WebSphere Message Broker v6.0

For Linux

Performance report

Version 1.2

January, 2006

Tim Dunn

Kevin Braithwaite

Rich Bicheno

WebSphere Message Broker Development
IBM UK Laboratories
Hursley Park
Winchester
Hampshire
SO21 2JN

Take Note!

Before using this report be sure to read the general information under "Notices".

Third Edition, January 2006.
This edition applies to *WebSphere Message Broker V6 for Linux* and to all subsequent releases and modifications until otherwise indicated in new editions.

Notices

This report is intended for Architects, Systems Programmers, Analysts and Programmers wanting to understand the performance characteristics of **WebSphere Message Broker V6 for Linux**. The information is not intended as the specification of any programming interfaces that are provided by WebSphere MQ or WebSphere Message Broker V6 for Linux. It is assumed that the reader is familiar with the concepts and operation of WebSphere Message Broker V6.

References in this report to IBM products or programs do not imply that IBM intends to make these available in all countries in which IBM operates.

Information contained in this report has not been submitted to any formal IBM test and is distributed “asis”. The use of this information and the implementation of any of the techniques is the responsibility of the customer. Much depends on the ability of the customer to evaluate these data and project the results to their operational environment.

The performance data contained in this report was measured in a controlled environment and results obtained in other environments may vary significantly.

Trademarks and service marks

The following terms, used in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

- IBM
- WebSphere MQ
- WebSphere Message Broker
- DB2

The following terms are trademarks of other companies:

- Windows 2000, Windows XP, Microsoft Corporation, RedHat, Linux

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

Summary of Amendments

Date	Changes
27th October 2005	Initial Release
1st December 2005	Correct the Aggregation Results in Release Highlights section.
11th January 2006	Correct the comments about LogBufferPages and the results for the Large Messaging usage scenario

Table of Contents

Introduction

The purpose of this report is to illustrate the key processing characteristics of WebSphere Message Broker. This has been done by measuring the message throughput which is possible for a number of different types of message processing, covering multiple message formats, types and sizes.

This report consists of two parts. These meet different requirements:

1. **Part I** contains the release highlights and some background information to help understand the context of the results. It shows:

    a. The improvement in performance with WebSphere Message Broker V6 when compared with WebSphere Business Integration Message Broker V5.

    b. The level of message throughput that is achievable when using WebSphere Message Broker in different ways. These tests use **multiple copies of the message flow and utilise as much of the server machine as possible** to illustrate the maximum message rate which can be sustained for the individual types of processing.

    The information in this part is presented at a high level and is intended to help you quickly understand WebSphere Message Broker throughput capabilities.

2. **Part II** contains measurement data for a wide variety of tests which examine the processing costs of individual functions **using a single copy of the message flow**. This information is provided for those who wish to understand the processing costs of different components within WebSphere Message Broker such as the differences in CPU cost between Fixed Length Tagged Delimited Strings and All Elements Delimited Tagged Delimited Strings. This information is intended for the more experienced WebSphere Message Broker user who is familiar with the product concepts and functions. **As these tests run a single copy of the message flow. They do not utilise the whole of the server machine and do not therefore represent the maximum message throughput which is achievable.**

There are a number of changes from previous performance reports. The most significant are:

1. Re-engineered tests to better reflect the processing costs which are encountered when processing messages with a WebSphere Message Broker message flow. The previous tests are deprecated and do not appear in this report.

2. Measurement of a selection of product samples which are available with WebSphere Message Broker V6. This is done for two reasons: Firstly it makes it easier for you to understand the volume of messages which can be processed for a variety of common use cases. Previous reports focused on the use of individual nodes which made it difficult to visualise particular applications. Secondly by using samples it is easy for you to take exactly the same message flows and message sets and run them in your own environment. You can then compare the results obtained in your environment against those published in this report. This can be very useful in validating that a broker environment is well configured.

3. More extensive analysis of product function, including incremental test cases.

4. Larger range of message sizes including a greater range of persistent message sizes.

5. A change in layout to separate the overview of message processing capability from the detailed data which shows the costs of using individual functions.

The performance measurements focus on the throughput capabilities of the broker using different message formats and processing node types. The aim of the measurements is to help you understand how many messages a second can be processed in different situations as well as helping you to understand the relative costs of the different node types and approaches to message processing.

You should not attempt to make any direct comparisons of the test results in this report with what may appear to be similar tests in previous performance reports. This is because the contents of the test messages are significantly different as is the processing in the tests. It is not meaningful to make such comparisons.

Some optimisations to the test environment and procedures have been implemented to minimise the effect of logging for example and to ensure that messages do not build up on output queues (which has a detrimental effect on message throughput). These are detailed in the section Summary of Tuning Information.

In many of the tests the business logic used is minimal so the results presented represent the best throughput that can be achieved for that node type. This should be borne in mind when performing sizing for WebSphere Message Broker.

Part I

This part contains an overview of the improvements in performance which were obtained with WebSphere Message Broker V6 when compared with WebSphere Business Integration Message Broker V5.

It contains the following sections:

- *Release Highlights* which outlines the main differences in performance when using WebSphere Message Broker V6 compared with WebSphere Business Integration Message Broker V5.

- *Additional Information* which provides links to other sources of information about WebSphere Message Broker and related products.

Release Highlights

Improvements over WebSphere Business Integration Message Broker V5

Improving Message Broker runtime performance has been a specific focus with WebSphere Message Broker V6 and as a result there are many improvements in the level of performance when compared with WebSphere Business Integration Message Broker V5. The improvements come from two sources - updates to existing function and provision of new function.

All key areas of Message Broker runtime function have been investigated and improvements made to improve performance. The main areas of focus were:

- The parsing and streaming of messages
- The cost of processing ESQL
- Message aggregation
- Message Broker infrastructure
- The calling of Java and database procedures

The improvements in these areas can be obtained by upgrading to WebSphere Message Broker V6. **No code or message model changes are required to benefit from the improvements**.

Further improvements are available if you take advantage of new functions such as

- The support for shared variables in ESQL which provides the capability to build an in-memory cache. This allows an in memory table to be built and accessed within message flows for example. The function can remove the need to access a database for read only routing or data validation. Previously a message flow had to issue a read against a database for each message flow invocation. The Message Routing sample shipped with the product provides an illustration of such processing.
- The MQGET node which makes it possible to use WebSphere MQ queues as an intermediate data store for communication between request and reply message flows for example. The Coordinated Request Reply sample provides an illustration of how such processing can be implemented. Previously a database had to be used to store the intermediate data.
- The extended and improved DATETIME functions which make it possible to perform complex date and time formatting operations using WebSphere Message Broker provided function. Previously a user had to write functions in ESQL or Java to perform such processing.

The Table below shows the results of running a series of use cases in WebSphere Business Integration Message Broker V5 and WebSphere Message Broker V6. The use cases are briefly described at the end of this section and more fully in Appendix D – Use Case Descriptions. The use cases are largely taken from the samples gallery of WebSphere Message Broker V6.

Use Case	Message Size	V6 Msgs/sec	Improvement Ratio (V6/V5)	Note
Aggregation	8K	135	**3.14**	**1**
Coordinated Request/Reply	1K	450	**4.50**	**2**
Data Warehouse	1K	495	**1.00**	**3**
Large Messaging	16K	230	**1.05**	**4**
Message Routing	1K	4279	**1.90**	**5**
SWIFT Message Parse	7K	119	**3.22**	**6**

Throughput Comparison for Use Cases.

Notes:

1. As Aggregation in WebSphere Message Broker V6 is now based on the use of WebSphere MQ queues and not a database the I/O bottleneck which was caused by database logging has been removed. This combined with a reduced CPU cost per message has made it possible to increase message throughput in V6 when compared with V5.

2. Use of a WebSphere MQ queue for intermediate data storage in the WebSphere Message Broker V6 edition of the message flow removed an I/O bottleneck (due to database logging requirements). This combined with a reduced cost CPU cost per message allowed a higher CPU utilisation and message rate to be obtained.

3. There was no change in message throughput in this use case.

4. A reduction in the CPU cost per message in WebSphere Message Broker V6 allowed a higher message rate to be obtained.

5. By using a routing table which was held in shared variables the CPU cost per message was reduced. In WebSphere Business Integration Message Broker V5 a database read was required for every invocation of the message flow. By using shared variables this could be removed. The reduced CPU cost per message allowed a higher message rate to be achieved.

6. The rewrite of the MRM TDS parser has significantly reduced the CPU cost per message of parsing a TDS message. As a result it is possible to achieve a significantly higher message rate in WebSphere Message Broker V6 when compared with WebSphere Business Integration Message Broker V5.

Each of the use cases was implemented in WebSphere Business Integration Message Broker V5 and WebSphere Message Broker V6 using the same hardware and prerequisite software.

The Aggregation, Data Warehouse, Large Messaging, SWIFT Message Parse and XML Transformation use cases contained no new code in the message flows. Exactly the same message flow was run on V5 and V6.

The results in the table above were obtained by running sufficient copies of each message flow so that system CPU utilisation was 80% or greater.

These results show that there are significant increases in the level of message throughput that is achievable with WebSphere Message Broker V6 when compared with WebSphere Business Integration Message Broker V5. Those use cases showing the best gains where the Coordinated Request/Reply, SWIFT Message Parse, and Aggregation all of which tripled message throughput or better.

Most of the gains in message throughput for the use cases came from improvements to existing broker function. The CPU cost of many aspects of the broker has been significantly reduced and as such message throughput can increase for a given amount of CPU power.

Improvements over WebSphere MQ Integrator V2.1

In this report WebSphere Message Broker V6 performance had been compared with that of WebSphere Business Integration Message Broker V5. The improvements in performance in WebSphere Message Broker V6. when compared to WebSphere MQ Integrator V2.1 are very much the same as WebSphere Business Integration Message Broker V5 performance was essentially the same as that of WebSphere MQ Integrator V2.1

Use Case Outline

This section contains a brief outline of the tests used to obtain the results presented in the table above. For more detail on individual test cases see the section Appendix D - Use Case Descriptions.

- **Aggregation**
  This represents the type of processing that is required when travel is booked and arrangements for a flight, hotel, car and money must be made. Requests to four different applications are made and the replies consolidated into a single reply. This test performs the processing required to split an incoming XML message and perform a four message aggregation using the Aggregation nodes which are supplied with WebSphere Message Broker.

- **Coordinated Request Reply**
  This performs the processing needed to enable two applications with different message formats to communicate with each other. One application has a message format of self-defining XML and the other uses Custom Wire Format (CWF) messages. The request and reply processing for a particular request must be coordinated so that data from the original request is restored to the reply message.

- **Data Warehouse**
  This demonstrates a scenario in which a message flow is used to perform the archiving of data, such as sales data, into a database. The data is stored for later analysis by another message flow or application.

- **Large Messaging**
  This is based on the scenario of end-of-day processing of sales data. Messages representing sales for the day are batched together for transmission to the IT center. On receipt at the IT center the batched messages are split back out into their constituent parts for subsequent processing.

- **Message Routing**
  This shows how a message flow can be used to route messages to different WebSphere MQ queues based on data stored in a database table. This is a commonly used scenario which is applicable to many different industries and applications.

- **SWIFT Message Parse**
  This demonstrates the use of WebSphere Message Broker to read and parse a SWIFT MT543 message for subsequent processing

- **XMLT**
  This shows how a message flow can be used to transform an XML message to another form of XML message, according to the rules provided by an XSL (eXtensible Stylesheet Language) stylesheet.

Additional Information

This section contains links to information about WebSphere Message Broker and associated products.

The Web Resources section in the development toolkit of WebSphere Message Broker V6 contains links to many additional pieces of information on topics such as Education, Technical Resources and SupportPacs. The Web resources section can be accessed by selecting `Web Resources` from the Help drop down on the development toolkit menu bar.

