A PERFORMANCE COMPARISON OF IBM MQSERIES 5.1 AND MICROSOFT MESSAGE QUEUE (MSMQ) 2.0 ON WINDOWS 2000

TRANSACTIONAL AND PERSISTENT DELIVERY MODES

(including a critique of Microsoft's recent NSTL performance report)

IBM SWG Competitive Technical Assessment

June 2000

ABSTRACT

In this paper, we present a performance comparison of the two major message queuing software products for Windows operating systems on the market today: ***IBM MQSeries 5.1*** *and* ***Microsoft Message Queue (MSMQ) 2.0****, using the transactional and persistent delivery modes. (This paper is a companion to the previously published paper* ***"A PERFORMANCE COMPARISON OF IBM MQSERIES 5.1 AND MICROSOFT MESSAGE QUEUE (MSMQ) 2.0 ON WINDOWS 2000: EXPRESS DELIVERY MODE"****.) The messages per second and throughputs for both running on the* ***Windows 2000*** *operating system were compared under a variety of conditions (different message sizes, number of clients and sessions per client, number of preloaded messages in the server response queue). Using a test setup and procedures that stressed the entire client-server response path (and were therefore representative of an actual customer environment), MQSeries significantly outperformed MSMQ.*

ITIRC KEY WORDS

Performance

Benchmarking

MQSeries

Message Queuing

MSMQ

Windows 2000

Persistent

Transaction

CONTENTS

INTRODUCTION

Message queuing performance is essential to any e-business solution

Message queuing is a strategic component of any e-business solution. By providing an encapsulated, asynchronous means by which different programs can communicate, it provides the distributed benefits of remote procedure calls and traditional transaction processing, without many of their inherent problems [1]. Robustness is introduced by encapsulating remote procedure calls and other requests into asynchronous, independent messages that are then handled by a designated queue manager. The queue manager oversees the course of those messages as they travel between different machines across a network, or between different programs within the same machine. The client application and/or machine originating the message can move on to processing other tasks as it awaits the message's response. Meanwhile, the message queuing application can manage and respond to a wide variety of networking or other problems that might otherwise hang up or crash the original requesting program.

Message queuing therefore allows an e-business application designer to (a) increase program efficiency and system capacity by utilizing multiple servers to execute e-business applications, and (b) create e-business solutions that require a number of (otherwise) incompatible programs. In the latter case, through their use of messages and message brokers, message queuing solutions allow an e-business application programmer to write separate translation or other bridging programs (that might run on separate servers) between two otherwise incompatible programs. This allows the programmer to utilize the broadest possible array of advanced application software in his/her final application, in a timely and easily manageable manner.

To provide this capability, message queuing clients and servers must be able to process a large number of messages per unit time, for the full range of possible message sizes. A relatively large single client/server capacity to process messages implies an efficient processor execution path, which in turn implies less processor utilization consumed by the message queuing application software. This translates to better overall performance of simultaneously executing applications, which are presumably the important end user applications (the "ends") that a message queuing solution should be facilitating (as the "means"). Likewise, a higher server capacity implies greater system scalability, i.e., the ability to support increasingly larger numbers of clients without an exorbitantly expensive increase in supporting resources.

Furthermore, any message queuing solution should provide robust performance in the face of server congestion, especially due to the accumulation of unretrieved messages. In a real customer environment, clients may need to share request/response queues, so that messages will naturally accumulate in a server's message queues over time. Some of these messages may remain unretrieved for some extended period of time. Any inefficiencies in the ability of a client to locate its own message(s) within its response queue is potentially detrimental to a solution's ability to scale. If the performance impact is severe, the affected message queuing solution should be deemed unacceptable for use in any e-business solution.

IBM[2] MQSeries[2] and **Microsoft[1] Message Queue (MSMQ[1])**, two major message queuing software products for the **Windows[1] 2000** operating system on the market today, provide support for three different modes of message delivery: express, persistent and transactional. The express delivery mode provides the best performance of all three modes (in terms of messages per second and/or throughput), and was therefore extensively studied in [2]. However, it does not guarantee that messages can be recovered following system failure (e.g., when the server containing the queue manager suddenly crashes or is shut off. This guarantee is, however, provided by the persistent (or recoverable) delivery mode. Since this mode of operation often requires that the server copy a message to disk, it typically yields lower performance numbers than those obtained using the express delivery mode. The transactional delivery mode allows a user to bundle message queuing operations inside a transaction. A transaction represents a defined unit of work, whereby every operation inside the transaction must execute successfully for

the transaction as a whole to be deemed successful (at which point the transaction is *committe*d). If any single operation inside the transaction fails, then all the other operations that did execute successfully are reversed, and the system is rolled back to its state prior to the start of the unsuccessful transaction. Because of the additional overhead entailed in guaranteeing that all operations inside a transaction execute successfully, the performance of the transactional delivery mode is, once again, usually significantly less than that of the express delivery mode.

Both the transactional and persistent delivery modes are therefore very important and commonly used in a wide variety of e-business applications, and most especially in those involving the use of databases (e.g., banking, reservations, inventory, etc.). Therefore, both modes of message delivery need to provide excellent performance and scalability under realistic testing conditions, if they are to be deemed usable in an actual customer environment.

Therefore, in order to provide an e-business solution designer with meaningful consumer data so that he or she can make an informed choice, in this paper, we compared the performance of MQSeries and MSMQ using the transactional and persistent (recoverable) delivery modes. The two were compared under a variety of conditions and system parameters, including different message sizes, number of clients, number of threads per client and number of preloaded messages (in the server response queue). The TCP/IP (Transmission Control Protocol / Internet Protocol) was used throughout, since it is the prevalent communications protocol used within most customer networks, as well as over the Internet. (As mentioned above, the express delivery mode was examined previously in [2].)

For both applications, great care was taken to individually tune the systems in accordance with performance recommendations made by IBM [3], [4] and Microsoft [5], [6]. However, in this particular study, we examined the performance of a system using a test scenario that stressed the entire client-server response path. Such a test scenario is therefore more indicative of message queuing performance in an actual customer environment (in contrast to test scenarios specifically manufactured to generate unrealistically large benchmark statistics; see, e.g., [5] [7]). The test setup and procedures are carefully described in the section entitled **Test Configuration and Procedures**.

Executive Summary of the Results

As the data will show:

(1) For the transactional delivery mode, IBM MQSeries significantly outperformed MSMQ for every combination of system parameters examined (i.e., for every combination of number of clients, number of threads per client and message size).

As Table 1 below shows, MQSeries' transactional delivery mode performance was over 5 times better than that provided by MSMQ for the most frequently occurring (and therefore most important) small message sizes. This MQSeries performance superiority was in fact maintained across all message sizes. These performance numbers were collected using the parameter combination that maximized MSMQ performance for the important message size of 1024 bytes (i.e., 4 clients running 3 threads each; see Figures 2 and 3 below). However, regardless of the parameter combination chosen (i.e., for every combination of number of clients, number of threads per client and message size examined), the results were always the same: MQSeries' transactional delivery mode performance was substantially greater than that of MSMQ.

For all these tests, both client and server were issuing, respectively, their Put (i.e., put a message in the server request queue) and Get/Put (i.e., get a message from the server request queue, and then put it in the server response queue) inside syncpoint, i.e., these commands were issued inside a transaction, controlled by a final *commit* (see the more detailed discussion below). This was the only way in which the MSMQ transactional delivery mode would operate. MQSeries provides greater flexibility, in that the client and server message queuing commands could have been defined to operate inside or outside of syncpoint independent of one another.

Table 1. Transactional Performance

Message Size (bytes)	Throughput (kilobytes per second)		Ratio of MQSeries to MSMQ throughput
	IBM MQSeries	MSMQ	MQSeries/MSMQ
100	4.8	0.9	5.6
512	23.2	4.3	5.4
1,024	46.4	8.8	5.3
2,048	83.4	16.4	5.1
3,072	116.3	24.2	4.8
4,096	142	32.6	4.4
5,120	182.8	41.2	4.4
10,240	297.6	80.4	3.7
20,480	463.3	162	2.9
30,720	579.8	247.8	2.3
40,960	625.7	336.2	1.9
65,536	705	530.8	1.3

Table 1. MQSeries and MSMQ transactional performance (measured in kilobytes per second) for 100 through 65,536 byte message sizes.

MQSeries transactional performance significantly exceeded that of MSMQ for every situation examined. For the table above, the parameter combination that maximized MSMQ performance at a 1024 byte message size (i.e., 4 clients, 3 threads per client) was used.

(2) For the persistent delivery mode operating outside syncpoint on the server (the only option available to MSMQ), IBM MQSeries significantly outperformed MSMQ for every combination of parameters examined (i.e., number of clients, number of threads per client, and initial response queue size). Furthermore, even when IBM MQSeries was at a performance disadvantage, operating *inside syncpoint* on the server, its persistent delivery mode still outperformed MSMQ operating *outside syncpoint*, for nearly every combination of parameters examined.

As Table 2 below shows, for a 1024 byte message, MQSeries' persistent overall peak performance (outside syncpoint) of 452.7 messages per second (mps) was 5.4 times (or 441%) better than that provided by MSMQ (at 83.6 mps). However, since both these peaks were achieved when server congestion was light, and little data was being stored on disk to guarantee recoverability (yielding numbers comparable to those obtained using the express delivery mode; see [2]), a more meaningful performance comparison was obtained using the "steady state" average mps. This metric averaged performance over those values collected when the system had settled down into the relatively stable, flat portion of its mps vs. threads per client curve (following the very large initial peak values described above; see Figures 7 and 8 below). It is over this portion of the curve that messages were being consistently stored on disk to guarantee recoverability.

For the persistent delivery mode operating outside syncpoint on the server, MSMQ achieved its highest steady state average mps for 5 clients, yielding 39.9 mps. For MQSeries running outside syncpoint, the corresponding steady state average of 74 mps (for 5 clients) was 85% better than that provided by MSMQ. More surprisingly, for MQSeries running *inside* syncpoint, the corresponding 5-client average of 43 mps was once again better than that of MSMQ running *outside* syncpoint. The latter represents a significant performance advantage provided by MQSeries, since running inside syncpoint necessarily involved more overhead to encapsulate the server commands inside a transaction, providing enhanced system integrity at the expense of overall performance.

(Note: Since MQSeries persistent delivery mode running inside syncpoint always stored messages to disk, there were no data points yielding express-like peak performance numbers that would allow for any meaningful and fair peak performance comparisons.)

Table 2. Persistent Performance, Inside and Outside Syncpoint (sp) (1,024 Byte Message Size)

Number of Clients	Peak Messages per Second (mps)		Steady State Average Messages per Second (mps)		
	MQSeries	MSMQ	IBM MQSeries		MSMQ
	outside sp	outside sp	inside sp	outside sp	outside sp
1	452.7	31.2	44.7	74.6	30.6
2	75.1	57.7	42.9	74.1	37.3
3	75.2	75.1	44	74.4	37.8
4	74.5	83.6	45.6	72.7	38.3
5	75.5	50.8	43	74	39.9
6	74.7	52.6	43.3	72.6	39.4
7	75.5	51.9	43.6	72.5	38.6
8	74.1	51.9	44.5	70.2	37.7

Table 2. MQSeries and MSMQ persistent peak and "steady state" average performance (measured in messages per second, or mps) as a function of the number of clients simultaneously transmitting to the server, for a 1024 byte message size. sp = syncpoint

MQSeries persistent performance outside syncpoint significantly exceeded that of MSMQ outside syncpoint for every situation examined (number of clients and threads per client). More importantly, MQSeries persistent performance *inside* syncpoint likewise exceeded that of MSMQ *outside* syncpoint for almost every situation examined, a surprising result given that operating inside syncpoint is more costly than operating outside it.

