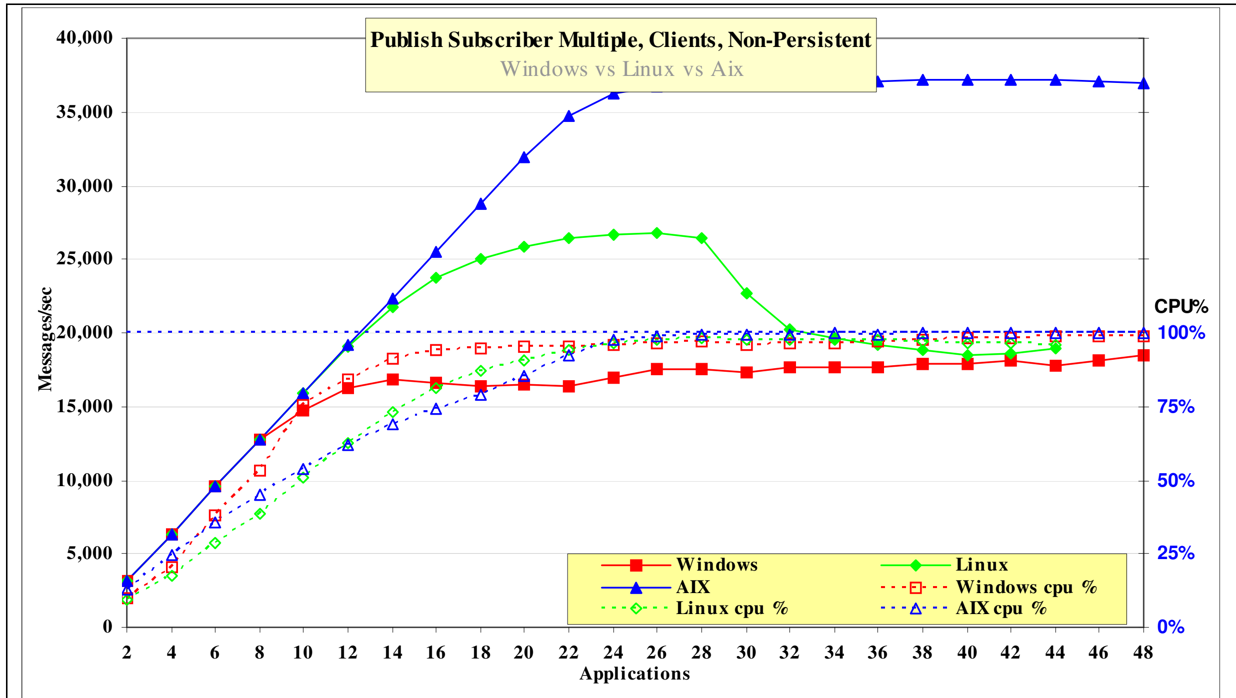


Figure 20 – Publish Subscribe Multiple, non persistent, client

Test name: PSMCN	Apps	Messages Per second	Server CPU
Linux	26	26808	98%
Windows	48	18459	99%
Aix	30	37228	100%

Table 15 – Publish/Subscribe Multiple, non Persistent messages, Client connection

Each publisher creates 1600 non persistent messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 11 producers and 11 consumers (22 Applications) on AIX, the expected throughput is 1600*11*2=35200 whereas the measured throughput is 34712 messages per second

3.4.2 Publish Subscribe (Multiple P/T/S), Persistent messages, Client

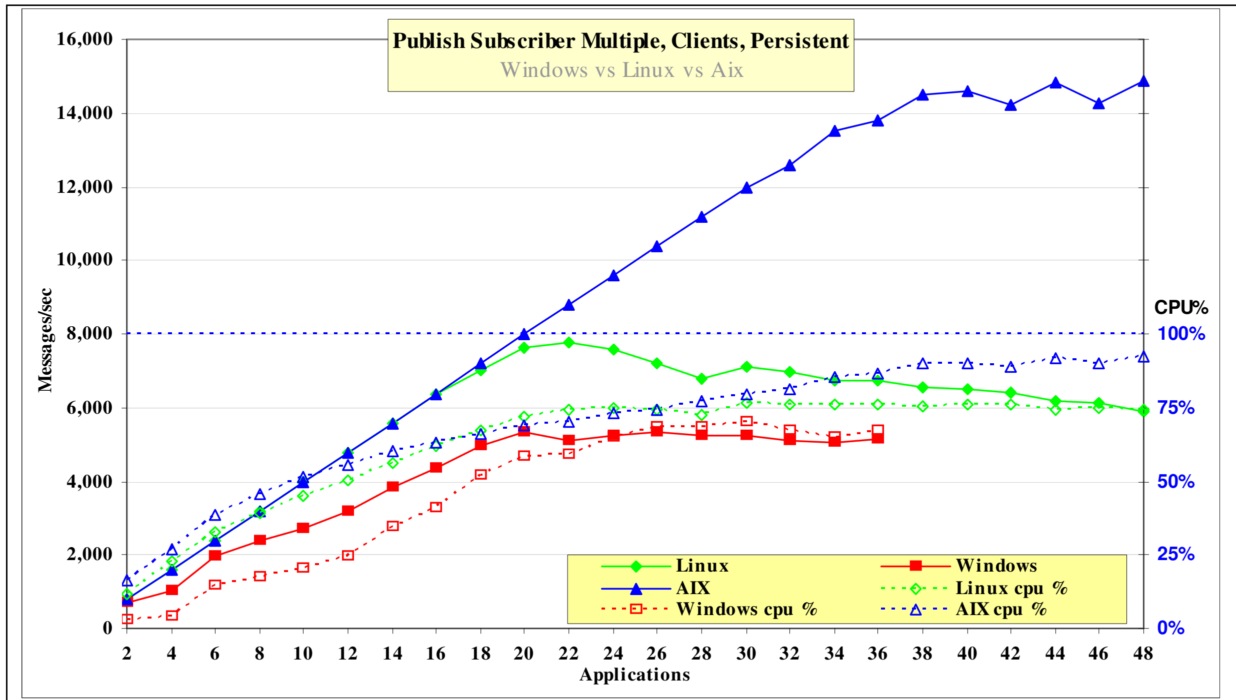


Figure 21 – Publish Subscribe multiple, persistent, client

Test name: **PSMCP**	**Apps**	**Messages Per second**	**Server CPU**
Linux	**22**	**7784**	**75%**
Windows	**20**	**5386**	**59%**
Aix	**48**	**14868**	**93%**

Table 16 – Publish/Subscribe Multiple, Persistent messages, Client connection

Each message producer creates 400 messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 17 producers and 17 consumers (34 Applications) on AIX, the expected throughput is 400*17*2=13600 whereas the measured throughput is 13517 messages per second.

4 z/OS – Local Binding & Client

This chapter shows the throughput achieved when the zOS Queue Manager is used with the benchmarks run in Binding and/or Client mode. Bindings mode means the JMS applications are executing on zOS and Client mode means the JMS applications are running on Linux with a client channel to the zOS server.

4.1 Point to Point Put Get 4Q Scenario

4.1.1 Put/Get 4Q Non persistent Messages

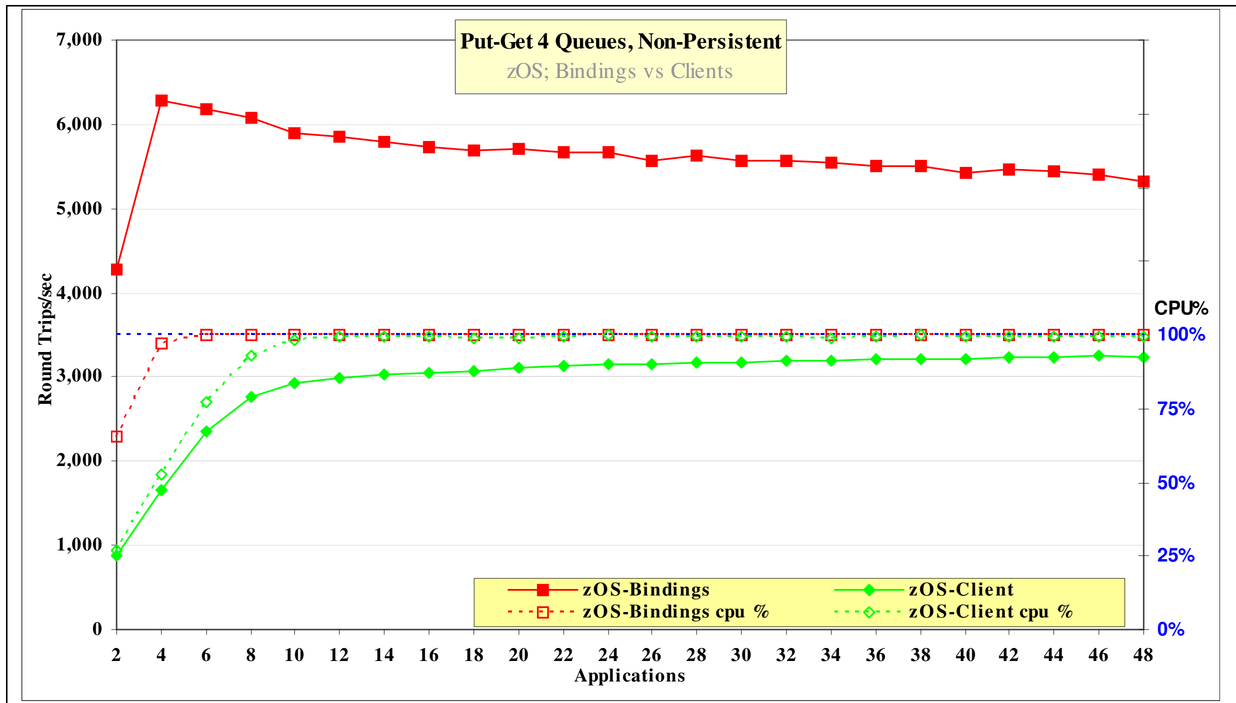


Figure 22 – Put/Get 4Q , non persistent

Test name: **PG4QZN**	**Apps**	**Round Trips/sec**	**CPU**
Binding	**4**	**6293**	**99%**
Client	**36**	**3209**	**100%**

Table 17 – Put/Get 4Q, non persistent messages

Reducing the number of queues from four to one has minimal effect.