For additional suggestions consider the following:

- See the announcement letters for
  - IBM WebSphere Message Broker V6 which is available at http://www.ibm.com/software/integration/wbimessagebroker/v6
  - IBM WebSphere Message Broker V6 for z/OS which is available at http://www.ibm.com/software/integration/wbimessagebroker/v6/zos.html
- IBM WebSphere MQ SupportPacs provide you with a wide range of downloadable code and documentation that complements the WebSphere MQ family of products. Additional performance reports are also available. These are available at http://www.ibm.com/software/integration/support/supportpacs.
- For more information about WebSphere Message Broker V6, go to the WebSphere Message Broker Web site. Product documentation is also available. This is available at http:/www.ibm.com/software/integration/wbimessagebroker.
- For more information about WebSphere MQ V6, go to the WebSphere MQ Web site. Product documentation is also available. This is available at http://www.ibm.com/software/integration/wmq.
- For more information about business integration software from IBM go to WebSphere Business Integration Web site. This is available at http://www.ibm.com/software/info1/websphere/index.jsp?tab=products/businessint.
- Get the latest WebSphere Message Broker technical resources at the WebSphere Business Integration zone. This is available at http://www.ibm.com/developerworks/websphere/zones/businessintegration.
- The JMS Node testing which was run for this report used a tool called the Performance Harness for JMS to generate and consume JMS messages. The tool is useful as a simple way to send and receive JMS messages. It also has the capability to send and receive WebSphere MQ messages. This makes it ideal for testing message flows which use the JMS nodes and which perform transformations to/from WebSphere MQ. The documentation for the tool contains examples of how to run it to send/receive messages to/from a JMS Provider. More information is available at http://www.alphaworks.ibm.com/tech/perfharness?open&S_TACT=105AGX21&S_CMP=AWRSS.
- In order to obtain the maximum message rate for your implementation it is important that you understand the current best practices for WebSphere Message Broker. These practices cover the architecture of message flow processing, the coding of message flows as well as the configuration and tuning of the message broker and associated components. Such information can be found in the Business Integration Zone of WebSphere Developer Domain. A suggested starting place is the article at http://www.ibm.com/developerworks/websphere/library/techarticles/0403_dunn/0403_dunn.html which highlights the information available and where it may be found.

Part II

This part contains the description and results of a series of tests which have been run in order to identify the processing costs of the different functions which are provided with WebSphere Message Broker.

It contains the following sections:

- *Routing and Transformation Processing Profiles* which describes the tests and shows the results obtained when a **single** copy of the message flow was run.

- *Resource Requirements* which provides a recommended minimum specification machine on which to install the product as well as some guidance on virtual memory use for execution groups running a variety of message flows.

- *Tuning* which describes the changes made to the default settings for WebSphere Message Broker V6 and WebSphere MQ in order to obtain the results detailed in this report.

- *Conclusion* which summarises the report.

Routing and Transformation Processing Profiles

This section contains the results of a series of micro tests which illustrate the costs of performing different types of processing using WebSphere Message Broker such as message parsing, message streaming, use of Filter nodes etc. These tests are not intended to represent applications. They are an illustration of the processing costs of specific functions.

The test results were all run using the same methodology. This was to run a single copy of the message flow (unless specified otherwise) to maximum CPU utilisation and to observe the message rate obtained. From this a CPU cost per message was calculated. This is presented in the results table for each measurement.

When comparing the costs of different functions it is recommended to compare them on the basis of CPU cost per message rather than message rate.

There are many comparisons which can be made using the data in this section which will give some insight into the relative costs of different implementations such as what is the relative cost of ESQL and XSLT to process the same message.

The data in this section will allow you to make a comparison on the basis of CPU costs. Other factors such as the potential for code re-use and the operational considerations of using a particular technology are not discussed.

Messages Used in Processing

For the majority of tests the message content was common. Different formats (in XML, CWF, TDS) of a common input message were used. The output message varied dependent on the test case. The messages are described in the section Appendix C – Test Messages.

For the Publish Subscribe tests a 1K JMS Bytes message was used. This was a sequence of random data. In these tests the message content was not of interest.

Results Presentation

Each of the tests are described below and accompanied by a table of data which has a format such as this:

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1K	No			
4K	No			
16K	No			
64K	No			
256K	No			
1K	Yes			
4K	Yes			
16K	Yes			
64K	Yes			
256K	Yes			

The data in the columns is as follows:

Msg Size records the approximate size of the message used as input to the test. This is the size of the message payload and does not include the size of any message header. For the Message Repository Manager (MRM) tests which use CWF and TDS message formats the input message will be smaller. This is due to the differences in the way in which data is formatted. In these cases the input message will still contain the same amount of information but it will be the CWF or TDS representation of the generic XML representation of the same

data. Most test cases used messages of 1K, 4K, 16K, 64K and 256K. In some cases a more limited range of message sizes was run where the test was not suitable for the whole range of message sizes.

Persistent: Indicates whether the messages used in the test were persistent or not

Message Rate: The number of round trips or message flow invocations per second

% CPU Busy: System busy CPU percentage on the server machine. This includes the CPU used by all processes (WebSphere Message Broker, WebSphere MQ queue manager, database manager etc) on the system under test. The rate is expressed as a percentage utilisation of all processors on the machine.

Note: As Hyper-threading was running in the hardware and this was recognised by the operating system all system CPU utilisations are reported as though the machine is an 8 CPU machine although there are only physically 4 CPUs. See also the note on CPU ms/msg below. A CPU utilisation of 12.5% represents the equivalent of fully utilising one CPU on the machine.

CPU ms/msg: Overall CPU cost per message, expressed as CPU milliseconds per message. The value is obtained using the calculation:

((Number of CPUs * 1000) * (%CPU/100)) / Message Rate.

This cost includes WebSphere Message Broker, WebSphere MQ, DB2, operating system costs etc. The CPU ms/msg figures reported are specific to the machine on which they were obtained and if projections of message processing capacity are to be made for other machines a suitable adjustment must be made in the costs to allow for differences in the capacity of the two systems.

Note: Although Hyper-threading was in effect and this increased the number of effective CPUs to 8 there were only 4 physical CPUs installed in the machine. The CPU cost per message is based on the use of 4 CPUs.

Response Times

Response time data for the message flow execution is not reported. The tests are configured to maximise message throughput and minimise CPU costs. As such tests always have a number of messages waiting on the input node of the message flow so that there is a message ready to be processed immediately after processing of the current message has completed. This means that the processing of each message involves queuing time at the input node. Because of this it is not meaningful to report message processing times as observed by the client as it will not reflect the true execution time in the message flow.

It is possible to estimate the elapsed time within a message flow in milliseconds from the results of these tests by dividing 1000 (representing the number of milliseconds in 1 second) by the message rate for the test.

For example let us suppose that a test achieved a message rate of 2000 per second. The message flow average execution time is 1000 / 2000 = 0.5ms. For a message rate of 200 per second the average execution time is 1000/200 = 5ms.

These times are an estimate of the execution time in the message flow and as such represent the elapsed time between the message being read from the input queue and the result being placed on the output queue.

If messages are generated or consumed by remote clients an allowance needs to be made for network delays.

The test descriptions and results follow.

Minimal Processing

The test in this section illustrates some of the simplest processing which can be performed with WebSphere Message Broker. As such it illustrates the smallest processing cost that you could expect for a message flow. This is not typical of the majority of implementations of Message Broker though. The data is provided for reference purposes only to help you understand the maximum rate that could be expected for one copy of the message flow.

Typically the processing within a message flow involves message parsing, processing logic and message serialisation. Under these circumstances the CPU processing costs can increase significantly and as such the message rate obtained for given amount of CPU will be lower than for the very simple type of flow presented in this section.

Setting of the MQ Message Headers

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers for the outgoing message are created using ESQL. To minimise processing costs only the CodedCharSetId and Encoding fields in the MQMD header are set. The message body is ignored and therefore not used in the output message.

This test identifies the cost of setting the message header only and creating an output message with no payload.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	2308.00	16.20	0.28
1k	Yes	316.20	4.00	0.51

Message Parsing and Writing

The tests in this section illustrate the cost of parsing input messages and writing output messages for different message formats.

Parsing a Message in the MRM Domain

The tests in this section illustrate the CPU processing costs of parsing different message formats in the MRM domain.

Parsing a Tagged Delimited String, All Elements Delimited Input Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a variable is declared and set to the last element in the incoming message. This causes a full parse of the incoming message. The output message consists of a message header only and no payload.

This test identifies the cost of parsing an All Elements Delimited, Tagged Delimited String input message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	513.60	13.60	1.06
4k	No	237.40	13.00	2.19
16k	No	75.40	13.00	6.90
64k	No	20.12	13.00	25.84
256k	No	5.10	13.00	101.96
1k	Yes	174.80	6.00	1.37
4k	Yes	163.80	10.60	2.59
16k	Yes	66.20	11.60	7.01
64k	Yes	19.18	12.00	25.03
256k	Yes	5.06	13.00	102.77

Parsing a Tagged Delimited String, Fixed Length Input Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a variable is declared and set to the last element in the incoming message. This causes a full parse of the incoming message. The output message consists of a message header only and no payload.

This test identifies the cost of parsing a Fixed Length, Tagged Delimited String input message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	494.60	14.00	1.13
4k	No	224.80	13.00	2.31
16k	No	70.40	13.00	7.39
64k	No	18.38	13.00	28.29
256k	No	4.64	13.00	112.17
1k	Yes	170.00	6.40	1.51
4k	Yes	150.20	11.00	2.93
16k	Yes	60.20	12.00	7.97
64k	Yes	17.44	12.20	27.98
256k	Yes	4.59	13.00	113.39

Parsing a Tagged Delimited String, Tagged Delimited Input Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a variable is declared and set to the last element in the incoming message. This causes a full parse of the incoming message. The output message consists of a message header only and no payload.

This test identifies the cost of parsing a Tagged Delimited String, Tagged Delimited input message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	430.60	13.80	1.28
4k	No	175.20	13.00	2.97
16k	No	52.00	13.00	10.00
64k	No	13.42	13.00	38.75
256k	No	3.39	13.00	153.48
1k	Yes	173.20	7.20	1.66
4k	Yes	129.40	11.00	3.40
16k	Yes	46.78	12.00	10.26
64k	Yes	13.02	12.40	38.10
256k	Yes	3.36	13.00	154.67

Parsing a Tagged Delimited String, Tagged Fixed Length Input Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a variable is declared and set to the last element in the incoming message. This causes a full parse of the incoming message. The output message consists of a message header only and no payload.

This test identifies the cost of parsing a Tagged Fixed Length, Tagged Delimited String input message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	414.20	14.00	1.35
4k	No	166.60	13.00	3.12
16k	No	48.98	13.00	10.62
64k	No	12.70	13.00	40.94
256k	No	3.19	13.00	163.11
1k	Yes	187.40	8.00	1.71
4k	Yes	123.00	11.00	3.58
16k	Yes	43.94	12.00	10.92
64k	Yes	12.20	13.00	42.62
256k	Yes	3.14	13.00	165.61

Parsing an MRM XML Input Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a variable is declared and set to the last element in the incoming message. This causes a full parse of the incoming message. The output message consists of a message header only and no payload.

This test identifies the cost of parsing an MRM XML input message.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	388.60	13.00	1.34
4k	No	182.80	13.00	2.84
16k	No	58.20	13.00	8.93
64k	No	15.92	13.00	32.66
256k	No	4.07	13.00	127.70
1k	Yes	183.40	8.00	1.74
4k	Yes	128.00	11.00	3.44
16k	Yes	51.40	12.40	9.65
64k	Yes	15.26	12.00	31.45
256k	Yes	4.02	12.60	125.25

Parsing a Custom Wire Format Input Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a variable is declared and set to the last element in the incoming message. This causes a full parse of the incoming message. The output message consists of a message header only and no payload.

This test identifies the cost of parsing a Custom Wire Format input message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	680.00	14.00	0.82
4k	No	285.00	13.00	1.82
16k	No	74.00	13.00	7.03
64k	No	18.94	13.00	27.46
256k	No	4.70	13.00	110.59
1k	Yes	209.60	6.00	1.15
4k	Yes	176.00	10.00	2.27
16k	Yes	64.20	11.80	7.35
64k	Yes	18.00	12.40	27.56
256k	Yes	4.63	13.00	112.36

Parsing a Comma Separated Value Input Message using Data Patterns

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the entire incoming message is copied to the outgoing message. In addition the format of the outgoing message is set to XML. This causes a full parse of the incoming message using the Tagged Delimited String Parser and a full write of the outgoing message using the Generic XML Writer.

This test identifies the cost of converting an incoming Comma Separated Value input message using the Data Pattern function with the Tagged Delimited String Parser, to an outgoing Generic XML Message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	240.20	13.00	2.16
1k	Yes	165.80	10.80	2.61

Parsing a SWIFT 543 Input Message using the Tagged Delimited String Parser

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a variable is declared and set to the last element in the incoming message. This causes a full parse of the incoming message. The output message consists of a message header only and no payload.

This test identifies the cost of parsing a SWIFT MT543 message using the Tagged Delimited String format. A single implementation of this message was used which was approximately 7K in size.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
7k	No	33.44	13.20	15.79
7k	Yes	31.68	12.00	15.15

Parsing and Writing a SWIFT 543 Input Message using the Tagged Delimited String Parser

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the Envelope within the incoming SWIFT Message is copied over to the outgoing message. This causes a full parse of the incoming message and a full serialisation of the outgoing message.

