A PERFORMANCE COMPARISON OF IBM MQSERIES 5.2 AND MICROSOFT MESSAGE QUEUE (MSMQ) 2.0 ON WINDOWS 2000

EXPRESS AND PERSISTENT DELIVERY MODES

TR 29.3410

March 2001

Andrew Rindos
Mitchell Loeb
Alan Kittel
Steven Woolet
Todd Shanaberger

IBM SWG Competitive Technical Assessment
Research Triangle Park, NC

Peter Toghill
Andrew Hickson

MQSeries Development
Hursley, UK

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® **5.2** and **Microsoft® Message Queue (MSMQ) 2.0**, using the express and persistent delivery modes. (This paper describes the significant performance improvements provided by MQSeries 5.2, and represents an update to the previously published papers **"A PERFORMANCE COMPARISON OF IBM MQSERIES 5.1 AND MICROSOFT MESSAGE QUEUE (MSMQ) 2.0 ON WINDOWS 2000: EXPRESS DELIVERY MODE"** and **"A PERFORMANCE COMPARISON OF IBM MQSERIES 5.1 AND MICROSOFT MESSAGE QUEUE (MSMQ) 2.0 ON WINDOWS 2000: TRANSACTIONAL AND PERSISTENT DELIVERY MODES"**.) 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, different number of process threads). As was the case with MQSeries 5.1, using a test setup and procedures that stressed the entire client-server response path (and were therefore representative of an actual customer environment), MQSeries 5.2 once again significantly outperformed MSMQ.*

ITIRC KEY WORDS

Performance

Benchmarking

MQSeries

Message Queuing

MSMQ

Windows 2000

Express

Persistent

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 (possibly) 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 (possibly) incompatible programs. This allows the programmer to utilize the broadest possible array of advanced application software in the 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 (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® MQSeries® and **Microsoft® Message Queue (MSMQ)**, two major message queuing software products for the **Windows® 2000** operating system on the market today, provide support for both express and persistent modes of message delivery. The express delivery mode provides the best performance (in terms of messages per second and/or throughput), and was previously studied for MQSeries 5.1 and MSMQ 2.0 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, which was previously studied for MQSeries 5.1 and MSMQ 2.0 in [3]. 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.

Bundling a sequence of message queuing operations *inside syncpoint* encapsulates them within 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 a particular delivery mode operating inside syncpoint is usually significantly less than that of operating outside syncpoint.

The persistent delivery mode operating inside syncpoint is 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.). Consequently, this mode of message delivery needs to provide excellent performance and scalability under realistic testing conditions, if it is 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 express (outside syncpoint) and persistent (inside syncpoint) 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, number of preloaded messages (in the server response queue) and number of process threads per server. 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.

For both applications, great care was taken to individually tune the systems in accordance with performance recommendations made by IBM [4], [5] and Microsoft [6], [7]. 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., [6], [8]). 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 express and persistent delivery modes, IBM MQSeries 5.2 significantly outperformed MSMQ 2.0 for every combination of system parameters examined. MQSeries overall peak server performance was up to 5 better than that of MSMQ. As shown later in the paper, MQSeries single client peak performance was nearly 14 times better than that of MSMQ (for both delivery modes, for a 1024 byte message).

As Table 1 below shows, MQSeries 5.2 express delivery mode peak overall performance was 5 times better than that provided by MSMQ 2.0, for one of the more frequently occurring (and therefore important) message sizes, i.e., 1024 bytes. Similarly, MQSeries 5.2 persistent delivery mode performance was 58% better than that provided by MSMQ 2.0. Additionally (as demonstrated later in the report), single client express and persistent delivery mode performance of MQSeries for a 1024 byte message was, in both cases, 14 times that of MSMQ. This MQSeries performance superiority was in fact maintained across all message sizes. Furthermore, 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 express and persistent delivery mode performance was substantially greater than that of MSMQ.

Please note that the persistent delivery mode throughputs were obtained using a disk architecture specifically defined by Microsoft MSMQ tuning guidelines [6] and [7] to obtain their best possible performance. In a subsequent paper, we will report the still larger performance differences obtained for the persistent delivery mode, using a far less expensive disk architecture for which IBM MQSeries performance has been optimized (i.e., greater performance at less the cost).

Table 1. Peak Performance

	Express Delivery Mode (outside syncpoint)		Persistent Delivery Mode (outside syncpoint)	
Message Size (Bytes)	Throughput (Mbps)		Throughput (Mbps)	
	IBM MQSeries	MSMQ	IBM MQSeries	MSMQ
1,024	35.81	7.18	4.71	2.99
2,048	61.52	16.23	8.98	5.47
5,120	108.05	32.99	19.88	11.32
10,240	135.43	41.88	31.41	17.38
20,480	154.1	53.13	41.58	25.3
30,720	164.26	58.14	45.87	29.57

Table 1. MQSeries and MSMQ express and persistent delivery mode performance (measured in Megabits per second or Mbps, one-way) for 1024 through 30,720 byte message sizes.

MQSeries express and persistent delivery mode performance significantly exceeded that of MSMQ for every situation examined. The performance numbers reported above represent the overall peak performance throughputs obtained over all examined combinations of number of clients, number of threads per client and number of process threads, for a given message size. (The peaks always occurred within the parameter combinations examined.).

(2) For the express delivery mode, IBM MQSeries 5.2 performance was extremely robust as the server response queue was preloaded with an increasing number of initial messages. In sharp contrast, MSMQ 2.0 performance effectively collapsed as the initial response queue size increased. For 1,000 preloaded messages, MQSeries performance was 264 times better than that of MSMQ.

For the persistent delivery mode, MSMQ performance effectively collapsed when multiple clients shared the same server response queue, regardless of whether or not that queue was preloaded. MQSeries persistent performance, on the other hand, was relatively unaffected by sharing a single (even preloaded) response queue.

Preloading the server response 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 2 below shows, for the express delivery mode (operating outside syncpoint), MQSeries 5.2 performance remained relatively unchanged as the initial request queue size (Q_0) was increased. In contrast, MSMQ 2.0 performance decreased for even a small number of preloaded messages. For Q_0 = 200, MSMQ performance dropped from its peak performance of 662 messages per second (mps) to merely 41 mps (a reduction of over 90%). At Q_0 = 1000 messages, MSMQ performance effectively collapsed to 9 mps. In contrast, the corresponding MQSeries performance was 2,275 mps.

For the persistent delivery mode (operating inside syncpoint), the simple act of having multiple clients share a single server response queue caused MSMQ performance to effectively collapse (regardless of whether or not that queue was preloaded). For 6 or more clients, MSMQ peak persistent performance ranged between 9-15 mps. In contrast, MQSeries persistent performance was relatively unaffected by sharing a single (even preloaded) response queue, yielding over 500 mps for 6 or more clients, for various values of Q_0.

Table 2. Messages/sec vs. Initial Request Queue Size

Initial Request Queue Size	Messages/sec (mps)		Ratio of MQSeries to MSMQ mps
(Q_0, in Messages)	IBM MQSeries	MSMQ	MQSeries/MSMQ
0	3,277	662	5
50	3,265	137	24
100	3,255	78	42
150	3,226	60	60
200	3,219	41	78
300	3,221	28	115
500	3,067	17	178
1,000	2,275	9	264

Table 2. MQSeries and MSMQ express performance (measured in messages per second or mps) 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. The performance advantage of MQSeries over MSMQ increased as the number of preloaded messages increased (tracking the collapse of MSMQ performance). 1024 byte messages were used throughout, with 6 clients running a single thread each. This was the parameter combination that maximized MSMQ peak server performance in [2] and [3].

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. The MSMQ setup was tuned in accordance with the recommendations given by Microsoft in [5] and [6].

Previously published benchmark results

Microsoft previously published performance numbers for MSMQ in [6] and [8] using the benchmark tests MSMQBench and MQSRBench. However, as we discussed in [2] and [3], the benchmark tests by themselves, and as run in the NSTL tests, represented a trivial scenario that had little relevance to a real world message queuing application. The extremely large performance numbers they obtained using these benchmarks should have immediately alerted their testers that they were actually measuring low-level, primitive functions. These low-level functions had nothing to do with the higher level bottlenecks that would actually define the performance of their message queuing software. In this study, the server was far more powerful (containing eight 550 MHz Pentium® III Xeon™ processors vs. their original servers with at most two 200 MHz Pentium Pro™ processors) and configured with the complex multiple disk architecture that Microsoft defined to be optimal for MSMQ performance. However, even when using this vastly more powerful server, the MSMQ performance numbers obtained (and reported herein) are far below those previously reported by Microsoft (verifying the unrealistic nature of the MSMQBench and MQSRBench benchmark programs).

