# *High-performance Messaging with JMS*

**A Benchmark Comparison of**
**Progress® SonicMQ™ and IBM® MQSeries®**


*Point Solutions*
*297 Allston Street*
*Cambridge MA 02139*

## CONTENTS



## About Point Solutions

Point Solutions is a consulting firm focusing on Internet-based commerce and large-scale system design technologies. The company's clients include software vendors and Internet startups as well as utility and insurance companies. Recently, Point Solutions developed a logistics system for international product delivery and designed reporting and transactional systems that were deployed on a nationwide scale.

## INTRODUCTION

The Sun Microsystems Java Message Service (JMS) specification defines the programming interface for a messaging system that uses Java-based clients. With this new interface standard, users can now compare messaging systems on a one-to-one basis, regardless of whether or not they are completely written in Java.

One critical measure of a messaging system is its performance and scalability under heavy workloads. This paper defines real-world scenarios in which messaging may be used, and provides a detailed performance comparison between Progress® SonicMQ™ and IBM® MQSeries.

### Progress SonicMQ

SonicMQ from Progress Software is a recent entry into the messaging market. SonicMQ provides a full JMS implementation of publish/subscribe and point-to-point messaging that is written entirely in Java. In addition to a native Java JMS client, SonicMQ also provides an ActiveX®/COM client. (A C client is currently in beta test.) The messaging clients communicate with one or more Message Brokers, which are implemented in Java and tested on a number of different platforms.

### IBM MQSeries

MQSeries from IBM is one of the most established messaging products in the industry. MQSeries has traditionally been oriented towards queue-based messaging, but was enhanced in late 1999 to support publish/subscribe messaging and a JMS interface. The MQSeries JMS client is a native Java implementation and does not require any platform-specific code. The messaging clients communicate with one or more Queue Managers, which are implemented in native code and are available for a wide range of platforms.

## TEST DESCRIPTION AND METHODOLOGY

Performance was measured under maximum load by sending messages as quickly as possible within each test configuration, using "out-of-the-box" default settings for each messaging product. Test were conducted under the following conditions:

- All tests were repeated until the recorded test data from multiple runs varied by less than 1%.
- Each client was run in a single JMS connection.
- All results were recorded after client connections, publishers, and subscribers were established.
- Except during the many-to-one tests, no client machine exceeded 75% CPU utilization.
- Auto acknowledge and asynchronous message receipt were used at the message consumers.
- Each product's message store was emptied between tests. In SonicMQ this was done by reinitializing the database and log files. For MQSeries, all queues (including those used for publish/subscribe) were cleared of messages using the JMS administration tool.

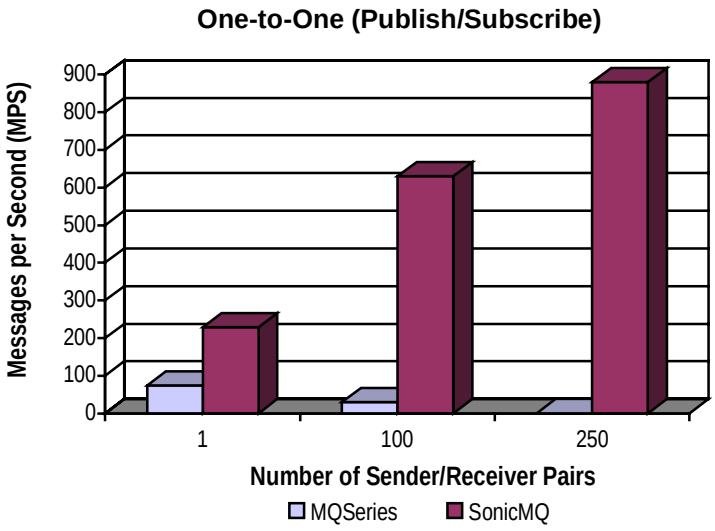
## USAGE MODELS AND PERFORMANCE GRAPHS

The following usage models describe typical application scenarios where messaging servers may be deployed. Though not complete, this list reflects the varied performance characteristics of messaging applications and allows us to examine the relative performance impact of different messaging configurations.

The following graphs represent selected test results using durable, persistent messages. For a complete list of test results, see Appendix A. For a detailed description of the hardware and software test environment, see Appendix B.