This test identifies the cost of parsing a SWIFT MT543-message and serializing it again using the Tagged Delimited String format. A single implementation of this message was used which was approximately 7K in size.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
7k	No	18.34	12.80	27.92
7k	Yes	17.54	12.40	28.28

Parsing a TLOG SA Input Message using the Tagged Delimited String Parser

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a variable is declared and set to the last element in the incoming message. This causes a full parse of the incoming message. The output message consists of a message header only and no payload.

This test identifies the cost of parsing a TLOG input message using the Tagged Delimited String format.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	33.48	13.00	15.53
2k	Yes	31.36	12.00	15.31

Parsing a JMS SOAP message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

The incoming JMS SOAP message is parsed. The output message consists of a message header only and no payload.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	433.80	13.40	1.24
1k	Yes	185.20	7.00	1.51

Parsing a JMS SOAP Message with Attachments

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

The incoming MIME JMS message is parsed and the soap envelope extracted and parsed using the MIME Parser within the MRM Domain. The output message consists of a message header only and no payload.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	118.60	8.00	2.70
1k	Yes	95.40	9.00	3.77

Parsing a JMS SOAP Message with Attachments and converting it to a Standard SOAP message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

The incoming MIME JMS message is parsed and the soap envelope extracted and parsed using the MRM domain parser. The extracted SOAP message is written to the output queue as a standard SOAP message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	52.00	14.20	10.92
1k	Yes	30.04	10.00	13.32

Parsing and Writing a JMS SOAP Message and Modifying a Field

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

The incoming JMS SOAP message is parsed. One field is modified and the resulting message is written to the output queue.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	114.20	13.40	4.69
1k	Yes	93.40	12.00	5.14

Writing a Message in the MRM Domain

The tests in this section illustrate the CPU processing costs of creating an output message with different formats in the MRM domain. This is the processing associated with taking a message tree in OutputRoot and flattening it to create a bitstream which is the output message.

Writing a Tagged Delimited String, All Elements Delimited Output Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition the incoming Generic XML message is converted to an All Elements Delimited, Tagged Delimited String outgoing message. This causes a full parse of the incoming message payload which is then written as the payload of the output message.

This test identifies the cost of parsing a Generic XML message and writing out an All Elements Delimited, Tagged Delimited String output message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	382.00	14.00	1.47
4k	No	168.60	13.00	3.08
16k	No	51.40	13.00	10.12
64k	No	13.62	13.00	38.18
256k	No	3.40	13.00	152.94
1k	Yes	186.40	8.60	1.85
4k	Yes	117.00	11.00	3.76
16k	Yes	45.86	12.00	10.47
64k	Yes	13.02	12.00	36.87
256k	Yes	3.35	12.80	152.93

Writing a Tagged Delimited String, Fixed Length Output Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition the incoming Generic XML message is converted to a Fixed Length, Tagged Delimited String outgoing message. This causes a full parse of the incoming message payload which is then written as the payload of the output message.

This test identifies the cost of parsing a Generic XML message and writing out a Fixed Length, Tagged Delimited String output message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	378.00	14.00	1.48
4k	No	164.60	13.00	3.16
16k	No	50.40	13.00	10.32
64k	No	13.24	13.00	39.27
256k	No	3.28	13.00	158.54
1k	Yes	182.00	8.00	1.76
4k	Yes	117.00	11.00	3.76
16k	Yes	44.84	12.60	11.24
64k	Yes	12.62	12.00	38.03
256k	Yes	3.23	12.80	158.51

Writing a Tagged Delimited String, Tagged Fixed Length Output Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition the incoming Generic XML message is converted to a Tagged Fixed Length, Tagged Delimited String outgoing message. This causes a full parse of the incoming message payload which is then written as the payload of the output message.

This test identifies the cost of parsing a Generic XML message and writing out a Tagged Fixed Length, Tagged Delimited String output message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	373.80	14.00	1.50
4k	No	163.80	12.80	3.13
16k	No	50.20	13.00	10.36
64k	No	13.42	13.00	38.75
256k	No	3.33	12.80	153.85
1k	Yes	183.20	8.40	1.83
4k	Yes	115.40	10.80	3.74
16k	Yes	44.74	12.40	11.09
64k	Yes	12.66	12.00	37.91
256k	Yes	3.25	12.00	147.69

Writing a Tagged Delimited String, Tagged Delimited Output Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition the incoming Generic XML message is converted to a Tagged Delimited String, Tagged Delimited outgoing message. This causes a full parse of the incoming message payload which is then written as the payload of the output message.

This test identifies the cost of parsing a Generic XML message and writing out a Tagged Delimited String output message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	355.20	13.80	1.55
4k	No	148.40	13.00	3.50
16k	No	44.32	13.00	11.73
64k	No	11.70	13.00	44.44
256k	No	2.95	13.00	176.15
1k	Yes	180.40	9.00	2.00
4k	Yes	112.20	11.80	4.21
16k	Yes	39.80	12.00	12.06
64k	Yes	11.20	12.00	42.86
256k	Yes	2.88	12.00	166.55

Writing an MRM XML Output Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition the incoming Generic XML message is converted to an MRM XML outgoing message. This causes a full parse of the incoming message payload which is then written as the payload of the output message.

This test identifies the cost of parsing a Generic XML message and writing out an MRM XML output message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	308.40	13.80	1.79
4k	No	118.20	13.00	4.40
16k	No	33.88	13.00	15.35
64k	No	8.84	13.00	58.82
256k	No	2.22	13.00	234.66
1k	Yes	177.20	9.00	2.03
4k	Yes	94.60	12.00	5.07
16k	Yes	31.16	12.00	15.40
64k	Yes	8.42	12.00	57.01
256k	Yes	2.14	12.00	224.72

Writing a Custom Wire Format Output Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition the incoming Generic XML message is converted to a Custom Wire Format outgoing message. This causes a full parse of the incoming message payload which is then written as the payload of the output message.

This test identifies the cost of parsing a Generic XML message and writing out a Custom Wire Format output message.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	453.00	14.00	1.24
4k	No	201.60	13.00	2.58
16k	No	62.20	12.80	8.23
64k	No	16.38	13.00	31.75
256k	No	4.10	13.00	126.83
1k	Yes	187.80	7.40	1.58
4k	Yes	137.40	11.00	3.20
16k	Yes	54.20	12.80	9.45
64k	Yes	15.58	12.00	30.81
256k	Yes	4.01	12.60	125.75

Parsing a Message in the XML Domain

The tests in this section illustrate the CPU processing costs of parsing different message formats in the XML domain.

Parsing a Generic XML Input Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a variable is declared and set to the last element in the

incoming message. This causes a full parse of the incoming message. The output message consists of a message header only and no payload.

This test identifies the cost of parsing a Generic XML input message. As the message body is ignored there are no writing costs.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	1109.00	14.40	0.52
4k	No	651.80	14.00	0.86
16k	No	242.40	14.00	2.31
64k	No	68.00	13.20	7.76
256k	No	17.24	14.00	32.48
1k	Yes	249.60	5.00	0.80
4k	Yes	230.00	7.00	1.22
16k	Yes	151.60	10.00	2.64
64k	Yes	55.80	12.00	8.60
256k	Yes	15.82	13.00	32.87

Parsing a Generic XML Input Message Containing XML Entities

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a variable is declared and set to the last element in the incoming message. This causes a full parse of the incoming message including the tags and entities. The output message consists of a message header only and no payload.

This test identifies the cost of parsing a Generic XML input message containing many XML Entities to see what the effect of having entities present in the message is.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	1723.80	15.00	0.35
64k	No	95.20	14.00	5.88

Writing a Message in the XML Domain

The test in this section illustrates the CPU processing costs of using the XML domain to create an output message. This is the processing associated with taking a message tree in OutputRoot and flattening it to create a bitstream which is the output message.

Writing a Generic XML Output Message

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the entire Message from the incoming message is copied over to the outgoing message. In addition the last element in the incoming message is modified. This causes a full parse of the incoming message which is then written as the payload of the output message.

This test identifies the cost of parsing a Generic XML input message and writing a Generic XML output message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	763.40	14.00	0.73
4k	No	346.80	13.60	1.57
16k	No	108.00	12.40	4.59
64k	No	28.96	13.00	17.96
256k	No	7.10	13.20	74.37
1k	Yes	224.00	6.00	1.07
4k	Yes	169.20	8.00	1.89
16k	Yes	81.80	11.00	5.38
64k	Yes	25.10	12.80	20.40
256k	Yes	6.50	13.00	80.00

External Resources

The tests in this section illustrate the processing cost of accessing resources such as a database or external procedure.

Accessing a Database from a Message Flow

The tests in this section illustrate the processing cost of performing operations on a DB2 database.

Reading from a Database

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition a SELECT is performed to obtain a piece of data from the Database. This data is used to validate an element in the input message.

This test identifies the cost of performing a Database SELECT.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	736.60	13.60	0.74
4k	No	685.60	14.00	0.82
16k	No	548.80	15.00	1.09
64k	No	182.00	7.00	1.54
256k	No	45.98	3.80	3.31
1k	Yes	241.20	6.60	1.09
4k	Yes	226.40	7.00	1.24
16k	Yes	198.40	8.00	1.61
64k	Yes	137.20	7.80	2.27
256k	Yes	43.32	6.00	5.54

Inserting into a Database

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition an INSERT is performed to populate the database with a piece of data. This data is obtained from an element in the input message.

This test identifies the cost of performing a Database INSERT.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	301.20	6.00	0.80
4k	No	292.20	6.00	0.82
16k	No	268.00	6.80	1.01
64k	No	181.60	6.00	1.32
256k	No	46.08	3.20	2.78
1k	Yes	158.80	4.00	1.01
4k	Yes	165.00	5.00	1.21
16k	Yes	149.40	5.00	1.34
64k	Yes	114.00	6.00	2.11
256k	Yes	42.22	6.00	5.68

Updating a row in a Database

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. In addition an UPDATE is performed to update a piece of data in the database with a new value. This value is obtained from an element in the input message.

This test identifies the cost of performing a Database UPDATE.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	688.80	13.80	0.80
4k	No	674.00	14.40	0.85
16k	No	618.80	15.00	0.97
64k	No	181.80	6.00	1.32
256k	No	46.02	3.00	2.61
1k	Yes	240.20	7.00	1.17
4k	Yes	228.80	7.00	1.22
16k	Yes	211.20	8.00	1.52
64k	Yes	141.40	7.00	1.98
256k	Yes	43.14	6.00	5.56

Calling External Procedures

The tests in this section illustrate the processing cost of invoking an external procedure such as a Java class or database stored procedure with different parameters.

Calling an External Java Procedure with no Parameters

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. Two thousand identical calls are made to an external Java procedure. The procedure receives zero input parameters and passes back zero parameters returning immediately.

This test identifies the cost of calling a Java procedure with zero parameters.

The results of running this test are given in the table below. The CPU ms/msg figure has been adjusted to report a per procedure invocation cost by dividing the CPU cost obtained from the test results by 2000.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	172.000	13.000	0.002
1k	Yes	131.600	11.200	0.002

Calling an External Java Procedure with One Integer Input Parameter

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. Two thousand identical calls are made to an external Java procedure. The procedure receives one Integer parameter and passes back zero parameters returning immediately.

This test identifies the cost of calling a Java procedure with one Integer parameter.

The results of running this test are given in the table below. The CPU ms/msg figure has been adjusted to report a per procedure invocation cost by dividing the CPU cost obtained from the test results by 2000.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	69.600	13.000	0.004
1k	Yes	61.800	11.400	0.004

Calling an External Java Procedure with Twenty Integer Input Parameters

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. Two thousand identical calls are made to an external Java procedure. The procedure receives twenty parameters all of which are integers and passes back zero parameters returning immediately.

This test identifies the cost of calling a Java procedure with twenty parameters which are integers.

The results of running this test are given in the table below. The CPU ms/msg figure has been adjusted to report a per procedure invocation cost by dividing the CPU cost obtained from the test results by 2000.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	17.060	13.000	0.015
1k	Yes	16.400	12.000	0.015

Calling an External Database Stored Procedure with no Parameters.

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. Two thousand identical calls are made to an external database stored procedure. The procedure receives zero input parameters and passes back zero parameters returning immediately.

This test identifies the cost of calling a Database Stored procedure with zero parameters.

The results of running this test are given in the table below. The CPU ms/msg figure has been adjusted to report a per procedure invocation cost by dividing the CPU cost obtained from the test results by 2000.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	4.416	13.000	0.059
1k	Yes	4.298	13.000	0.060

Calling an External Database Stored Procedure with One Integer Input Parameter

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. Two thousand identical calls are made to an external database stored procedure. The procedure receives one parameter which is an integer and passes back zero parameters returning immediately.