The overall peak mps for a given number of clients was determined as the maximum over all number of threads per client examined. This peak typically occurred at a small number of threads per client.

The steady state average mps was obtained by averaging performance over those values collected when the system had settled down into the relatively stable, flat portion of its mps vs. threads per client curve (following the very large initial peak values listed above; see Figures 7 and 8 below). It was over this portion of the curve that messages were being consistently stored on disk to guarantee recoverability.

(3) For persistent delivery mode, IBM MQSeries performance was extremely robust as the server request queue was preloaded with an increasing number of initial messages. In sharp contrast, MSMQ performance collapsed as the initial request queue size increased.

Preloading the server request queue mimicked the expected conditions that would arise within a real customer environment, as this queue would grow over time with increasing customer requests. Performance results yielded by this test therefore determined the ability of a message queuing solution to scale up with an increasing number of users.

As Table 3 below shows, for persistent delivery mode, MQSeries performance remained relatively unchanged as the initial request queue size (Q_0) was increased. In contrast, MSMQ performance was decreased by even a small number of preloaded messages. For Q_0 = 200, MSMQ performance dropped from its peak performance of 43.0 messages per second (mps) to 26.3 mps (a reduction of nearly 39%). At Q_0 = 1000 messages, MSMQ performance completely collapsed to 7.9 mps. In contrast, the corresponding MQSeries performance was 72.8 mps.

Table 3. Messages per Sec vs. Initial Request Queue Size (Persistent Delivery Mode)

Initial Request Queue Size (Q_0, in Messages)	Messages per second (mps)	
	IBM MQSeries	MSMQ
0	73	43
50	72.9	39.9
100	72.9	36.7
150	73	32.4
200	72.9	26.3
300	72.8	20.3
500	72.9	14.1
1,000	72.8	7.9

Table 3. MQSeries and MSMQ persistent performance (measured in messages per second) as the server response queue was preloaded with a fixed number of messages.

This test scenario mimicked expected conditions within an actual message queuing customer environment. MQSeries' performance advantage over MSMQ increased as the number of preloaded messages increased (tracking the collapse of MSMQ performance). 1024 byte messages were used throughout, with 4 clients running 5 threads each. This was the parameter combination that yielded near optimal MSMQ persistent performance, while guaranteeing that messages were truly being stored on the disk (as discussed above).

(Note: The performance of neither product was appreciably affected by server request queue preloading when transactional delivery mode was used.)

It should once again be emphasized that MQSeries and MSMQ were tested using the identical setup (given in Figure 1), using the same test procedures. This setup was tuned in accordance with the recommendations given by Microsoft in [5] and [6]. A subsequent report will examine transactional and persistent performance using Microsoft's recommended hard drive architecture in [5].

Published benchmark results avoid MSMQ's performance Achilles' heels

Microsoft has published unrealistically inflated MSMQ performance numbers [5] [7], using a test procedure and setup specifically designed to avoid the performance "Achilles' heels" demonstrated by Table 3 above. In those reports, a typical performance test consisted of measuring how fast a server could empty a preloaded message queue, unrealistically discarding all messages. The client was never required to search the server response queue to find a given return message. In the real world, such a return message would typically contain processed data or information requested by the client in the corresponding original message it initially sent to the server. Microsoft's test procedures therefore minimized the processing costs associated with both the client and server. As the data above shows (which will be amplified by the data presented below), the MSMQ queue searching algorithm performed extremely poorly. This document will therefore demonstrate that when a test scenario is used that realistically stresses the entire client-server response path, the introduction of these very real costs yields quite a different performance story from the one Microsoft would like to sell to consumers. **These performance shortfalls seriously compromise MSMQ's ability to scale, as customer environments grow to accommodate ever larger numbers of clients. Furthermore, peak single client and server throughputs obtained for MSMQ corresponded to the saturation of, respectively, the client and server processors (i.e., processor utilizations were 100%). Therefore, MSMQ's relatively lower server and client performance, with its associated costs to processor efficiency, will likely mean poorer performance of the other (e-business) applications that must simultaneously run on an MSMQ-enabled client or server.**

Throughout this paper, sufficient details regarding the test setup, measurement process and system parameters will be provided, so that the reader may easily replicate these tests. For a detailed comparison between the test setup and methods used in this study and those used by Microsoft in [5], please see our companion study of express delivery mode performance [2]. A detailed analysis of the methods used by Microsoft in its most recent (NSTL) report [7] are given at the end of this paper.

An extensive analysis of the effects of message size on express delivery mode performance is given in [2]. In this paper, we have reduced the number of message sizes examined in order to provide more detailed analyses of the effects on performance of message delivery mode, operation inside vs. outside syncpoint, and initial response queue size.

Note: The comparative information published in this document reflects laboratory tests undertaken at IBM's facility in Research Triangle Park, NC. Performance in individual cases may vary depending on customer environment, workload, and any unique characteristics of the products in the customer location. The versions of MQSeries and MSMQ examined were the very latest available from IBM and Microsoft respectively as of June 30, 2000.

MESSAGE QUEUING PERFORMANCE

TEST CONFIGURATION AND PROCEDURES

Figure 1 shows the configuration that was used to measure the performance of the IBM and Microsoft message queuing products. The test setup consisted of up to 8 IBM PC300GL client workstations (300 MHz Intel[3] Pentium[3] II processor; 32-bit, 33 MHz PCI bus), each with an IBM Etherjet[2] PCI (Peripheral Component Interconnect) adapter (100 Mbps Ethernet, full-duplex mode). The server was an IBM Netfinity[2] 7000 M10 (4 x 400 MHz Xeon[3] Pentium II processors; 32- and 64-bit, 33 MHz PCI buses; 1 GB RAM; 18.2 GB SCSI hard drive), with an IBM Netfinity Gigabit Ethernet SX adapter (1 Gbps, full-duplex, 64-bit). The operating system on all clients was Windows 2000 Professional (General Availability- or GA-level), while that on the server was Windows 2000 Advanced Server (GA-level). The clients and server were interconnected via an Intel Express[3] 510T Ethernet switch (using multimode optical fiber to the server, and twisted pair copper cabling to the clients).

The message queuing applications examined were MQSeries version 5.1 and MSMQ version 2.0 (the latter provided with the Windows 2000 operating system). Message queuing clients were configured as dependent. (Messages were not prestored on the client.) When multiple clients were examined, they shared a single request and response queue, managed by a single queue manager. The performance of this scenario was deemed extremely important in assessing the scalability of the two applications. Transactional (inside syncpoint) and persistent (inside and outside syncpoint) delivery mode performance is reported in this study. (In the companion paper [2], express delivery mode alone was examined. Please note that the setup and test procedures used in this study are the same those used in [2].) We will discuss the differences between express, persistent and transactional delivery modes, as well as the differences between, and various permutations of, inside and outside syncpoint, in the relevant sections that follow.

The communication protocol throughout was TCP/IP (Microsoft's version), using the default maximum frame and TCP window sizes.

All power management features (screen saver, etc.) were disabled, to preclude interruptions to the clients'/server's processors during testing. A 16-way Black Box[4] ServSwitch[4] was used to allow all client and server machines to be controlled using a single monitor, keyboard and mouse. Given the length of the tests, the ServSwitch was set to monitor a machine not involved in the testing during data collection, so that no test machine could be inadvertently interrupted through the mouse or keyboard.

IBM MQSeries performance was tuned in accordance with the recommendations given in [3] and [4]. As prescribed, MQIBindType was set to FASTPATH using the MQSeries Services interface. In addition, the MaxChannels and MaxActiveChannels were set to a value greater than the product of the largest number of clients (here, 8) times the largest number of threads per client examined (here, 15), again using MQSeries Services. For this study, we used MaxChannels = MaxActiveChannels = 950 (well above the maximum number of clients times threads per client). Applications performance optimization was enabled, since it yielded the best MQSeries performance numbers.

MSMQ performance was tuned in accordance with the recommendations given in [5] and [6]. As prescribed, auditing was deactivated. In addition, applications performance optimization was disabled and the size of the paging file was increased to the recommended size.

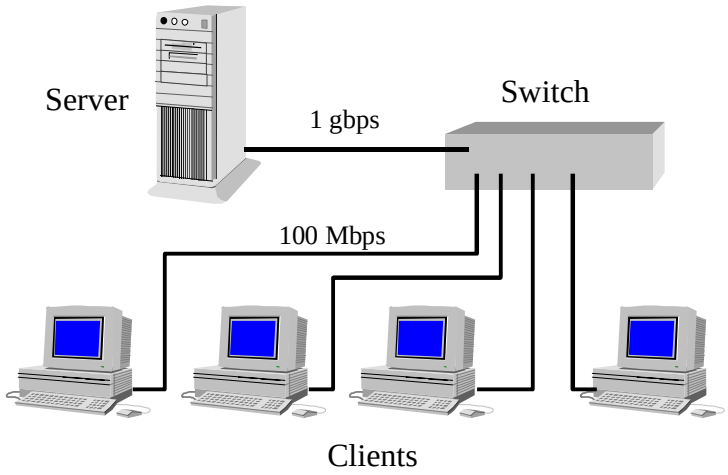

FIGURE 1. Test Setup

Up to 8 clients (300 MHz Pentium II; 100 Mbps full-duplex Ethernet; Windows 2000 Professional) connected to a server (4 x 400 MHz Xeon; 1 Gbps full-duplex Ethernet; Windows 2000 Advanced Server) via a Gigabit Ethernet switch, using TCP/IP.

A server with only a single SCSI (Small Computer System Interface) hard drive was used in this particular study, being representative of the lower cost systems more typically found in a small to medium sized business. In a forthcoming study, we will compare the performance of MQSeries and MSMQ running on more expensive systems, with the multiple striped hard drive configuration used and recommended in [5]. In [2], we discuss in great detail the sources of any differences in results between the two test setups and procedures.

A specialized test application was written to send a message of a fixed size to the queue manager on the server, which would then transmit the message back to the client. For each thread running on the client, another message would not be sent to the server until the previously sent message had been successfully received. Therefore, in order to maximize the number of messages transmitted per unit time by a given client, the benchmark application could define multiple (i.e., ***n***) threads to be run on each client (allowing for up to ***n*** simultaneously outstanding messages per client to exist at any given moment). The benchmark application would then measure the number of successfully transmitted messages for a predefined sampling period, across all threads. In order to allow server queues to reach equilibrium, as well as to guarantee that all threads had started transmitting messages, a waiting period between the start of message transmission and actual measurement collection could be defined.

When multiple clients were used, all were programmed to start transmitting messages at the exact same time. Prior to each test, all client clocks were synchronized to the server's clock. In

addition (unless the response queue was deliberately preloaded with a fixed number of messages), the request and response queues within the server were cleared before every test. (If the response queue was not cleared prior to each test, it was observed that MSMQ performance would deteriorate over time; discussed later.)

Reported response times were calculated as the reciprocal of the number of messages per second measured for a single client running a single thread. Because the transmission rate was gated by the round trip response time (as described above), this represented an accurate measure of the response time from a minimally loaded server. (See [2] for caveats associated with this approach.)