In fact, because the disk architecture has been optimized to favor MSMQ (in accordance with Microsoft tuning guidelines [6] and [7]), the MSMQ persistent delivery mode throughputs have in fact improved significantly over those reported in [3]. However, even given this advantage, as the data will show, IBM MQSeries express and persistent delivery mode performance remained significantly better than that of MSMQ, for every combination of system parameters examined. (In a subsequent paper, we will report the much larger performance superiority of MQSeries persistent delivery mode over that of MSMQ, using a far less expensive disk architecture for which IBM MQSeries performance has been optimized, i.e., greater performance at less the cost).

In all these previous analyses, Microsoft 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. As Table 2 above shows (which will be amplified by the data presented below, and was reported originally in [2] and [3]), under test conditions that mimicked these very real customer environments, MSMQ performance effectively collapsed, even under very mild server congestion. 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. 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 clearing benefits provided by the test procedures used in our study. This seriously compromises the ability of MSMQ to scale to the larger client population sizes often found in actual customer environments. Furthermore, MSMQ excessively 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.

In their previously published studies, Microsoft was clearly interested in avoiding MSMQ 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.

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 [6], 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 [8] are given in [3].

An extensive analysis of the effects of message size on express delivery mode performance is given in [2]. Similarly, an extensive analysis of transactional and persistent delivery mode performance is given in [3]. 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 and 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 February 28, 2001.

MESSAGE QUEUING PERFORMANCE

TEST CONFIGURATION AND PROCEDURES

Configurations

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 twenty IBM Personal Computer 300PL client workstations (667 MHz Intel® Pentium® III processor; 32-bit, 33 MHz PCI bus, 256 MB RAM), each with an IBM Etherjet™ PCI (Peripheral Component Interconnect) adapter (100 Mbps Ethernet, full-duplex mode, based on Intel 82559 technology). The server was an IBM Netfinity® 8500R (8 x 550 MHz Pentium III Xeon processors; 32- and 64-bit, 33 MHz PCI buses; 1 GB RAM; 18.2 GB SCSI hard drive; up to 9 x 9 GB (8.3 GB useable) hardware-striped RAID0 drives configured into 2 logical disks, IBM ServeRaid™-3HB Ultra2 SCSI Controller), with an IBM Netfinity Gigabit Ethernet SX adapter (1 Gbps, full-duplex, 64-bit, based on Intel 82542 technology). The operating system on all clients was Windows 2000 Professional, while that on the server was Windows 2000 Advanced Server. The clients and server were interconnected via an Intel Express™ 510T Ethernet switch (using multimode optical fiber to the server, and twisted pair copper cabling to the clients). The networking hardware (adapters, switch) were chosen to guarantee that the network was not the system bottleneck in this study (providing more than enough bandwidth).

The message queuing applications examined were MQSeries version 5.2 and MSMQ version 2.0 (the latter provided with the Windows 2000 operating system). Message queuing clients were configured as dependent (and therefore, messages were not prestored on the client). Express (outside syncpoint) and persistent (inside syncpoint on both the server and clients) delivery mode performance is reported in this study. We will discuss the differences between express and persistent delivery modes, and between inside and outside syncpoint, in the relevant sections that follow. Because the persistent delivery mode was run inside syncpoint (on both the server and clients), the MSMQ queues were necessarily defined as transactional for those specific tests.

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. Two 8-way Cybex® AutoView™ Commander™ switches were linked together in the master/slave configuration, effectively yielding a 16-way switch. It allowed all client and server machines to be controlled using a pair of monitors, etc.

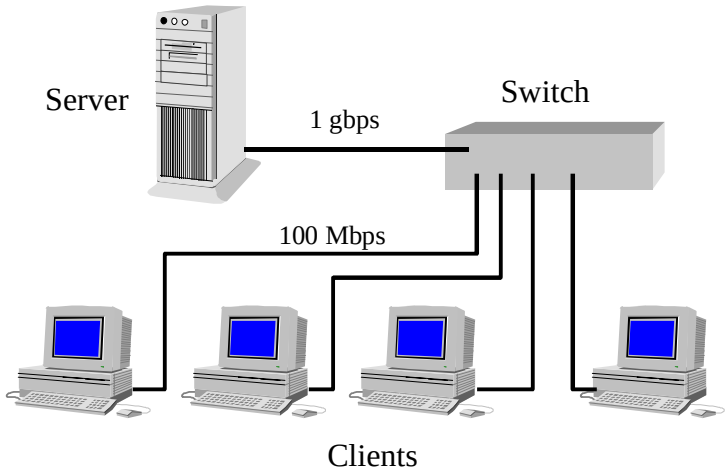

FIGURE 1. Test Setup

Up to 20 clients (667 MHz Pentium III; 100 Mbps full-duplex Ethernet; Windows 2000 Professional) connected to a server (8 x 550 MHz Pentium III Xeon; 1 Gbps full-duplex Ethernet; Windows 2000 Advanced Server) via an Intel Express 510T Ethernet switch, using TCP/IP.

Performance Tuning

MSMQ performance was tuned in accordance with the recommendations given in [6] and [7]. As prescribed, auditing was deactivated, applications performance optimization was disabled and the size of the paging file was increased to the recommended size (on the server and all clients).

IBM MQSeries performance was tuned in accordance with the recommendations given in [4] and [5]. As prescribed, MQIBindType was set to FASTPATH using the MQSeries Services interface (on the server). The FASTPATH option allows an application process (in this case, the "internal" programs represented by the channels/listeners) to avoid a very costly process switch, by permitting it to directly update the message queue state. Normally, in order to guarantee MQSeries integrity, the message queue state is isolated from all user processes, requiring a separate process (under MQSeries control) to be called to update the queue appropriately when an MQPUT or MQGET command is issued.

In addition, the MaxChannels and MaxActiveChannels were set to a value greater than the product of the largest number of clients (here, 20) 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 (on the server and all clients), since it yielded the best MQSeries performance numbers.

When testing the express delivery mode (outside syncpoint), all message queuing code, queues and logs were stored on the same physical drive (and logical disk), i.e., the server's standard 18.2 GB SCSI drive. Because messages are not stored to disk for the express delivery mode, the use of a single standard physical disk did not represent a potential performance bottleneck. However, when testing the persistent delivery mode (inside syncpoint), the code, queues and logs were distributed across the following three logical disks, representing distinct physical drives: (a) disk C, residing on a single standard 18.2 GB SCSI drive; (b) disk F, logically created using 6 hardware-striped 8.3 GB RAID0 drives (50 GB total); and (c) disk G, created using 3 hardware-striped 8.3 GB RAID0 drives (25 GB total). For MQSeries, the program files were stored on disk C, the queue managers on disk F, and the logs on disk G. For MSMQ, the program files and message logs were stored on disk C, the messages on disk F, and the transaction logs on disk G (in precise accordance with the tuning recommendations provided by Microsoft in [5] and [6]). During testing, all three disks were operating well below saturation (typically at less than 30% utilization). Our automated test code was on a small (one GB) logical disk (disk D), which was physically on the same SCSI drive as disk C above. It was only active at the setup of (and not during) any testing. (Note: The individual physical RAID0 drives were physically 9 GB, but after Windows 2000 formatting, 8.3 GB remained as usable.)

All of the above performance tuning enhancements were identical to those used in [2] and [3]. The following additional test scenarios were examined and/or test procedure modifications were made (to both MSMQ and MQSeries, as appropriate), in order to increase performance over and above what was reported in [2] and [3] by maximizing the utilization of all 8 server processors:

(1) When the response queue was not preloaded (described below), each client was serviced by a dedicated request/response queue pair, each of which was in turn serviced by one or more copies of the queue manager.

(2) For both MQSeries and MSMQ, the response queue handles were cached so that the applications avoided reopening and then closing the response queues with every message processed.

(3) The buffer size requested when “putting” or “getting” a message was equivalent to the actual message size used (plus anticipated overhead).