This test identifies the cost of calling a Database Stored procedure with one parameter which is an integer.

The results of running this test are given in the table below. The CPU ms/msg figure has been adjusted to report a per procedure invocation cost by dividing the CPU cost obtained from the test results by 2000.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	4.156	13.000	0.063
1k	Yes	4.054	13.000	0.064

Calling an External Database Stored Procedure with Twenty Integer Input Parameters

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. Two thousand identical calls are made to an external database stored procedure. The procedure receives twenty parameters which are integers and passes back zero parameters returning immediately.

This test identifies the cost of calling a Database Stored procedure with twenty parameters which are integers.

The results of running this test are given in the table below. The CPU ms/msg figure has been adjusted to report a per procedure invocation cost by dividing the CPU cost obtained from the test results by 2000.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	3.038	13.000	0.086
1k	Yes	3.014	12.800	0.085

JMS Nodes

The tests in this section illustrate the processing cost of utilising JMS messages, which are new in WebSphere Message Broker V6, in a variety of ways.

Receiving and sending JMS Messages

This test consists of JMSInput Node -> JMSOutput Node

The JMSInput Node acts as a JMS Subscriber to a JMS Provider.

The JMS Output Node acts as a Topic Publisher and sends the same message to another JMS Provider.

For this test the JMS Provider was a separate WebSphere Message Broker running a Real-time Optimized flow.

This test uses a JMS Bytes message.

This test identifies the cost of receiving a JMS Message from a JMS Provider and publishing that same message to another JMS Provider.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	895.00	14.00	0.63
4k	No	780.00	15.00	0.77
16k	No	569.00	15.00	1.05
64k	No	156.00	10.00	2.56

Note: The maximum supported message size for the WebSphere MQ Real-time client is 100K and so it was not possible to measure with 256K messages.

JMS to MQ Protocol conversion

This test consists of JMSInput Node -> JMSMQTransform Node -> MQOutput Node

The JMSInput Node is configured to receive a JMS bytes message as a subscriber to a JMS Provider.

Within the JMSMQTransform node the tree built from the JMS input message is converted to one suitable for the MQ transport.

An MQ output message is written.

For this test the JMS Provider was a separate WebSphere Message Broker running a Real-time Optimized flow.

This test identifies the cost of converting a JMS Message to an MQ Message.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	950.00	15.00	0.63
4k	No	805.00	15.00	0.75
16k	No	590.00	16.00	1.08
64k	No	120.00	8.00	2.67

Note: The maximum supported message size for the WebSphere MQ Real-time client is 100K and so it was not possible to measure with 256K messages.

Receiving and sending JMS Messages with an XML Transformation

This test consists of JMSInput Node ->XMLT Node-> JMSOutput Node

The JMSInput Node is configured to receive a JMS Text message as a subscriber to a JMS Provider.

Within the XMLT node a compiled stylesheet is used to significantly change the structure of the incoming message. The new structure is written as the output message.

The JMS Output Node acts as a Topic Publisher and sends the same transformed message to another JMS Provider.

This test identifies the cost of using JMS input and output messages with an XSL stylesheet to perform message manipulation.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	274.00	15.00	2.19
4k	No	149.00	15.00	4.03
16k	No	55.00	15.00	10.91
64k	No	15.00	15.00	40.00

Note: The maximum supported message size for the WebSphere MQ Real-time client is 100K and so it was not possible to measure with 256K messages.

JMS to MQ Protocol conversion with an XML Transformation

This test consists of

JMSInput Node -> JMSMQTransform Node ->XMLT Node-> MQOutput Node

The JMSInput Node is configured to receive a JMS Text message as a subscriber to a JMS Provider.

Within the JMSMQTransform node the tree built from the JMS input message is converted to one suitable for the MQ transport.

Within the XMLT node a compiled stylesheet is used to significantly change the structure of the incoming message. The new structure is written as an MQ output message.

This test identifies the cost of using a JMS input message and MQ output message with an XSL stylesheet perform message manipulation.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	197.00	14.00	2.84
4k	No	98.00	14.00	5.71
16k	No	34.00	14.00	16.47
64k	No	9.00	14.00	62.22

Note: The maximum supported message size for the WebSphere MQ Real-time client is 100K and so it was not possible to measure with 256K messages.

Routing and Transformation Logic

The tests in this section illustrate the processing cost of simple routing and transformation logic using a variety of routing and transformation technologies (ESQL, JavaCompute node, XML Transformation). A number of the tests are performed for each of the technologies thus allowing a simple comparison of CPU processing costs to be made. In other cases a comparison is only made within a technology such as looking at the efficiency of different parsers whilst using ESQL.

These tests are not a definitive statement of the relative processing costs of the different technologies. They are provided for illustrative purposes only. Message processing performance will be affected by the complexity of the messages and processing to be performed on the messages.

Using ESQL

The tests in this section illustrate the processing costs of using ESQL for different routing and transformation operations.

Filter an Incoming Message based on the First Element in the Message using the XML Parser

This test consists of MQ Input Node -> Filter Node -> MQ Output Node.

Within the filter node the first element of the incoming message is examined. The result is always set to be true and thus the message is propagated to the MQOutput node.

This test identifies the cost of filtering on an element at the start of a message using the XML parser.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	2389.00	15.40	0.26
4k	No	1899.80	15.00	0.32
16k	No	663.00	10.00	0.60
64k	No	163.60	4.00	0.98
256k	No	38.52	3.00	3.12
1k	Yes	337.80	4.40	0.52
4k	Yes	292.20	4.00	0.55
16k	Yes	231.60	6.00	1.04
64k	Yes	101.20	6.00	2.37
256k	Yes	31.16	6.60	8.47

Filter an Incoming Message Based on the Last Element in the Message using the XML Parser

This test consists of MQ Input Node -> Filter Node -> MQ Output Node.

Within the filter node the last element of the incoming message is examined. The result is always set to be true and thus the message is propagated to the MQOutput node.

This test identifies the cost of filtering on an element at the end of a message using the XML parser.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	1569.80	14.20	0.36
4k	No	742.20	13.80	0.74
16k	No	238.80	14.00	2.35
64k	No	63.00	13.80	8.76
256k	No	15.52	14.00	36.08
1k	Yes	251.20	4.00	0.64
4k	Yes	183.40	5.00	1.09
16k	Yes	136.20	9.00	2.64
64k	Yes	43.46	10.40	9.57
256k	Yes	12.62	11.80	37.40

Filter an Incoming Message Based on the First Element in the Message using the XMLNSC Parser

This test consists of MQ Input Node -> Filter Node -> MQ Output Node.

Within the filter node the first element of the incoming message is examined. The result is always set to be true and thus the message is propagated to the MQOutput node.

This test identifies the cost of filtering on an element at the start of a message using the XMLNSC parser.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	2479.20	16.20	0.26
4k	No	2035.20	16.00	0.31
16k	No	661.80	8.00	0.48
64k	No	163.80	4.00	0.98
256k	No	38.66	3.00	3.10
1k	Yes	332.20	4.20	0.51
4k	Yes	295.40	4.20	0.57
16k	Yes	224.40	5.20	0.93
64k	Yes	102.40	6.00	2.34
256k	Yes	31.16	6.00	7.70

Filter an Incoming Message Based on the Last Element in the Message using the XMLNSC Parser

This test consists of MQ Input Node -> Filter Node -> MQ Output Node.

Within the filter node the last element of the incoming message is examined. The result is always set to be true and thus the message is propagated to the MQOutput node.

This test identifies the cost of filtering on an element at the end of a message using the XMLNSC parser.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	1334.60	14.20	0.43
4k	No	562.20	14.00	1.00
16k	No	170.00	13.00	3.06
64k	No	45.16	13.00	11.51
256k	No	11.14	13.40	48.11
1k	Yes	243.40	4.80	0.79
4k	Yes	178.40	6.00	1.35
16k	Yes	116.20	10.00	3.44
64k	Yes	36.60	12.00	13.11
256k	Yes	9.94	12.40	49.90

Computation on an Input Message using the XML Parser

This test consists of MQ Input node -> Compute Node -> MQ Output Node.

Within the compute node ESQL is used to calculate the total of all items and prices within a repeating structure which is in the input message. The totals along with a copy of the input message are written in the outgoing message.

This test identifies the cost of using ESQL to perform computation and message parsing using the XML parser.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	630.80	14.00	0.89
4k	No	236.00	13.60	2.31
16k	No	66.40	13.00	7.83
64k	No	16.64	13.00	31.25
256k	No	3.93	13.00	132.25
1k	Yes	222.00	6.80	1.23
4k	Yes	155.20	10.60	2.73
16k	Yes	55.60	12.00	8.63
64k	Yes	15.30	12.40	32.42
256k	Yes	3.72	12.40	133.26

Computation on an Input Message using the XMLNSC Parser

This test consists of MQ Input node -> Compute Node -> MQ Output Node.

Within the compute node ESQL is used to calculate the total of all items and prices within a repeating structure which is in the input message. The totals along with a copy of the input message are written out in the outgoing message.

This test identifies the cost of using ESQL to perform computation and message parsing using the XMLNSC parser.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	552.40	14.00	1.01
4k	No	201.80	13.00	2.58
16k	No	57.00	13.00	9.12
64k	No	14.44	13.20	36.57
256k	No	3.45	13.00	150.55
1k	Yes	217.80	7.00	1.29
4k	Yes	141.40	11.00	3.11
16k	Yes	49.30	12.20	9.90
64k	Yes	13.42	12.00	35.77
256k	Yes	3.32	12.00	144.58

Manipulation of an Input Message using the XML Parser

This test consists of MQ Input node -> Compute Node -> MQ Output Node.

Within the compute node ESQL is written to significantly change the structure of the incoming message. The new structure is written as the output message

This test identifies the cost of using ESQL to perform message manipulation and message parsing using the XML parser.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	580.20	14.00	0.97
4k	No	206.80	13.00	2.51
16k	No	56.60	13.00	9.19
64k	No	13.34	13.00	38.98
256k	No	2.58	12.80	198.76
1k	Yes	223.80	7.00	1.25
4k	Yes	144.20	11.00	3.05
16k	Yes	49.34	13.00	10.54
64k	Yes	12.52	12.00	38.34
256k	Yes	2.51	12.60	200.64

Manipulation of an Input Message using the XMLNSC Parser

This test consists of MQ Input node -> Compute Node -> MQ Output Node.

Within the compute node ESQL is written to significantly change the structure of the incoming message. The new structure is written as the output message

This identifies the cost of using ESQL to perform message manipulation and message parsing using the XMLNSC parser.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	566.60	14.00	0.99
4k	No	202.40	13.40	2.65
16k	No	55.20	13.00	9.42
64k	No	13.20	13.00	39.39
256k	No	2.58	13.00	201.71
1k	Yes	227.20	7.20	1.27
4k	Yes	135.00	10.40	3.08
16k	Yes	48.44	12.20	10.07
64k	Yes	12.46	12.00	38.52
256k	Yes	2.51	12.00	191.08

Manipulation of an Input Message using the EVAL Function on all Lines of ESQL in One Invocation

This test consists of MQ Input node -> Compute Node -> MQ Output Node.

Within the compute node ESQL is written to significantly change the structure of the incoming message. The new structure is written as the output message The ESQL processing is run within a single EVAL statement.

This test identifies the cost of using the EVAL function to run a large amount of ESQL

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	132.20	13.00	3.93
4k	No	81.00	13.00	6.42
16k	No	23.26	13.00	22.36
64k	No	12.10	12.60	41.65
256k	No	2.51	13.00	207.34
1k	Yes	106.00	11.80	4.45
4k	Yes	78.20	11.60	5.93
16k	Yes	37.70	12.00	12.73
64k	Yes	11.62	12.00	41.31
256k	Yes	2.46	12.40	201.63

Manipulation of an Input Message using the EVAL Function on Each Line of ESQL

This test consists of MQ Input node -> Compute Node -> MQ Output Node.

Within the compute node ESQL is written to significantly change the structure of the incoming message. The new structure is written as the output message. Each line of ESQL is run individually in an EVAL statement

This test identifies the cost of using the EVAL function on many lines of ESQL.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	66.60	12.40	7.45
4k	No	17.78	12.40	27.90
16k	No	4.49	12.80	113.98
64k	No	1.12	12.60	450.81
256k	No	0.27	12.60	1852.94
1k	Yes	59.20	12.00	8.11
4k	Yes	17.00	12.00	28.24
16k	Yes	4.42	12.00	108.70
64k	Yes	1.11	13.00	468.47
256k	Yes	0.27	12.40	1837.04

Manipulation of an Input Message Using the SELECT Function

This test consists of MQ Input node -> Compute Node -> MQ Output Node.