4.1.2 Put/Get 4Q Persistent Messages

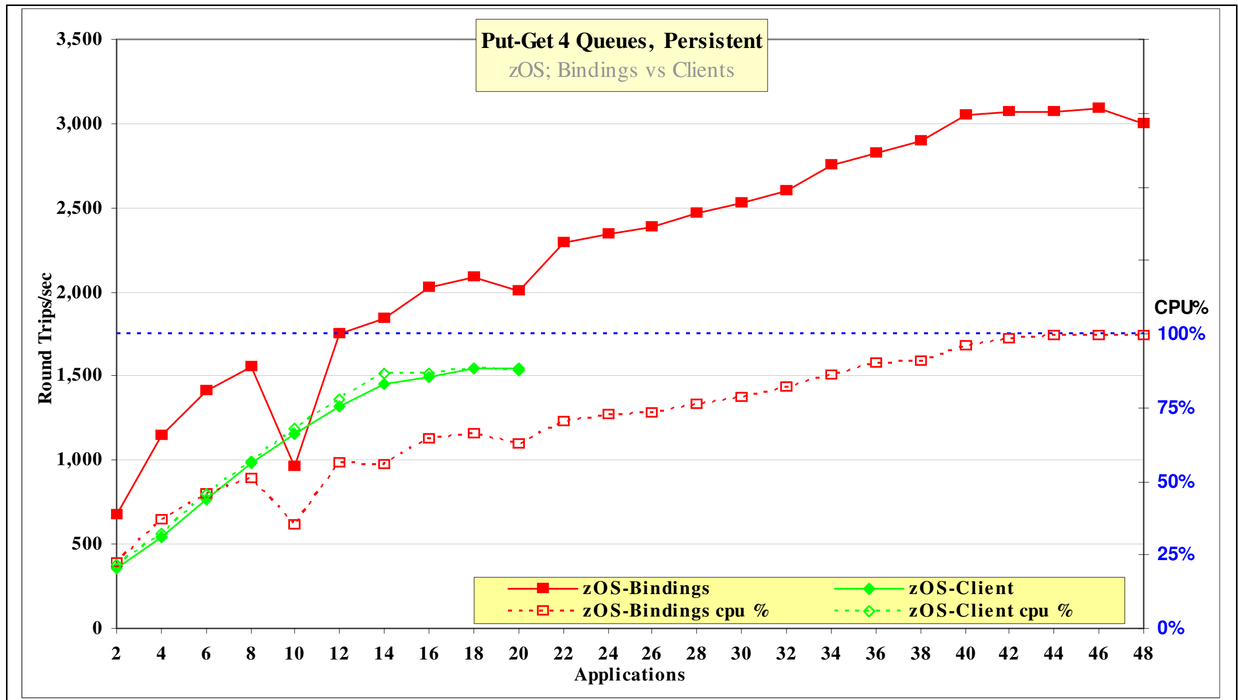


Figure 23 – Put/Get 4Q, persistent

Test name: PG4QZP	Apps	Round Trips/sec	Server CPU
Binding	46	3089	99%
Client	20	1549	88%

Table 18 – Put/Get 4Q, Persistent messages

With Persistent messages, the log is the bottleneck and reducing the number of queues to one has minimal effect.

4.1.3 Put/Get 1Q Persistent Messages – Client

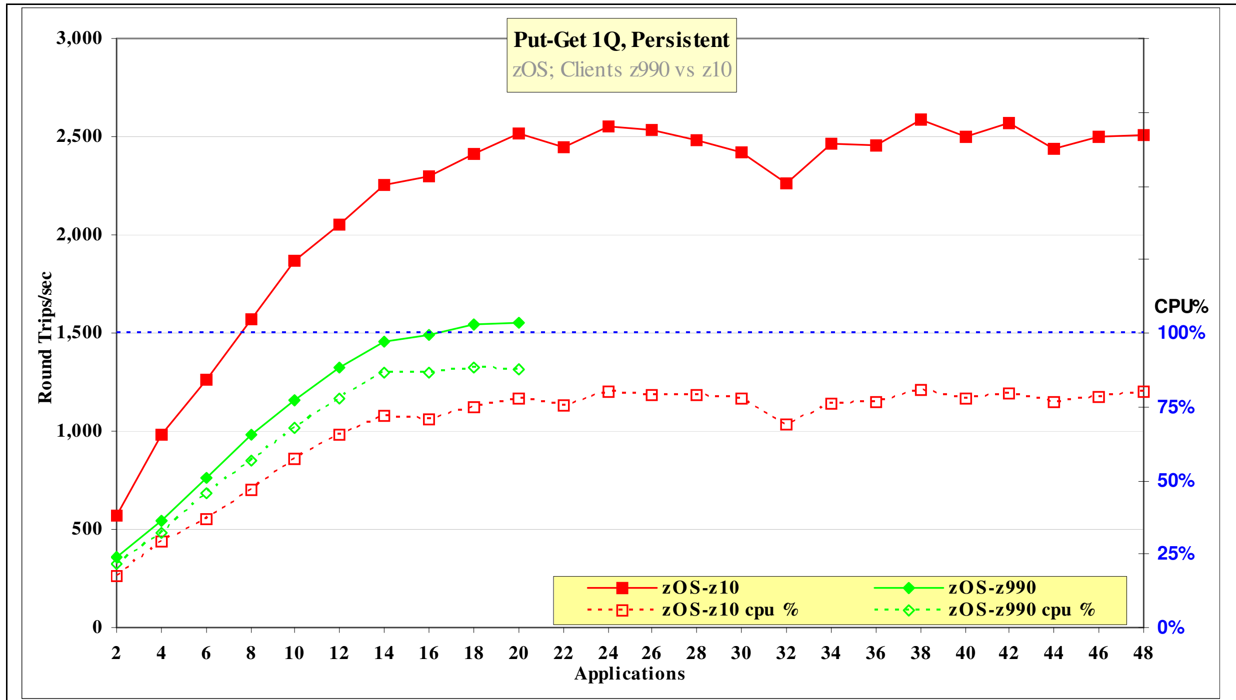


Figure 24 – Put/Get 1Q, persistent messages, Client connection

This comparison using a z10 is the only measurement of this system. 3 CPU(dedicated) on z10(2097-E64) with the same disk and network connectivity as the z990 described in Chapter 8 . The scenario uses a single Queue shared among all clients. The increased power of the z10 provides a significant increase in throughput.

4.2 Point to Point , Multiple (Producer, Consumer, Queue) Scenario

4.2.1 Producer Consumer, Non Persistent Messages

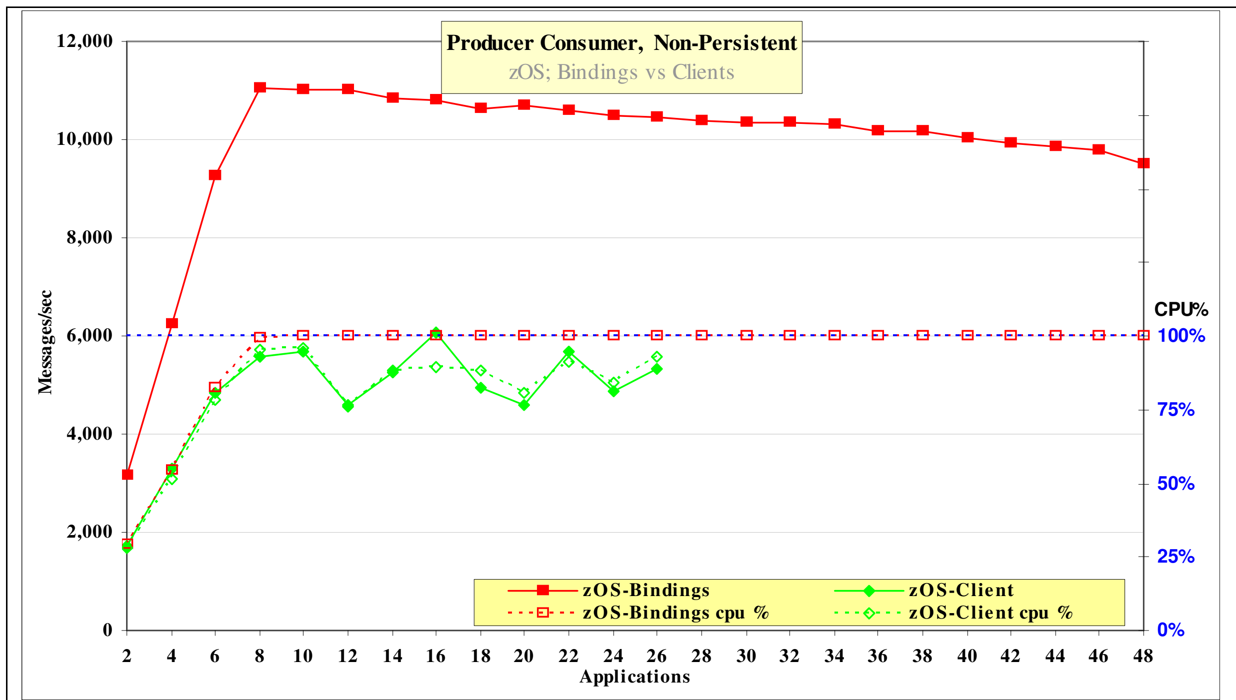


Figure 25 – Producer/Consumer, non persistent

Test name: PCZN	Apps	Messages Per second	Server CPU
Binding	8	11043	100%
Client	16	6075	90%

Table 19 – Producer/Consumer, non Persistent messages

Each message producer creates 1600 non persistent messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 4 producers and 4 consumers (8 Applications) on zOS/Client mode, the expected throughput is 1600*4*2=12800 whereas the measured throughput is 11043 messages per second

4.2.2 Producer Consumer, Persistent Messages

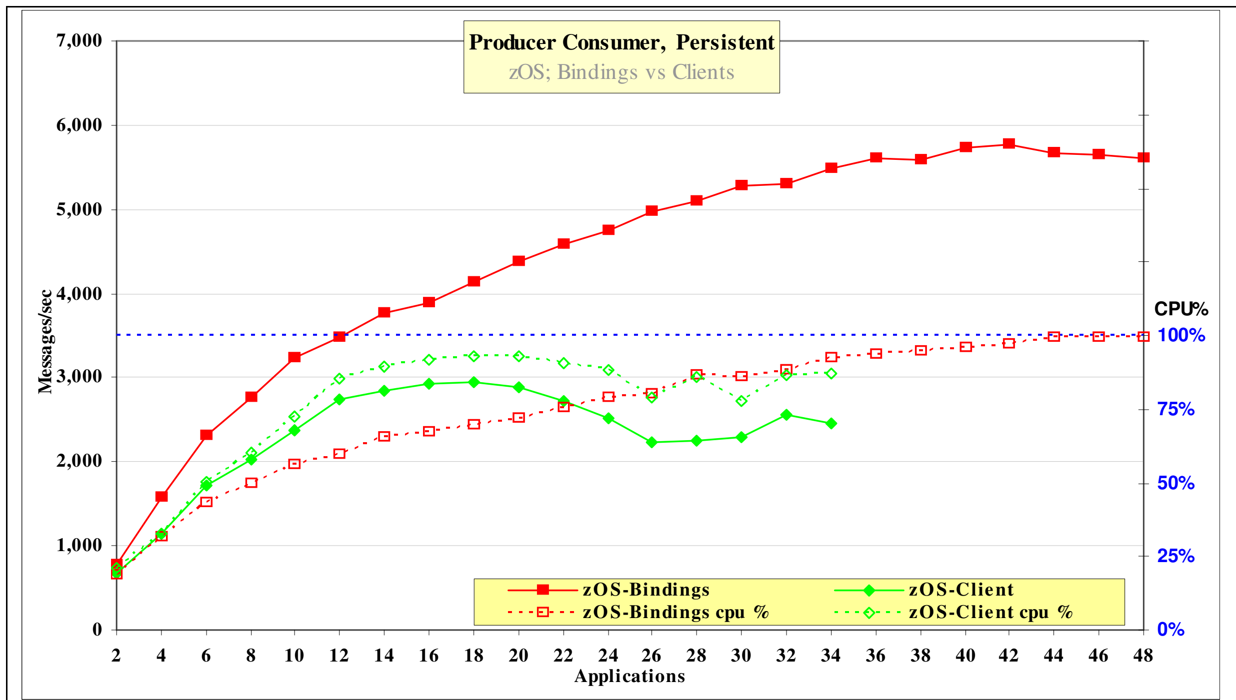


Figure 26 – producer/Consumer, persistent, client

Test name: PCZP	Apps	Messages Per second	Server CPU
Binding	42	5773	97%
Client	18	2939	93%