(4) For MQSeries, in addition to configuring the channel(s) and listener(s) to use the FASTPATH option (as dictated by setting MQIBindType to FASTPATH; stated above), the server applications were similarly FASTPATH-bound to the server queue manager. (This was accomplished by connecting the queue manager using the MQCONNX verb, rather than MQCONN, with the MQCNO_FASTPATH_BINDING option. The rules for FASTPATH applications are stated in the application programming reference, or APRM.)

Test 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 (essentially resulting in the transmission of two identical messages: one from the client to the server, followed by one from the server 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. All messages per second and throughputs reported in this paper represent one-way performance measures. Two-way performance measures can simply be obtained by multiplying all reported values by 2, since a

given message was always transmitted twice, in opposite directions, as stated above. 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 to drive the server, two situations were examined: (1) when the response queue was not preloaded prior to testing (see explanation below), each client was assigned a dedicated pair of request and response queues (i.e., for ***m*** clients driving the server, ***m*** separate request queues and ***m*** separate response queues were used); (2) when the response queue was preloaded prior to testing, all clients shared a single request and response queue. The performance of this latter scenario was deemed extremely important in assessing the scalability of the two applications.

In order to overcome inherent bottlenecks associated with single process threads, the effect on performance of running multiple process threads was examined for two different important processes: (1) the message queuing server application, which moved a client message from the request to the response queue; and (2) the queue manager, which provided overall management of the queues associated with it (including, most importantly, writing a message to disk for the persistent delivery mode).

A given pair of request/response queues could be managed by one or more dedicated copies of the message queuing server application. For example, when it is reported below that ***q*** copies of the server application were servicing ***m*** clients (each with its own dedicated request/response queue pair), ***q x m*** server application copies were actually running on the server. (However, as reported later, additional server application copies were not observed to improve peak performance.)

When the persistent delivery mode (inside syncpoint) was examined, the ability of the queue manager to access and write messages to disk represented a significant potential system bottleneck. It was observed for MQSeries that both processor and disk utilizations were very low when the peak performance for a single queue manager was obtained. This bottleneck could be overcome (with a subsequent increase in messages per second and/or throughput) by running multiple queue managers, thereby increasing the efficiency with which messages were written to disk.

For MSMQ, with its significantly less efficient code, all 8 server processors were very close to saturation when single-queue-manager peak performance was achieved. Since the server itself is defined as the queue manager for MSMQ, running multiple queue managers was not an option available to MSMQ. However, near-saturation of the server processors suggested that running multiple queue managers would not have improved its performance anyway.

When multiple queue managers were examined, a given client was assigned to a specific queue manager. In those cases where the number of clients exceeded the number of queue managers, the clients were evenly distributed (as best as possible) among the queue mangers. Each queue manager maintained a dedicated request/response queue pair for each client assigned to it.

All (multiple) clients 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 Administrative Tools) was used to verify that all 8 server Pentium III Xeon processors were being utilized (and for example, saturated when 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 3 minutes was optimal in reducing variations in measurements between runs under identical circumstances. They also suggested a 1 minute waiting interval prior to data collection. These values were used for all tests whose results are summarized below.

TEST RESULTS

EXPRESS DELIVERY MODE (OUTSIDE SYNCPOINT)

The express delivery mode is typically used in applications that require the best possible performance, since (unlike the persistent delivery mode) messages are not stored to disk. For these same performance reasons, this mode is typically run outside syncpoint (i.e., the message queuing operations are not encapsulated in a transaction). Therefore, the performance of the express delivery mode for both message queuing applications, operating outside syncpoint, was analyzed below.

Effects of Varying the Number of Clients and Threads
Express Delivery Mode (inside syncpoint); 1024 byte Message Size

In this section, 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 that can be processed by the client and server, using the express delivery mode (outside syncpoint) for either IBM MQSeries 5.2 or MSMQ (Microsoft Message Queue) 2.0. 1024 bytes is a frequently occurring (small) message size, and therefore its performance is carefully examined below.

The effects on performance of a wide variety of message sizes are examined in the later section **Effects of Varying Message Size (Express Throughput): *Express Delivery Mode (outside syncpoint).***

Figure 2 compares the performance of IBM MQSeries 5.2 vs. Microsoft MSMQ 2.0, using overall server messages per second (mps) as a function of the number of simultaneous threads per client, for 1 through 4 clients, as well as 15 clients for MSMQ only. (MSMQ required 15 clients simultaneously driving its server in order to achieve its peak performance. Because of the far greater efficiency of IBM's MQSeries clients, peak server performance was achieved using only 3 clients.) As stated above, all data is for a message size of 1024 bytes, using express delivery mode. Each client had a dedicated request/response pair of queues, with each pair serviced by a dedicated copy of the message queuing server application.

As Figure 2 demonstrates, at a message length of 1024 bytes, IBM MQSeries peak performance for this system was 4,372 mps. This was achieved using 3 clients each running 8 threads. MQSeries single client/single thread response time was 1.2 msec. In contrast, MSMQ server peak performance for this system was only 876 mps (i.e., only 1/5 that of IBM MQSeries). This was achieved using 15 clients each running 3 threads. MSMQ single client/single thread response time was 8.1 msec.

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

It should be noted that the effect of varying the number of copies of the message queuing server application associated with each client request/response pair is examined in the later section *Effects of increasing the number of server application copies*. However, as that section will explain, the overall peak server performance was achieved using a single message queuing server application copy

Effects of increasing the number of clients: IBM MQSeries single client peak performance was 2,389 mps (achieved running 6 threads). Since this was more than half the overall server peak performance, the addition of a second client increased performance by 70%. Overall server performance peaked with the addition of a third client, as a result of a small increase in

performance. Additional clients yielded no increase and/or a very small decrease (due to increased server congestion) in MQSeries performance.

MSMQ single client peak performance was quite low, i.e., 174 mps (which was only 7% of the 2,389 mps achieved by a single IBM MQSeries client). Since this was approximately 1/5 of the capacity of the MSMQ server, overall messages per second increased linearly with added clients, up to 5 clients. Above 5 clients, slight marginal increases in peak performance were observed, up to 15 clients. However peak performance actually declined for 16 or more clients (once again presumably due to increased server congestion).

Effects of increasing the number of threads per client: For up to 9 clients, MQSeries performance increased and then plateaued as more threads per client were initially added. As the number of clients increased, the number of threads per client required to achieve the peak and/or plateau onset steadily decreased. For 10 clients and above, additional threads per client yielded a modest decrease in performance. At that point, additional threads per client only added more overhead time for switching between threads, with no possible benefit to overall performance, thereby reducing single client efficiency and subsequent mps and throughput (see the discussion below).

As was the case with MQSeries, MSMQ performance increased and then plateaued as more threads per client were initially added, with a similar decrease in performance for a large number of clients and/or threads (see, e.g., the MSMQ curve for 15 clients shown in Figure 2, exhibiting a noticeable decline in performance following the achievement of its overall peak,). While (as stated above) IBM MQSeries performance also declined with additional threads per client (given a large number of clients), it remained significantly higher than that of MSMQ throughout

In general, for both MQSeries and MSMQ, a single thread's performance was gated by its round trip response time (i.e., it had to wait for the return of a transmitted message before sending the next one). Therefore, the only way to increase single client performance was to add more threads. The point at which performance peaked represented the client's capacity to execute client code (a function of the client processor capacity and the code pathlengths).

On the server side, once peak server performance had been achieved, overall server messages per second decreased as additional clients and/or threads per client were added. This was 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).

It should be noted that all 8 server processors were effectively saturated for both message queuing applications when overall peak performance was obtained for the express delivery mode (and remained saturated as additional clients and/or threads were added). This was also true for the persistent delivery mode peak performance numbers (reported later).

Effects of increasing the number of server application copies: As mentioned previously, the test paradigm included the ability to increase the number of server application copies (which, e.g., move a message from the request queue, after it has been first received by a client, to the response queue, for subsequent retrieval by the same client). This was done in anticipation of requiring multiple application threads to saturate the capacity of the server's 8 powerful processors. However, it was observed that the overall peak performance was easily obtained using only a single server application copy per request/response queue pair assigned to each client. The addition of more server application copies per request/response queue pair yielded no increase in peak performance (and for a large number of additional server application copies, yielded a very slight decrease in performance, due to additional overhead experienced by the processor in switching between processes).

This behavior was also true for the persistent delivery mode. However, as will be shown later, running multiple queue mangers proved to be extremely important in overcoming the disk access bottleneck and yielding significantly higher performance.