In preliminary runs (prior to collection of the data presented below), the Performance/System Monitor application (part of Windows 2000's Administrative Tools) was used to verify that all 4 server Xeon processors were being utilized (and for example, saturated when MSMQ overall peak server performance numbers were obtained). It was also used to verify that system memory was available and more than adequate for peak capacity testing.

In addition, numerous tuning tests suggested that a sampling period of 5 minutes was optimal in reducing variations in measurements between runs under identical circumstances. They also suggested a 3 minute waiting interval prior to data collection. These values were used for all tests whose results are summarized below.

TEST RESULTS

Transactional Delivery Mode

In this section, for the transactional delivery mode, we analyze in detail the effects of varying both the number of clients and the number of threads per client on the overall number of messages per second (mps) that can be processed by the client and server, using either IBM MQSeries 5.1 or MSMQ (Microsoft Message Queue) 2.0. Since performance also varied between the two message queuing solutions with message size, we present the same detailed performance data for 1024 and 65,536 byte messages. 1024 bytes is a frequently occurring (small) message size, while 65,536 bytes is a relatively large message size that typically maximizes network throughput and minimizes client/server processor utilizations in standard network benchmark tests. Finally, to complete this analysis, we present mps as a function of message size (varying between 100 and 65,536 bytes) for the parameter combination that optimized MSMQ performance during the more detailed analysis of the 1024 byte message size. All transactional data presented below was collected running inside syncpoint.

Both MQSeries and MSMQ support a transactional delivery mode, which provides the ability to incorporate message queuing operations inside a transaction [1]. A transaction is a unit of work, defining a set of operations that must all occur together in order for the transaction to be deemed successfully completed. If a transaction completes successfully, it is ***committed***, and all operations within the transaction (e.g., sending out a message, making a change to a database, etc.) take effect. However, if one fails (for whatever reason), then none of the operations within the transaction take affect (or are reversed), and the system rolls back to its state prior to initiation of the unsuccessful transaction. One important practical advantage to using transactions is their protection of the integrity of commercial databases accessed by many users. For example, the withdrawal of money from one account might need to be simultaneous to deposit into another account. The use of transactions would guarantee that only both could occur. If one or the other could not, then neither would occur. It is therefore a mode of message delivery that is extremely important and commonly used in a wide variety of e-business applications.

In all of the testing done throughout this paper (as well as in [2]), the client issued a Put command, to place its message on the server's request queue, followed by a Get command to retrieve the message from the server's response queue. (In an actual customer environment, the message might typically receive some type of service, as it moved between the request and response queues.) The server correspondingly issued a Get command, to get the message from its request queue, followed by a Put command to move it to its response queue. In testing the performance of the transactional delivery mode below, the client Put was issued inside syncpoint, i.e., it represented a single transaction, controlled by a final commit. Likewise, the server Get/Put commands were issued inside syncpoint, representing another single transaction. MSMQ required that both client and server commands be bundled into a pair of transactions. MQSeries provided greater flexibility, in that the client and server message queuing commands could have been defined to operate inside or outside of syncpoint independent of one another.

1024 byte Message Size

Figure 2 compares the performance of IBM's MQSeries 5.1 vs. Microsoft's Message Queue (MSMQ) 2.0 server, using overall server messages per second (mps) as a function of the number of simultaneous threads per client, for 1 through 4 clients. Figure 3 represents the same comparison using 5 through 8 simultaneously transmitting clients. All data is for a message size of 1024 bytes, using transactional delivery mode (inside syncpoint).

As Figure 2 demonstrates, at a message length of 1024 bytes, IBM MQSeries peak server performance for this system was 45.4 mps. This peak was achieved using 2 clients each running

a single thread. MQSeries single client/single thread response time was 25.8 msec. In contrast, MSMQ server peak performance for this system was only 8.6 mps (i.e., less than 1/5 that of IBM MQSeries, as was the case for express delivery mode; see [2]). This was achieved using 4 clients running 3 threads each. MSMQ single client/single thread response time was 189.0 msec.

This superiority was not limited to peak performance, however. **For the transactional delivery mode**, **using a 1024 byte message size, IBM MQSeries outperformed MSMQ for every combination of number of simultaneous clients and threads per client examined.**

Effects of increasing the number of clients: In general, for both MQSeries and MSMQ, additional clients yielded no increase in performance (since a single client was typically capable of saturating this particular server). The only situation in which an additional client yielded a very slight increase in performance was when a second client was added to a system running a single thread per client (see Figure 2).

Effects of increasing the number of threads per client: In general, for both MQSeries and MSMQ, additional threads per client had only a small effect on overall performance (at least, for up to the 15 threads per client examined here). For both applications, a very slight increase in performance occurred when a second thread was added for the single client situation. For a large number of simultaneous clients, MSMQ experienced a slight decrease in performance as the number of threads increased e.g., for 8 clients, MSMQ performance dropped from 7.8 mps (peak over threads per client examined) to 7.0 mps, when the number of threads per client increased from 2 to 15. Corresponding MQSeries performance measurements for 8 clients were respectively 43.9 mps (2 threads) and 42.0 (15 threads), representing a smaller percent decrease. These decreases with additional clients and/or threads per client were due to (a) queuing delays at the server (expected as the load was increased), which depressed the request rate of a given thread by increasing the time between the messages it could send, and (b) additional overhead time associated with switching between threads on a given client, when multiple threads were running. (Please note that these effects were more pronounced as message size increased; see the next section ***65,536 byte Message Size***. For this larger message size, MSMQ transactional performance decreased by as much as 40% from its peak, as the number of clients and/or number of threads per client increased.)

It should be noted that there are no transactional performance numbers reported by Microsoft for the networked machine (i.e., machine-to-machine) scenario in [5], the scenario that is relevant to this particular study.

It should also be noted that the exact combination of number of clients and threads per client at which peak throughputs are obtained may vary slightly between different setups, and even between tests run on the same setup at different times. Likewise, there can be small experimental variations in the actual values measured between test runs, as would be expected.

In summary, for the transactional delivery mode using a 1024 byte message size, MQSeries provided more than 5 times the performance of that of MSMQ. Most importantly, the IBM MQSeries solution provided far more messages per second for a given unit of client or server processor utilization than did MSMQ. For the same performance, the IBM solution therefore utilized far fewer cycles of a client or server processor, as compared with the Microsoft solution. The end result to the customer should be far better performance of all the other important applications that are running simultaneously with message queuing.

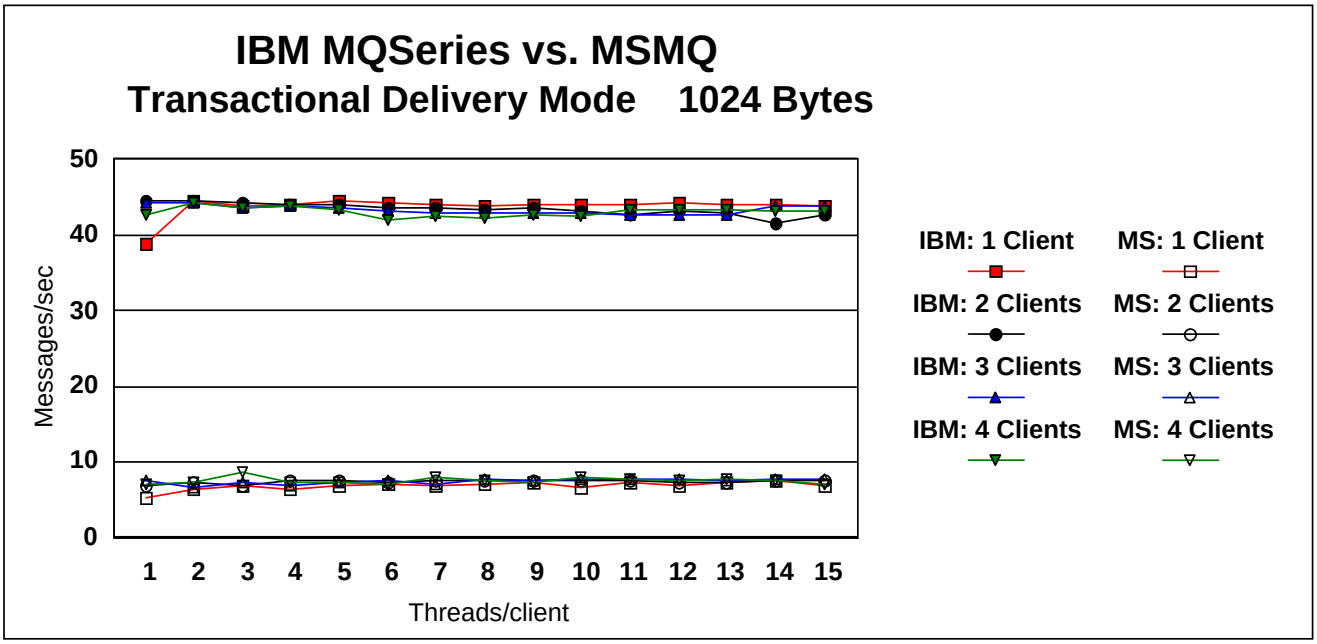

FIGURE 2 IBM MQSERIES vs. MSMQ
MESSAGES/SECOND vs. THREADS/CLIENT
1, 2, 3 and 4 CLIENTS
TRANSACTIONAL DELIVERY MODE
1024 BYTE MESSAGE SIZE

IBM MQSeries and MSMQ messages/second (mps) vs. number of threads/client, for one, two, three and four clients communicating to a server, using a 1024 byte message size and transactional delivery mode.

In the legend above: IBM = IBM MQSeries; MS = Microsoft Message Queue (MSMQ)

MQSeries peak performance = 45.4 mps
MSMQ peak performance = 8.6 mps

MQSeries single client/single thread response time = 25.8 msec
MSMQ single client/single thread response time = 189.0 msec

IBM MQSeries outperformed MSMQ for every combination of number of simultaneous clients and threads/client examined.