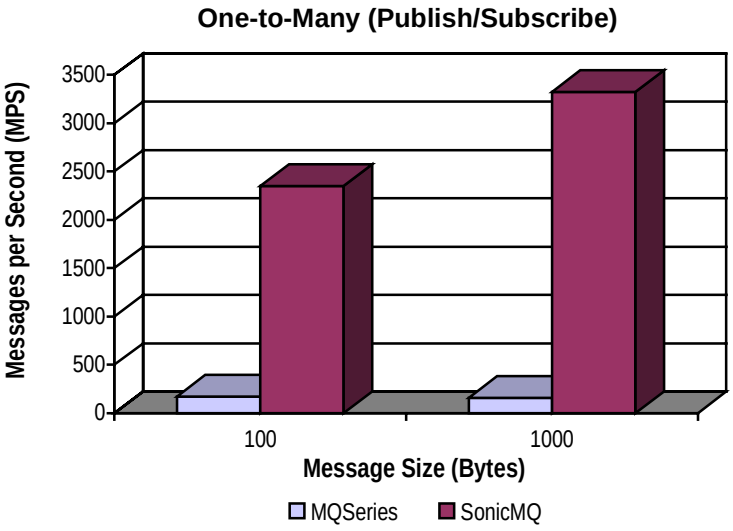
### Bulk Data Transfer: One-to-One (Publish/Subscribe)

Transferring bulk data between applications or databases typically implies a one-to-one relationship, e.g. a manufacturer sending upcoming shipment information to a distributor. In a messaging environment, one-to-one implies a single message producer sending information to a single consumer. Each message is sent to one predefined destination that is associated with a specific sender/receiver pair.



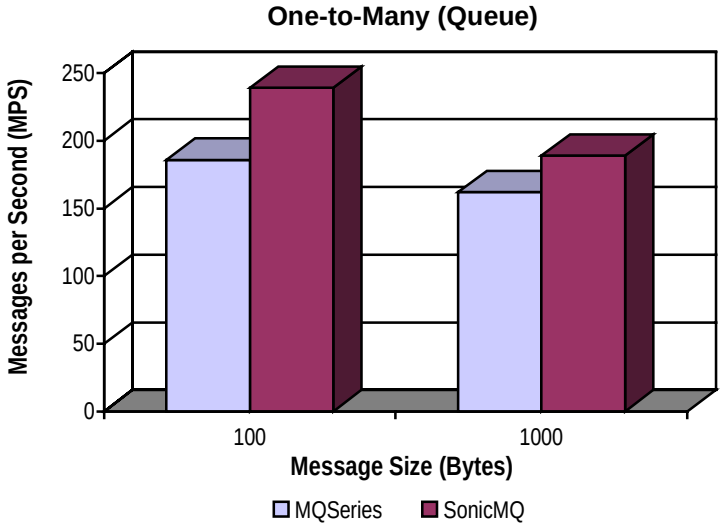

### Broker Services: One-to-Many (Publish/Subscribe)

Reliable information delivery to a wide audience is the basis for many of today's leading Web-based brokerage applications. Users indicate interest by subscribing to specific topics; messages published to those topics are automatically sent to all registered subscribers. An application may require that each message be guaranteed to arrive at each recipient (e.g. a stock transaction that must update multiple back-office systems), or it may allow some messages to be lost (e.g. a stock ticker that updates prices every few seconds).



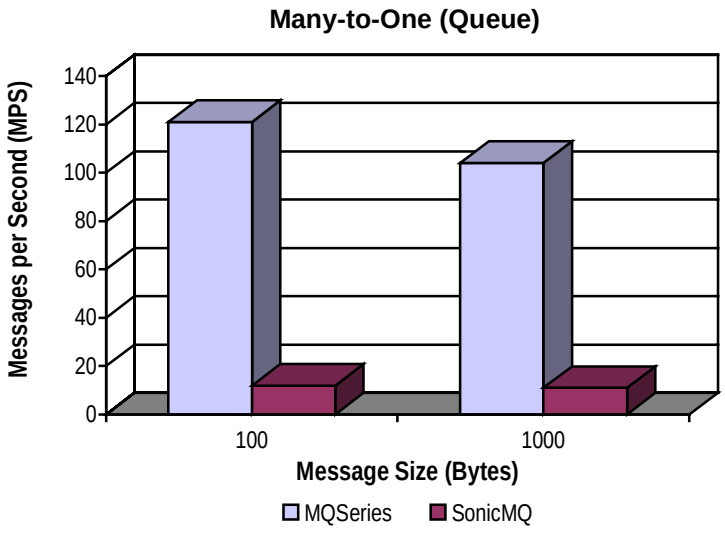

## Service Call Dispatch: One-to-Many (Queue)

A field service organization would typically use a one-to-many queuing model to dispatch service call requests to remote service technicians. A number of recipients may request messages from a specific queue. Unlike publish/subscribe, only one recipient will receive each message, ensuring that each technician receives unique service calls. Messages are delivered on a first-come/first-served basis, and requests (even for an empty queue) are saved until a new message is available.