The performance numbers reported herein for both MQSeries and MSMQ were significantly greater than those reported in [2] and [3] as a result of: (1) the significantly more powerful client and server processors; (2) a dedicated request/response queue pair for each client; (3) a dedicated copy of the message queuing server application associated with each request/response queue pair; and (4) additional performance tuning as described in **TEST CONFIGURATION AND PROCEDURES**.

(It should be noted that the exact combination of number of clients and threads per client at which peak messages per second and/or 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 using the express delivery mode (outside syncpoint), MQSeries 5.2 provided almost 14 times the single client performance of that of MSMQ 2.0. Similarly, MQSeries 5.2 provided 5 times the overall server capacity of that provided by MSMQ 2.0.

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 will be far better performance of all the other important applications that are running simultaneously with message queuing.

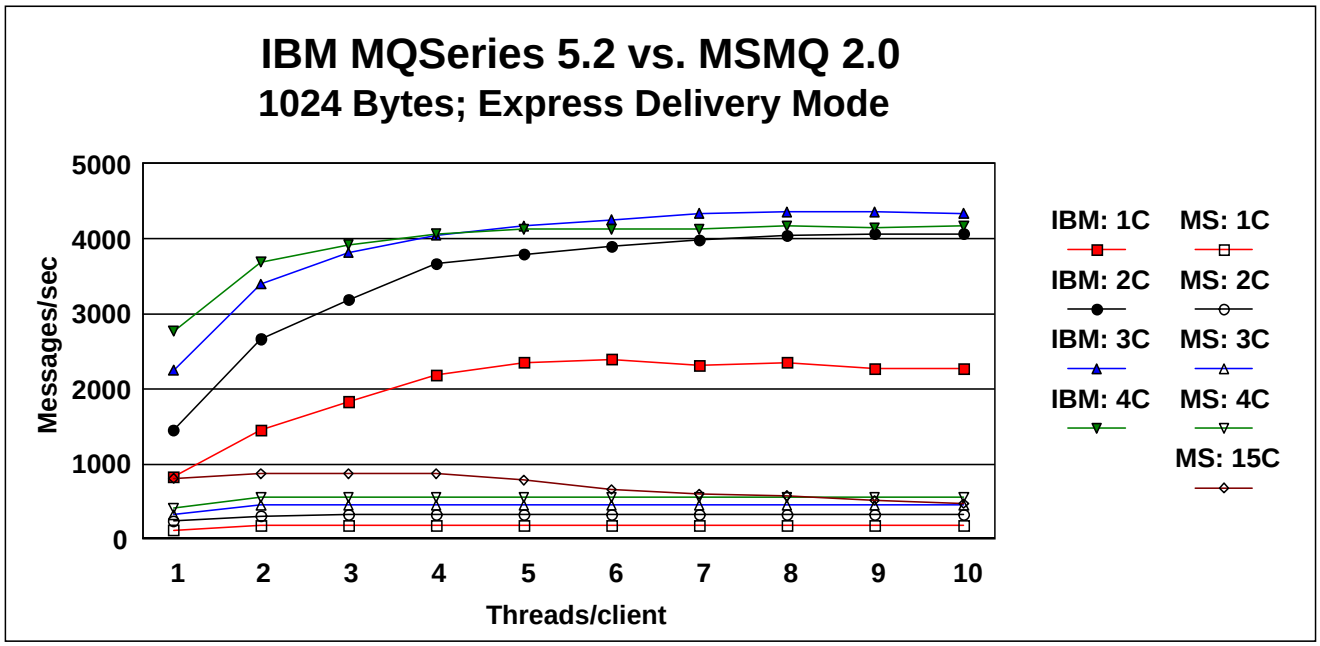

FIGURE 2 IBM MQSERIES 5.2 vs. MSMQ 2.0 MESSAGES/SECOND vs. THREADS/CLIENT EXPRESS DELIVERY MODE 1024 BYTE MESSAGE SIZE

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

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

MQSeries single client peak performance = 2,389 mps
MSMQ single client peak performance = 174 mps

MQSeries overall peak server performance = 4,372 mps
MSMQ overall peak server performance = 876 mps

MQSeries single client/single thread response time = 0.42 msec
MSMQ single client/single thread response time = 5.75 msec

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

Note: MSMQ mps vs. number of threads/client for fifteen clients is plotted above, because that was the number of clients required to finally obtain MSMQ peak performance. In contrast, MQSeries peak performance was easily obtained with only three clients.

Effects of Preloading the Server Response Queue

Express Delivery Mode (inside syncpoint); 1024 byte Message Size

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 [2] and [3], only a single pair of request and response queues were shared by all clients. In those studies, it was therefore 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. These messages were not retrieved in subsequent runs (e.g., messages left by client number 8, used only when 8 clients were tested, could not be retrieved during later tests of 1 through 7 clients). Therefore, unretrieved messages slowly accumulated in the response queue over multiple test runs. 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, all server response and request queues were systematically cleared prior to each performance test. This was done for all testing reported in the section **Effects of Varying the Number of Clients and Threads** above, as well as in the section **Effects of Varying Message Size (Throughput)** below. 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, as in [2] and [3], only a single pair of request and response queues were shared by all clients. 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). The parameter pair of number of clients and threads per client that yielded the peak performance for MSMQ at this message size in [2] and [3] (i.e., when a single request/response queue pair was shared by all clients) was 6 clients running a single thread each. This combination was therefore used throughout. The response queue was preloaded with 0, 50, 100, 150, 200, 300, 500 and 1000 messages.

Table 3 and Figure 3 summarize the effects of this preloading on MQSeries and MSMQ performance, presenting messages per second (mps) as a function of the initial response queue size (in messages). Table 3 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, **MSMQ performance was devastated by even a very small number of initial messages in the response queue**. For an initial response queue size of 50 messages,

MSMQ performance dropped to 137 mps, a decrease of nearly 80% from its peak of 662 mps (the latter achieved when no messages were preloaded). **In contrast, MQSeries performance remained relatively unaffected,** yielding 3,265 mps (as compared to the unloaded value of 3,277 mps). When the initial response queue size was preloaded with 1,000 messages, MSMQ performance virtually collapsed, yielding barely 9 mps. In contrast, MQSeries performance was 2,275 mps.

Table 3. Messages/sec vs. Initial Request Queue Size Express Delivery Mode (outside syncpoint)

Initial Request Queue Size (Messages)	Messages per second (mps)		Ratio of MQSeries to MSMQ mps
	IBM MQSeries	MSMQ	MQSeries/MSMQ
0	3,277	662	5
50	3,265	137	24
100	3,255	78	42
150	3,226	60	60
200	3,219	41	78
300	3,221	28	115
500	3,067	17	178
1,000	2,275	9	264

Table 3. MQSeries and MSMQ performance (measured in messages per second or mps) for the express delivery mode (operating outside syncpoint) 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 264 times better than that of MSMQ, or 26,300 % 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 6 clients running a single thread each. This was the parameter combination that maximized MSMQ peak server performance in [2] and [3].