Table 20 – Producer/Consumer, Persistent messages

Each message producer creates 400 messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 3 producers and 3 consumers (6 Applications) on zOS Bindings mode, the expected throughput is 400*3*2=2400 whereas the measured throughput is 2312 messages per second

4.3 Publish/Subscribe Single Publisher, Many Subscribers Scenario(1:N)

4.3.1 Publish Subscribe 1:N, Non Persistent messages

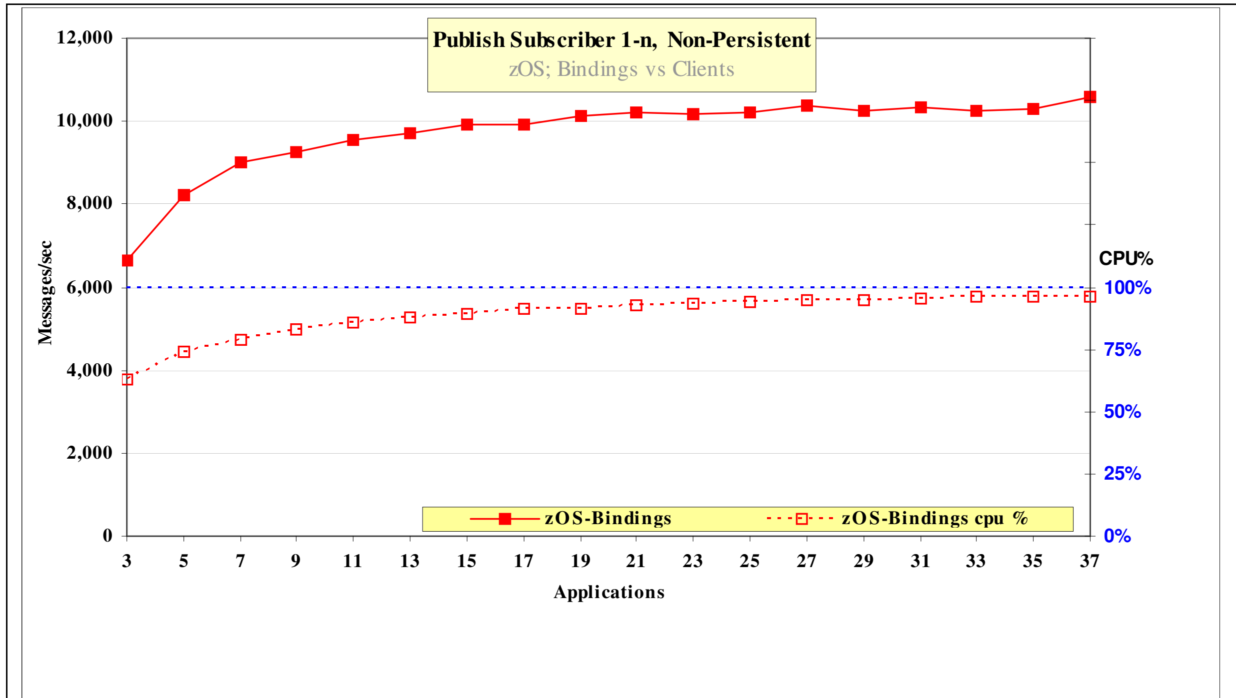


Figure 27 – Publish Subscribe 1:N, non persistent

Test name: PS1NZN	Apps	Messages Per second	Publications per second	Server CPU	Pubs per second With 2 subscribers Per publication
Binding	**37**	**10597**	**286**	**97%**	**2221**

Table 21 – Publish/Subscribe 1:N, non Persistent messages

The publisher produces messages as fast as possible. Initially there are 2 subscribers and one publisher when 2221 publications per second can be achieved on zOS Bindings mode. The response time for the publish command increases as the number of subscribers increase. On zOS Bindings mode with 36 subscribers, the publisher creates 286 messages per second which are all consumed by the subscribers.

4.3.2 Publish Subscribe 1:N, Persistent messages

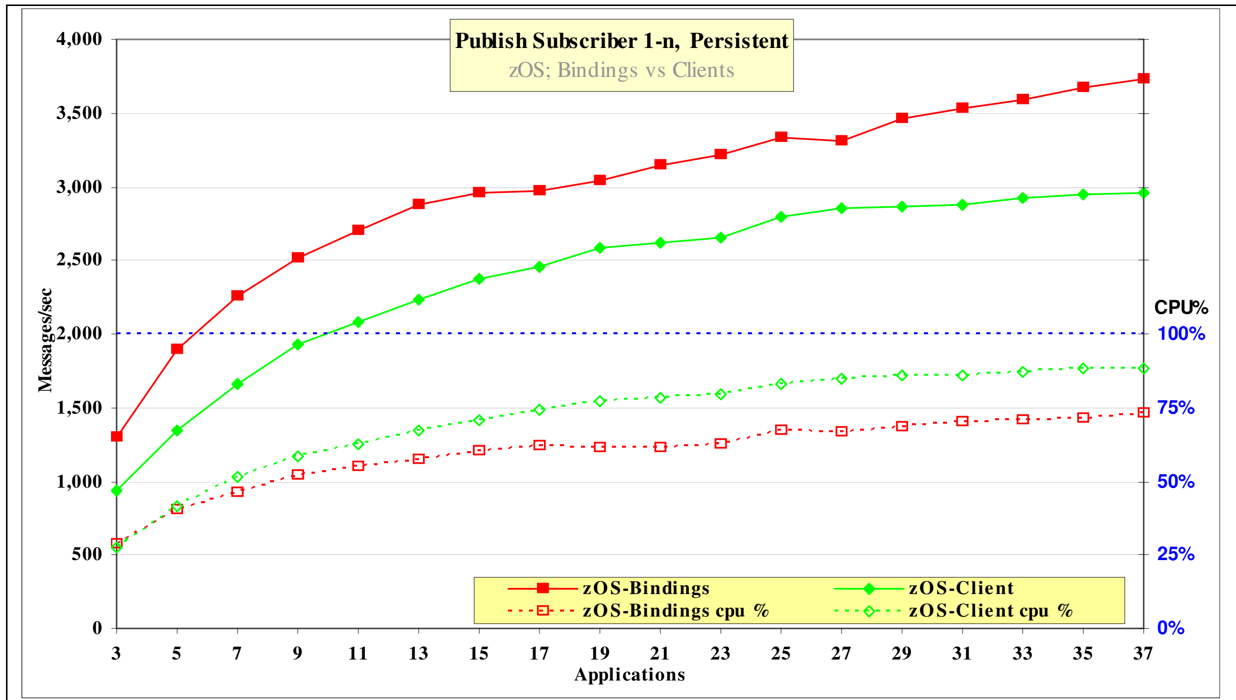


Figure 28 – Publish Subscribe 1:N, persistent

Test name: PS1NZP	Apps	Messages Per second	Publications per second	Server CPU	Pubs per second With 2 subscribers Per publication
Binding	37	3728	101	72%	433
Client	37	2960	80	89%	309

Table 22 – Publish/Subscribe 1:N, Persistent messages

The publisher produces messages as fast as possible. Initially there are 2 subscribers and one publisher when 309 publications per second can be achieved on zOS/Client mode. The response time for the publish command increases as the number of subscribers increase. On zOS Client mode with 36 subscribers, the publisher creates 80 messages per second which are all consumed by the subscribers

4.4 Publish Subscribe multiple (Publisher, Topic, Subscriber) scenario

4.4.1 Publish Subscribe (Multiple P/T/S), Non Persistent messages

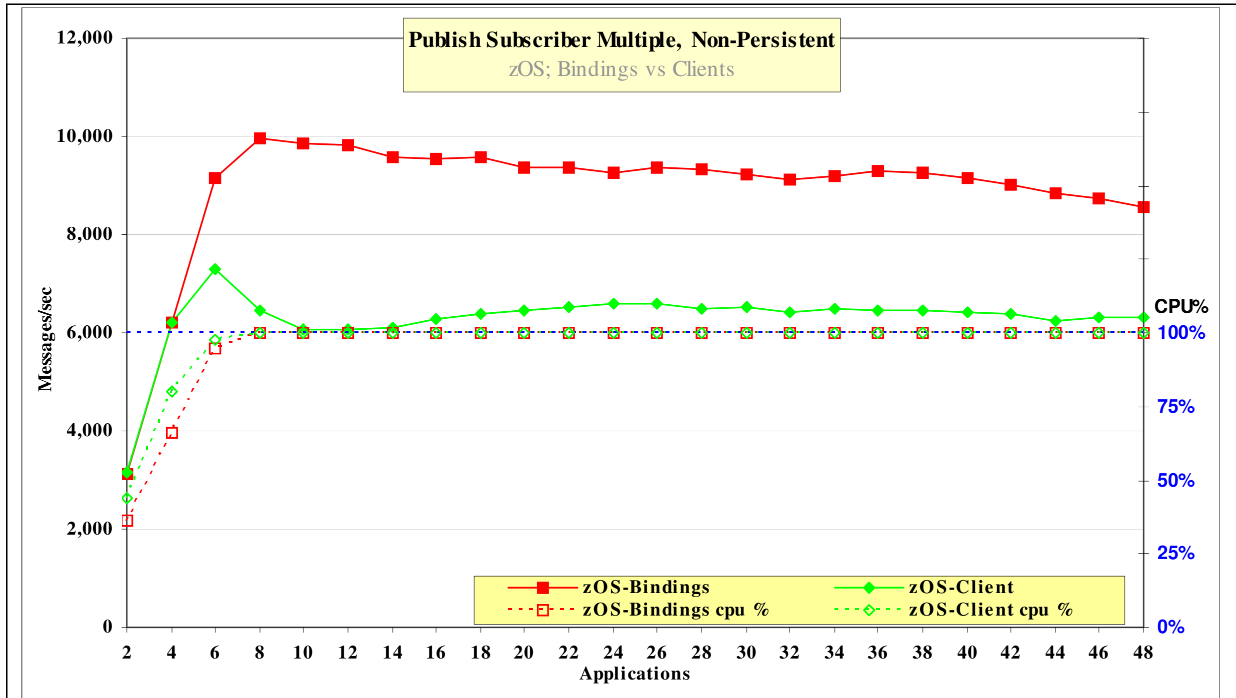


Figure 29 – Publish Subscribe Multiple, non persistent

Test name: PSMZN	Apps	Messages Per second	Server CPU
Binding	8	9947	100%
Client	6	7309	98%