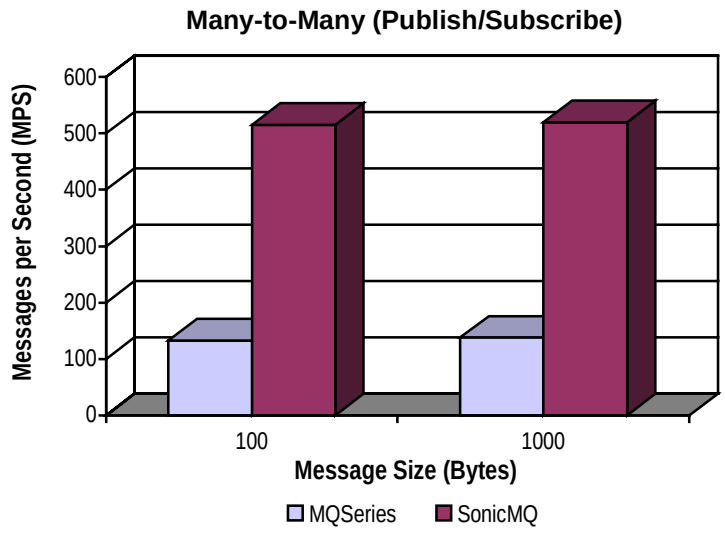
## Sales Office Reporting: Many-to-One (Queue)

A many-to-one queuing model may be utilized when collecting sales data from remote offices for reporting or data warehousing purposes. Messages may accumulate from many sources into a single location, and may be handled at an appropriate time by the message recipient. When guaranteed messages are required, the sender does not need to wait for the recipient to process the message before being assured of its delivery.



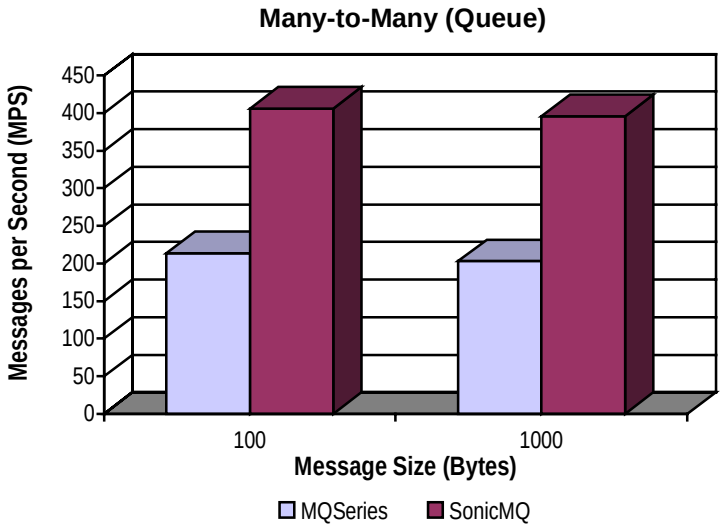

## Online Auctions: Many-to-Many (Publish/Subscribe)

In online auctions, information is communicated from each bidder directly to each participant in the auction. Using publish/subscribe in a many-to-many model is similar to one-to-many, except that each recipient is also a message sender. All messages are sent and received on a single topic, and every messaging client shares information with all other clients.



### Web Server Integration: Many-to-Many (Queue)

Robust Web sites typically require the effective integration of multiple Web servers with multiple application servers. Using many-to-many queuing, each Web server can make processing requests to the application servers by sending messages to a queue on which they all listen. An application server (or queue receiver) will listen for requests only when it is able to process them; only one server will ever process a given request. A specific Web server (or message sender) may have its messages processed by a different application server from one request to the next.




## ANALYZING THE RESULTS

Test results indicate that SonicMQ consistently achieves higher throughput than MQSeries, yielding better performance results in all cases except many-to-one queuing. The following sections explore several aspects of SonicMQ's underlying architecture that are the likely cause of its overall high performance.