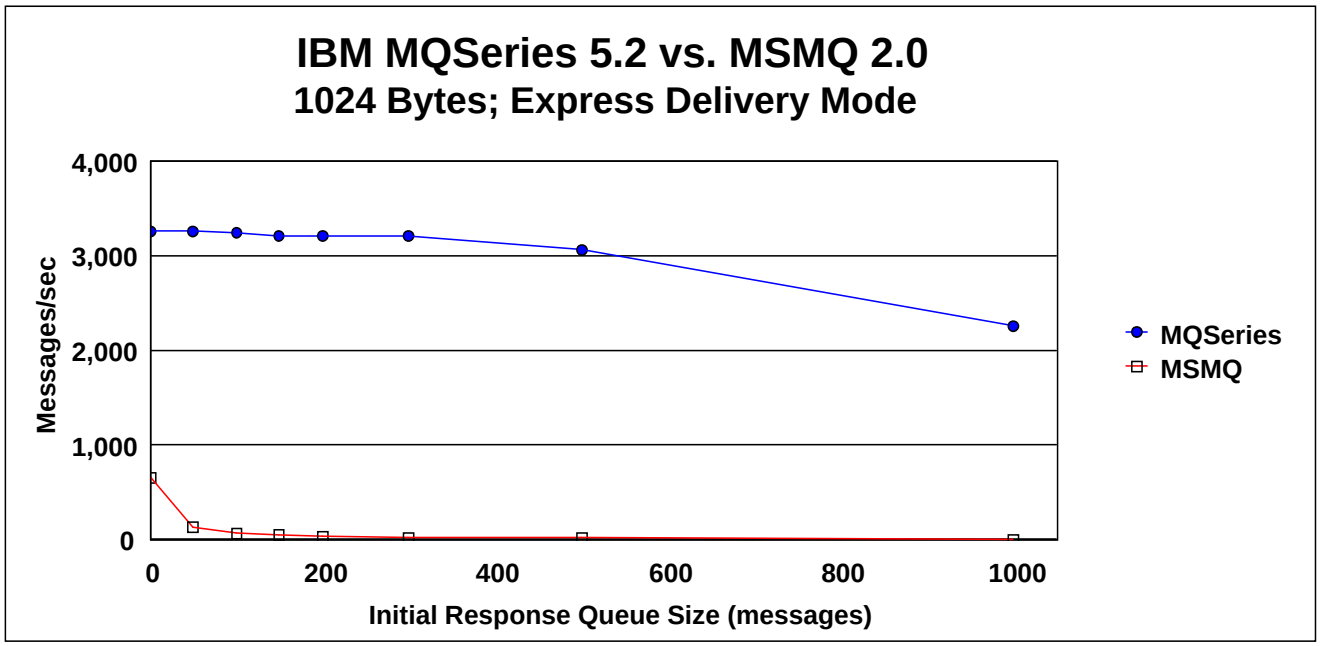

FIGURE 3 IBM MQSERIES 5.2 vs. MSMQ 2.0
MESSAGES/SECOND vs. INITIAL RESPONSE QUEUE SIZE
6 CLIENTS, 1 THREAD/CLIENT
EXPRESS DELIVERY MODE
1024 BYTE MESSAGE SIZE

IBM MQSeries and MSMQ messages/second (mps) vs. initial response queue size, using six clients running a single thread each, a 1024 byte message size and express delivery mode.

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

Q_0 = 0 messages	**MQSeries performance =**	**3,277 mps**
	MSMQ performance =	**662 mps**
Q_0 = 100 messages	**MQSeries performance =**	**3,255 mps**
	MSMQ performance =	**78 mps**
Q_0 = 1,000 messages	**MQSeries performance =**	**2,275 mps**
	MSMQ performance =	**9 mps**

Effects of Varying Message Size (Throughput)

Express Delivery Mode (inside syncpoint)

In this section, we rigorously analyze the effects of varying the message size on the overall throughput (measured in Megabits per second, or Mbps) that can be processed by the server, using either IBM MQSeries 5.2 or MSMQ 2.0. Throughput was calculated here as the number of messages processed per second (mps) times the message size. (All throughputs reported throughout this paper therefore represent one-way throughputs. Note that since a given message was transmitted twice for each successfully processed message, throughputs as observed by the system network switch and adapters, server and client buses, etc, were actually 2 times the reported numbers below.) As expected, mps decreased with increasing message size. Throughput, on the other hand, actually increased with increasing message size (as the data below will show). Discussions of why mps increase, while throughputs decrease, with increasing message size are given in [2] and [3].

The message sizes examined were 1024, 2048, 5120, 10240, 20480 and 30720 bytes. Detailed performance measurements were made for each of these message sizes. These measurements were identical in nature to those taken in the section **Effects of Varying the Number of Clients and Threads** above, i.e., using 1 through as many as 20 clients, each running 1 through (in this case) as many as 20 threads. The comparative results were the same as those reported in that previous section, i.e., **for every message size examined, IBM MQSeries outperformed MSMQ for every combination of number of simultaneous clients and threads per client examined**.

Figure 4 represents overall peak server throughputs (Megabits per second, or Mbps) for IBM MQSeries and MSMQ. All data was collected using express delivery mode, running outside syncpoint. For a given message size, overall peak server throughput was defined to be the maximum throughput measured over every examined combination of number of clients and threads per client (i.e., over 1 through up to 20 clients, running 1 through up to 20 threads per client). Peak throughputs were always achieved over this range of combinations, with the number of clients and/or threads per client increased (as necessary) to determine the overall peaks.

Table 4 summarizes the individual peak throughput values obtained for each message size examined. Table 4 also presents the ratio of MQSeries to MSMQ throughput, indicating the number of times (not percent) that MQSeries performance was better than that of MSMQ.

Figure 4 and Table 4 demonstrate that IBM MQSeries and MSMQ overall peak server throughput increased with increasing message size. They also demonstrate that **overall peak server performance for IBM MQSeries was significantly greater than that of MSMQ for every message size examined**. MQSeries consistently provided several times the peak server performance of that offered by MSMQ.

Table 4. Throughput vs. Message Size
Express Delivery Mode (outside syncpoint)

Message Size (Bytes)	Throughput (Mbps)		Ratio of MQSeries to MSMQ mps
	IBM MQSeries	MSMQ	MQSeries/MSMQ
1,024	35.82	7.18	5
2,048	61.52	16.23	3.8
5,120	108.05	32.99	3.3
10,240	135.43	41.88	3.2
20,480	154.1	53.13	2.9
30,720	164.26	58.14	2.8

Table 4. IBM MQSeries and MSMQ overall peak server throughputs obtained for a given message size (measured in Megabits per second, or Mbps), for the express delivery mode (operating outside syncpoint).

MQSeries performance substantially exceeded that of MSMQ, for every situation examined. For a given message size, overall peak server throughput was the maximum throughput achieved over all combinations of number of clients and threads per client examined (i.e., over 1 through up to 20 clients, running 1 through up to 20 threads per client). It represented the overall server capacity to process messages This maximum was always achieved over the range of combinations that were examined.

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., **for a 1024 bytes message size, MQSeries performance was 5 times better than that of MSMQ, or 400% better**).

In summary, for the express delivery mode (operating outside syncpoint), MQSeries 5.2 outperformed MSMQ 2.0 for every message size examined, providing up to 5 times the peak performance of that of MSMQ

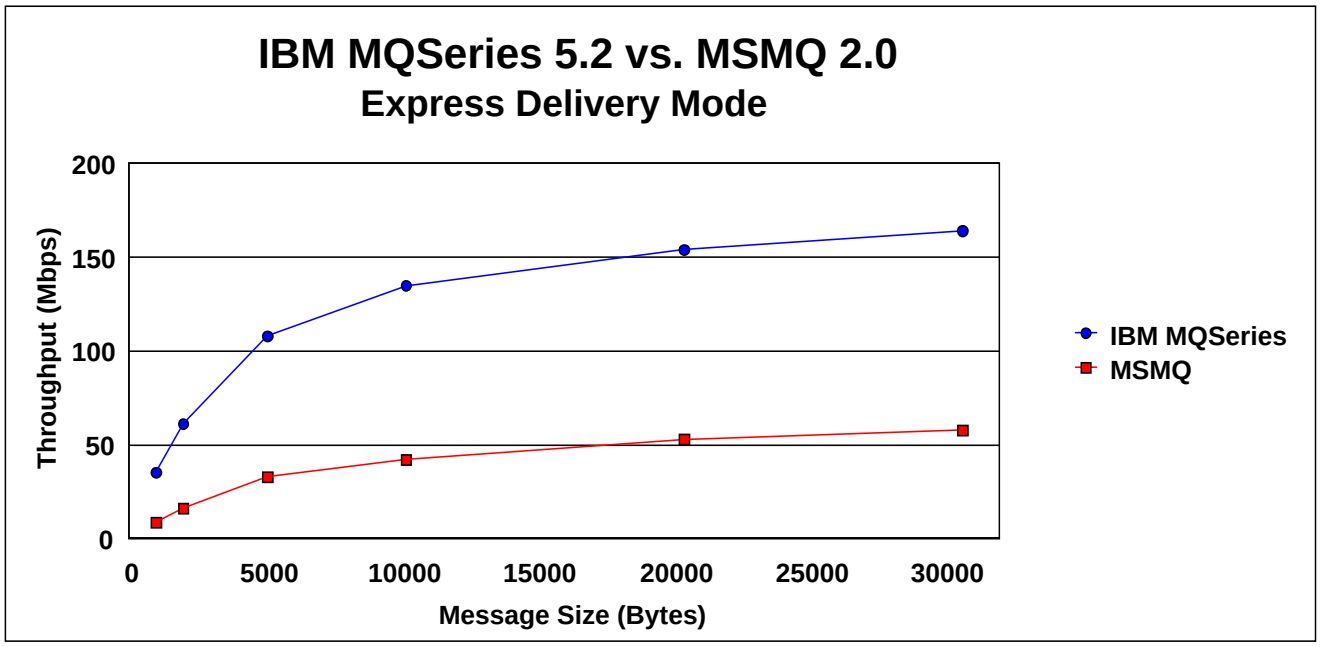

FIGURE 4 IBM MQSERIES 5.2 vs. MSMQ 2.0 THROUGHPUT vs. MESSAGE SIZE 1,024 through 30,720 BYTE MESSAGE SIZES EXPRESS DELIVERY MODE

IBM MQSeries and MSMQ overall peak server throughput (Megabits per second, or Mbps) vs. message size (1024, 2048, 5120, 10240, 20480, and 30720 bytes), using express delivery mode (outside syncpoint). The throughput plotted for each message size represents the maximum obtained over all examined combinations of number of clients and threads/ client (i.e., over 1 through up to 20 clients, running 1 through up to 20 threads/client). For all message sizes examined, the overall peak server throughput always occurred within this range of combinations.