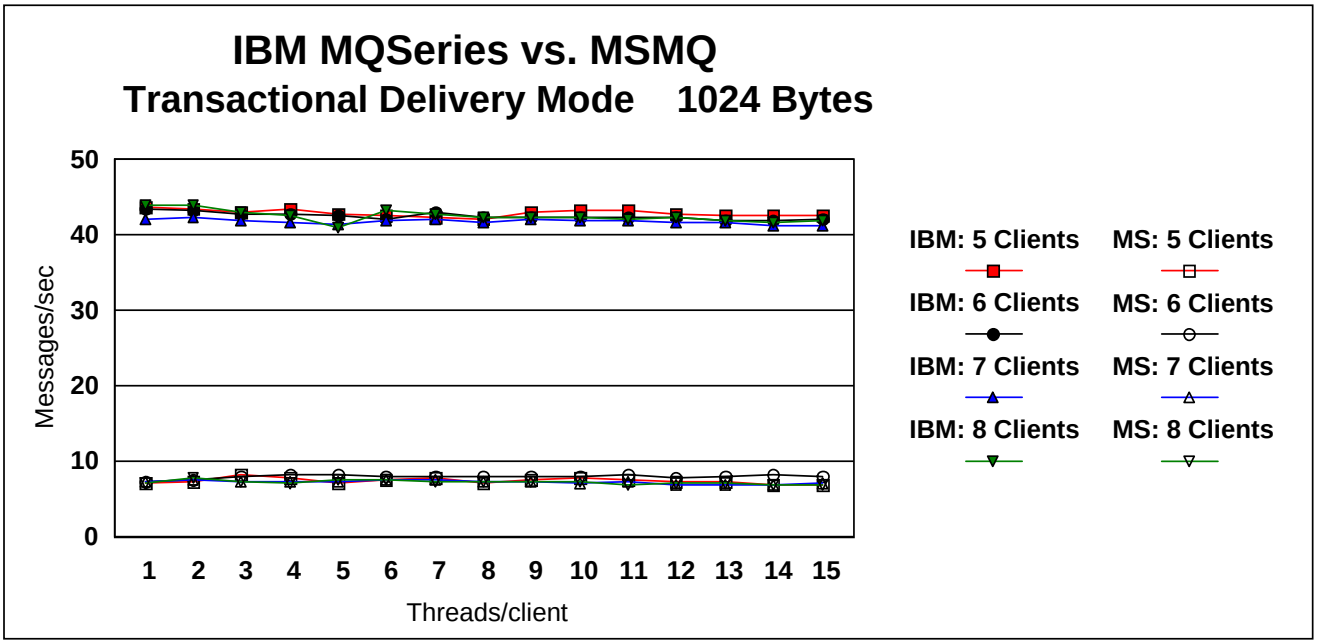

FIGURE 3 IBM MQSERIES vs. MSMQ
MESSAGES/SECOND vs. THREADS/CLIENT
5, 6, 7 and 8 CLIENTS
TRANSACTIONAL DELIVERY MODE
1024 BYTE MESSAGE SIZE

IBM MQSeries and MSMQ messages/second (mps) vs. number of threads/client, for five, six, seven and eight clients communicating to a server, using a 1024 byte message size and transactional delivery mode.

In the legend above: IBM = IBM MQSeries; MS = Microsoft Message Queue (MSMQ)

MQSeries peak performance = 45.4 mps (see Figure 2)
MSMQ peak performance = 8.6 mps (see Figure 2)

IBM MQSeries outperformed MSMQ for every combination of number of simultaneous clients and threads/client examined.

In general (as discussed in the text), overall server mps decreased very slightly as additional clients and threads/client were added, due to (a) queuing delays at the server (expected as the load was increased), and (b) additional overhead time associated with switching between threads.

65,536 byte Message Size

Figure 4 compares the performance of IBM's MQSeries vs. MSMQ server, using overall server messages per second (mps) as a function of the number of simultaneous threads per client, for 1 through 4 clients. Figure 5 represents the same comparison using 5 through 8 simultaneously transmitting clients. All data in this section is for a message size of 65,536 bytes, using transactional delivery mode (inside syncpoint).

As Figure 4 demonstrates, for a message length of 65,536 bytes, IBM MQSeries achieved its peak performance of 12.4 mps at 2 clients each running a single thread. MQSeries single client/single thread response time was 101.9 msec. MSMQ server peak performance for this system was 8.3 mps, achieved using 4 clients each running a single thread. MSMQ single client/single thread response time was 202.4 msec. As was the case for the 1024 byte message size, this superiority was not limited to peak performance, since **IBM MQSeries outperformed MSMQ for every combination of number of simultaneous clients and threads per client examined, for a 65,536 byte message size.**

For both IBM's and Microsoft's message queuing solutions, the performance as measured in mps (for any given number of clients and threads per client) achieved for a message size of 65,536 bytes was always less than that achieved for a 1024 byte message size. This is as expected, since the larger message size would typically require more processing and network transmission time. However, as was demonstrated in [2], system throughputs (obtained by multiplying mps by the message size) actually increased with larger message sizes. This phenomenon has been commonly observed for a wide variety of performance benchmark tests. Network transmission, client/server processing, etc. times for a given unit of data (e.g., file size, physical frame size, protocol window size, etc.) tend to be dominated by a fixed overhead time, which is more efficiently amortized with increasing data unit size. The result is typically an increase in throughput with data unit size, up to a point, with a simultaneous decrease in packets, messages, transactions, etc. per second.

MQSeries performance decreased slightly when going from 1 to 2 clients. Additional clients yielded little or no effect on performance. MSMQ performance was highly variable, with a measurable decrease in performance for 7 and 8 clients, given 10 or more threads per client.

MQSeries performance decreased slightly with an initial increase in the number of threads per client, but stabilized after 4 to 8 threads per client, plateauing to about 10 mps. (MQSeries performance continuously decreased as more threads per client were added, up to the 15 threads per client examined here. However, performance always remained above the 10 mps asymptote.) Given a large number of clients, MSMQ performance continuously declined as the number of threads per client increased. (For example, for 8 clients, MSMQ performance decreased by 32% when going from a single thread per client to 15 threads per client.)

As stated previously, observed performance degradation with an increase in the number of clients and/or threads per client was due to increasing server congestion (and subsequent growth of the server response queue length) and/or cumulative thread switching overhead.

In summary, for the transactional delivery mode using a 65,536 byte message size, MQSeries provided 49% more performance than that delivered by MSMQ. Once again, the overall benefit of MQSeries was far greater performance, at less processor cost, which should provide customers with better system response times and better overall performance of all their other simultaneously running applications.

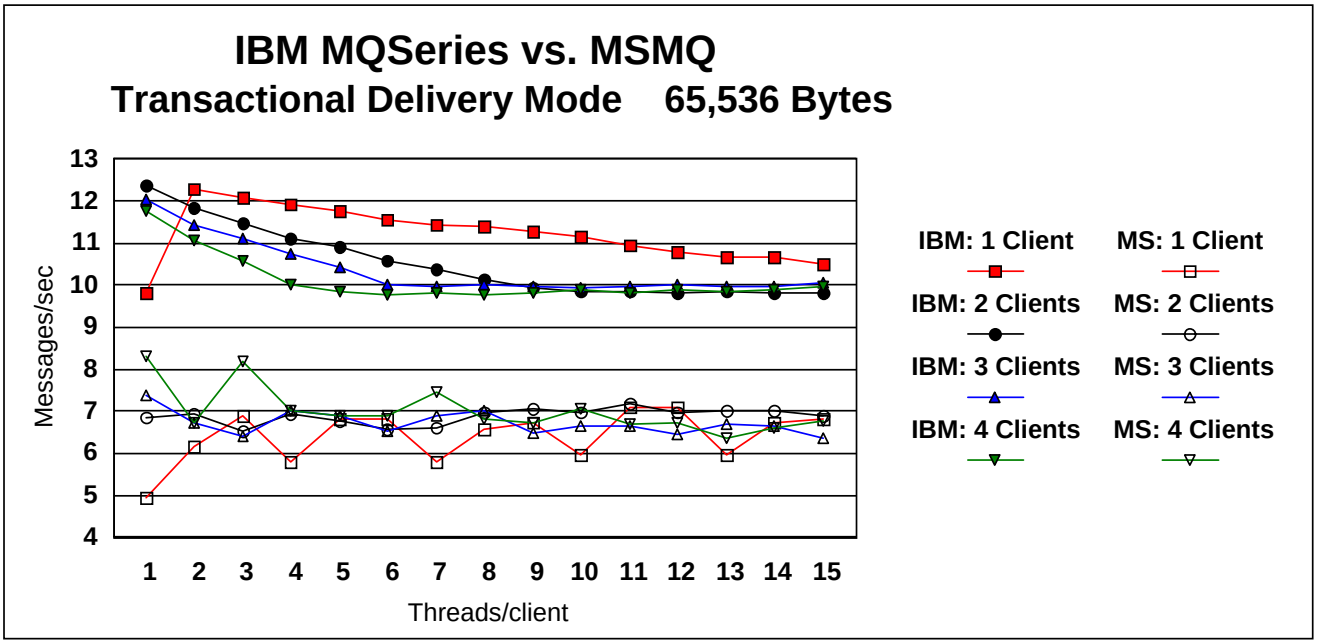

FIGURE 4 IBM MQSERIES vs. MSMQ
MESSAGES/SECOND vs. THREADS/CLIENT
1, 2, 3 and 4 CLIENTS
TRANSACTIONAL DELIVERY MODE
65,536 BYTE MESSAGE SIZE

IBM MQSeries and MSMQ messages/second (mps) vs. number of threads/client, for one, two, three and four clients communicating to a server, using a 65,536 byte message size and transactional delivery mode.

In the legend above: IBM = IBM MQSeries; MS = Microsoft Message Queue (MSMQ)

MQSeries peak performance = 12.4 mps
MSMQ peak performance = 8.3 mps

MQSeries single client/single thread response time = 101.9 msec
MSMQ single client/single thread response time = 202.4 msec

IBM MQSeries outperformed MSMQ for every combination of number of simultaneous clients and threads/client examined.

For both, peak performance was less than that obtained for 1024 byte message sizes due to increased processing times associated with larger message sizes.

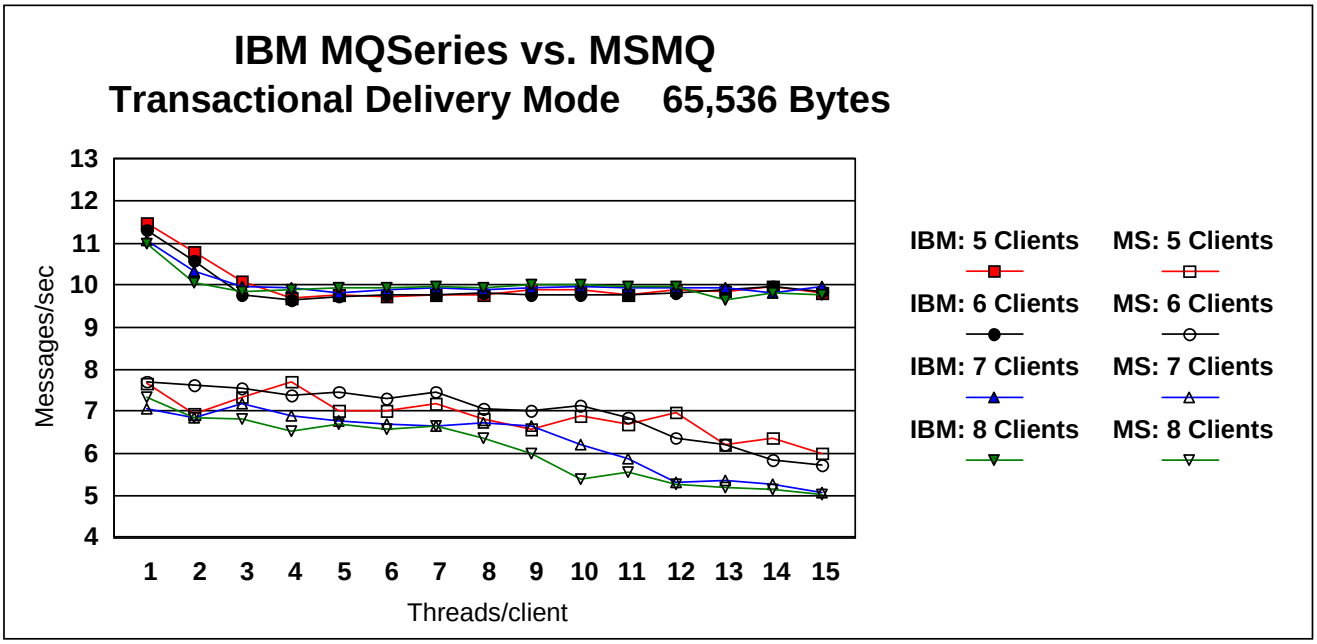

FIGURE 5 IBM MQSERIES vs. MSMQ
MESSAGES/SECOND vs. THREADS/CLIENT
5, 6, 7 and 8 CLIENTS
TRANSACTIONAL DELIVERY MODE
65,536 BYTE MESSAGE SIZE

IBM MQSeries and MSMQ messages/second (mps) vs. number of threads/client, for five, six, seven and eight clients communicating to a server, using a 65,536 byte message size and transactional delivery mode.