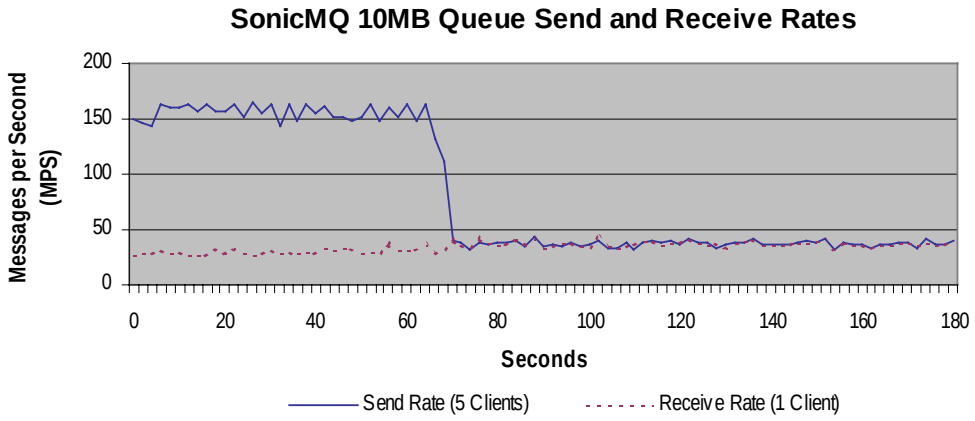
Application requirements and individual deployment scenarios both play a major role in determining whether system performance is sufficient, especially for high-volume Internet applications. As a result, custom performance tuning of both products would be expected to improve overall test results.

### Message Flow Control

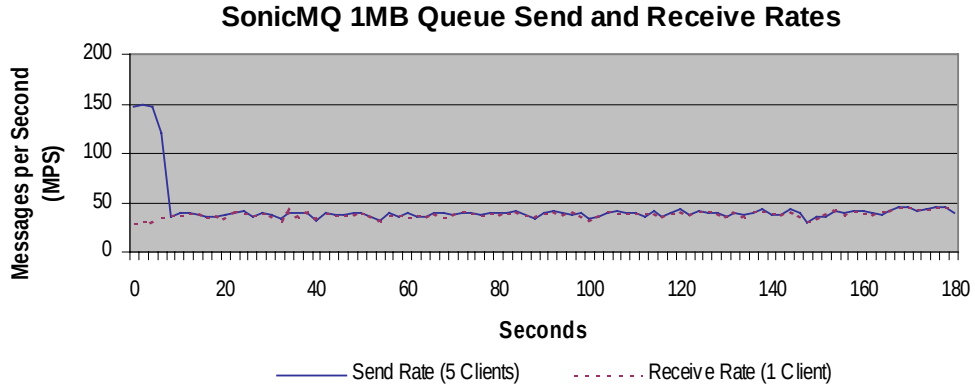
When messages are sent at a faster rate than they can be received at their destination, a broker must save them for delivery. When capacity limits within the broker are reached, the sending client must be throttled using flow control to avoid losing messages. The capacity limit may be predefined administratively or determined by limitations in memory or disk space. How and when flow control is applied can significantly alter the performance results of the messaging system.

When many messages are allowed to accumulate in a broker's memory before flow control is applied, the sending client will attain a high level of performance until the flow control point is reached. While this may be desirable in some cases, this increase in speed comes at a price. Improved service to one sender may reduce the CPU resources and memory available to other clients, potentially causing slowdowns in overall message throughput.

The following graphs depict the send and receive rates for a test where multiple senders send as many messages as possible to a single receiver. In the graph below, SonicMQ is configured with a 10MB queue limit, a single queue receiver, and five queue senders. The initial send rate achieved for the 10MB queue demonstrates the effect of filling memory before flow control is applied, which occurs approximately 70 seconds into the test. After flow control is applied, 10MB of memory is being used at the broker by messages buffered for delivery.




In the next graph, the queue limit is set to 1MB, which is the default configuration for SonicMQ. A 1MB queue limit provides a smaller amount of time where the send rates are higher than the receive rate, and less memory is used for messages.




Another effect of high queue size limits is increased delivery time for each message, since buffered messages will spend more time in memory and will take longer to arrive at a receiver.