Within the compute node an ESQL SELECT function is written to significantly change the structure of the incoming message. The new structure is written as the output message.

This test identifies the cost of using an ESQL SELECT function to perform message manipulation.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	612.00	14.00	0.92
4k	No	223.20	13.00	2.33
16k	No	63.00	13.00	8.25
64k	No	16.10	12.80	31.80
256k	No	4.04	13.00	128.78
1k	Yes	226.40	7.00	1.24
4k	Yes	152.20	10.80	2.84
16k	Yes	53.80	12.20	9.07
64k	Yes	15.02	12.00	31.96
256k	Yes	3.87	12.00	124.16

Manipulation of an Input Message using the ROW Function

This test consists of MQ Input node -> Compute Node -> MQ Output Node.

Within the compute node the ESQL SELECT function within an ESQL ROW function is written to significantly change the structure of the incoming message. The new structure is written as the output message

This test identifies the cost of using an ESQL ROW function to perform message manipulation.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	607.60	14.00	0.92
4k	No	221.80	13.00	2.34
16k	No	62.20	13.00	8.36
64k	No	16.04	12.80	31.92
256k	No	4.00	13.00	130.00
1k	Yes	227.00	7.00	1.23
4k	Yes	150.20	10.80	2.88
16k	Yes	53.60	12.40	9.25
64k	Yes	14.90	12.00	32.21
256k	Yes	3.82	12.40	129.71

Manipulation of an Input Message using the ITEM Function

This test consists of MQ Input node -> Compute Node -> MQ Output Node.

Within the compute node an ESQL SELECT function using the ITEM clause is written to significantly change the structure of the incoming message. The new structure is written as the output message.

This test identifies the cost of using an ESQL ITEM function to perform message manipulation.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	577.20	13.60	0.94
4k	No	215.20	12.00	2.23
16k	No	61.20	12.40	8.10
64k	No	15.80	13.00	32.91
256k	No	3.97	13.00	131.11
1k	Yes	226.00	7.00	1.24
4k	Yes	151.40	11.00	2.91
16k	Yes	53.80	12.20	9.07
64k	Yes	14.94	12.20	32.66
256k	Yes	3.85	12.20	126.75

Calling an Internal ESQL Procedure with No Parameters

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. Two thousand identical calls are made to an internal ESQL procedure. The procedure receives zero input parameters and passes back zero parameters returning immediately.

This test identifies the cost of calling an ESQL procedure with zero parameters.

The results of running this test are given in the table below. The CPU ms/msg figure has been adjusted to report a per procedure invocation cost by dividing the CPU cost obtained from the test results by 2000.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	477.800	14.000	0.001
1k	Yes	193.600	7.200	0.001

Calling an Internal ESQL Procedure with One Integer Input Parameter

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message. Two thousand identical calls are made to an internal ESQL procedure. The procedure receives one integer parameter and passes back zero parameters returning immediately.

This test identifies the cost of calling an ESQL procedure with one Integer parameter.

The results of running this test are given in the table below. The CPU ms/msg figure has been adjusted to report a per procedure invocation cost by dividing the CPU cost obtained from the test results by 2000.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	329.200	13.200	0.001
1k	Yes	181.000	8.800	0.001

Calling an Internal ESQL Stored Procedure with Twenty Integer Input Parameters

This test consists of MQ Input Node -> Compute Node -> MQ Output Node.

Within the compute node the message headers from the incoming message are copied over to the outgoing message Two thousand identical calls are made to an internal ESQL procedure. The procedure receives twenty parameters all of which are integers and passes back zero parameters returning immediately.

This test identifies the cost of calling an ESQL procedure with twenty parameters which are integers.

The results of running this test are given in the table below. The CPU ms/msg figure has been adjusted to report a per procedure invocation cost by dividing the CPU cost obtained from the test results by 2000.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	121.600	13.000	0.002
1k	Yes	100.200	11.600	0.002

Using Java

The tests in this section illustrate the processing costs of using the JavaCompute node for different routing and transformation operations.

Filter an Incoming Message Based on the First Element in the Message using the Java Compute Nodes XPath Capability

This test consists of MQ Input Node -> Java Compute Node -> MQ Output Node.

Within the Java Compute Node the first element of the incoming message is examined using the XPath capability. The result is always set to be true and thus the message is propagated to the MQOutput node.

This test identifies the cost of filtering on an element at the start of a message using the Java Compute Node XPath capability.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	2296.00	16.40	0.29
4k	No	1835.00	14.60	0.32
16k	No	661.60	10.00	0.60
64k	No	163.60	4.40	1.08
256k	No	38.62	3.00	3.11
1k	Yes	328.80	4.20	0.51
4k	Yes	292.40	4.40	0.60
16k	Yes	218.00	6.00	1.10
64k	Yes	95.80	5.80	2.42
256k	Yes	29.64	6.00	8.10

Filter an Incoming Message Based on the Last Element in the Message using the Java Compute Nodes XPath Capability

This test consists of MQ Input Node -> Java Compute Node -> MQ Output Node.

Within the Java Compute Node the last element of the incoming message is examined using the XPath capability. The result is always set to be true and thus the message is propagated to the MQOutput node

This test identifies the cost of filtering on an element at the end of a message using the Java Compute Node XPath capability.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	1469.00	14.20	0.39
4k	No	717.60	14.00	0.78
16k	No	239.80	14.00	2.34
64k	No	62.20	13.60	8.75
256k	No	14.54	13.00	35.76
1k	Yes	252.00	4.60	0.73
4k	Yes	188.00	5.00	1.06
16k	Yes	132.00	9.00	2.73
64k	Yes	43.02	10.20	9.48
256k	Yes	12.02	11.40	37.94

Filter an Incoming Message Based on the First Element in the Message using the Java Compute Nodes GetByPath Capability

This test consists of MQ Input Node -> Java Compute Node -> MQ Output Node.

Within the Java Compute Node the first element of the incoming message is examined using the GetByPath capability. The result is always set to be true and thus the message is propagated to the MQOutput node.

This test identifies the cost of filtering on an element at the start of a message using the Java Compute Node GetByPath capability.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	2358.60	15.80	0.27
4k	No	1874.20	15.00	0.32
16k	No	660.00	10.00	0.61
64k	No	164.40	4.00	0.97
256k	No	38.56	3.00	3.11
1k	Yes	330.20	4.40	0.53
4k	Yes	295.00	4.20	0.57
16k	Yes	223.20	6.00	1.08
64k	Yes	98.60	5.80	2.35
256k	Yes	31.20	6.20	7.95

Filter an Incoming Message Based on the Last Element in the Message using the Java Compute Nodes GetByPath Capability

This test consists of MQ Input Node -> Java Compute Node -> MQ Output Node.

Within the Java Compute Node the last element of the incoming message is examined using the GetByPath capability. The result is always set to be true and thus the message is propagated to the MQOutput node.

This test identifies the cost of filtering on an element at the end of a message using the Java Compute Node GetByPath capability.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	1563.00	15.00	0.38
4k	No	695.60	14.00	0.81
16k	No	220.80	14.00	2.54
64k	No	58.80	13.40	9.12
256k	No	14.44	14.00	38.78
1k	Yes	242.40	4.00	0.66
4k	Yes	181.40	5.00	1.10
16k	Yes	131.40	9.00	2.74
64k	Yes	41.70	11.00	10.55
256k	Yes	11.92	12.00	40.27

Computation on an Input Message using the Java Compute Nodes XPath Capability

This test consists of MQ Input node -> Java Compute Node -> MQ Output Node.

Within the Java Compute Node Java code is used to calculate the total of all items and prices within a repeating structure which is in the input message. The totals along with a copy of the input message are written in the outgoing message.

This test identifies the cost of using Java to perform computation and message parsing using the XML parser.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	658.60	14.00	0.85
4k	No	229.40	13.20	2.30
16k	No	65.80	12.60	7.66
64k	No	16.80	12.80	30.48
256k	No	4.15	13.00	125.18
1k	Yes	222.40	6.60	1.19
4k	Yes	155.40	10.20	2.63
16k	Yes	56.00	12.20	8.71
64k	Yes	15.52	12.80	32.99
256k	Yes	3.96	13.00	131.31

Manipulation of an Input Message using the Java Compute Nodes XPath Capability

This test consists of MQ Input node -> Java Compute Node -> MQ Output Node.

Within the Java Compute Node Java code, utilising the XPath capability is used to significantly change the structure of the incoming message. The new structure is written as the output message.

This test identifies the cost of using Java code and XPath to perform message manipulation.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	411.40	14.00	1.36
4k	No	143.80	13.00	3.62
16k	No	42.72	13.00	12.17
64k	No	11.14	13.00	46.68
256k	No	2.81	13.00	184.92
1k	Yes	185.00	7.80	1.69
4k	Yes	111.80	11.60	4.15
16k	Yes	39.34	12.40	12.61
64k	Yes	10.76	12.20	45.35
256k	Yes	2.73	12.20	178.62

Manipulation of an Input Message using the Java Compute Nodes GetByPath Capability

This test consists of MQ Input node -> Java Compute Node -> MQ Output Node.

Within the Java Compute Node Java code, utilising the GetByPath capability is used to significantly change the structure of the incoming message. The new structure is written as the output message.

This test identifies the cost of using Java code and GetByPath to perform message manipulation.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	674.00	14.00	0.83
4k	No	223.00	13.00	2.33
16k	No	61.40	13.00	8.47
64k	No	15.56	13.00	33.42
256k	No	3.80	13.00	136.91
1k	Yes	227.00	7.00	1.23
4k	Yes	154.40	11.00	2.85
16k	Yes	52.80	12.20	9.24
64k	Yes	14.50	12.00	33.10
256k	Yes	3.68	12.00	130.36

Using XMLT

The tests in this section illustrate the processing costs of using an XML Transformation node to perform a computation and manipulation of an input message.

Computation on an Input Message

This test consists of MQ Input node -> XMLT Node -> MQ Output Node.

Within the XMLT Node a compiled stylesheet is used to calculate the total of all items and prices within a repeating structure which is in the input message. The totals along with a copy of the input message are written in the outgoing message.

This test identifies the cost of using an XSL stylesheet to perform computation and message parsing using the XML parser.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	260.00	10.00	1.54
4k	No	173.60	13.00	3.00
16k	No	64.80	12.00	7.41
64k	No	18.26	13.00	28.48
256k	No	5.58	14.00	100.36
1k	Yes	175.40	8.00	1.82
4k	Yes	111.80	9.00	3.22
16k	Yes	54.00	12.00	8.89
64k	Yes	16.44	12.40	30.17
256k	Yes	5.22	13.00	99.62

Manipulation of an Input Message

This test consists of MQ Input node -> XMLT Node -> MQ Output Node.

Within the XMLT Node a compiled stylesheet is used to significantly change the structure of the incoming message. The new structure is written as the output message.

This test identifies the cost of using an XSL stylesheet to perform message manipulation.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	270.80	9.80	1.45
4k	No	200.40	12.80	2.55
16k	No	79.00	12.00	6.08
64k	No	23.48	13.00	22.15
256k	No	7.46	14.00	75.07
1k	Yes	180.60	8.00	1.77
4k	Yes	127.20	9.20	2.89
16k	Yes	65.20	11.20	6.87
64k	Yes	20.90	12.40	23.73
256k	Yes	6.90	13.00	75.36

Publish Subscribe

The tests in this section illustrate the processing costs of using the publish/subscribe functions within WebSphere Message Broker with different message protocols and varying numbers of subscribers.

Topic Based Publish/Subscribe using Non Persistent MQ Messages

This test consists of MQInput node -> Publication node.

A publisher publishes a message on a single topic. The test is run repeatedly with varying numbers of subscribers (1, 10, 100 and 1000). All subscribers are registered to receive messages on the single topic.

Non persistent MQ messages 1K in size are used by the publisher.
This test identifies the cost of using the Publication node for a single publisher, varying subscribers, single topic and a single copy of the message flow when using non persistent MQ messages.

The results of running this test are given in the table below.

Publishers	Topics	Subscribers	Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1	1	1	1k	No	2430.00	11.00	0.18
1	1	10	1k	No	6325.00	15.00	0.09
1	1	100	1k	No	5979.20	14.00	0.09
1	1	1000	1k	No	4864.86	26.00	0.21

Topic Based Publish/Subscribe using MQ Real-time Messages

This test consists of a Real-time OptimizedFlow Node.

A publisher publishes a message on a single topic. The test is run with a single subscriber which is registered to receive messages on the single topic.

Both the publisher and subscriber use the WebSphere MQ Real-time transport to publish and subscribe to messages.