In the legend above: IBM = IBM MQSeries; MS = Microsoft Message Queue (MSMQ)

MQSeries peak performance = 12.4 mps (see Figure 4)
MSMQ peak performance = 8.3 mps (see Figure 4)

IBM MQSeries outperformed MSMQ for every combination of number of simultaneous clients and threads/client examined.

In general (as discussed in the text), overall server mps decreased as additional clients and threads/client were added, due to (a) queuing delays at the server (expected as the load was increased), and (b) additional overhead time associated with switching between threads.

For both, peak performance was less than that obtained for 1024 byte message sizes due to increased processing times associated with larger message sizes.

All Message Sizes

As the previous sections above demonstrated, IBM MQSeries performance was significantly greater than that of MSMQ for both relatively small and large message sizes (i.e., for 1024 and 65,536 byte message sizes), for every combination of number of clients and threads per client examined. To quickly verify that this MQSeries performance superiority over MSMQ held across all possible message sizes, we examined the performance of the two message queuing applications for a wide variety of message sizes (i.e., from 100 to 65,536 bytes), using the parameter combination that maximized MSMQ performance for the important 1024 byte message size (i.e., 4 clients, running 3 threads per client). Table 4 and Figure 6 summarize the results of this comparison.

As can be seen below, **IBM MQSeries significantly outperformed MSMQ for every message size examined**. Most importantly, **for the important smaller message sizes (i.e., 100 through 2,048 bytes) that typically comprise up to 80% of most networks' messages/packets, IBM MQSeries throughput was over 5 times better than that of MSMQ**.

Generally, messages per second (mps) decreased (or remained relatively constant) with increasing message size. This was because certain processing costs (e.g., bus transmission times) would be expected to increase with increasing message size, resulting in fewer messages per unit time being processed by the same system hardware/software. (If little change was observed, then processing costs that would be fixed per unit message presumably dominated total processing costs per message.) However, throughput measured in kilobytes per second (KBps), and obtained using the formula *(mps x message size in bytes) /1000*, increased with increasing message size, indicating that the benefits of amortizing fixed costs per unit message (due to larger message sizes) increased more rapidly than message size itself. This amortization phenomenon, resulting in higher throughputs with increasing data unit size, is commonly observed for a wide variety of physical systems.

Table 4. Transactional Performance

Message Size (bytes)	Transactional Performance 4 clients, 3 threads per client				Ratio of MQSeries to MSMQ mps
	IBM MQSeries		MSMQ		MQSeries/ MSMQ
	KBps	mps	KBps	mps	
100	4.8	47.9	0.9	8.6	5.6
512	23.2	45.3	4.3	8.4	5.4
1,024	46.4	45.3	8.8	8.6	5.3
2,048	83.4	40.7	16.4	8.2	5.1
3,072	116.3	37.8	24.2	7.9	4.8
4,096	142	34.7	32.6	8	4.4
5,120	182.8	35.7	41.2	8	4.4
10,240	297.6	29.1	80.4	7.9	3.7
20,480	463.3	22.6	162	7.9	2.9
30,720	579.8	18.9	247.8	8.1	2.3
40,960	625.7	15.3	336.2	8.2	1.9
65,536	705	10.8	530.8	8.1	1.3

Table 4. MQSeries and MSMQ transactional performance (measured in both throughput, or kilobytes per second, and messages per second) for 100 through 65,536 byte message sizes. KBps = kilobytes per second; mps = messages per second.

MQSeries transactional performance significantly exceeded that of MSMQ for every situation examined. For the table above, the parameter combination that maximized MSMQ performance at a 1024 byte message size (i.e., 4 clients, 3 threads per client) was used.

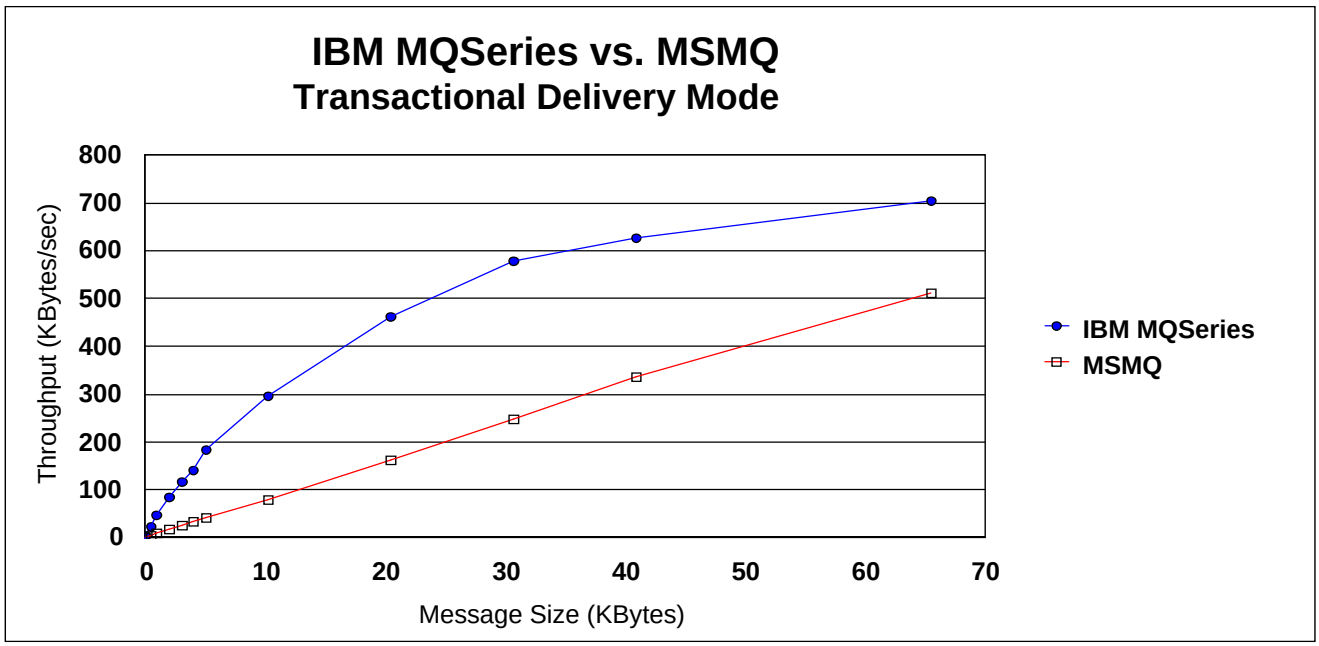

FIGURE 6 IBM MQSERIES vs. MSMQ THROUGHPUT vs. MESSAGE SIZE 100 through 64K BYTE MESSAGE SIZES 4 CLIENTS, 3 THREADS TRANSACTIONAL DELIVERY MODE

IBM MQSeries and MSMQ throughput (kilobytes per second, or KBps) vs. message size (100, 512, 1024 5120, 10240, 20480, 30720, 40960 and 65536 bytes), obtained with four clients running 3 threads per client, communicating to a server using transactional delivery mode (running inside syncpoint on the server). The parameter combination of four clients running 3 threads per client yielded the peak MSMQ throughput, for the important 1024 byte message size.

For every message size examined, MQSeries substantially outperformed MSMQ. For both applications, throughput increased with increasing message size (in contrast to messages per second, which decreased with increasing message size; see Figures 2 through 4).

Persistent Delivery Mode

In this section, for the persistent delivery mode, we once again analyze in detail the effects of varying both the number of clients and the number of threads per client on the overall number of messages per second that can be processed by the client and server, using either IBM MQSeries 5.1 or MSMQ 2.0. We also examine the performance effects of running inside vs. outside syncpoint on the server (the former for MQSeries only), as well as the effects of preloading the server response queue. Throughout, a 1024 byte message size is used (being representative of the message sizes most often used in actual customer environments).

As previously described, the persistent (or for MSMQ, recoverable) delivery mode guarantees that all outstanding messages can be recovered following system failure. For the tests presented below, this typically entailed storing messages to the server disk (unless the server could immediately hand off a received message back to the originating client, which occurred only when the server was lightly loaded).

As stated above, for MQSeries, data was collected (and is presented below) for persistent delivery mode, with the server Get/Put operations running both inside and outside syncpoint (see the previous section entitled **Transactional Delivery Mode** for a detailed explanation of what is meant by running inside/outside syncpoint). For MSMQ, persistent delivery mode would only operate outside syncpoint on the server, so only this data is presented below.

Effects of Number of Clients, Number of Threads per Client and Operating Inside or Outside Syncpoint

Figure 7 compares the performance of IBM's MQSeries 5.1 vs. Microsoft's Message Queue (MSMQ) 2.0 server, using overall server messages per second (mps) as a function of the number of simultaneous threads per client, for 1 through 4 clients. Figure 8 represents the same comparison using 5 through 8 simultaneously transmitting clients. All data is for a message size of 1024 bytes, using persistent (or in Microsoft terminology, recoverable) delivery mode, running both inside syncpoint (IBM MQSeries only) and outside syncpoint (IBM MQSeries and MSMQ).

As Figure 7 suggests, for some curves representing a given number of clients, there were a number of initial data points for 1 to 3 threads per client that were considerably larger in value than those obtained for a larger number of threads per client. Specifically for MSMQ, for 2 through 4 clients, running from 1 to 3 threads per client, the observed mps were significantly larger than those obtained for 4 through 15 threads per client. For example, for 4 clients, the performance measured when a single thread was running was 83.6 mps. (This was in fact the overall peak performance obtained for MSMQ's persistent delivery mode.) However, at 4 threads per client, performance had dropped by 48% to 43.4 mps. (At 15 threads per client, MSMQ performance was down to 34.5 mps.) These 1-to-3-thread-per-client larger values (as well as the entire MSMQ single client persistent curve) were similar to those obtained using their express delivery mode (see [2]). Furthermore, while these express-like numbers were being collected, there was very little or no disk activity (as indicated by the server's disk access light), suggesting that very little or none of the data was actually being stored on disk. Presumably, given that the server was lightly loaded, the server could immediately hand off a received message back to the originating client, thereby precluding the need for storage to the disk. (This was the case for the 1-client/ 1-thread-per-client case described below for MQSeries.) For the smaller performance values obtained for 4 through 15 threads (as well as for all data collected for 5 or more clients), the disk activity light was on continuously, indicating that data was indeed being stored on disk.

For MQSeries persistent delivery mode, only the single data point represented by 1 client running a single thread (outside syncpoint) was unusually large and comparable to its corresponding express delivery mode value. At 452.7 mps, it represented the peak performance obtained for

MQSeries' persistent delivery mode. Subsequent points on the same curve (i.e., 1 client running 2 through 15 threads) ranged between 73.8 and 75.4 mps. For these latter and all other MQSeries data points (for 2 or more clients, as well as for all persistent delivery mode data *inside* syncpoint), the disk activity light was on continuously, indicating that data was being stored on disk for possible subsequent recovery.