Send rates alone do not accurately reflect broker performance. In addition to send rates, the total number of messages delivered in a system under load must be considered in “real world” performance evaluations. Excessive buffering typically hinders absolute throughput. Any time the send rate exceeds the receive rate, flow control has taken effect. As a result, messages may be buffered in the broker and remain undelivered for a measurable period of time.

The appropriate size limits that govern flow control will vary between applications. It may be advantageous to enable a high send rate for a client, particularly if the number of messages will be small and buffering will not have a great effect. SonicMQ allows the effect of flow control to be adjusted by providing tunable buffer sizes for publish and subscribe. For publish/subscribe messages, the parameter OUTPUT_QUEUE_SIZE can be used to adjust the buffer size for reliable messages and GUAR_QUEUE_SIZE can be used for persistent messages. For queue-based messaging, the size of the queue affects when flow control is applied.

### Save/Retrieve Extent

For certain applications, limiting the size of a queue to save memory may be desirable. However, it may not be possible to allow a message sender to be stopped by flow control. For those cases SonicMQ provides the Save Extent parameter, which defines the queue size at which messages will be saved in the database. This technique saves memory by using disk space for queue storage, allowing large queues to be handled efficiently with minimal affect on overall broker performance.

### Syncpoint

A syncpoint in SonicMQ is the time where the running state of the message broker is saved in the recovery log files. The information needed to ensure the delivery of guaranteed messages is stored during a syncpoint. Reliable messaging does not require syncpoints to be performed. Syncpoints provide a safe starting point for recovery operations in the case of broker machine failure, and allow older recovery information to be discarded once the syncpoint is complete.

In SonicMQ, a syncpoint is performed when the broker fills one log file and switches to the second. The length of the log files therefore determines how often syncpoints will occur. Because the syncpoint process consumes resources in the broker, longer log files will yield higher performance levels overall. The SonicMQ broker will provide a warning when syncpoint operations account for more than 50% of the total log file size.

## PERFORMANCE CONSIDERATIONS

### Java Virtual Machine

Both the SonicMQ broker and standard client are written in Java; as a result, the Java Virtual Machine (JVM) used to run SonicMQ can have a significant impact on overall messaging performance. Recent JVM advances allow for just-in-time compilation of Java classes, enhanced garbage collection, efficient input and output processing, and other significant capabilities. These advances can improve overall performance by a factor of 300% in some cases, making the choice of JVM critical to attaining high performance levels.

Another significant factor is the size of the Java heap, which is typically specified on the JVM command line with a parameter such as –mx. Typically this parameter is set to 128 or 256 MB. If this parameter exceeds the memory available to the JVM process, performance may significantly degrade as a result of page swapping in the underlying operating system. The memory available to the JVM may not match the total memory in the broker machine due to the memory requirements of other processes. In this case, lowering the total heap for the JVM will increase performance.

### Client Acknowledge

When using publish/subscribe messaging, guaranteed messages may either be acknowledged automatically by message receivers or acknowledged through "client acknowledge," which is under the control of the calling program. When client acknowledge is used, the message sender will not be able to send subsequent messages until the acknowledgement occurs. To avoid unnecessary slowdown when using guaranteed messages, client acknowledge should be performed as quickly as possible.

### Disk Drive Caching

Disk file access from the broker can have a major influence on overall performance. Increased drive speeds directly translate to higher message throughput when handling guaranteed messages. Many disk drive controllers support write caches that allow disk writes to be delayed, increasing write speeds for the operating system. While a write cache increases performance, it also increases the possibility that messages will be lost in a broker machine failure. For message recoverability, the SonicMQ broker requires that saved messages actually be written to disk (in addition to being cached). As a result, caching controllers should only have cache enabled if the controller supports write-through capability, or if message recoverability is not required.

### Queue Prefetch

SonicMQ supports prefetching messages from a queue to optimize overall throughput. Prefetching allows a client to receive messages from the SonicMQ broker before being explicitly requested by the client, eliminating the overhead of broker requests on a per-message basis. However, prefetching also changes the operation of the SonicMQ system by allowing messages to accumulate at the client until an application-defined count is reached.

The performance gain using prefetching is primarily realized on lightly loaded brokers, where a receiving client tends to govern overall throughput. When the broker is operating at full capacity, other factors (such as queue size and disk I/O) tend to limit message delivery rates.