Table 23 – Publish/Subscribe Multiple, non Persistent messages

Each publisher creates 1600 non persistent messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 3 producers and 3 consumers (6 Applications) on zOS Bindings mode, the expected throughput is 1600*3*2=9600 whereas the measured throughput is 9172 messages per second

4.4.2 Publish Subscribe (Multiple P/T/S), Persistent messages

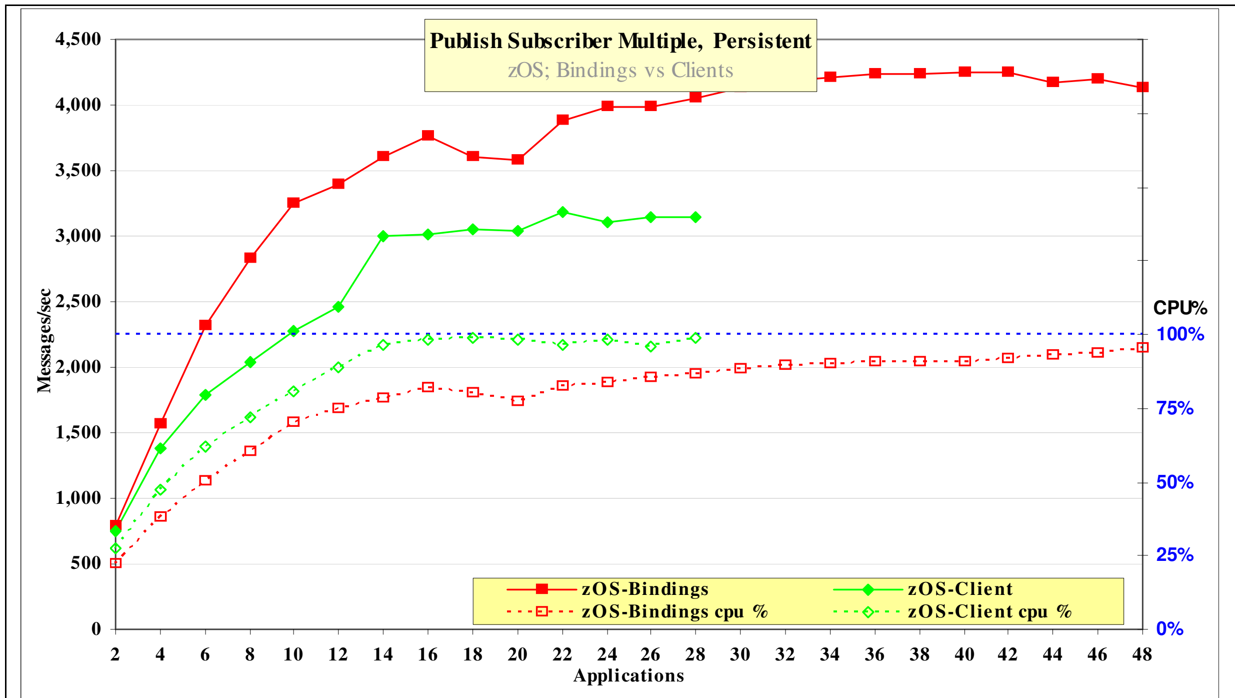


Figure 30 – Publish Subscribe multiple, persistent,

Test name: PSMZP	Apps	Messages Per second	Server CPU
Binding	42	4254	92%
Client	28	3150	99%

Table 24 – Publish/Subscribe Multiple, Persistent messages

Each message producer creates 400 messages per second and the system throughput increases as a straight diagonal line until the system capacity is achieved. With 4 producers and 4 consumers (8 Applications) on zOS Bindings mode, the expected throughput is 400*4*2=3200 whereas the measured throughput is 2829 messages per second.

4.5 JMS Selectors and Correlation Identifiers on z/OS

The basic model for these scenarios is to insert a message to a common queue and then retrieve the message using a selector. Each thread will generate a correlator for the inserted message and then retrieve the message specifying the correlator. All of the messages inserted by thread-n will contain the unique key used by thread –n thus ensuring that each thread retrives the message previously inserted. There are 4 varients for this test

1. The message is retrived without specifying a selector.
2. The message is produced with JMSCorrelationID set to bytes
3. The message is produced with JMSCorrelationID set to string
4. The message is produced with an arbitrary selector

4.5.1 JMS Selectors and Correlation Identifiers on z/OS – Client Bindings

Figure 31 – JMS Selector and Correlid Performance on z/OS, client bindings.

Maximum Sustainable throughput of 1k messages using thread id as a selector/correlid, for 1 to 10 threads, single queue.

4.5.2 JMS Selectors and Correlation Identifiers, z/OS - Local Bindings.

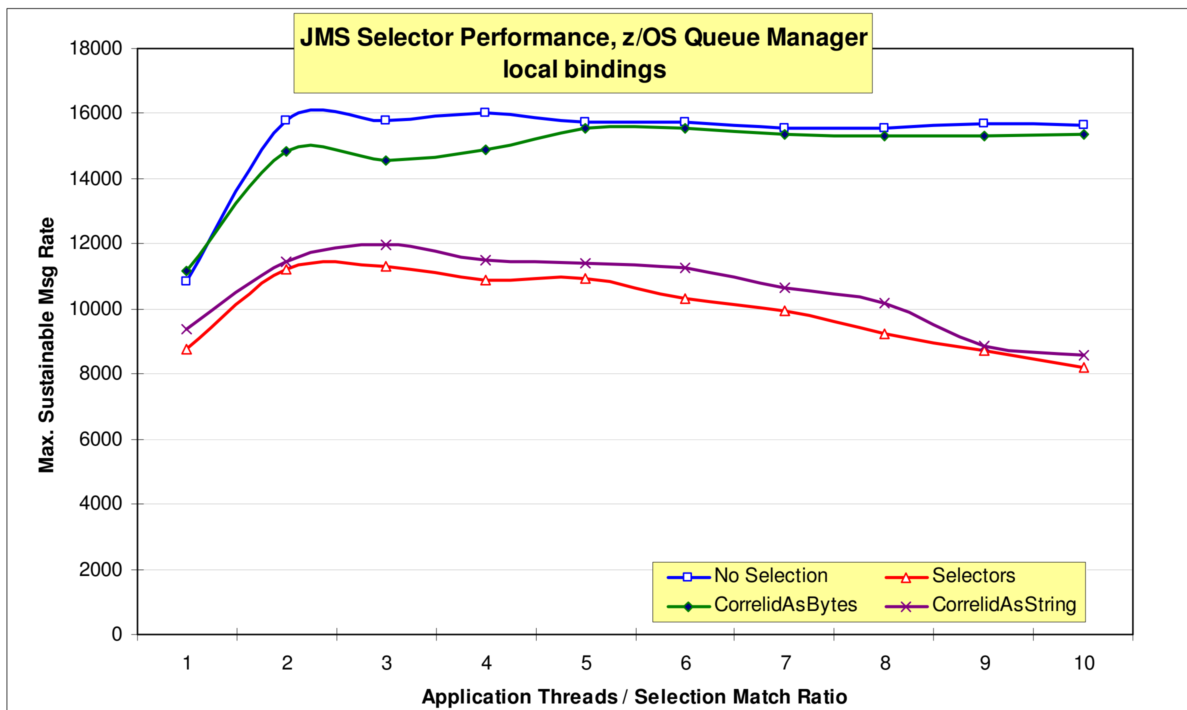


Figure 32 – JMS Selector and Correlid Performance on z/OS, Local Bindings.

Maximum Sustainable throughput of 1k messages using thread id as a selector/correlid, for 1 to 10 threads, single queue.

The two figures above illustrate the relative performance of a typical JMS application when using the various selection mechanisms available with a z/OS Queue Manager. This highlights the performance benefits of using the optimized CorrelidAsBytes. See section "Use of correlation identifiers" on page 43 for details of how to use this.

5 Message Driven Beans

5.1 Overview

A Message Driven Bean (MDB) is a relatively simple message consumer application which runs within an EJB container such as Websphere Application Server. The container manages issues such as transactional integrity, security and concurrency, making MDBs easier to develop than stand-alone JMS applications.
The MDB consumes messages asynchronously, being activated when a new message arrives.
Typically a messaging provider such as WMQ connects to the EJB container using a J2EE Connector Architecture (JCA) resource adapter. There measurements used ASF mode and WAS V6.

5.2 Scalability Improvements in WMQ V7.0

5.2.1 Queue with message rate gradually increasing

In a typical scenario we have a message producer writing messages at a steady rate to a single MQ queue, and an MDB consuming the messages from this queue. The MDB is running within a Websphere Application Server. The producer, queue manager and MDB/Application Server are running on separate machines connected by a network as shown in Figure 33.

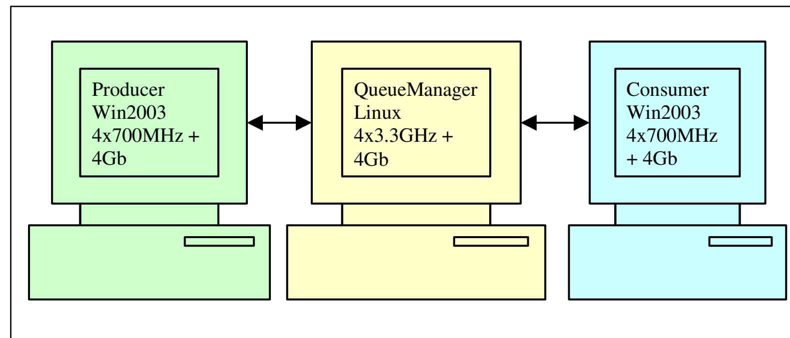


Figure 33 – MDB Scenario, Single Consumer

In a situation where the rate of message production exceeds the capacity of the MDB to consume, then queue depths can grow to provide a short term solution. If the situation persists however, the queue will fill up and the system will be unable to continue to process messages.

One solution is to add additional MDB/Application Server instances to consume messages from the same queue, as shown in Figure 34.