Therefore, given this bimodal behavior of persistent performance, in order to compare the two applications, two different metrics were computed (for a given number of clients): (1) peak mps; and (2) steady state average mps.

The overall peak performance for a given number of clients was determined as the maximum over all number of threads per client examined. This peak typically occurred at one or two threads per client.

The steady state average mps for a given number of clients was computed by averaging over those mps measurements taken when the system was in "steady state". The system was deemed to be in steady state when significant disk activity occurred (indicating that messages were being regularly stored to disk). Steady state typically yielded mps values that lay in the performance plateau region, i.e., a relatively flat region of the mps vs. threads per client curve that followed the initial peaks in performance described above. (As stated previously, the peak region typically yielded performance numbers similar to those obtained for express delivery mode, as reported in [2].) For MSMQ, this average was computed over 4-15 threads per client when 1-4 clients were tested. For fairness, this same method of computing the average was used for MQSeries when 1-4 clients were tested, when the server operated outside syncpoint (even though MQSeries achieved steady state for every parameter combination except a single client running a single thread). For both MSMQ and MQSeries, when 5-8 clients were tested, this average was computed over all numbers of threads per client examined, i.e., 1-15. This same method of computing the average was used for MQSeries running inside syncpoint.

The results are summarized in Table 5 below.

Table 5. Peak and Average Persistent Messages/sec vs. Number of Clients (1,024 Byte Message Size)

Number of Clients	Persistent Average Messages per Second (mps)					
	IBM MQSeries (inside sp)		IBM MQSeries (outside sp)		MSMQ (outside sp)	
	Peak mps	Steady State Av.	Peak mps	Steady State Av.	Peak mps	Steady State Av.
1	45.5	44.7	452.7	74.6	31.2	30.6
2	44.6	42.9	75.1	74.1	57.7	37.3
3	45	44	75.2	74.4	75.1	37.8
4	46.2	45.6	74.5	72.7	83.6	38.3
5	43.8	43	75.5	74	50.8	39.9
6	44.4	43.3	74.7	72.6	52.6	39.4
7	44.9	43.6	75.5	72.5	51.9	38.6
8	44.5	44.5	74.1	70.2	51.9	37.7

Table 5. MQSeries and MSMQ persistent peak and "steady state" average performance (measured in messages per second, or mps) as a function of the number of clients simultaneously transmitting to the server, for a 1024 byte message size. sp = syncpoint

MQSeries persistent performance *outside* syncpoint significantly exceeded that of MSMQ *outside* syncpoint for every situation examined. More importantly, MQSeries persistent performance *inside* syncpoint likewise exceeded that of MSMQ *outside* syncpoint for almost every situation examined, a surprising result given that operating inside syncpoint is more costly than operating outside syncpoint.

The overall peak mps for a given number of clients was determined as the maximum over all number of threads per client examined. This peak typically occurred at a small number of threads per client.

The steady state average mps was obtained by averaging performance over those values collected when the system had settled down into the relatively stable, flat portion of its mps vs. threads per client curve (following the very large initial peak values listed above; see Figures 7 and 8 below). It is over this portion of the curve that messages were being consistently stored on disk to guarantee recoverability.

As Figure 7 and Table 5 demonstrate, at a message length of 1024 bytes, IBM MQSeries peak server performance for this system (running outside syncpoint on the server) was 452.7 mps (as mentioned above). This was achieved by a single client running a single thread. MQSeries single client/single thread response time was 2.2 msec. In contrast, MSMQ peak server performance for this system (running outside syncpoint on the server) was only 83.6 mps (i.e., less than 1/5 that of IBM MQSeries, similar to the case for express delivery mode; see [2]). This was achieved using 4 clients running a single thread each. MSMQ single client/single thread response time was 36.6 msec.

However, as discussed above, these peak numbers were obtained under lightly loaded conditions, where messages were clearly not being stored consistently to disk. (Recoverability was presumably guaranteed by the fact that the server was almost immediately able to hand a message back to the client, following its receipt.) Under these conditions, persistent delivery mode performance (operating outside syncpoint on the server) was comparable to that obtained for express delivery mode, and therefore simply reflected the substantial superiority of MQSeries' express performance over that of MSMQ [2]. Since MQSeries clients (for express delivery mode) were so much more efficient than those of MSMQ at generating messages to the server,

MQSeries persistent performance was only comparable to that of its express delivery mode for the single data point of 1 client running 1 thread. For any additional clients and/or threads per client, the MQSeries clients were powerful enough to congest the server, requiring it to store most messages to disk to guarantee their recoverability. Correspondingly, since MSMQ single client (express) performance was only a fraction of that of MQSeries [2], MSMQ required many more clients and/or threads per client before congesting the server (that would in turn then require persistent messages to be stored to disk). Therefore, MSMQ yielded unusually high (express-like) performance numbers for 2-4 clients running up to 3 threads per client (as stated above).

As a result, except for the single data point of 4 clients running a single thread each (for which MSMQ was not required to store messages to disk), **for the persistent delivery mode running outside syncpoint on the server**, **IBM MQSeries outperformed MSMQ for every other combination of number of simultaneous clients and threads per client examined.**

More significantly, as Figures 7 and 8 demonstrate, for every data point examined (representing a given combination of number of clients and threads per client) in which both applications were truly storing data to disk, **MQSeries persistent performance running *inside* syncpoint on the server was usually better than, and at worse, only slightly less than, that of MSMQ running *outside* syncpoint.** Since running inside syncpoint is more costly to performance than running outside it, this superiority by IBM MQSeries was quite exceptional. (Please note that we could not run MSMQ persistent delivery mode inside syncpoint.)

In order to develop a more meaningful metric that would compare the two applications when both were storing persistent messages to disk, the persistent steady state average mps was defined. As Table 5 above shows, **for every number of clients examined, IBM MQSeries steady state average persistent performance, running *either inside or outside* syncpoint on the server, significantly exceeded that of MSMQ running *outside* syncpoint.**

For the persistent delivery mode operating outside syncpoint on the server, MSMQ achieved its highest steady state average mps for 5 clients, yielding 39.9 mps. For MQSeries running outside syncpoint, the corresponding steady state average of 74 mps (for 5 clients) was 85% better than that provided by MSMQ. For MQSeries running *inside* syncpoint, the corresponding 5-client average of 43 mps was once again better than that of MSMQ running *outside* syncpoint.

Effects of increasing the number of clients and/or number of threads per client: As described above for both applications, when only a very few clients and threads per client were examined, both applications yielded performance numbers that were similar to those obtained for their corresponding express delivery modes (due to the fact that little or data was being stored to disk, thereby avoiding costly disk access delays). It should be noted that this phenomenon never occurred for IBM MQSeries persistent delivery mode inside syncpoint (and only for a single client running a single thread for MQSeries outside syncpoint). As the number of clients and/or threads was further increased, performance would then drop to a much lower value, as data was then stored to disk, in order to guarantee recoverability (as evidenced by the server disk activity light being on continuously during testing).

Just prior to this data-storage transition point being achieved, performance for both applications typically increased slightly as either the number of clients or number of threads per client was increased. However, once this data-storage transition point had been achieved, as the number of clients and/or threads per client continued to increase, the performance of both applications typically decreased, due to server congestion and/or overhead due to switching between threads. For MQSeries, this performance decrease was slight, for the parameter combinations examined. However, for MSMQ, this performance decrease was quite significant when the product of the number of clients and threads per client was large. For example, for 8 clients, MSMQ persistent performance (outside syncpoint) was 51.8 mps when each client was running a single thread. However, for 15 threads per client, MSMQ performance dropped by almost 45% to 28.8 mps. (In contrast, MQSeries persistent performance outside syncpoint for 8 clients each running 15

threads was more than double, at 67.3 mps. Even more significantly, for the same parameter pair, MQSeries persistent performance inside syncpoint was 44.0 mps, i.e., 53% greater than that achieved by MSMQ *outside* syncpoint.) This same phenomenon of significant performance degradation by MSMQ with server congestion was observed in [2] for its express delivery mode, and is presumably due in part to problems associated with its response queue searching algorithm (see the section **Effects of Preloading the Server Response Queue** below).

In general, performance outside syncpoint was better than that obtained inside syncpoint. This is as expected, since running inside syncpoint necessarily involved more overhead to encapsulate the server commands inside a transaction. This provided enhanced system integrity at the expense of overall performance.

Once again, it should also be noted that the exact combination of number of clients and threads per client at which peak throughputs are obtained may vary slightly between different setups, and even between tests run on the same setup at different times. Likewise, there can be small experimental variations in the actual values measured between test runs, as would be expected.

In summary, for a 1024 byte message size, the persistent delivery mode of MQSeries, operating outside syncpoint, significantly outperformed that of MSMQ for every combination of number of clients and threads per client examined (when data was actually being stored to disk). More significantly, the persistent delivery mode of MQSeries, operating *inside* syncpoint, outperformed that of MSMQ operating *outside* syncpoint, for most of these same parameter combinations. This is most surprising, since operating inside syncpoint is more costly to overall performance than operating outside syncpoint. As was the case for the transactional delivery mode, MQSeries provided far greater performance, at less processor cost, which should provide customers with better system response times and better overall performance of all their other simultaneously running applications.

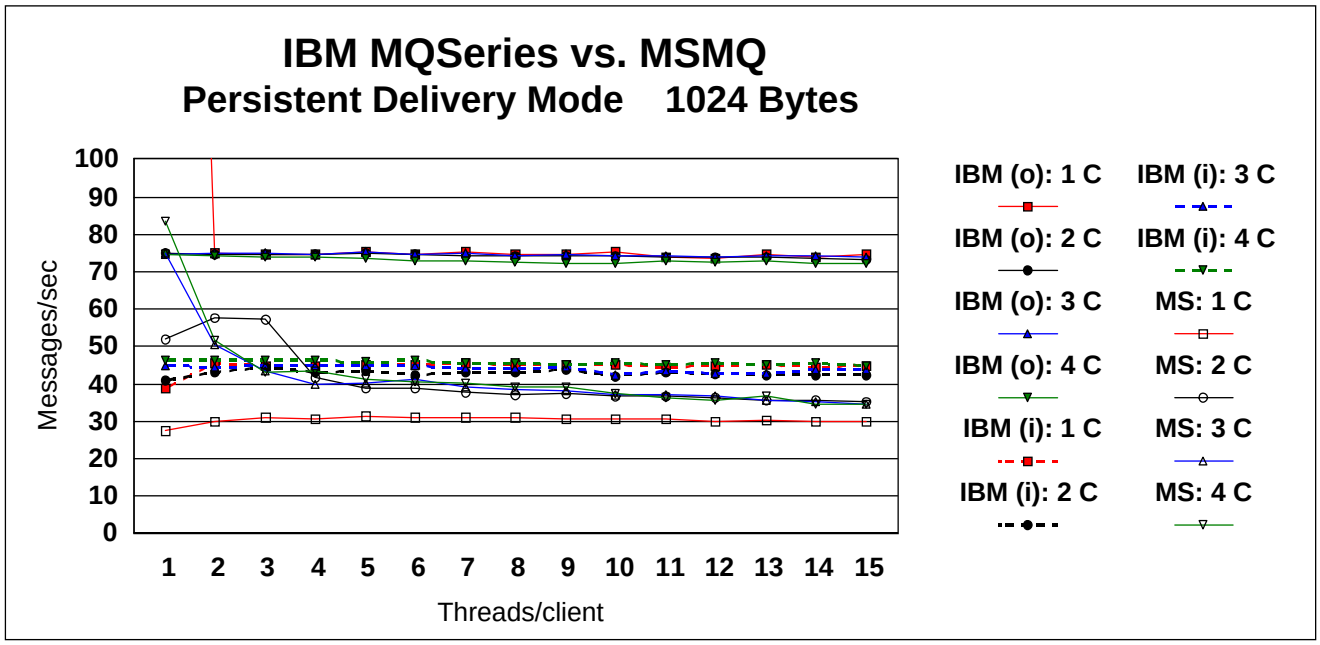

FIGURE 7 IBM MQSERIES vs. MSMQ
MESSAGES/SECOND vs. THREADS/CLIENT
1, 2, 3 and 4 CLIENTS
PERSISTENT DELIVERY MODE
1024 BYTE MESSAGE SIZE

IBM MQSeries and MSMQ messages/second (mps) vs. number of threads/client, for one, two, three and four clients communicating to a server, using a 1024 byte message size and persistent delivery mode.