## CONCLUSION

Though the Java Message Service (JMS) specification defines a standard programming interface for messaging systems, implementation of the specification can result in products with significant performance differences. This report shows a variety of typical high-volume business scenarios and how two JMS implementations, Progress SonicMQ and IBM MQSeries, perform differently under those conditions.

The test results demonstrate that technology design and underlying architecture are critical to the performance and scalability of a JMS solution. This report shows superior results in most scenarios for Progress SonicMQ.

Application performance will vary based on functional requirements and the individual deployment environment. It is important to remember that, while these tests reflected an "out-of-the-box" configuration, fine-tuning may lead to overall performance improvements for both implementations.

## APPENDIX A: DETAILED TEST RESULTS

| *Test Description (Client Configuration)* | *Message Size (Bytes)* | *Durable/ Persistent* | *SonicMQ Msgs/Sec Sent* | *SonicMQ Msgs/Sec Received* | *MQSeries Msgs/Sec Sent* | *MQSeries Msgs/Sec Received* | *SonicMQ Received % Faster* | *MQSeries Received % Faster* |
|---|---|---|---|---|---|---|---|---|
| **One-to-One (Publish/Subscribe)** | | | | | | | | |
| 1 sender, 1 receiver | 100 | No | 2,365 | 2,355 | 141 | 106 | 2,122% | |
| 1 sender, 1 receiver | 100 | Yes | 92 | 91 | 112 | 78 | 17% | |
| 1 sender, 1 receiver | 1,000 | No | 559 | 559 | 134 | 100 | 459% | |
| 1 sender, 1 receiver | 1,000 | Yes | 229 | 229 | 108 | 74 | 209% | |
| 100 senders, 100 receivers | 100 | No | 5,655 | 5,653 | 30 | 14 | 40,279% | |
| 100 senders, 100 receivers | 100 | Yes | 580 | 497 | 56 | 48 | 935% | |
| 100 senders, 100 receivers | 1,000 | No | 2,011 | 2,011 | 21 | 4 | 50,175% | |
| 100 senders, 100 receivers | 1,000 | Yes | 632 | 630 | 47 | 30 | 2,000% | |
| 250 senders, 250 receivers | 100 | No | 4,977 | 4,975 | 21 | 4 | 124,275% | |
| 250 senders, 250 receivers | 100 | Yes | 918 | 870 | 38 | 29 | 2,900% | |
| 250 senders, 250 receivers | 1,000 | No | 1,330 | 1,330 | 7 | 4 | 33,150% | |
| 250 senders, 250 receivers | 1,000 | Yes | 880 | 880 | *N/A* | *N/A* | | |
| **One-to-Many (Publish/Subscribe)** | | | | | | | | |
| 1 sender, 50 receivers | 100 | No | 264 | 13,111 | 27 | 470 | 2,690% | |
| 1 sender, 50 receivers | 100 | Yes | 47 | 2,353 | 22 | 174 | 1,252% | |
| 1 sender, 50 receivers | 1,000 | No | 95 | 4,770 | 26 | 458 | 941% | |
| 1 sender, 50 receivers | 1,000 | Yes | 66 | 3,322 | 22 | 158 | 2,003% | |
| **One-to-Many (Queue)** | | | | | | | | |
| 1 sender, 50 receivers | 100 | No | 303 | 303 | 226 | 225 | 35% | |
| 1 sender, 50 receivers | 100 | Yes | 239 | 239 | 187 | 186 | 28% | |
| 1 sender, 50 receivers | 1,000 | No | 287 | 287 | 196 | 196 | 46% | |
| 1 sender, 50 receivers | 1,000 | Yes | 189 | 189 | 162 | 162 | 17% | |
| **Many-to-One (Queue)** | | | | | | | | |
| 50 senders, 1 receiver | 100 | No | 232 | 227 | 183 | 179 | 27% | |
| 50 senders, 1 receiver | 100 | Yes | 16 | 12 | 124 | 121 | | 908% |
| 50 senders, 1 receiver | 1,000 | No | 549 | 548 | 153 | 150 | 265% | |
| 50 senders, 1 receiver | 1,000 | Yes | 12 | 11 | 107 | 104 | | 845% |
| **Many-to-Many (Publish/Subscribe)** | | | | | | | | |
| 50 senders, 50 receivers | 100 | No | 241 | 11,914 | 25 | 374 | 3,086% | |
| 50 senders, 50 receivers | 100 | Yes | 11 | 515 | 20 | 133 | 287% | |
| 50 senders, 50 receivers | 1,000 | No | 91 | 4,533 | 24 | 337 | 1,245% | |
| 50 senders, 50 receivers | 1,000 | Yes | 10 | 520 | 4 | 138 | 277% | |
| **Many-to-Many (Queue)** | | | | | | | | |
| 50 senders, 50 receivers | 100 | No | 750 | 745 | 694 | 690 | 8% | |
| 50 senders, 50 receivers | 100 | Yes | 411 | 406 | 218 | 214 | 90% | |
| 50 senders, 50 receivers | 1,000 | No | 1,190 | 1,188 | 635 | 632 | 88% | |
| 50 senders, 50 receivers | 1,000 | Yes | 397 | 396 | 206 | 203 | 95% | |