This test identifies the cost of using the Publication node for a single publisher, subscriber, topic and a single copy of the message flow when using the WebSphere MQ Real-time transport.

The results of running this test are given in the table below.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	16300.00	9.00	0.02

Scaling Message Throughput

The tests in this section show the effect of using two different approaches to increase message throughput for a message flow. These are the use of additional instances and assigning one copy of the message flow to each of multiple execution groups.

Using Additional Instances

This test consists of running the Large Messaging sample with a varying number of instances of the message flow in a single execution group.

The purpose of this is to see how effective the use of additional instances is in increasing message throughput and achieving higher system CPU utilisation. The benefits observed in any given situation will depend on the processing requirements of the message flow. CPU bound message flows will have different scaling characteristics from those which are I/O bound for example.

The results of running this test are given in the table below.

Instances	Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1	1k	No	467.6	14.00	1.20
2	1k	No	841.8	27.00	1.28
4	1k	No	1138.8	49.40	1.74
8	1k	No	937.6	59.20	2.53

The results in the table show that when running with 2 instances of the message flow there is an increase in message throughput. It was possible to achieve 1.80 times that achieved when with running one instance.

When running with 4 instances of the message flow message throughput continued to increase. It was possible to achieve 2.44 times that of one instance. T

When using 8 instances of the message flow, message throughput actually declined when compared with the 4 instance case. This is for what was a CPU bound workload. The results are very likely to be different for a workload which has more I/O processing in it.

From these measurements we can see that use of additional instances does provide a mechanism to increase message throughput and achieve greater CPU utilisation of the system. From an initial message rate of 467.6 messages per second it was possible to get a maximum rate of 1138.8 messages per second.

The results with other workloads are likely to vary dependent on the nature of the processing and whether it is CPU or I/O bound. Accordingly you are recommended to determine the optimum number of copies to use for each message flow individually through experimentation with a varying number of copies.

Using Mulitple Execution Groups

This test consists of running the Large Messaging sample with a single instance of the message flow in a varying number of execution groups.

The purpose of this is to see how effective the use of multiple execution groups is in increasing message throughput and achieving higher system CPU utilisation. The benefits observed in any given situation will depend on the processing requirements of the message flow. CPU bound message flows will have different scaling characteristics from those which are I/O bound for example.

The results of running this test with are given in the table below.

Execution Groups	Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1	1k	No	467.6	14.00	1.20
2	1k	No	916.4	27.60	1.20
4	1k	No	1489.8	54.40	1.46
8	1k	No	1817.8	98.00	2.16

The results in the table show that when running with 2 execution groups there is an increase in message throughput. It was possible to 1.96 times that achieved when with running one instance.

When running with 4 instances of the message flow message throughput continued to increase. It was possible to achieve 3.19 times that of one instance.

When using 8 copies of the message flow, message throughput further increased, as did CPU utilisation. It was possible to achieve 3.89 times that of one instance. At this point the system was 98% CPU busy and so fully utilised.

From these measurements we can see that use of execution groups can be an effective mechanism for increasing message throughput and allowing a machine to be fully utilised.

The results with other workloads are likely to vary dependent on the nature of the processing and whether it is CPU or I/O bound. Accordingly you are recommended to determine the optimum number of copies to use for each message flow individually through experimentation with a varying number of copies.

Overheads

The tests in this section indicate the processing costs of using Accounting and Statistics and Trace on a message flow.

Using Accounting and Statistics

This test consists of running a single copy of the Large Messaging sample with basic thread level and advanced node level accounting activated.

Using a 1K message size there was a 6% reduction in message throughput. This is a CPU overhead and reflects the additional cost of processing needed to collect the data.

Using a lower level of reporting would have resulted in a lower overhead.

Using Trace

This test consists of running a single copy of the Large Messaging sample whilst taking a user trace of type normal at the same time.

Using a 1K message size there was a 21% reduction in message throughput. This reflects the CPU and I/O overhead of writing user trace.

With debug trace the overhead will be even higher as debug trace is more extensive.

You are strongly recommended not to use WebSphere Message Broker trace in a production system. You are also strongly recommended not to have any trace nodes in the main processing paths of message flows. Even if trace is not active a penalty is still incurred to evaluate the expression specified in the Trace Node.

Resource Requirements

This section details the recommended minimum specification of a machine on which to install the development toolkit and Message Broker runtime. It also illustrates virtual memory use for message flows.

Recommended Minimum Specification

The recommended minimum specification machine to install and run the development toolkit is:

- Any Intel Pentium III (or higher) processor-based IBM PC or compatible with 700 or more MHZ processor speed. This is the minimum supported level. For improved performance use a 2 GHz or faster processor.
- Up to 6.1 GB of disk space
    - 4.5 GB disk plus 1.5 GB temporary space for WebSphere Message Broker
    - 105 MB for ODBC drivers for Cloudscape
- 512MB memory. This is the minimum requirement though and 1GB is recommended.

The recommended minimum specification machine to install and run the broker runtime is:

- Any Intel Pentium III (or higher) processor-based IBM PC or compatible with 700 or more MHZ processor speed. This is the minimum supported level. For improved performance use a 2 GHz or faster processor. For production a multi-processor machine is recommended.
- Up to 915 MB disk space
    - 315 MB disk plus 300 MB temporary space for WebSphere Message Broker
    - 300 MB for DB2 Enterprise Server compact version (assuming DB2 as the Message Broker database)
- 512 MB memory. This is the minimum requirement though and more is recommended. How much will depend on the complexity of the messages and message flows. For development 1GB+ is recommended. For production a suggested minimum would be 4GB.

These are recommended minimum specifications which are suitable to enable the processing of simple messages with simple message transformation or routing. Situations requiring more intensive processing are likely to need greater resources.

For more guidance on the support processors and configuration requirements see the WebSphere Message Broker *Managing Your Installation* manual.

Memory Use

The amount of virtual and real memory used by a message flow running within an execution group will vary, dependent on the complexity of the message flow, the style of processing within the message flow and the size of the messages being processed. This is a complex subject and a detailed discussion is beyond the scope of this document. However to assist with planning the memory used for a variety of tests is reported.

Virtual memory size is the total of all private (not shared) bytes allocated for the process, whether currently in physical memory or on disk. Real Memory is the amount of physical RAM allocated for the process. Memory utilisations are reported to the nearest 1MB.

Note that the recorded virtual and real memory size is dependent on the platform specific memory and swap space allocation algorithms. These values vary on a per platform basis.

The figures in the table below record the amount of virtual and real memory used by an execution group for the message flow when it is initially deployed and after it has processed a number of messages and the size stabilised.

In each case a single copy of the message flow was deployed to a single execution group. Each use case was deployed to a new execution group.

Use Case	Virtual memory usage with message flow deployed but no messages processed	Real memory usage with message flow deployed but no messages processed	Virtual memory peak usage after processing messages	Real memory peak usage after processing messages
Empty Execution Group	75	75	N/A	N/A
Aggregation	77	77	82	82
Coordinated Request Reply	78	78	81	81
Data Warehouse	76	76	79	79
Large Messaging	76	76	78	78
Message Routing	76	76	77	77
SWIFT Message parse	134	134	155	155
XMLT	76	76	198	198

Virtual and Real Memory Use in MB for a Variety of Use Cases.

In taking these figures the minimum heap size of the WebSphere Message Broker Java Virtual Machine (JVM) was allowed to default to the value of 128MB.

Tuning

This section details the parameters which were reviewed or changed in the course of obtaining the measurement results.

The description of each parameter is brief as a detailed discussion of the effects of any changes are beyond the scope of this document.

Message Broker

The Message Broker used in the measurements was configured in the following ways for all tests:

1. Transactional support was used where appropriate. When processing persistent messages it was used, with non persistent messages it was not. The use of transaction control means that message processing takes place within a WebSphere MQ unit of work. This involves additional CPU and I/O processing by WebSphere MQ because the unit of work is recoverable. The result is inevitably a reduction in message throughput for persistent messages. By default the transaction parameter on the MQInput node was set to automatic. This is the recommended value to use for transaction mode unless there is a specific requirement to use a particular value since persistent messages will be processed within transactional control and non persistent messages will not.

Additional tuning was performed for the **publish subscribe** tests. This was as follows:

1. The heap size of the The WebSphere Message Broker Java Virtual Machine (JVM) (in which much of the publish subscribe code is executed) was set to 512MB. For the non Publish Subscribe tests the default value of 128MB was used.

2. The thread settings of the RealtimeOptimizedNode used a default value of 10 read and write threads. These were sufficient to cater for the test cases run in this report. However if more clients are used increasing these values could be beneficial.

3. Client Pinging - The ping protocol implements a “keep alive” protocol where the broker is periodically verifying that connected clients are alive. This process allows the broker to detect disconnected clients and maintain an updated subscription list. In situations where there are a large number of clients connected to a broker, this pinging process may account for a large proportion of the messages exchanged between the broker and clients and can impact the broker's message throughput. In such circumstances, the ping interval can be turned off or alternatively increased to reduce the amount of traffic generated by pinging. For the tests in the report the value was set to 0.

4. Client Queue Size - The broker employs a set of internal queues which are used to regulate the delivery of messages to subscribers. Note: these are not the queues used by the MQ Transport. The size of the queue specifies the number of bytes of data that the broker will store for one client. If this maximum is exceeded, the broker will take action which is determined by the value of the parameter "Client disconnection due to queue overflow". The default queue size is 100,000 bytes. Setting the value to zero allows the broker to grow the queue size as required. In this case, the queue size will only be limited by the available system memory. In the tests detailed in this report a value of 0 was used.

5. Client disconnection due to queue overflow - When the depth of the client queue exceeds the Client Queue Size value, the broker can choose between two courses of action. The default action is to disconnect the client; in this case, all the queued messages are lost. This is specified through a value of true for the parameter. The alternative course of action, specified with a value of false, is to keep the client connection alive but remove any excess messages from the client's queue. In the tests detailed in this report a value of false was used.

6. Maximum message size - If the broker receives a message that is bigger than the maximum message size value, it will disconnect the client that sent the message. This feature is useful for protecting the broker from applications sending excessively large messages. The default maximum message size is 100,000 bytes and this was adequate for the tests run in the report so the value was left unchanged.

7. Maximum number of client connections - The broker has the ability to limit the number of client connections that it will handle. This is useful in situations where the number of client applications that will connect to the broker is unknown. In such cases limiting the number of connections will allow the broker to maintain a particular level of service (this level will depend upon the particular environment in which the broker is being used). The default setting is unlimited. This was also the value used for the tests run in the report.

8. Interval statistics reporting was enabled and set to an interval of 10000 (10 seconds) so that the value of ClientBytesQueued could be monitored.

Additional tuning was performed for the **JMS nodes**. These was as follows:

The performance of the JMS Nodes is dependant on the performance of the JMS Provider and the JMS Providers client code. You should check the JMS Providers documentation for tuning details, in particular look for details of how to tune the client code which is supplied.

- In the tests run for this report the max buffer size for the TopicConnectionFactory used by the JMS Input Node was increased to a value of 3000. This is important for a subscriber using WebSphere MQ Real-time as it offers protection from message rate spikes. For information on how to set the subscriber max buffer size see the WebSphere MQ "Using Java" manual.

There were no error processing or error conditions in any of the measurements. All messages were successfully passed from one node to another through the out or true terminal. No messages were passed through the failure terminal of a node.

WebSphere MQ

The following changes were made to all queue managers used in the tests:

1. The value of DefaultQBufferSize was increased to a value of 1000000 for each queue used in the tests.

2. Given the use of persistent messages in the tests the following MQ log parameters were modified:
   - LogBufferPages was set to 0 allowing the value to default to the value of 128 (for WebSphere MQ V6)
   - LogFileSize was set to 1024
   - LogType was set to circular
   - LogPrimaryFiles was set to 3
   - LogSecondaryFiles was set to 2

3. Circular logging was set for all WebSphere MQ queue managers used in the tests.

4. The Message Broker queue manager MQ listener and channels were run as trusted applications. In the queue manager qm.ini the value MQIBindType was set to FASTPATH in the channel stanza. The environment variable MQ_CONNECT_TYPE=FASTPATH was present in the environment in which the broker queue manager was started.

TCP/IP

No specific tuning was performed for TCP/IP. All machines used the operating system default values.

Database

The DB2 instance used with the message broker was a default configuration and the only tuning performed on the instance was placement of the database data and log files on different disks.