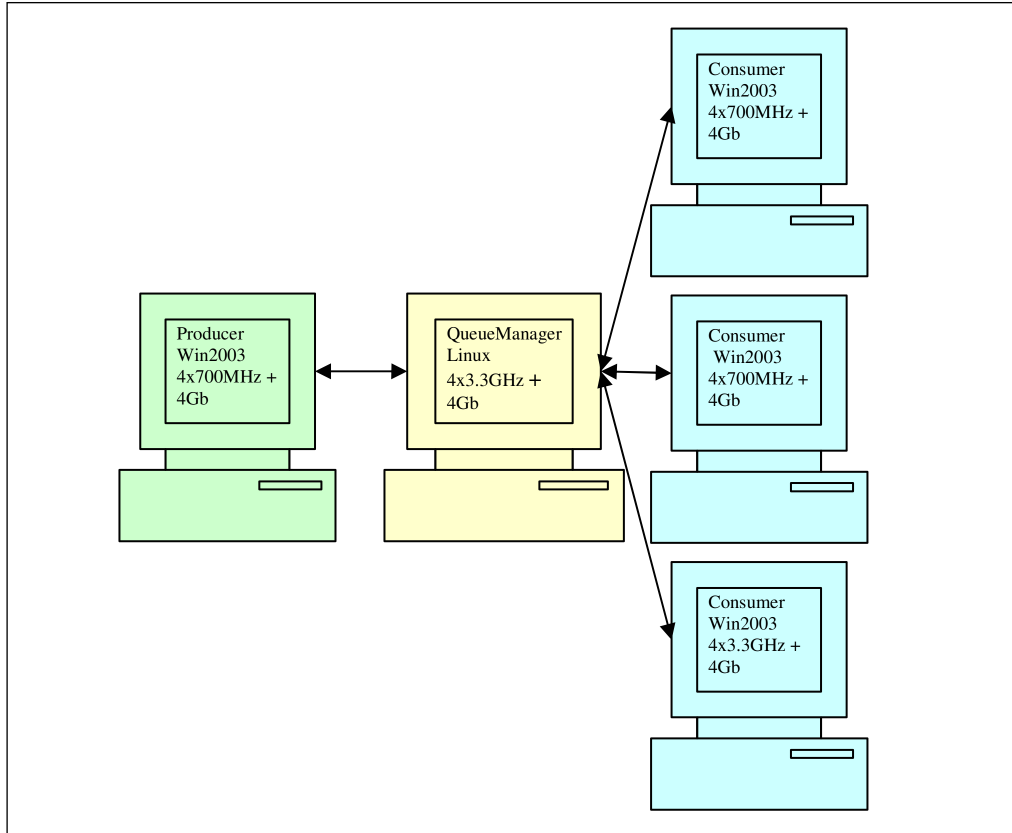


Figure 34 – MDB Scenario, Multiple Consumers

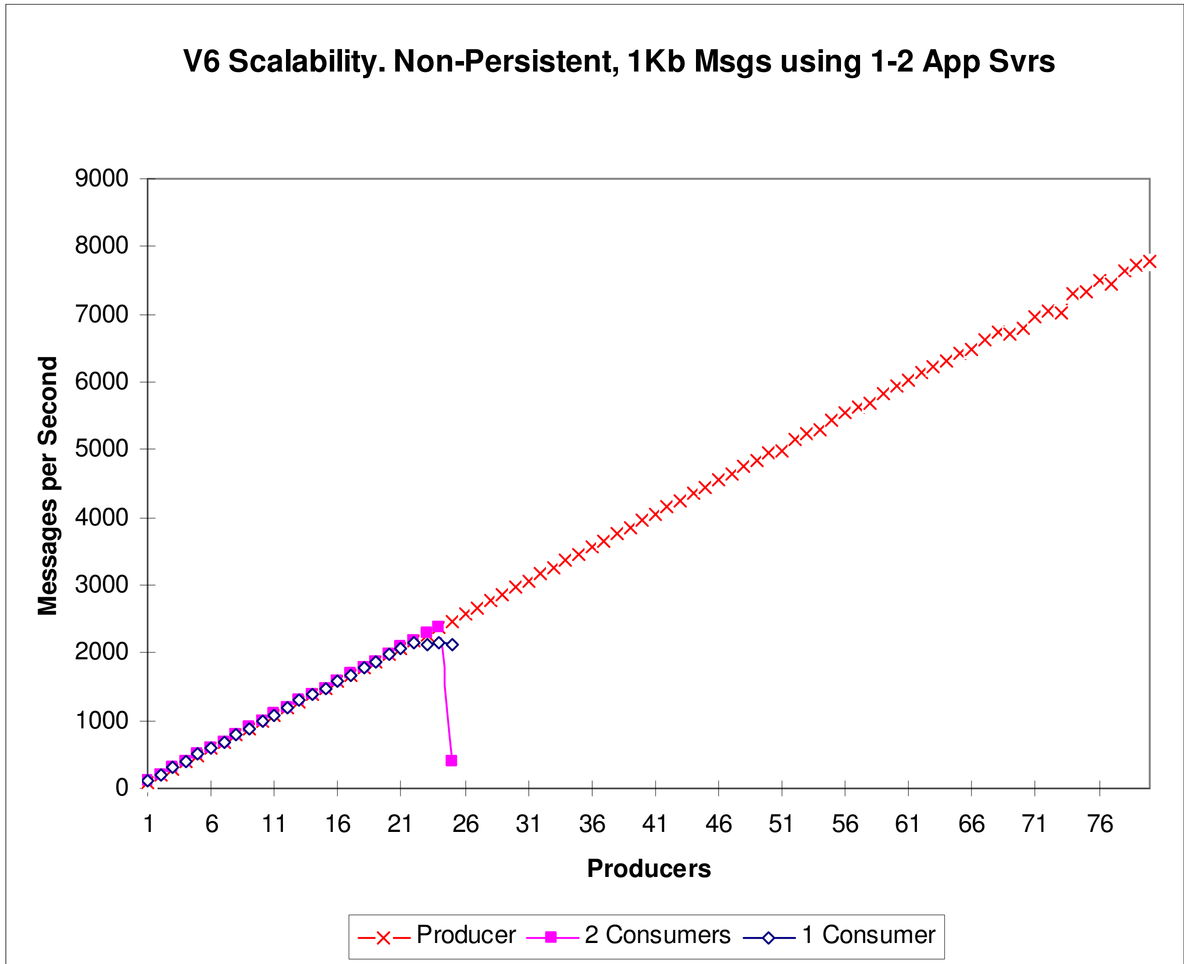


Figure 35 – MDB Scalability and Performance WMQ V6

WMQ V6 has some architectural limitations which prevent this approach from solving the problem. Adding more consumers can have little or no effect on total message throughput rates.
This is a known issue with V6 and in fact additional consumers can cause contention within the queue manager and a drop-off in performance when message production exceeds the maximum capacity of the consumers. See **Figure 35**.

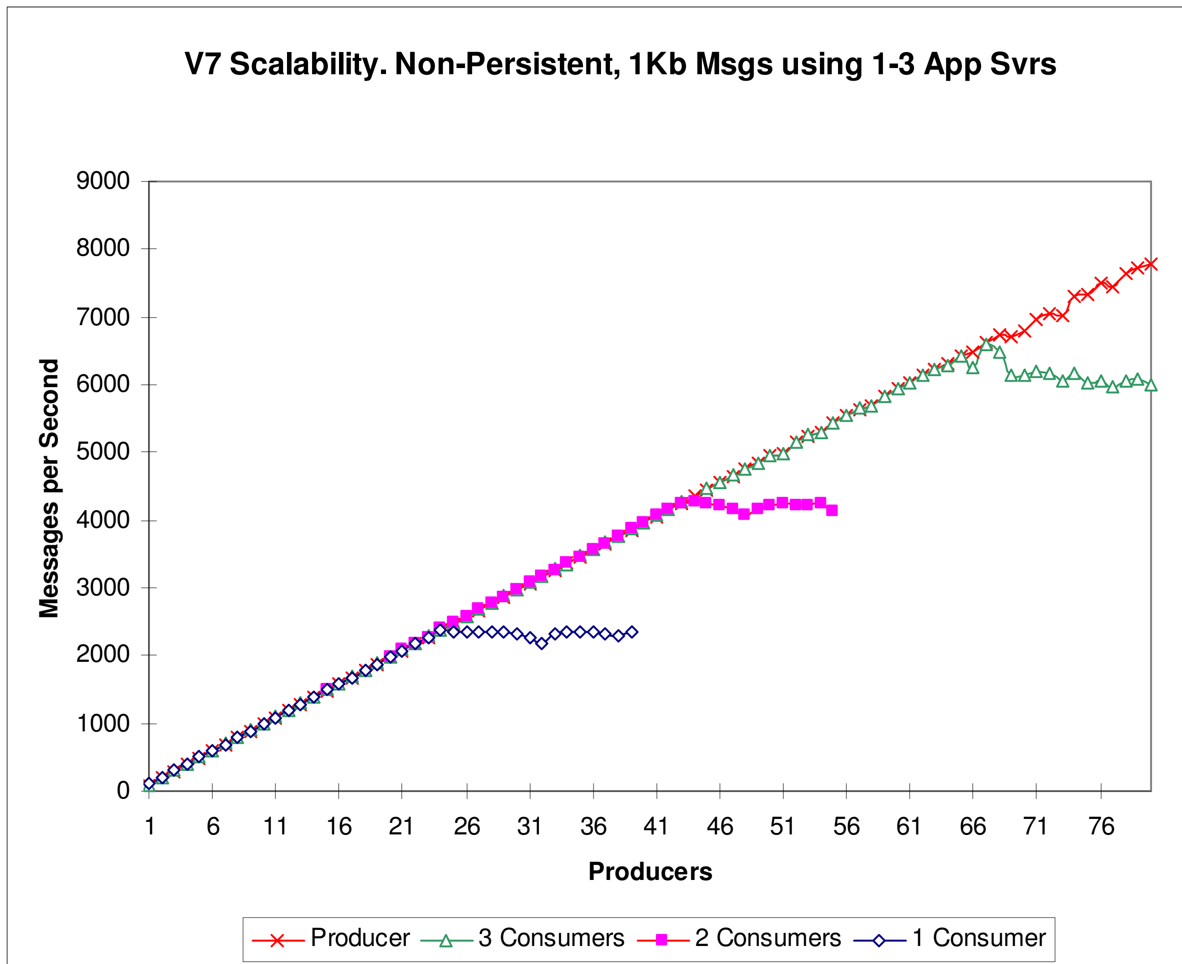


Figure 36 – MDB Scalability and Performance WMQ V7

The architectural limitations in the queue manager have been removed in WMQ V7 and total throughput now increases in a linear way even with 3 consumers as shown in **Figure 36**. Each consumer is on a separate WAS. Note also that once the maximum capacity has been exceeded, the MDBs continue to process messages at almost the same rate as the maximum which improves the systems ability to tolerate occasional peaks in message production. The benchmark caused each message producer to insert 10 messages per second. Thus 20 producers inserted 400 messages per second. By observing the queue depth, the message processing ability of WAS servers can be calculated. The bottleneck is the capacity of the individual WAS system since the QM system was lightly loaded.

5.2.2 Queue with maximum sustainable message rate manager

WAS 7.0.0.5 with both WMQ 6.0.2.7 and WMQ 7.0.1.0 (plus APAR IC63705). WAS with listener ports. Hardware specification.