For every message size examined, MQSeries substantially outperformed MSMQ. For both applications, throughput increased with increasing message size.

PERSISTENT DELIVERY MODE (INSIDE SYNCPOINT)

The persistent (or for MSMQ, recoverable) delivery mode is typically used in applications that must guarantee that messages sent to a given queue (on a possibly remote machine) are never lost, even, for example, when that machine (containing the target queue) suddenly crashes. Therefore, in the persistent delivery mode, all messages that are not immediately claimed by their intended recipient client are stored to disk, thereby guaranteeing recoverability in the face of any system failures.

In addition, both MQSeries and MSMQ support the ability to imbed their message queuing operations inside a transaction (i.e., they can operate inside syncpoint). 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 (and/or are reversed), and the system rolls back to its state prior to initiation of the unsuccessful transaction.

Applications that typically utilize the persistent delivery mode option running inside syncpoint include those which require a high degree of database integrity, e.g., applications that modify and maintain bank accounts. With these types of applications, procedures such as those involved in modifying account balances (that may include message queuing instructions) are encapsulated inside a transaction. This guarantees that an account is never modified unless all operations associated with that modification are successful (e.g., a deposit to a target account cannot occur without the simultaneous and successful withdrawal of the same amount from a corresponding source account). Therefore, the persistent delivery mode, operating inside syncpoint, is extremely important and commonly used in a wide variety of e-business applications. For this reason, its performance is carefully examined below.

Because of the heavy overhead associated with writing messages to disk, as well as encapsulating all client and server message queuing operations inside transactions, the performance of MQSeries and MSMQ using the persistent delivery mode operating inside syncpoint (on the server and clients) will necessarily be significantly less that obtained for the express delivery mode operating outside syncpoint. Data security is therefore provided at a necessary cost to performance.

In all of the testing done throughout this paper (as well as in [2], [3]), 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 persistent delivery mode operating inside syncpoint below, the client PUT and GET were each issued inside syncpoint, i.e., they each represented a single transaction, controlled by a final commit. (Note: The client PUT and GET were contained inside two separate transactions. If they were contained within the same transaction, the client threads issuing the requests would become deadlocked.) Likewise, the server GET/PUT commands were issued inside syncpoint, representing another single transaction. (The MSMQ queues were therefore necessarily defined to be transactional queues. MQSeries, with its greater flexibility in combining transactions with different delivery modes, does not require special queue types to be permanently defined.)

Effects of Varying the Number of Clients and Threads
Persistent Delivery Mode (outside syncpoint); 1024 byte Message Size

In this section, 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 that can be processed by the client and server, this time using the persistent delivery mode (inside syncpoint) for either IBM MQSeries 5.2 or MSMQ (Microsoft Message Queue) 2.0. As was the case for the express delivery mode, the performance of the frequently occurring 1024 bytes message size is carefully examined below.

The effects on performance of a wide variety of message sizes are examined in the later section **Effects of Varying Message Size (Throughput): *Persistent Delivery Mode (outside syncpoint***).

Figure 5 compares the performance of IBM MQSeries 5.2 vs. Microsoft MSMQ 2.0, using overall server messages per second (mps) as a function of the number of simultaneous threads per client, for 1 through 4 clients, as well as for 7 clients for MQSeries only and 11 clients for MSMQ only. (MQSeries and MSMQ required respectively 7 and 11 clients simultaneously driving the server in order to achieve their individual peak performances.) As stated above, all data is for a message size of 1024 bytes, using persistent (or in Microsoft terminology, recoverable) delivery mode. The message queuing applications were running inside syncpoint (i.e., within transactions) at both the server and clients (which for MSMQ, required the use of transactional queues). As was the case for the express delivery mode, each client had a dedicated request/response pair of queues, with each pair serviced by a dedicated copy of the message queuing server application.

As Figure 5 demonstrates, at a message length of 1024 bytes, IBM MQSeries peak performance for this system was 575 mps. This was achieved using 7 clients each running 4 threads. MQSeries single client/single thread response time was 4.7 msec. In contrast, MSMQ server peak performance for this system was only 365 mps (i.e., less than 2/3 that of IBM MQSeries). This was achieved using 11 clients each running 3 threads. MSMQ single client/single thread response time was 63.8 msec. (Single client peak performance for MQSeries was 214 mps, while that of MSMQ was only 16 mps.)

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

As was discussed in the previous section, *Effects of increasing the number of server application copies*, overall peak server performance was achieved using a single server copy (and was therefore used throughout). However, the ability to access the disk was a major bottleneck for the persistent delivery mode, which was overcome by increasing the number of simultaneously running queue managers (see the discussion in the previous section entitled ***Test Procedures***).

Effects of increasing the number of queue managers: As mentioned above, it was observed that when only a single MQSeries queue manager was running, processor and disk utilizations were low, even when peak performance numbers were obtained. In contrast, MSMQ peak performances corresponded to near saturation of all 8 server processors. Therefore, given only a single queue manager, for the same level of performance, MQSeries processor utilizations were substantially less than those observed for MSMQ (indicating that the MQSeries code was far more efficient in its usage of the processors).

MSMQ does not have the option to increase the number of queue managers (since the queue manager is defined to be the server itself). However, given the very high utilizations already experienced by the server's processors, the addition of more queue managers would not have significantly improved MSMQ performance. In contrast, the number of queue managers servicing

a set of queues can be increased for MQSeries. By utilizing additional queue managers, the system was able to more effectively access the disk (in storing messages, in accordance with the persistent delivery mode paradigm), with a subsequent increase in performance. The overall peak server performance cited above (i.e., 575 mps) was obtained for 7 queue managers (although just 4 queue managers were required to get within 95% of this peak; with all log files on the same disk). All 8 processors were close to saturation for such peak performance. The ability of MQSeries to run multiple queue managers provided the necessary scalability to take full advantage of the system processing capability

Effects of increasing the number of clients: As stated previously, IBM MQSeries persistent single client peak performance was 213 mps (achieved running 6 threads). While performance did not increase linearly with each thread added, performance increased smoothly with each new client (up to about 4 clients). Overall server performance peaked with the addition of a seventh client (with performance increasing very slowly with the addition of the fifth, sixth and seventh clients). Additional clients yielded no increase and/or a very small decrease in MQSeries performance (due to increased server congestion).

MSMQ persistent single client peak performance was extremely low, i.e., 16 mps (which, just as it was the case for the express delivery mode, was only 7% of the 213 mps achieved by a single IBM MQSeries client). This was achieved running 2 threads. Peak performance increased linearly with the addition of a second and third client. However, with the addition of a fourth and fifth client, performance increased dramatically (suggesting possibly a significantly more efficient code path, operational mode, etc. employed by MSMQ with increasing server congestion). Additional clients yielded small increases in performance, with peak performance obtained with 11 clients. Peak performance remained stable and/or declined slightly for 12 or more clients (once again presumably due to increased server congestion). As discussed below, for 5 or more clients, performance was observed to decline linearly with each client per thread added (over and above 3 to 4 threads per client).

Effects of increasing the number of threads per client: IBM MQSeries performance increased and then plateaued as additional threads per client were added. For a very large number of threads per client, performance could decrease very slightly from overall peak performance (but in no way like the steep decline observed for MSMQ; see below). As discussed previously, additional threads per client presumably only added more overhead time for switching between threads, with no possible benefit to overall performance, thereby reducing single client efficiency and subsequent mps and throughput.