Miscellaneous

Although not implemented in all cases the following additional tuning changes are recommended

- Locate the log of any WebSphere MQ queue manager through which persistent messages pass on a dedicated disk.
- Locate the WebSphere MQ queue manager log on a very fast disk such as one with a non-volatile fast write cache. Such disks are consistently capable of I/O times of 1ms compared with a time of 6 ms for a 10,000 RPM SCSI disk. When using a disk with a fast write cache it is essential that it has a non-volatile capability as the log data is critical to the integrity of your queue manager.
- Note that there is no need to locate the WebSphere queue manager queue file on a fast disk. It is advisable to locate it on a dedicated disk in order to improve the efficiency of queue manager checkpoint processing.
- Locate the log of the Message Broker database on a dedicated disk.
- Locate the log of the Message Broker database on a very fast disk such as one with a non-volatile fast write cache.
- When performing BLOB inserts to a database locate the data portion of the database on a very fast disk such as one with a non-volatile fast write cache. BLOB I/O is not buffered by a database such as DB2 and is written to disk immediately.
- When using the aggregation node follow the message flow coding advice provided in **Supportpac IP05, WebSphere MQ Integrator V2.1 - Optimizing Use of Aggregation Nodes** which is available at http://www.ibm.com/software/integration/support/supportpacs/individual/supportpacs/ip05.pdf.

  NOTE: When using WebSphere Message Broker V6 there is no need to follow the recommendations in the document about Message Broker database configuration. This is because the aggregation mechanism is now based on the use of WebSphere MQ queues, rather than a database table as with previous versions of the Message Broker.

Additional Tuning Information

In order to obtain the maximum message rate for your implementation it is important that you understand the current best practices for WebSphere Message Broker. These practices cover the architecture of message flow processing, the coding of message flows as well as the configuration and tuning of the message broker and associated components.

Such information can be found in the Business Integration Zone of WebSphere Developer Domain. A suggested starting place is the article http://www.ibm.com/developerworks/websphere/library/techarticles/0403_dunn/0403_dunn.html which highlights the information available and where it may be found.

Conclusion

This report has detailed the key performance characteristics of the WebSphere Message Broker V6 runtime. The primary focus in the report has been on identifying the CPU costs of different functions. Additional information has been supplied on virtual memory requirements for the use cases.

Part I showed the level of message throughput that can be expected for a variety of common use cases. You have the ability to run these same tests in your own environment as the messages flows are shipped as product samples. Using machines with even faster processors it will be possible to achieve high message rates.

From the data supplied in Part I of the report it is possible to see that there have been some significant reductions in CPU costs and as a result increases in message throughput in WebSphere Message Broker V6 when compared with WebSphere Business Integration Message Broker V5. Most notably improvements in message throughput of

- 3.1 times for the Aggregation use case
- 4.5 times for the Coordinated Request/Reply use case
- 1.9 times for the Message Routing use case
- 3.2 times for the SWIFT Message Parse use case

No application changes are needed to obtain the majority of these performance improvements. They come as standard with version 6 and are available immediately on installation and migration of the message flows.

Some additional gains in performance are available by using new function such as shared variables.

WebSphere Message Broker V6 has significantly reduced the CPU of using all key functions. This allows you to noticeably reduce the CPU cost of running an existing workload or achieve a higher level of message throughput for the same amount of CPU.

Appendix A - Measurement Environment

All throughput measurements where taken on a single server machine. The client type and machine on which they ran varied with the test. The details are given below.

Server Machine

The hardware consisted of

- An IBM xSeries 350 with 4 * 2.00 GHz Intel Xeon processors with Hyper-threading
- Three 69 GB SCSI hard drives
- Two 150 GB SAN controlled drives
- 4 GB RAM
- 1 Gb Ethernet card

The software consisted of:

- Red Hat Enterprise Linux AS release 3 (Taroon Update 3)
- WebSphere MQ V6
- WebSphere Message Broker V6
- DB2 for Solaris V8.1 with Fixpack 2

Client Machines

A number of different client machines were used dependent on the tests being run. The different configurations are described below.

Point to Point Testing
The hardware consisted of:

- An IBM xSeries 350 with 4 * 3.00 GHz Intel Pentium 4 Xeon processors
- Six 68 GB SCSI hard drives formatted with NTFS
- 3.5 GB RAM
- 1 Gb Ethernet card

The software consisted of:

- Microsoft Windows 2000 with Service Pack 4
- WebSphere MQ V5.3 CSD5

Publish Subscribe Testing
The hardware consisted of multiple machines of this specification:

- An IBM xSeries processor with 4 * 1800 MHz Pentium 4 Xeon processors
- 2 GB SCSI hard drives
- 4 GB RAM
- 1 Gb Ethernet card

The software on all client machines consisted of:

- Linux Red Hat Advanced Server Version 2.1
- IBM Java 1.4.2
- WebSphere MQ V5.3.6.

JMS Node Testing
Different machines were used for the JMS provider and JMS client components. They are described below:

JMS Provider
The JMS provider machine hardware consisted of:

- An IBM xSeries processor with 8 * 700 MHz Pentium 4 Xeon processors
- 2 GB SCSI hard drives
- 4 GB RAM
- 1 Gb Ethernet card

The software consisted of:

- Linux Red Hat Advanced Server Version 2.1
- IBM Java 1.4.2
- WebSphere Message Broker V6.
- WebSphere MQ V6.

Client Machine
The client machine hardware consisted of:

- An IBM xSeries processor with 4 * 1800 MHz Pentium 4 Xeon processors
- 2X GB SCSI hard drives
- 4 GB RAM
- 1 Gb Ethernet card

The software consisted of:

- Linux Red Hat Advanced Server Version 2.1
- IBM Java 1.4.2
- WebSphere MQ V5.3

Network Configuration

The client and server machines were connected using a full duplex 1 Gigabit Ethernet LAN with a single hub.

Appendix B - Evaluation Method

This section outlines the software components that were used to produce the measurement results which are contained in this report.

Three different configurations were used in the generation and consumption of input and output messages. This is because different test cases required different types of input and output messages. The methods used were:

1. Point to Point Message Processing

2. Publish Subscribe Message Processing

3. JMS Node Message Processing.

These are described in the remainder of this section.

A series of parameter configuration changes were made to improve message throughput. These are discussed in the section Tuning.

Point to Point testing

This section describes how messages were generated and consumed for the point to point messaging tests, such as the Database Read tests or Filter an Incoming Message based on the First Element in the Message. The configuration of the software components is also discussed.

Message Generation and Consumption

A multi threaded WebSphere MQ Client program written in C was used to generate input messages for the test case being run and to consume the WebSphere MQ output messages.

The client program used the Message Queue Interface (MQI). Both persistent and non persistent messages were generated from this program.

Sufficient threads were run in the multi threaded client to ensure that there were always messages on the input queue waiting to be processed. This is important when measuring message throughput.

Any thread within the client program was able to retrieve any message which had been processed by a message flow. No use was made of the WebSphere MQ correlation identifiers to limit consumption of a message to the thread which created it.

Machine Configuration

The WebSphere MQ client program used to generate and consume messages for the message flows was run on a dedicated machine, the Client Machine. The Message Broker, its dedicated WebSphere MQ queue manager and broker database were all located on a dedicated machine, the Server Machine.

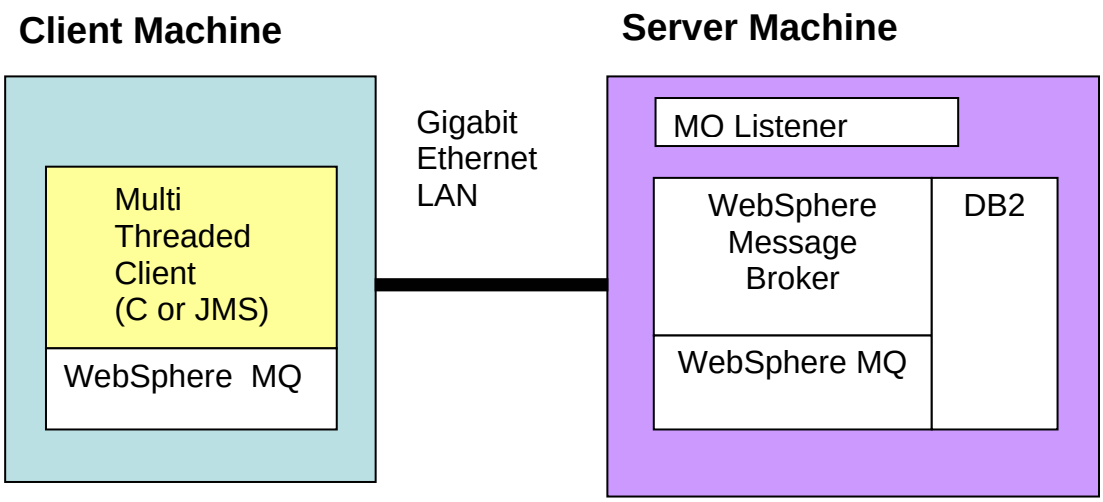
There was a single client machine.

Messages were transmitted from the client machine to the server machine over WebSphere MQ SVRCONN channels. The messages were received on the server machine through use of a WebSphere MQ queue manager listener process. This was run as a trusted MQ application in order to improve message throughput.

The database used for the database related test cases used the same database instance as the Message Broker.

Messages were transmitted from the client machines to the server machine using the WebSphere MQ transport.

The diagram below illustrates the major components in the measurement environment and their location.



Both the client and server machine were configured with sufficient memory to ensure that no paging took place during the tests.

Publish Subscribe testing

This section describes how messages were generated and consumed for tests which used the publish subscribe message processing model.

Message Generation and Consumption

A multi threaded JMS client application was used to publish JMS messages and consume messages received by JMS Subscribers. The JMS client application used WebSphere Message Broker as the JMS Provider. The same client application was able to generate JMS messages using the WebSphere MQ and WebSphere MQ Real-time transports.

JMS bytes messages were used in the testing. The message content was not of interest in the tests only the topic under which it was published.

Messages were transmitted from the client machines to the server machine using either the WebSphere MQ or WebSphere Real-time transport depending on the test case.

When using the WebSphere MQ transport the publish rate was set to a high value, this publish rate was then throttled by the MQ acknowledgement protocol to a rate which was sustainable by the broker. The publisher acknowledgement interval was set to ensure messages were always available on the brokers input queue. Details of how to set the broker acknowledgement interval can be seen in the WebSphere MQ "Using Java" manual. Each subscriber was allocated its own temporary dynamic queue to store its messages on the broker.

The WebSphere MQ Real-time transport does not have a self throttling protocol like that of MQ. As a result the publish rate was manually adjusted until the optimum message throughout for the broker is found. The optimum level of message throughput was determined by monitoring the ClientBytesQueued value in broker statistics.

The value for ClientBytesQueued shows the number of bytes waiting to be delivered to subscriber clients. When the broker becomes overloaded it is unable to service this buffer fast enough and so the number of bytes that are queued increases. For a test to be successful the buffer size must not continually increase during the test run. Constant growth of this buffer indicates too high a publish rate. The point at which the buffer starts to fill is dependent on a combination of factors such as network bandwidth, system memory and client performance.

The subscriber, when using WebSphere MQ Real-time client, contains a message buffer to protect itself from message rate spikes. For these tests this was increased from the default to hold 3000 messages. For information on how to set the subscriber max buffer size see in the WebSphere MQ "Using Java" manual.

In all of the tests it was verified that all publications were delivered to subscribers without any loss of messages. As part of ensuring this all subscribers were started first before the publishing of messages commenced.

Queue depths and buffer sizes were monitored to ensure that the system was running in a stable manner and that there was no backlog of messages to be processed.

Publishers

The JMS Publisher sent non-persistent publications only. A non transacted JMS Session was used. The publisher produced publications at a constant rate, i.e. a fixed number of publications per second.

Subscribers

The JMS Subscribers were non durable and non transacted. Each subscriber had a separate TopicConnection and a TopicSession therefore each subscriber was associated with one physical TCP/IP connection (socket pair). A single topic was used for all tests and so all

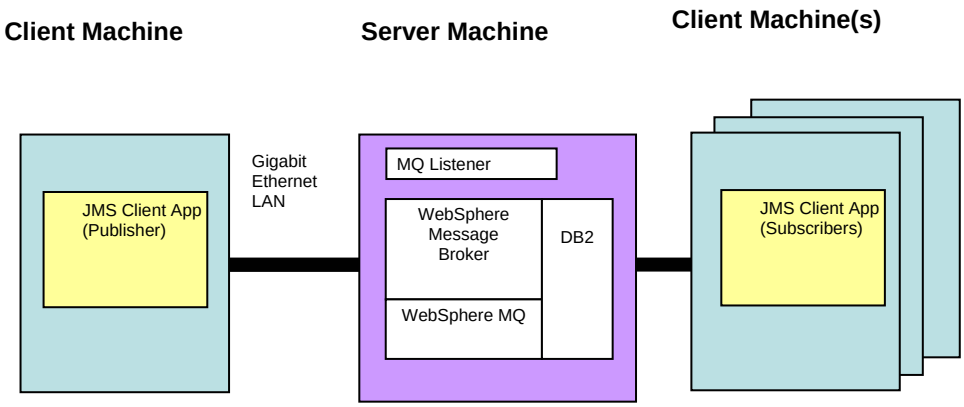
subscribers were subscribed to the same topic. This meant that for every message published a copy was received by each subscriber.