In the legend above:

IBM (o) = IBM MQSeries, outside syncpoint;
IBM (i) = IBM MQSeries, inside syncpoint;
MS = Microsoft Message Queue (MSMQ), outside syncpoint (only option)
C = Client(s)

MQSeries peak performance = 452.7 mps (off chart, occurring for 1 client, 1 thread/client)
MSMQ peak performance = 83.6 mps

MQSeries single client/single thread response time = 2.2 msec
MSMQ single client/single thread response time = 36.6 msec

IBM MQSeries outperformed MSMQ for essentially every combination of number of simultaneous clients and threads/client examined.

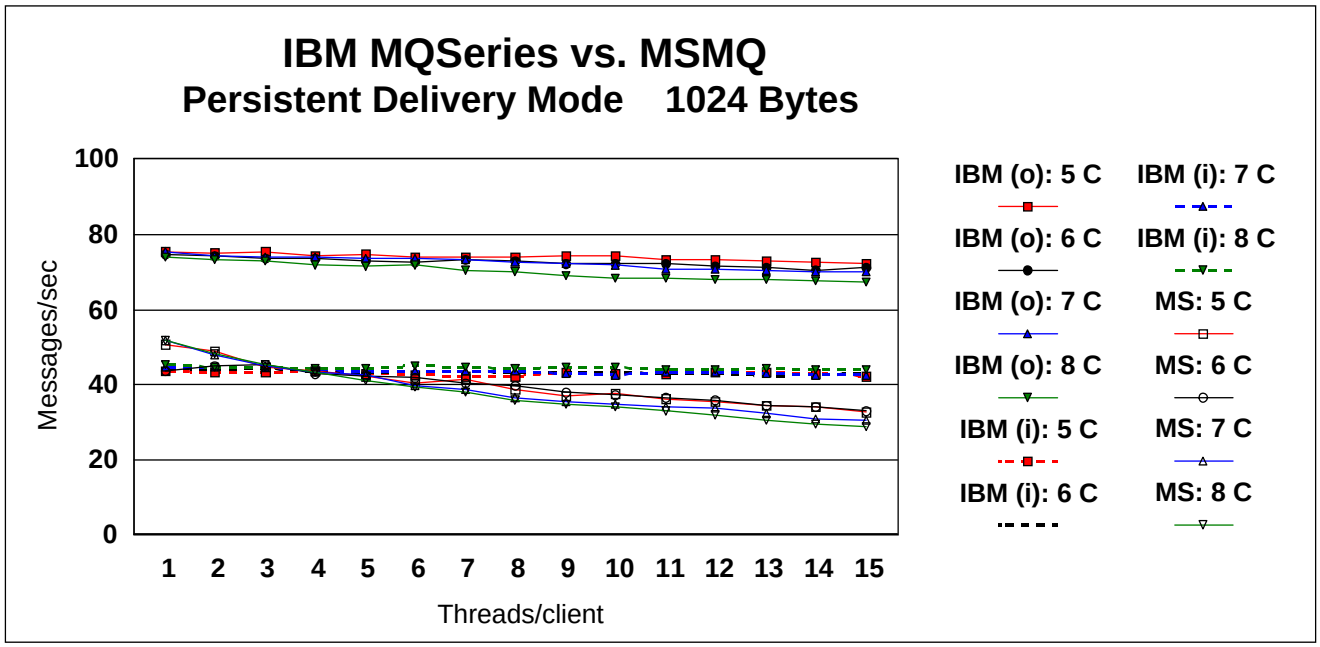

FIGURE 8 IBM MQSERIES vs. MSMQ
MESSAGES/SECOND vs. THREADS/CLIENT
5, 6, 7 and 8 CLIENTS
PERSISTENT DELIVERY MODE
1024 BYTE MESSAGE SIZE

IBM MQSeries and MSMQ messages/second (mps) vs. number of threads/client, for five, six, seven and eight clients communicating to a server, using a 1024 byte message size and persistent delivery mode.

In the legend above:

IBM (o) = IBM MQSeries, outside syncpoint;
IBM (i) = IBM MQSeries, inside syncpoint;
MS = Microsoft Message Queue (MSMQ), outside syncpoint (only option)
C = Client(s)

MQSeries peak performance = 452.7 mps (see Figure 7)
MSMQ peak performance = 83.6 mps (see Figure 7)

IBM MQSeries outperformed MSMQ for essentially every combination of number of simultaneous clients and threads/client examined.

In general (as discussed in the text), overall server mps decreased as additional clients and threads/client were added, due to (a) queuing delays at the server (expected as the load was increased), and (b) additional overhead time associated with switching between threads

Effects of Preloading the Server Response Queue

In this section, we analyze the effects of preloading the server response queue with a fixed number of messages, prior to measuring performance (using the same setup and test procedures as outlined above). In our original study of express delivery mode performance [2], it was observed that when the response queue was not cleared prior to each performance test (for a given parameter pair of number of clients and threads per client), MSMQ performance deteriorated over time. It was determined that occasionally, for a given test, a small number of unretrieved messages remained in the response queue following test termination, that were not retrieved in subsequent runs (thereby slowly accumulating). When the response queue was systematically cleared prior to each test, MSMQ performance remained stable over time. MQSeries performance was not significantly affected by this process of unretrieved message accumulation.

These initial observations suggested that, over the course of the normal operation of a message queuing server, unretrieved messages should be expected to slowly accumulate with time in the server response queue. These messages would result from their source clients shutting down (perhaps for days or weeks due to vacation), crashing, etc., prior to their retrieval. Our test process simply accelerated the rate at which these unretrieved messages would accumulate.

Therefore, as stated above, in order to optimize MSMQ performance, the server response and request queues were systematically cleared prior to each performance test. This was done for all testing reported in the section above. However, given that unretrieved messages would be expected to naturally accumulate over time in any operational message queuing server's response queue, a series of tests was deemed necessary to determine the performance effects of preloading the response queue with a precise number of messages.

For the data presented below, prior to each test, both the request and response queues were cleared. A separate client, not involved in the subsequent performance testing, was then used to directly preload the response queue with a fixed number of messages. The client was then immediately shut off without retrieving the messages. The messages therefore sat in the response queue, awaiting retrieval by the (now unresponsive) client throughout subsequent testing. It was verified that the desired number of preloaded messages was in fact delivered to the response queue, using Administrative Tools for MSMQ and MQSeries Explorer for MQSeries.

A message size of 1024 bytes was used throughout this particular test (for comparison purposes with the performance optimized data previously presented above). For this message size, the parameter pair of number of clients and threads per client that yielded the best performance for MSMQ's persistent delivery mode, while also guaranteeing that messages were actually stored on disk, was 4 clients running 5 threads each. This combination was therefore used throughout. The response queue was preloaded with 0, 50, 100, 150, 200, 300, 500 and 1000 messages.

(Please note that when transactional delivery mode was used, the performance of neither product was appreciably affected by server request queue preloading. The unloaded transactional performance values given above were representative of those obtained with preloading, and are therefore not repeated in this section.)

Table 6 and Figure 9 summarize the effects of this preloading on MQSeries and MSMQ persistent performance, presenting messages per second (mps) as a function of the initial response queue size (in messages). Table 6 also presents the ratio of MQSeries to MSMQ mps, indicating the number of times (not percent) that MQSeries performance was better than that of MSMQ.

As the data indicates, **for persistent delivery mode, MQSeries performance remained relatively unchanged as the initial request queue size (Q_0) was increased. In contrast, MSMQ performance was decreased by even a small number of preloaded messages.** For Q_0 = 200, MSMQ performance dropped from its peak performance of 43.0 mps to 26.3 mps (a reduction of nearly 39%). At Q_0 = 1000 messages, MSMQ performance completely collapsed to 7.9 mps. In contrast, the corresponding MQSeries performance was 72.8 mps.

These results are consistent with the effects that were observed above when the server became congested (i.e., when the product of the number of clients and threads per client became large). Under those conditions, MSMQ performance was observed to likewise decline.

Table 6. Messages/sec vs. Initial Request Queue Size

Initial Request Queue Size (Q_0, in Messages)	Messages per second (mps)		Ratio of MQSeries to MSMQ mps
	IBM MQSeries	MSMQ	MQSeries/MSMQ
0	73	43	1.7
50	72.9	39.9	1.8
100	72.9	36.7	2
150	73	32.4	2.3
200	72.9	26.3	2.8
300	72.8	20.3	3.6
500	72.9	14.1	5.2
1,000	72.8	7.9	9.2

Table 6. MQSeries and MSMQ performance (measured in messages per second) as the server response queue was preloaded, prior to testing, with a fixed number of (unretrievable) messages.

This test scenario mimicked expected conditions within an actual message queuing customer environment. Also included is the ratio of MQSeries to MSMQ performance, indicating the number of times (not percent) that MQSeries performance was better than that of MSMQ (e.g., **at 1,000 preloaded messages, MQSeries performance was 9.2 times better than that of MSMQ, or 822% better**). This ratio increased as the number of preloaded messages increased, due to the collapse of MSMQ performance. 1024 byte messages were used throughout, with 4 clients running 5 threads each. This was the parameter combination that yielded near optimal MSMQ server performance, while guaranteeing that messages were truly being stored on disk.