## APPENDIX B: TEST ENVIRONMENT

### Progress SonicMQ

Tests were performed on SonicMQ Enterprise Edition 2000.1. No additional patches were required.

### IBM MQSeries

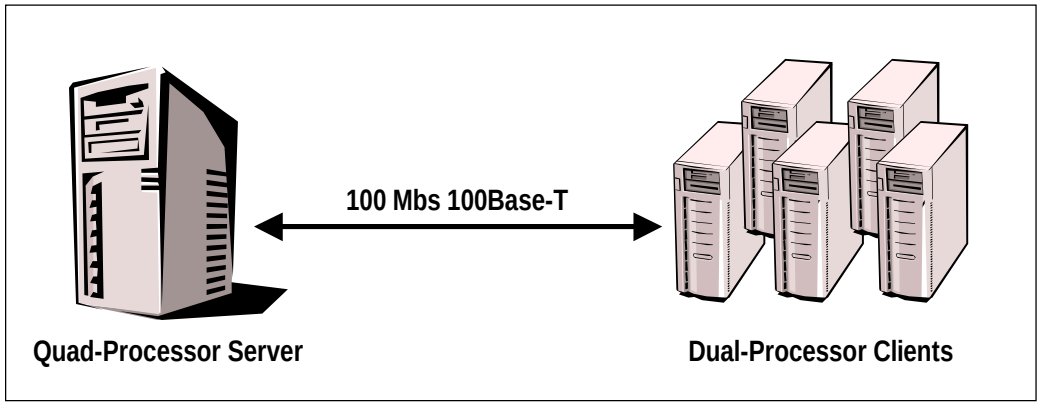
Tests were performed on MQSeries Version 5.1 for Windows® NT. The following IBM SupportPacs were required to complete the tests:

- PTF U200095 (12/20/99) – Cumulative release of fixes for MQSeries 5.1 on Windows NT
- MA0C (10/22/99) – Enables publish and subscribe capabilities
- MA0F (12/7/99) – Supports Application Messaging Interface 1.0 (API for point-to-point and publish/subscribe as well as administered settings in a repository)
- MA88 (12/17/99) – Adds Java client capabilities and JMS support

### Software Settings

- Queue limit set to 1MB in SonicMQ; set to 1000 entries in MQSeries
- MQSeries set to allow 1000 client connections
- IBM JVM V1.1.8 (with JIT enabled) used on both client and server
- Memory limit for SonicMQ set to 256MB (via server JVM); no memory limit set for MQSeries
- Client JVM used default settings (no command line options specified)
- Hard disk write caching was disabled
- No other tuning or product configuration was performed

### Hardware Configuration




***Server Machine*:**
Dell™ PowerEdge™ 6300 server
Four 550MHz XEON CPUs
2GB SD100 RAM
3Com® 3C905B Fast EtherLink® XL 10/100 PCI LAN card
Adaptec® RAID controller running RAID 0 on three IBM 9.1GB Ultra-2/LVD SCSI 7200 RPM hard disks
Windows NT 4.0 SP5 using NTFS volumes

***Client Machines (5)*:**
Dell 410 workstation
Two 500MHz Pentium® III CPUs
256MB SD100 RAM
EIDE ATA-33 controller
Maxtor™ 10GB 7200 RPM EIDE hard disk
3Com 3C905B Fast EtherLink XL 10/100 PCI LAN card
Windows NT 4.0 SP5 using NTFS volumes

***Networking*:**
Xylan OmniSwitch with guaranteed 100Mbs throughput for 12 connections
Dedicated 100Mbs LAN segment