- JMS Client

x365 4 x 3.0 GHz CPU, hyperthreaded, 4 MB L3 cache, 4.0 GB RAM

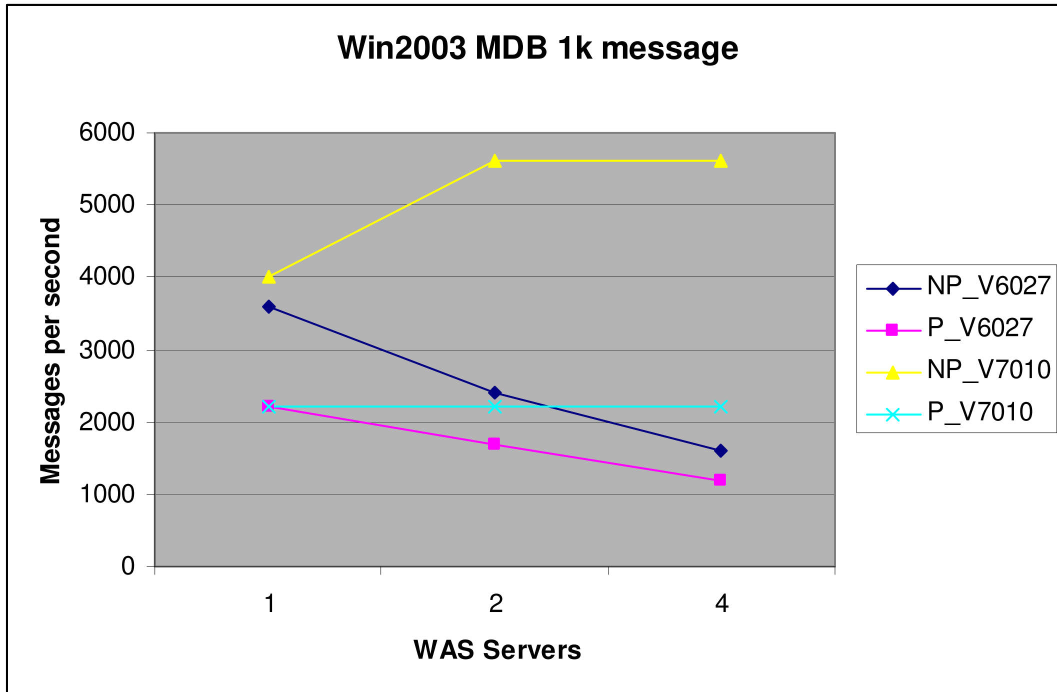
- WAS Server

x366 4 x 3.67 GHz, hyperthreaded, 0 MB L3 cache, 3.25 GB RAM WAS JVM Heap 256M

- WMQ Server

x365 4 x 2.8 GHz CPU, non-hyperthreaded, 2 MB L3 cache, 3.5 GB RAM

Multiple WAS servers were all on the same physical machine. Monitoring free memory, paging and CPU utilisation showed this was not a problem. The throughput is the maximum steady state rate that can be achieved with the varying number of WAS instances.



For non-persistent messages: (up to the limit of the systems used in these tests)

- WMQ 6 message throughput degrades as more WAS connections are made.
- WMQ 7 message throughput improves as more WAS connections are made.

For persistent messages: (up to the limit of the systems used in my tests)

- WMQ 6 message throughput degrades as more WAS connections are made.
- WMQ 7 message throughput is constant, irrespective of the number of connections

Monitoring & Statistics were turned on for the Queue and AMQSMON showed that with one WAS with MQ V6027 and all measurements with MQ 7010, there were zero GetFailCount queue statistic. With two WAS, there was a 46% failure rate and four WAS suffered a 73% failure rate for the GetFailCount queue statistic. The improvement to the GetFailCount queue statistic is caused by 'Browse and mark' in WMQ 7 and is the principle reason for the throughput improvement in V7. With MQ Version 6, a Browse was executed independently for each WAS instance followed by a destructive Get which could lead to multiple WAS instances issuing Get for the same message. Browse and mark prevent subsequent Browse from seeing the messages already earmarked for a different WAS.

6 Other Performance Enhancements in V7.0

6.1 JMS Selectors

A JMS message selector allows a client to filter the messages that it is interested in by using the message header. Messages with headers that match the selector are delivered to the application. In the model for point-to-point JMS in WebSphere MQ V7.0 this matching is now done in the server (Queue Manager) rather than the client which reduces message flows and hence can significantly improve performance. See **Figure 37**. Nonetheless, message selectors by their nature add overhead to the codepath and care should be taken to minimize their use.

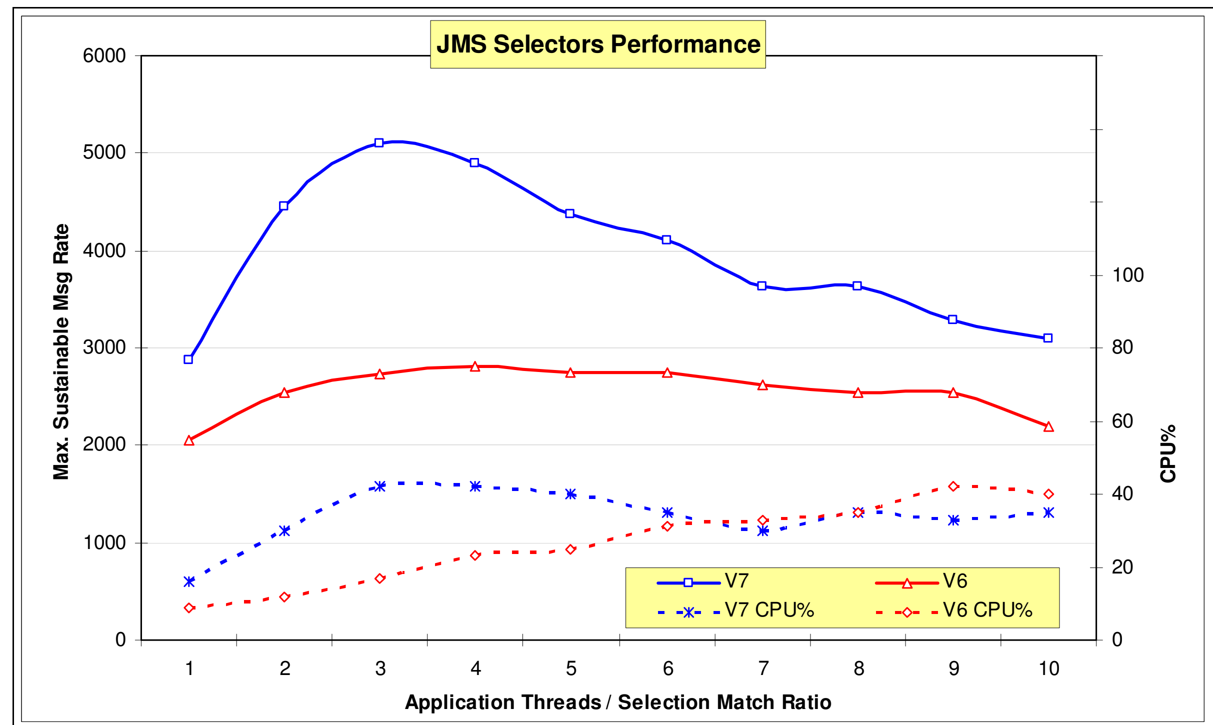


Figure 37 – JMS Selector Performance. Cpu figures are for QM only.

Maximum Sustainable throughput of 2k messages using thread id as a selector, for 1 to 10 threads, single queue.

6.1.1 Recommendations

- Disjoint selectors, i.e. multiple selectors which match on entirely separate sets of messages, can also be implemented by using multiple queues, topics or perhaps *correlationId*.
- When selectors *are* used, every effort should be made to minimise their complexity and also to make them fail as quickly as possible when a message does not match.

  Example:
  "height=183 AND gender=male" will fail faster than "gender=male AND height=183" since we can assume that in the selection domain there are fewer people with that exact height than there are males.

- CorrelationId should still be used in preference to Selectors where possible. This is because the correlationid field maps directly to the internal WMQ Queue Manager's representation of the message.
- For best performance try to keep queue depths down when using Selectors. Because selection is now done in the Queue Manager the additional workload of managing a deep queue can degrade throughput.

6.2 Asynchronous Put and Read Ahead

For some Non-Persistent Non-Transactional message workloads, the new Asynchronous Put and Read Ahead modes in V7.0 can offer significant response time improvements in systems where network latency is high. Both of these enhancements offer potential improvements at a cost of a slightly lower quality of service.

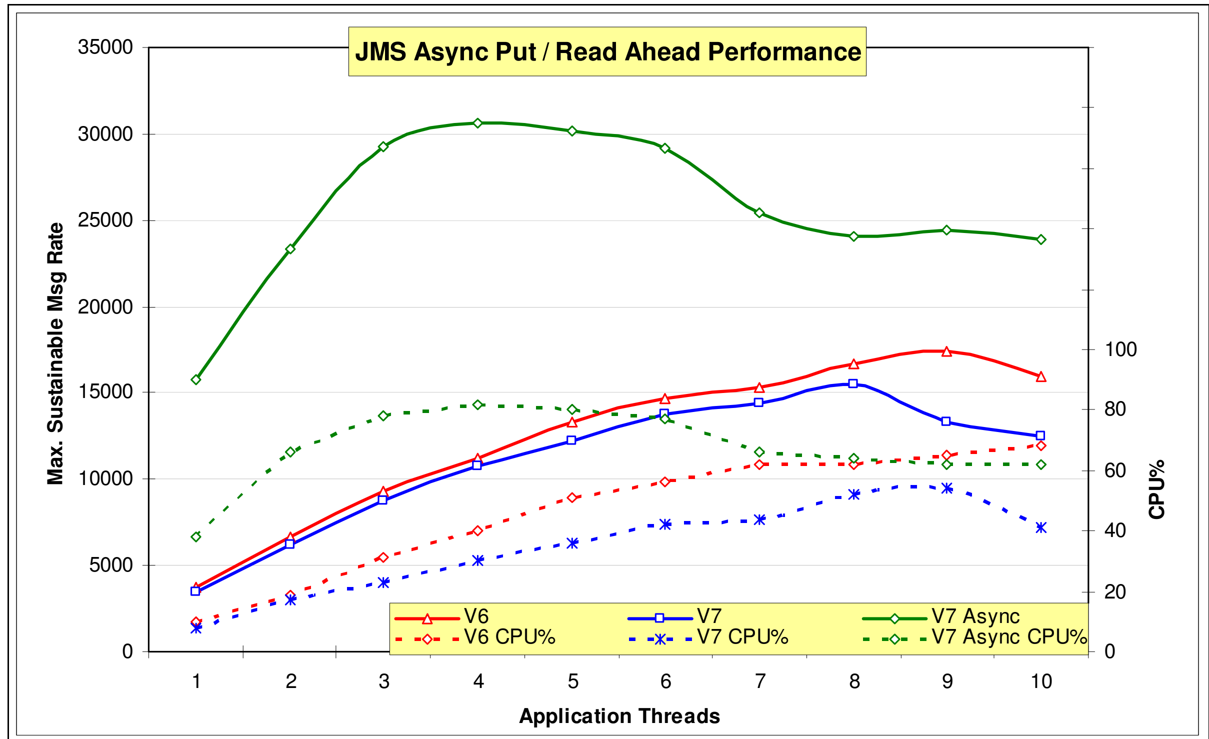


Figure 38 – New Asynchronous Put and Read Ahead features.