For up to 4 clients, MSMQ performance increased (and then typically plateaued) as more threads per client were initially added (just as was the case for MQSeries). However, for 5 or more clients, performance initially increased steeply as threads per client were added, peaking with just 3 to 4 threads per client. However, performance then declined linearly with each additional thread per client (see the MSMQ curve for 11 clients in Figure 5, which is highly representative of the curves obtained for 5 or more clients).

Effects of preloading the server response queue: In the previous section summarizing performance results obtained for the express delivery mode, we analyzed the effects of preloading the server response queue with from 0 to 1,000 messages. In that test paradigm, all clients shared a single request/response queue pair. As the results showed, MQSeries performance was relatively unaffected by preloading a common (shared) response queue because it utilizes a correlation ID to quickly find a given client's message in that queue. MSMQ performance was devastated by even a small number of preloaded messages in the server response queue, since it was required to sequentially search through the queue to find a given client's message.

However, with no preloading of a single queue shared by all clients, MSMQ performance was relatively high and similar to that obtained when MSMQ clients each had their own dedicated request/response queue pairs. This was because the express delivery mode was sufficiently fast

to keep up with incoming requests, which in turn prevented the growth of the server response queue.

For the persistent delivery mode (operating in syncpoint), the significant processing required by each incoming message guaranteed the growth of the server response queue when it was shared by all requesting clients. For MSMQ, this yielded extremely poor persistent performance for any test using multiple clients that shared a single request/response queue pair, regardless of whether the response queue was preloaded or not. Peak performance for 6 or more clients (over all threads per client examined) ranged between 9 and 15 mps, regardless of the number of preloaded messages, i.e., **for the persistent delivery mode (operating inside syncpoint), MSMQ performance effectively collapsed when multiple clients shared the same request/response queue pair, whether or not the response queue was preloaded. MQSeries persistent performance, on the other hand, was relatively unaffected by multiple clients sharing a single pair of queues, even when the shared response queue was preloaded with a fixed number of unretrievable messages** (i.e., for 6 or more clients, it delivered over 500 mps, even when preloaded). Multiple clients sharing a single common message queue represents a situation that would arise frequently in actual customer environments, especially in ones that have to support a very large population of clients.

As was the case for the express delivery mode analyzed above, the performance numbers reported herein for both MQSeries and MSMQ were significantly greater than those reported in [2] and [3], as a result of the considerably more powerful disk architecture and processor capacity of the server used, as well as the additional tuning, described above in **TEST CONFIGURATION AND PROCEDURES**.

(As before, it should be noted that the exact combination of number of clients and threads per client at which peak messages per second and/or 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 using the persistent delivery mode (inside syncpoint), MQSeries 5.2 provided almost 14 times the single client performance of that of MSMQ 2.0. Similarly, MQSeries 5.2 provided more than 50% the overall server capacity of that provided by MSMQ 2.0. Its ability to define additional queue managers provided MQSeries with the scalability required to increase its performance capacity in the face of a substantially larger offered client load (an ability lacking in MSMQ).

As was the case for the express delivery mode, MQSeries persistent performance was relatively unaffected by preloading a shared sever response queue. In contrast, MSMQ persistent performance effectively collapsed simply when multiple clients were required to share a single queue, with or without preloading. Multiple clients sharing a single common message queue represents a situation that would arise frequently in actual customer environments, especially in ones that have to support a very large population of clients.

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 will be far better performance of all the other important applications that are running simultaneously with message queuing.

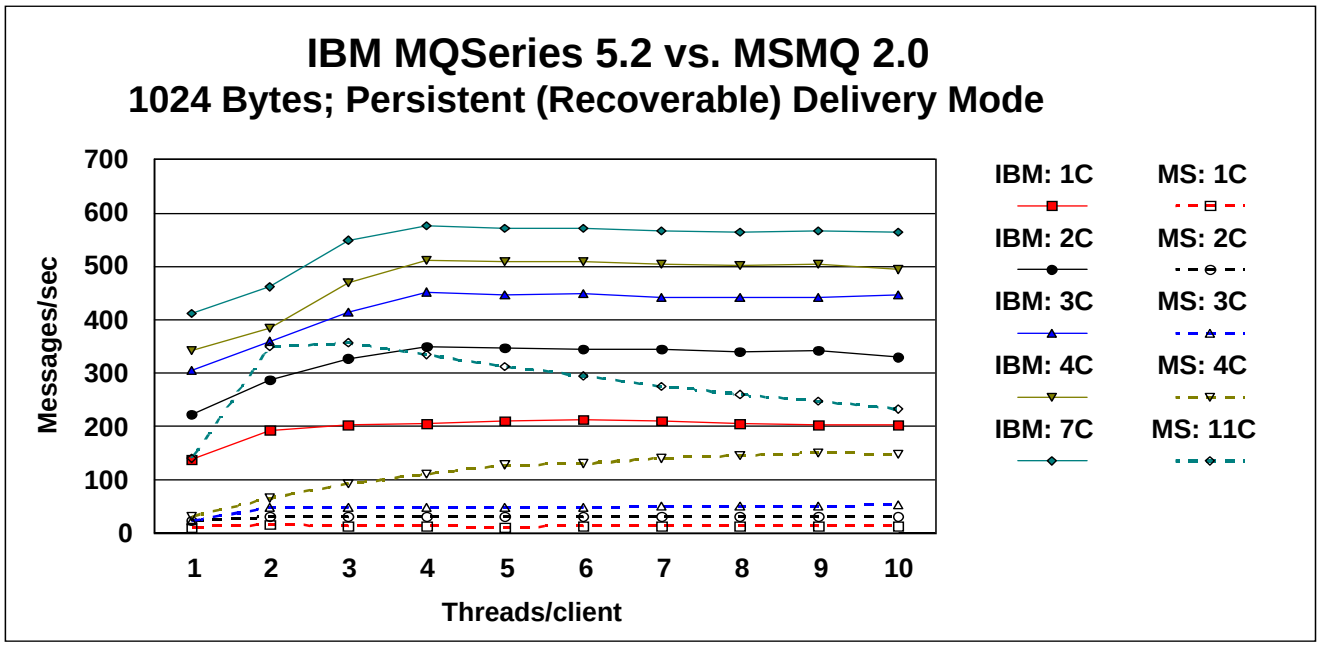

FIGURE 5 IBM MQSERIES 5.2 vs. MSMQ 2.0
MESSAGES/SECOND vs. THREADS/CLIENT
PERSISTENT DELIVERY MODE
1024 BYTE MESSAGE SIZE

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

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

MQSeries single client peak performance = 214 mps
MSMQ single client peak performance = 16 mps

MQSeries overall peak server performance = 575 mps
MSMQ overall peak server performance = 365 mps

MQSeries single client/single thread response time = 4.7 msec
MSMQ single client/single thread response time = 63.8 msec

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

Effects of Varying Message Size (Throughput)

Persistent Delivery Mode (outside syncpoint)

In this section, we analyze the effects of varying the message size on the overall throughput (measured in Megabits per second, or Mbps) for the persistent delivery mode (operating inside syncpoint), using either IBM MQSeries 5.2 or MSMQ 2.0. As before, throughput was calculated here as the number of messages processed per second (mps) times the message size. Once again (as expected), mps decreased with increasing message size, while throughput increased with increasing message size.

The message sizes examined were 1024, 2048, 5120, 10240, 20480 and 30720 bytes. Detailed performance measurements were made for each of these message sizes. Just as it was the case for the express delivery mode, **for the persistent delivery mode (operating inside syncpoint), for every message size examined, IBM MQSeries outperformed MSMQ for every combination of number of simultaneous clients and threads per client examined**.

Figure 7 represents overall peak server throughputs (Megabits per second, or Mbps) for IBM MQSeries and MSMQ, as a function of message size. All data was collected using persistent delivery mode, running inside syncpoint. For a given message size, overall peak server throughput was defined to be the maximum throughput measured over every examined combination of number of clients and threads per client (i.e., over 1 through up to 20 clients, running 1 through up to 20 threads per client). Peak throughputs were always achieved over this range of combinations, with the number of clients and/or threads per client increased (as necessary) to determine the overall peaks.