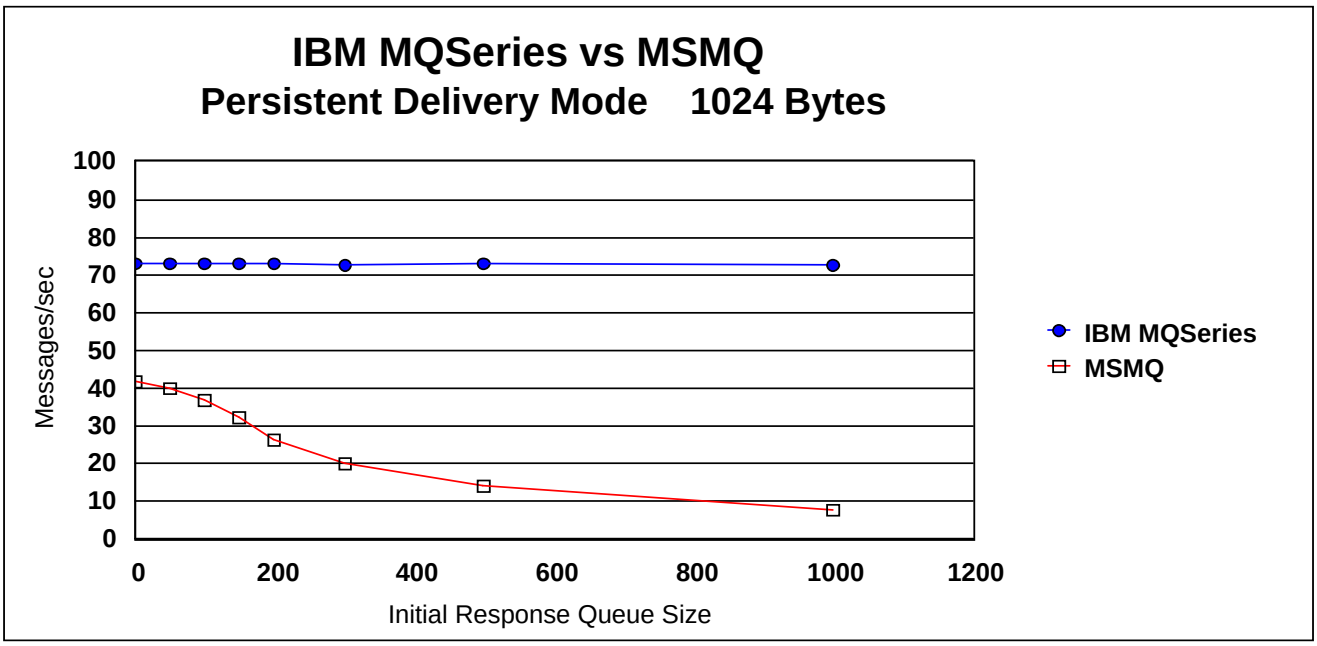

FIGURE 9 IBM MQSERIES vs. MSMQ
MESSAGES/SECOND vs. INITIAL RESPONSE QUEUE SIZE
4 CLIENTS, 5 THREADS/CLIENT
PERSISTENT DELIVERY MODE
1024 BYTE MESSAGE SIZE

IBM MQSeries and MSMQ messages/second (mps) vs. initial response queue size, using 4 clients running 5 threads each, a 1024 byte message size and persistent delivery mode. (The above parameter combination yielded near optimal MSMQ server performance, while guaranteeing that messages were truly being stored on the disc.)

For every initial response queue size (Q_0) examined, MQSeries substantially outperformed MSMQ. MSMQ performance completely collapsed as the initial queue size increased.

Q_0 = 0 messages **MQSeries performance = 73.0 mps**
MSMQ performance = 42.0 mps

Q_0 = 200 messages **MQSeries performance = 72.9 mps**
MSMQ performance = 26.3 mps

Q_0 = 1,000 messages **MQSeries performance = 72.8 mps**
MSMQ performance = 7.9 mps

RECENTLY PUBLISHED MSMQ PERFORMANCE RESULTS

In [2], we provided a detailed analysis and criticism of the measurement setup and techniques used in Microsoft's white paper entitled "Optimizing Windows NT Server performance in a Microsoft Message Queue Server environment" [5]. It presented benchmark measurement data for MSMQ only, in which messages were transmitted or received (a) within the same machine (i.e., the "single machine scenario"), and (b) between two different machines (i.e., the "networked machine scenario"). Several major flaws were present in the testing procedures of that study, that rendered the subsequent results of little value to actual customers. These flaws were carefully enumerated in [2].

Recently, Microsoft funded a similar study by NSTL entitled "Performance evaluation of Microsoft Messaging Queue, IBM MQSeries and MQBridge" [7], in which MQSeries performance numbers were also quoted. The study only examined the "single machine scenario" of [5] (as well as a bridged scenario between MSMQ and MQSeries, presumably to demonstrate that MSMQ could bridge to non-Windows platforms using "MQSeries Bridge"). Surprisingly, the NSTL tests [7] were run on the exact same hardware configuration that was used in the 1998 report, i.e., 200 MHz Pentium Pros. More importantly, Microsoft made the benchmark tests that were used available to the public (MSMQBench for MSMQ, MQSRBench for IBM MQSeries, both written by Microsoft). This study shares all (and more) of the design flaws inherent in the original study.

(1) *MSMQBench and MQSRBench measure low-level function only:* We carefully reviewed the source code for the two benchmark tests cited above (i.e., MSMQBench and MQSRBench). We also ran them in our labs. These benchmark tests by themselves, and as run in the NSTL tests, represented a very trivial scenario that had little relevance to a real world message queuing application. While a number of options existed for these benchmark tests (express, persistent or recoverable, etc.), the two major choices consisted of (a) measuring how fast a queue could be filled, and (b) measuring how fast a queue could be cleared. (These appear to correspond to the Messages Sent and Messages Received reported in [5] and [7].) An example of a test that was run in both [5] and [7], under the single machine scenario, was to measure how fast MSMQ cleared a queue, after all messages had already been received by that queue. The test yielded ridiculously large performance numbers, that should have immediately alerted Microsoft and NSTL that they were actually measuring extremely low-level, primitive functions, that had nothing to do with the higher level bottlenecks that would actually define the performance of their message queuing software. By themselves (and as reported), therefore, these benchmarks provided little insight as to how the application would perform in a real customer environment.

It also appears that MQSRBench does not take advantage of the correlation ID capability of MQSeries, deliberately crippling MQSeries performance in tests of shared response queues (discussed below).

(2) *Single machine scenario only:* As stated above, the NSTL report provided only a single machine test comparison, i.e., using message queuing to communicate between two programs running on the same machine. However, most customers use message queuing to communicate between two or more machines on a network (e.g., between a client and a server), with the initiating client expecting a response back from the server to which it sent the message. One of the most important benefits and, therefore, customer uses of message queuing is to remove handling of networking problems (network failures, congestion delays, etc.) from the client software program.

In our study, the client was required to first send a message to the server's request queue, and then retrieve that same message back from the server's response queue (once the server had moved the message between the two queues). As such, the full client and server code paths were exercised. This test scenario therefore mimicked a real customer's typical use of a message queuing application, i.e., whereby a message is sent by a client to a (possibly remote) server

requesting some service, the results of which are then returned to the client in a (possibly different) message.

The test scenarios chosen by Microsoft in [5] and [7] therefore minimized any application processing costs incurred by the client and server. Yet, as suggested above, these are the very costs that define the final real performance of any message queuing application in an actual customer environment. Their tests were more indicative of the capacity of their server and network hardware resources, than of the capacity of their message queuing server to provide meaningful processing of incoming messages (that would then yield return messages back to the requesting clients).

(3) *Dedicated server response queue*: As was the case with Microsoft's earlier tests in [5], the recent NSTL tests completely avoided testing any situations in which two or more clients shared a common response queue. In the real world, response queues on servers often have to be shared in customer environments that involve hundreds and even thousands of client machines. Under test conditions that mimicked these very real customer environments, MSMQ performance completely collapsed, even under very mild server congestion. This phenomenon was well documented above for the persistent delivery mode (in the section entitled **Effects of Preloading the Server Response Queue**). This same performance collapse was also observed for MSMQ's express delivery mode (see [2]). It is presumably the result of a poorly designed response queue searching algorithm.

Because MSMQ performance was so sensitive to even a very small number of messages accumulating in its server response queue, our study required that the message queues be cleared prior to each test. Without this, MSMQ performance was observed to significantly degrade over the course of testing. (MQSeries was not appreciably affected.) Unfortunately, the accumulation of messages in a response queue would occur naturally over time in a real customer environment, where Microsoft would not have the system initialization benefits provided by the test procedures used in our study. This seriously compromises MSMQ's ability to scale to the larger client population sizes often found in actual customer environments. Furthermore, MSMQ completely consumed the server processor as a given client searched for its messages in the shared queue. In the real customer world, this would mean that the processor would necessarily be kept from executing important e-business or other applications. The performance of these other applications would therefore suffer.

With the NSTL benchmarks, Microsoft was clearly interested in avoiding MSMQ's design problems and thereby inflating the results, whether or not the benchmarked environment had any bearing on real world scenarios. On the other hand, the benchmarks used in this report were designed to assess the real performance in an actual customer environment. It should be noted that the real world includes many servers that run LINUX and other non-Windows operating systems. In that real world, which characterizes a very large proportion of existing server farms, MSMQ cannot even directly participate.

CONCLUSIONS

As the data has shown, MQSeries transient delivery mode performance was several times that of MSMQ, for every combination of parameters examined. Likewise, this superiority in IBM performance held up when the persistent delivery mode was examined. Surprisingly, even when operating at a disadvantage running ***inside*** syncpoint, MQSeries persistent performance was generally better than that of MSMQ running ***outside*** syncpoint. When combined with the results from [2], in which MQSeries express performance was several times that of MSMQ, for every combination of parameters examined (including all message sizes), it affirms the performance superiority of MQSeries over MSMQ for all delivery mode options.

Furthermore, as was the case for its express delivery mode (as reported in [2]), the performance of MSMQ's persistent delivery mode was unstable as the response queue size increased. This very serious performance problem was understandably avoided by Microsoft in its recently published (NSTL) benchmark report (just as it was avoided in its earlier 1998 report). Such problems compromise MSMQ's ability to effectively scale for customer environments supporting a large number of clients. Furthermore, because MSMQ can quite easily consume the server's processor, as it inefficiently searches for a given client's messages in its response queue, other key applications can be left idly waiting for the processor to become available. The overall effect is likely to be poor performance of other simultaneously running applications. This is a cost that most e-businesses cannot afford to pay.

In contrast, MQSeries performance was stable as the response queue size increased (for all delivery modes). Therefore, MQSeries will scale effectively as the user population grows, without the catastrophic collapse in performance that was observed for MSMQ. This scalable high performance allows your server resources to be used for what they were originally intended, i.e., to provide your users and/or customers with applications and services that run smoothly and continuously, with excellent response times - for any delivery mode option chosen.

REFERENCES

[1] **Advanced Messaging Applications with MSMQ and MQSeries,** Rhys Lewis, QUE Professional (Macmillan), Indianapolis IN, 1999.

[2] "A performance comparison of IBM MQSeries 5.1 and Microsoft Message Queue (MSMQ) 2.0 on Windows 2000: Express delivery mode", March 2000. see http://www-4.ibm.com/software/ts/mqseries/library/articles/MQperf.pdf

[3] "MQSeries for Window NT - V5.1: Capacity planning guidance", Tim Pickrell, Edition 1.0, February 8, 2000.

[4] Getting the most out of MQSeries", Richard G. Nikula, BMC Software White Paper, 2000, see http://www.bmc.com/products/whitepaper.html

[5] "Optimizing Windows NT Server performance in a Microsoft Message Queue Server environment", Microsoft TechNet White Paper, January 17, 2000 (last updated), see http://www.microsoft.com/TechNet/winnt/msmqperf.asp .

[6] "HOWTO: Optimize MSMQ performance", Microsoft product Support Services article ID Q19942B, January 23, 1999 (last reviewed), see http://support.microsoft.com/support/kb/articles/Q199/4/28.ASP

[7] "Performance evaluation of Microsoft Messaging Queue, IBM MQSeries and MQBridge", NSTL Final Report for Microsoft Corporation, May 11, 2000, see http://www.microsoft.com/windows2000/guide/platform/performance/reports/msmq.asp

NOTES

[1] Trademarks of Microsoft Corp.

[2] Trademarks of IBM Corp.

[3] Trademarks of Intel Corp.

[4] Trademarks of Black Box Corp.