6.2.1 Asynchronous Put

Asynchronous Put provides considerable performance benefits to JMS client applications that transmit a sequence of messages in rapid succession but do not require immediate acknowledgement from the queue manager for every sent message. This is achieved by removing the requirement for the client to wait for a response from the Queue Manager to every message sent, thus enabling the client to continue processing in cases where it makes sense to do so.

6.2.2 Read Ahead

Similarly the ReadAhead enhancement sends messages to the client before being requested to do so, in anticipation that such a request will be made. The messages are queued locally and can be retrieved more quickly when the client is ready to do so.

The key considerations for using the read ahead feature in application designs are:

- If the JMS application using read ahead terminates abruptly before it consumes all the messages from the internal buffer, then any non-persistent messages currently stored in the buffer are discarded and the messages are lost.
- The read ahead feature is only applicable to non-persistent messages.
- Any existing JMS application can make use of the read ahead feature without any code modifications.
- If all the following conditions are true, messages sent to a queue in a session might not be received in the order in which they were sent:
  - An application uses two or more message consumers in the same session to consume the messages from the queue.
  - Each message consumer uses a different destination object for the WebSphere MQ queue.
  - Any or both of the destination objects are configured for read ahead.

Existing JMS applications can make use of the WebSphere MQ V7.0 asynchronous put and read ahead features without any modifications to the code.

6.2.3 Performance Notes

To achieve maximum message throughput the Async Send and ReadAhead features should be used together, and the rate at which messages are sent should be regulated to keep the queue depths to a minimum.

If left unregulated there is a risk that the receiver application will be unable to keep up with the sender and as a result queue depths will increase leading to inefficiencies in the Queue Manager and lower throughput.

6.3 Asynchronous Consume

Prior to WebSphere MQ V7.0, asynchronous message consumption was not natively supported by the Queue Manager. To perform asynchronous message consumption, the WebSphere MQ classes for JMS periodically polled the destination for suitable messages to arrive.
Both synchronous and asynchronous message consumption are now supported as native features in WebSphere MQ V7.0. An application that needs to consume a message asynchronously registers a callback function for a destination. When a suitable message is available at the destination, WebSphere MQ calls the function and passes the message as a parameter. The function can then process the message asynchronously.
From the JMS application design or programming perspective there are no changes for asynchronous message consumption. As previously, the JMS application registers an object implementing the JMS MessageListener interface to consume messages asynchronously.

In WebSphere MQ V7.0, the WebSphere MQ classes for JMS no longer polls a destination to check the availability of a message. As soon as a suitable message arrives at the destination the WebSphere MQ classes for JMS pass the message to the MessageListener callback function.
The advantages of using WebSphere MQ native asynchronous consume are summarized as follows:

- Improved performance of JMS message listeners, particularly when an application uses multiple message listeners in a session to monitor multiple destinations.
- Reduced CPU usage at both the JMS application and the WebSphere MQ queue manager.
- Fewer requests over TCP/IP and a reduction in network traffic between the classes and the queue manager.
- Message throughput is increased and the time taken to deliver a message to a message listener is reduced.

In the test scenario **Figure 39** a JMS client application registers message listeners for between 4 and 40 destinations, only 4 of which are actively receiving messages from a separate JMS client. The rate at which messages are arriving on the active destinations is regulated to avoid queue growth and hence to maximize total throughput.
In the V6.0 case, the overhead of polling the idle destinations causes performance to degrade as the number of message listeners increases. In contrast the V7.0 results show only a minor drop in performance even with 40 message listeners.

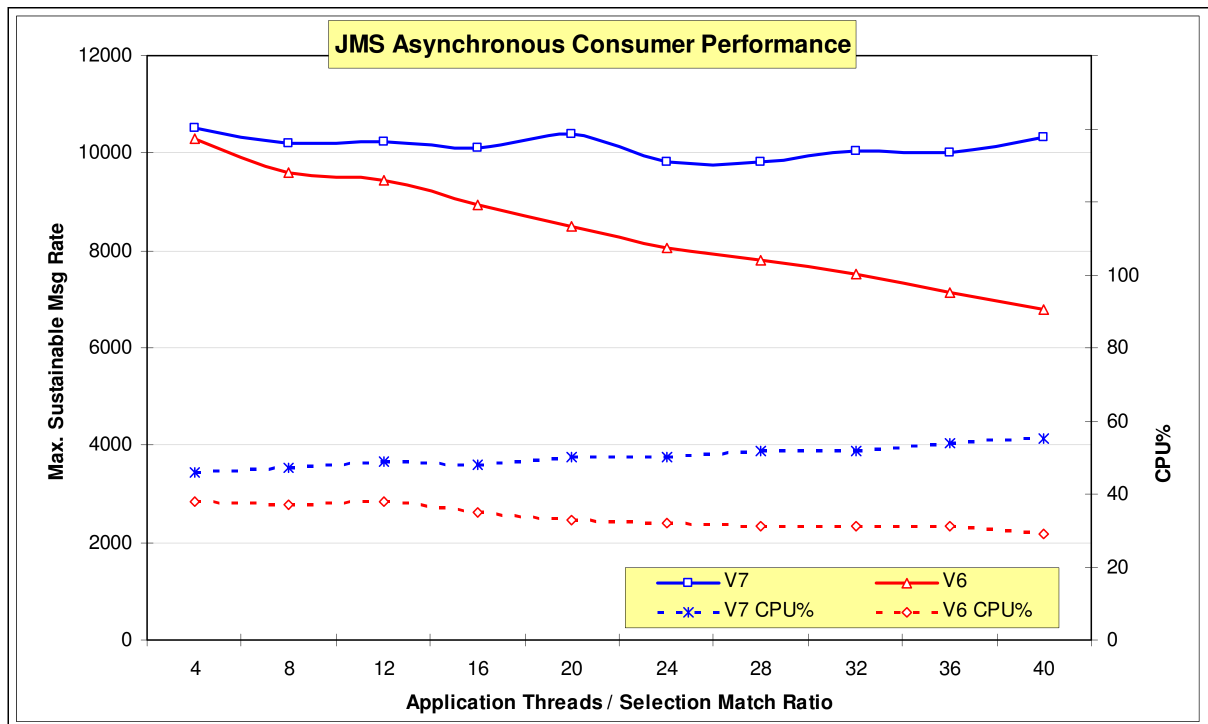


Figure 39 – Asynchronous Consumer Performance.

6.4 Multiplexed Sockets and Conversation Sharing

WebSphere MQ V7.0 introduces the ability for many threads in one JMS client program to connect to the same queue manager and share a running instance of a client channel. The client protocol conversations for the JMS calls made by each thread are transparently multiplexed over a single TCP/IP socket session.
Multi-threaded JMS client programs running on previous versions of WebSphere MQ use a separate channel instance for each thread. For a WebSphere MQ V7.0 JMS client connected to a WebSphere MQ V7.0 queue manager, the conversation sharing feature reduces the number of running TCP/IP sockets on the queue manager, allowing a greater number of concurrent client connections to be maintained.

WebSphere MQ V7.0 uses a full duplex protocol for JMS client connections when the conversation sharing parameter SHARECNV is greater than zero on both channels.

Full duplex means that information can be sent from either end of the session at any time.
Heartbeats are short flows of information that are sent at regular intervals to confirm that the session is still active.
Heartbeats can now be performed from both the client end and the queue manager end at a negotiated rate that is based on the HBINT parameter of both channels. This leads to earlier detection of communications network failures and other channel problems. The client program and the queue manager can now carry out recovery and reconnecting functions in a more timely manner, resulting in an overall improvement to the quality of service.
Previous versions of MQ use a half duplex protocol, where a heartbeat could only be performed from the queue manager end during a MQGET operation with a WaitInterval specified. This limited its usefulness for detecting problems.

For performance-critical applications the use of SHARECNV(1) is advised, unless a very large number of concurrent client connections is required. Setting SHARECNV(1) allocates a socket for each application thread which reduced the overhead of Multiplexing. The default value is SHARECNV(10) which allocates a single socket for up to 10 JMS client application threads.
Setting SHARECNV(0) disables Multiplexing and forces the JMS client into V6 compatibility mode which disables many of the V7 enhancements and is recommended only when connecting to a V6 Queue Manager.

6.5 Large Messages

WebSphere MQ V7.0.1 includes improvements to the internal processing of messages which enhance the steady-state throughput performance of larger messages in some scenarios.

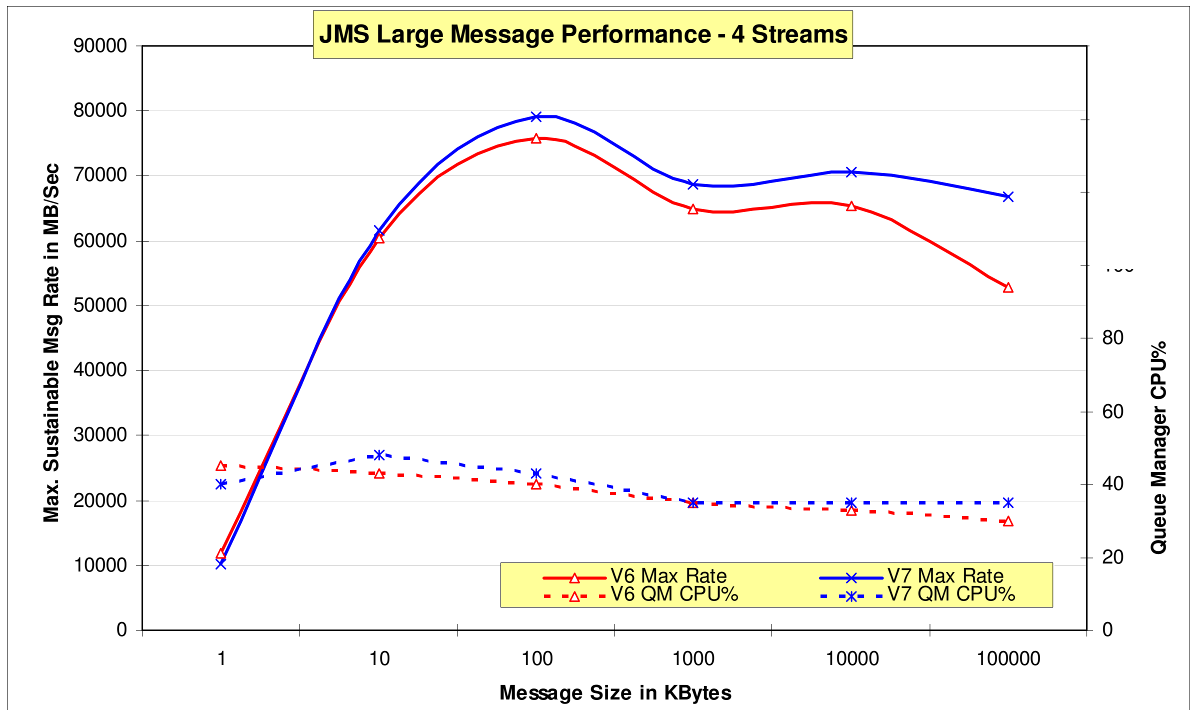


Figure 40 – Large Message Performance – 4 Streams.