Table 5 summarizes the individual peak throughput values obtained for each message size examined. Table 5 also presents the ratio of MQSeries to MSMQ throughput, indicating the number of times (not percent) that MQSeries performance was better than that of MSMQ.

Figure 7 and Table 5 demonstrate that IBM MQSeries and MSMQ overall peak server throughput increased with increasing message size. They also demonstrate that **overall peak server performance for IBM MQSeries was significantly greater than that of MSMQ for every message size examined**. MQSeries consistently provided several times the peak server performance of that offered by MSMQ.

Table 5. Throughput vs. Message Size
Persistent Delivery Mode (inside syncpoint)

Message Size (Bytes)	Throughput (Mbps)		Ratio of MQSeries to MSMQ mps
	IBM MQSeries	MSMQ	MQSeries/MSMQ
1,024	4.71	2.99	1.6
2,048	8.98	5.47	1.6
5,120	19.88	11.32	1.8
10,240	31.41	17.38	1.8
20,480	41.58	25.3	1.6
30,720	45.87	29.57	1.6

Table 5. IBM MQSeries and MSMQ overall peak server throughputs obtained for a given message size (measured in Megabits per second, or Mbps), for the persistent delivery mode (operating inside syncpoint).

MQSeries performance substantially exceeded that of MSMQ, for every situation examined. For a given message size, overall peak server throughput was the maximum throughput achieved over all combinations of number of clients and threads per client examined (i.e., over 1 through up to 20 clients, running 1 through up to 20 threads per client). It represented the overall server capacity to process messages This maximum was always achieved over the range of combinations that were examined.

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**.**

In summary, for the persistent delivery mode (operating inside syncpoint), MQSeries 5.2 outperformed MSMQ 2.0 for every message size examined, providing up to 1.8 times the peak performance of that of MSMQ.

Please note that these throughputs were obtained using a disk architecture specifically defined by Microsoft MSMQ tuning guidelines [6], [7], [8] to obtain their best possible performance. In a subsequent paper, we will report the still larger performance differences obtained for the persistent delivery mode, using a far less expensive disk architecture for which IBM MQSeries performance has been optimized (i.e., greater performance at less the cost).

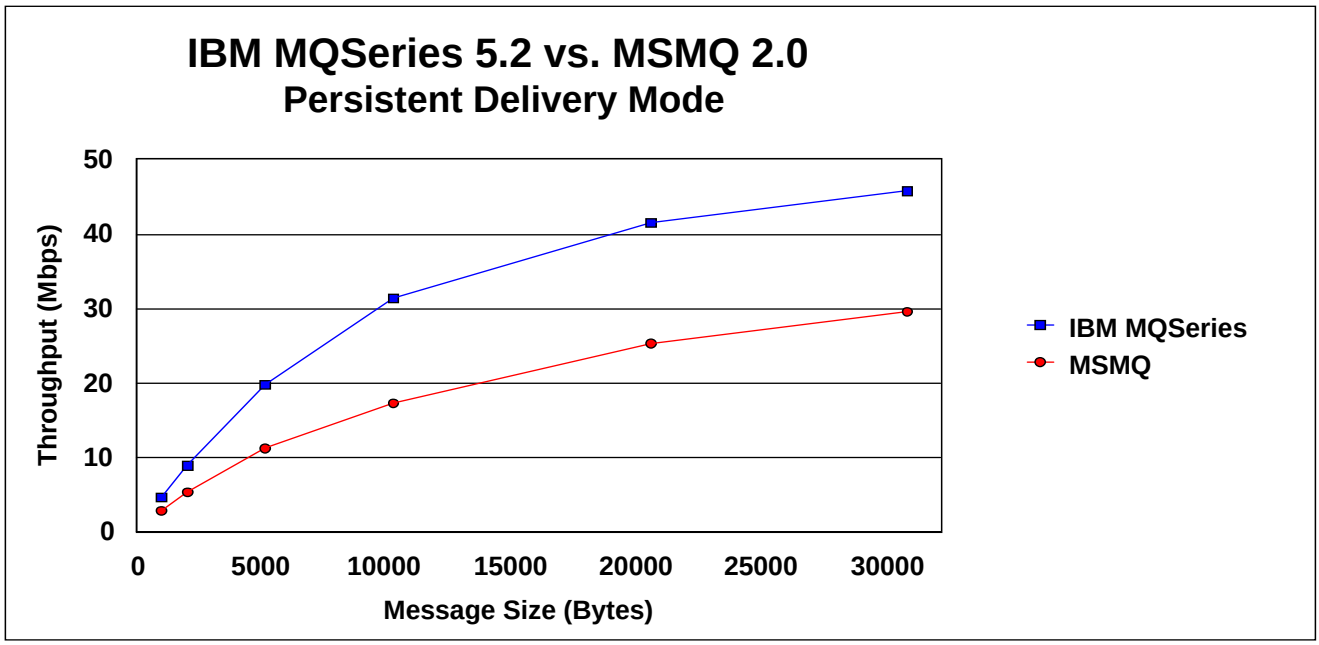

FIGURE 7 IBM MQSERIES 5.2 vs. MSMQ 2.0 THROUGHPUT vs. MESSAGE SIZE 1,024 through 30,720 BYTE MESSAGE SIZES PERSISTENT DELIVERY MODE

IBM MQSeries and MSMQ overall peak server throughput (Megabits per second, or Mbps) vs. message size (1024, 2048, 5120, 10240, 20480, and 30720 bytes), using persistent delivery mode (inside syncpoint). The throughput plotted for each message size represents the maximum obtained over all examined combinations of number of clients and threads/ client (i.e., over 1 through up to 20 clients, running 1 through up to 20 threads/client). For all message sizes examined, the overall peak server throughput always occurred within this range of combinations.

For every message size examined, MQSeries substantially outperformed MSMQ. For both applications, throughput increased with increasing message size.

CONCLUSIONS

As the data has shown, for both the express and persistent delivery modes, IBM MQSeries 5.2 significantly outperformed MSMQ 2.0 for every combination of system parameters examined. MQSeries overall peak server performance was up to 5 better than that of MSMQ. Furthermore, MQSeries single client peak performance was nearly 14 times better than that of MSMQ (for a 1024 byte message).

For the express delivery mode, IBM MQSeries 5.2 performance was extremely robust as the server response queue was preloaded with an increasing number of initial messages. In sharp contrast, MSMQ 2.0 performance effectively collapsed as the initial response queue size increased. For 1,000 preloaded messages, MQSeries performance was 264 times better than that of MSMQ.

For the persistent delivery mode, MSMQ performance virtually collapsed when multiple clients shared the same server response queue, regardless of whether or not that queue was preloaded. MQSeries persistent performance, on the other hand, was relatively unaffected by sharing a single (even preloaded) response queue.

MSMQ therefore has a number of very serious performance problems which, understandably, were avoided by Microsoft in its previously published benchmark reports. These problems compromise the ability of MSMQ to effectively scale for large customer environments. 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 poor performance of other simultaneously running applications. This is a cost that most e-businesses cannot afford to pay.

In contrast, MQSeries will scale effectively as the user population grows, without the catastrophic collapse in performance that was observed for MSMQ. This scalable high performance guarantees that your server resources can 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.

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", Andrew Rindos, Alan Kittel, Steven Woolet and Mitchell Loeb, IBM Technical Report, March 2000. see http://www-4.ibm.com/software/ts/mqseries/library/articles/MQperf.pdf

[3] "A performance comparison of IBM MQSeries 5.1 and Microsoft Message Queue (MSMQ) 2.0 on Windows 2000: Transactional and persistent delivery mode (including a critique of Microsoft's recent NSTL performance report", Andrew Rindos, Alan Kittel, Steven Woolet, Mitchell Loeb and Robert Maiolini, IBM Technical Report TR29.3334, June 2000. see http://www-4.ibm.com/software/ts/mqseries/library/articles/29.3334_ext.pdf

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

[5] Getting the most out of MQSeries", Richard G. Nikula, BMC Software White Paper, 2000, see http://www.bmc.com/available_doc.html?item_no=100029505

[6] "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 .

[7] "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

[8] "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

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both: IBM®, MQSeries®, Netfinity®, Intellistations®, Etherjet, ServeRaid.

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

Intel®, Pentium®, Pentium® III Xeon, Pentium® Pro and Express are trademarks of Intel Corp. in the United States, other countries, or both.

Cybex®, AutoView and Commander are trademarks of Cybex Computer Products Corp. in the United States, other countries, or both.

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