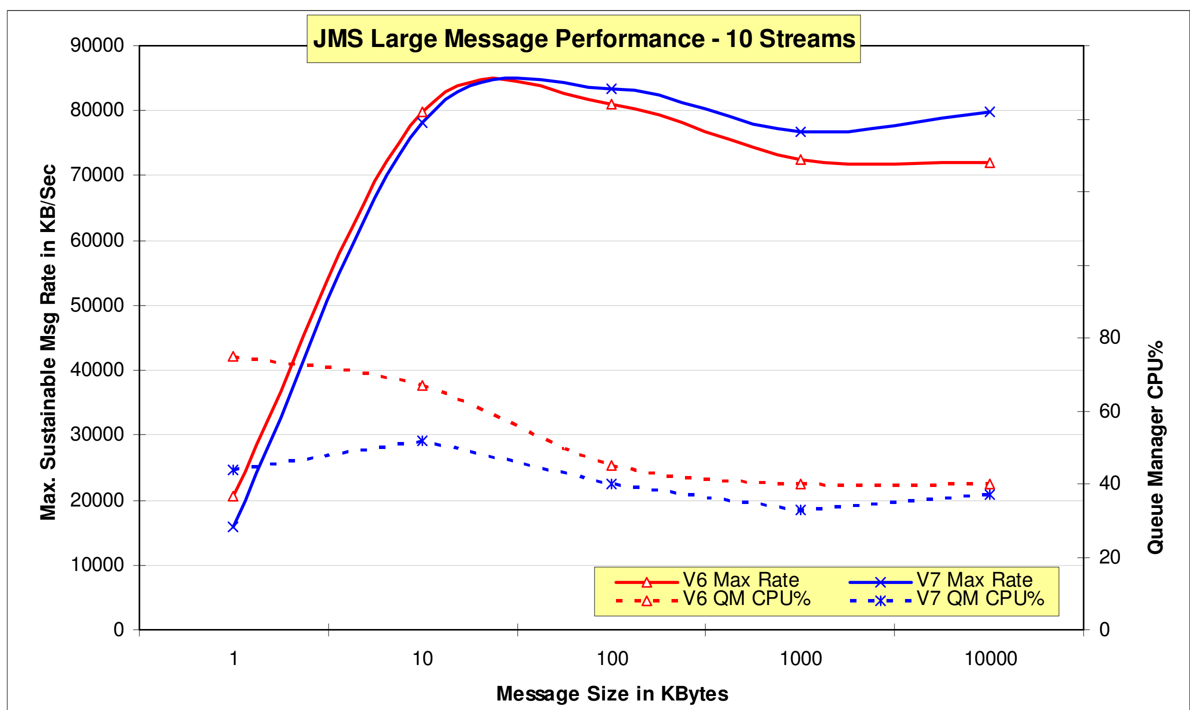


Figure 41 – Message Performance – 10 Streams.

Figures **Figure 40** and **Figure 41** show the maximum sustainable message rate relative to message size for a range of different message sizes from 1000 bytes to 100Mb.

Figures **Figure 40** uses 4 threads in each client and **Figure 41** uses 10.
In each case the application threads in the sender application send messages at a steady rate to the Queue Manager, and the application threads in the receiving application receive these messages. Each sender/receiver pair has a dedicated queue. The send rate of each thread is increased until the receiving threads are unable to keep up, leading to queuing within the Queue Manager.
The reported result at each message size is the maximum throughput achievable without queuing, multiplied by the message size to give a relative throughput rate.
The figures show that a message size of 100Kb is processed most efficiently, and also show the relative performance of WMQ V6 and V7.

It can be seen that the performance of V7 exceeds that of V6 for messages over 10Kb where 4 application threads are in use, and for messages over 100Kb where 10 application threads are in use.

7 Tuning/programming guidelines

7.1 Tuning the queue manager

Performance reports with tuning information for WebSphere MQ v7.0 on all supported operating systems can be found via. 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 Chapter 2 and 3 of 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=SingleWrite (suitable for write-cached disks)
- Channels / MQIBindType = FASTPATH (channels are an extension to QM address space)
- 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)

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.
- Do not leave the minimum heap size unset. If left unset, the heap may not be expanded. With a small (the default) minimum heap size, the *GC* may operate very frequently, reclaiming transient memory but not extending the heap (since we have already stated that most of the memory is released immediately after it is requested when messaging).
- Modern 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.

7.3 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 which can be useful when human interaction is necessary for each message. Benchmarks that produce or consume multiple hundreds of messages per second will bottleneck on the shared socket and should use SHARECNV=1 as explained in section “Multiplexed Sockets and Conversation Sharing” on page 40

7.4 Avoiding running in Migration/Compatibility Mode

MQ JMS 7.0 can connect to both V6 and V7 Queue Managers. When connected to a V6 Queue Manager a less optimised codepath is used, which facilitates migration from V6 to V7 but which should not be considered as a long term solution if performance is important.
It is also possible to connect V7 JMS to a V7 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 value should be used.

7.5 Use of correlation identifiers

- Selecting against *correlationId* or *messageId* follows an optimised path through WebSphere MQ 7.0 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.

7.6 JVM Warmup

- Modern 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 JVM 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. (e.g. –Xquickstart on IBM Java 5.0 jvms).
- 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.

7.7 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 for messages to specify whether messages are lost if the JMS provider fails. These are set by the producer via the *deliveryMode* property of javax.jms.MessageProducer#send(). The deliveryMode can be set on the send() call, but can also be set on the destination to which the messages are being sent, and in fact the example below does the latter.

The PERSISTENT delivery mode, which is the default, instructs the JMS provider to take extra care to ensure that a message is not 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. It is set via the acknowledgement Mode property of javax.jms.Session#createSession() on the consuming application.

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 16 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

The JMS applications used in chapters 2, 3 and 4 this report to generate the performance data are:

- Co-located on the server. (Local Measurements)
- Located on Linux clients communicating with the server. (Client measurements)

8.1 Linux, Windows and AIX

The Windows/Linux server machine is IBM eServer x360 with 4 * 2.8GHz P4 Xeon processors and 8GB RAM. The AIX server is IBM P570 with 1 dual core CPU (4.2GHz Power 6)

This means that the Linux and Windows throughput can be compared because they use similar hardware but AIX is using faster hardware.

The server operating system is

- Windows 2003 with MQ Log and Queues on 3 * IBM SSA 15,000 RPM drives
- Linux Redhat 3.4.6 (kernel 2.6.9) with MQ Log and Queues on SAN disks on DS6000
- AIX 6.1 with MQ Log and Queues on SAN disks on DS6000

Clients are hosted by four driver machines IBM Server x3850 with 4*3.3GHz Xeon are connected by 1Gb Ethernet LAN which is designed to ensure the server is the bottleneck. 64 bit RHEL

8.1.1 SAN disk subsystem

The MQ SAN consists of a pair of 2026 model 432 (McDATA ES-4700) switches running at 4Gb/s with 32 ports each. They are connected together via two inter-switch links to form a single SAN fabric.

The MQ hosts attach via this SAN to a DS6800 disk array (1750 model 511) with one expansion drawer.

Each drawer (controller + expansion) contains 16 x 73Gb 15K fibre channel disk drives, so there are a total of 32 physical drives.

The 32 drives are configured as four RAID-5 arrays, each of which is 6+Parity+Spare (the number of spares is defined by the configuration of the DS6800).

The controller has an effective cache size of 2.6Gb plus 0.3Gb of NVS

8.2 zOS

The hardware configuration is:
• CPU: 3-CPU logical partition (LPAR) of a zSeries 990 (2084-332).
CPUs are defined as floating but there are always 3 physical CPUs available.
Its capacity is similar to that of a 2084-303.
• DASD: FICON-connected Enterprise Storage Server (ESS) Model F20.

Software levels are:
• z/OS 1.9
• WebSphere MQ v7
• Java 1.5

8.3 MDB Test Configuration

The MDB Test configuration used in chapter 5 of this report consists of one Windows client (message producer), one Linux server (Queue Manager) machine and 3 Windows clients each running a single MDB message consumer within a Websphere Application Server.

The Linux server machine is an IBM eServer x3850 with 4 * 3.3GHz P4 Xeon processors and 4GB RAM.

The Windows client machines are:

- (Message Producer) IBM NetFinity 8500R with 4 * 700MHz P4 Xeon processors and 4GB RAM.
- (Message Consumer1) IBM eServer x3850 with 4 * 3.3GHz P4 Xeon processors and 4GB RAM.
- (Message Consumer2) NetFinity 8500R with 4 * 700MHz P4 Xeon processors and 4GB RAM.
- (Message Consumer3) NetFinity 8500R with 4 * 700MHz P4 Xeon processors and 4GB RAM.

The server operating system is

- Red Hat Enterprise Linux AS release 4 (Nahant Update 5) with MQ Log and Queues on SAN disks on DS6000

The client operating systems are

- Windows 2003 Server
- Windows 2003 R2 Enterprise x64 Edition SP1

The Websphere Application Server version is v6.1.0.19

All machines are connected by a 1Gb Ethernet LAN.

8.4 Other V7 Enhancements

The JMS applications used in chapter 6 of this report to generate performance data for the new V7 enhancements are located on Windows and Linux clients communicating with a Linux server.

The Linux server machine is an IBM eServer x366 with 4 * 3.66GHz P4 Xeon processors and 4GB RAM.

The Windows client machine is an IBM eServer x366 with 4 * 3.66GHz P4 Xeon processors and 4GB RAM.

The Linux client machine is an IBM eServer x3850 with 4 * 3.33GHz P4 Xeon processors and 4GB RAM.

The server operating system is

- Red Hat Enterprise Linux AS release 4 (Nahant Update 5) with MQ Log and Queues on SAN disks on DS6000

The client operating systems are

- Windows 2003 R2 Enterprise x64 Edition SP1
- Red Hat Enterprise Linux AS release 4 (Nahant Update 5)

Clients are connected to the server by 1Gb Ethernet LAN.

For performance reasons the tests in this section are run with the conversation sharing feature of WMQV7 disabled. See section “Multiplexed Sockets and Conversation Sharing” on page 40 for details on how to do this.