Machine Configuration

The programs used to publish and subscribe messages for the tests were run on dedicated Client Machines which were separate from the machine on which the message broker was run.

The number of client machines used varied depending on the test case, for example for a 1 to 1 test (1 publisher and 1 subscriber) 2 client machines were used but for a 1 publisher to 1000 subscriber test the 1000 subscribers were spread over 10 Client Machines. The publisher used a different dedicated machine. The distribution of subscribers over multiple client machines was essential to avoid the client Machines becoming the bottleneck in the test case.

The Message Broker, its dedicated WebSphere MQ queue manager and broker database were all located on a dedicated machine, the Server Machine. The figure below shows the configuration of software components and machines.



Both the client and server machine were configured with sufficient memory to ensure that no paging took place during the tests.

For those tests using WebSphere MQ as the transport a message flow consisting of an MQInput node wired to a Publication node was used.

For tests using WebSphere MQ Real-time as the transport a message flow consisting of a RealtimeOptimizedFlow node was used.

JMS Node Message Processing

JMS messages used in the JMS node tests used WebSphere MQ Real-time as the transport. WebSphere MQ messages were used in some of the tests where there was a conversion between protocols for example.

Message Generation and Consumption

A multi threaded JMS client application was used to generate and consume JMS messages. The JMS client application used WebSphere Message Broker as the JMS Provider. The same client application was able to generate JMS messages using the WebSphere MQ and WebSphere MQ Real-time transports.

JMS messages used both the WebSphere MQ and WebSphere MQ Real-time transports dependent on the tests. Which was used when is documented for each test.

Both JMS bytes and JMS Text messages were used in the testing. Again this is documented in the individual test details.

The client application used to generate and consume messages was a tool known as the Performance Harness for JMS. See the section Additional Information for more details on how to obtain a copy.

Machine Configuration

For the JMS node tests two different configurations were used. The first was for the JMS to JMS messaging test and a second for the JMS to MQ protocol conversion.

The JMS Input and Output Nodes were configured to use Publish Subscribe mode to connect to the JMS Provider. The JMSNodes can also be configured in Point to Point mode.

The Performance Harness for JMS was used to perform the role of publisher and subscriber for the tests. The Performance Harness was run on a dedicated Client Machine which was separate from the broker machine.

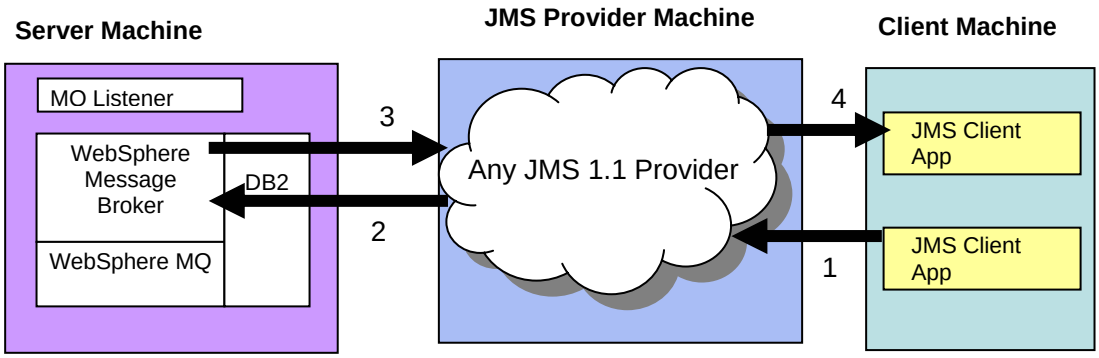
As part of the test a JMS provider was required. This is in addition to the broker under test. For the tests in the report the JMS provider was another copy of WebSphere Message Broker V6. However any JMS provider which conforms to the JMS 1.1 specification would be able to fulfil this role.

The Message Broker, its dedicated WebSphere MQ queue manager and broker database were all located on another dedicated machine, the Server Machine.

The distribution of components over multiple machines was essential to ensure that the testing of the JMS nodes running within the Message Broker was not throttled by a bottleneck which results from another component running on the same machine. It also allowed the cost of running the JMS nodes to be clearly identified.

The sections below show the configuration of software components and machines for the different test cases.

JMS to JMS Tests



.

The numbered arrows indicate the sequence of processing:

1) Messages were published by the client application to the JMS Provider on the JMS provider machine. The publication was destined for a topic called “INPUTTOPIC”.

2) The JMSInput Node running in the message flow in the message broker was configured to subscribe to the JMS Provider on topic “INPUTTOPIC”. It obtained the input message from the JMS provider running on the JMS provider machine.

3) The message received by the JMSInput Node (from the JMS provider running on the JMS provider machine) is processed in the message flow. The output message is written by the JMSOutput node which is configured as a topic publisher on topic “OUTPUTTOPIC”.

4) The second client application running on the client machine operates as a subscriber. It is configured to connect to the JMS Provider (on the JMS provider machines) as a subscriber with a subscription to the topic “OUTPUTTOPIC”. Hence it will receive the publication sent from the JMSOutput Node in the broker.

JMS Provider
In the configuration shown the JMS Provider could have been any JMS Provider that is compliant with the JMS 1.1 Specification. It would have been possible to have used a different JMS Provider for the JMSInput and JMSOutput nodes. For the tests run in this report WebSphere Message Broker was used as the JMS Provider. The Message Broker was configured with a RealtimeOptimisedFlow Node and was tuned as described in the Publish Subscribe Testing section of this report. We used Broker statistics to monitor the performance characteristics of this broker to ensure it was not the bottleneck in these tests.

Publisher
A single JMS publisher was used. It sent only non-persistent publications (step 1 in the diagram). A non transacted JMS Session was used. The publisher produced publications at a constant rate, i.e. a fixed number of publications per second. The rate was varied according to the individual test cases. There was no simulation of publishers that publish messages at variable rates. The JMS Publisher was configured to use the WebSphere MQ Real-time Transport.

Subscriber
A single non durable and non transacted JMS Subscriber was used as the receiving client (step 4 in the diagram). The JMS Subscriber was configured to use the WebSphere MQ Real-time Transport.

The message rate numbers in this report are the rates measured at step 4 i.e. number of messages a second that the JMS Subscriber application received.

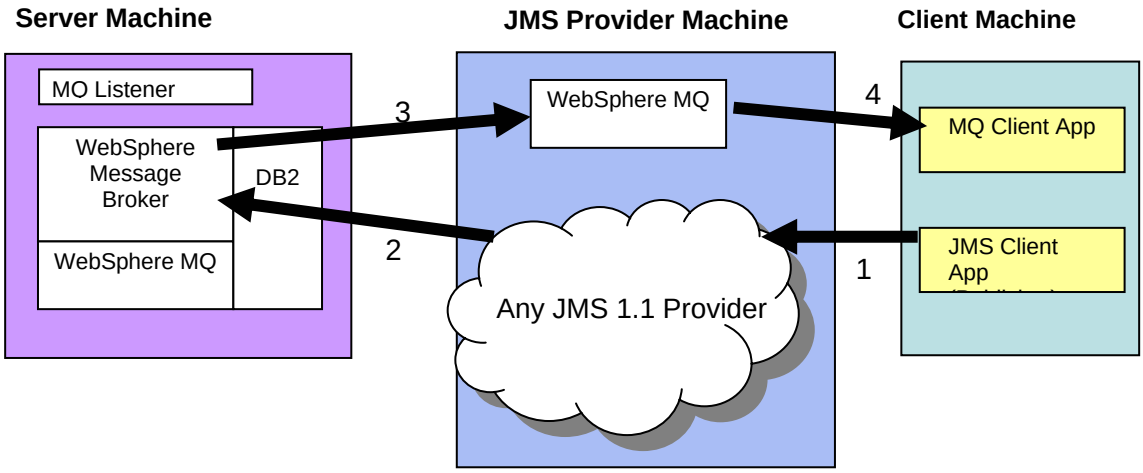
In all of the tests it was verified that all publications were delivered to subscribers without any loss of messages. As part of ensuring this all subscribers were started first before the publishing of messages commenced.

Queue depths and buffer sizes were monitored to ensure that the system was running in a stable manner and that there was no backlog of messages to be processed.

The message throughput reported for these tests is the maximum message rate that could be achieved whilst meeting the constraints mentioned above. Typically in all of these tests message throughput of the test was limited by the CPU capacity on the Server machine.

JMS to MQ Tests

The tests which involved transformation from JMS to MQ used a slightly different configuration from that used for the JMS to JMS tests. This is illustrated below.



The numbered arrows indicate the sequence of processing:

1) Messages were published by the client application to the JMS Provider on the JMS provider machine. The publication was destined for a topic called “INPUTTOPIC”.

2) The JMSInput Node running in the message flow in the message broker was configured to subscribe to the JMS Provider on topic “INPUTTOPIC”. It obtained the input message from the JMS provider running on the JMS provider machine.

3) The message received by the JMSInput Node was processed by the message flow and an output message was created and sent to the MQOutput node which was configured to write to a remote WebSphere MQ Queue called “OUTPUTQUEUE”. This queue was located on a WebSphere MQ Queue Manager on the JMS Provider machine. The remote WebSphere MQ Queue Manager was actually the same queue manager that the Message Broker acting as the JMS Provider on the JMS provider machine was using. Note that appropriate transmission queue and sender and receiver channels had to be defined between the two queue managers.

4) The second client application read the WebSphere MQ message from the remote queue “OUTPUTQUEUE”. The Performance Harness for JMS was used to implement this client.

Reported Message Rates

For tests which did not involve publish subscribe the message rates reported are the number of invocations of the message flow per second.

For tests involving several message flows such as the message aggregation test the rate reported is the number of complete operations or aggregations per second. Fan-out and Fan-in processing is counted as one rather than separately.

For tests using publish subscribe the message rate reported is the total message rate. That is the number processed by all publishers and all subscribers. The total number of messages reported is calculated using the formula (number of subscribers +1) * publication rate.

For a configuration consisting of one publisher and 10 subscribers where the publication rate was 10 messages/second the total message rate is (10 +1) * 10 = 110 messages second.

For tests using the JMS nodes the message rate is the number of message flow invocations per second.

The message rates quoted are an average taken over the measurement period. This starts once the system initialisation period has completed.

Appendix C - Test Messages

This section describes the input and output messages used for the tests detailed in this report.

The messages which are in this section have been formatted for this report and as such contain white space between tags. When used in measurements all such white space is removed.

Input Message

An input message of the type shown below was used for the non publish/subscribe tests in the report.

The publish/subscribe tests used a 1K JMS Bytes message.

The message shown below is in Generic XML format but it was also represented in a variety of other formats such as MRM XML, CWF and TDS where this was required in the test.

The different message sizes used in testing are achieved by repeating the content of the SaleList tag to give the required size. Larger messages thus result in more tags.
A Perl script ensures that the names and values in the tags are different as the SaleList structure is repeated. This is to stop a limited number of strings being used in very large messages which could lead to over optimistic results.

```
<Parent>
        <First>1</First>
         <SaleList>
                <Invoice>
                        <Initial>K</Initial>
                        <Initial>A</Initial>
                        <Surname>Braithwaite</Surname>
                        <Item>
                                <Code>00</Code>
                                <Code>01</Code>
                                <Code>02</Code>
                                <Description>Twister</Description>
                                <Category>Games</Category>
                                <Price>00.30</Price>
                                <Quantity>01</Quantity>
                        </Item>
                        <Item>
                                <Code>02</Code>
                               <Code>03</Code>
                               <Code>01</Code>
                                <Description>The Times Newspaper</Description>
                                <Category>Books and Media</Category>
                                <Price>00.20</Price>
                                <Quantity>01</Quantity>
                        </Item>
                        <Balance>00.50</Balance>
                        <Currency>Sterling</Currency>
                </Invoice>
                <Invoice>
                        <Initial>T</Initial>
                        <Initial>J</Initial>
                        <Surname>Dunnwin</Surname>
                        <Item>
```

```
                    <Code>04</Code>
                    <Code>05</Code>
                    <Code>01</Code>
                    <Description>The Origin of Species</Description>
                    <Category>Books and Media</Category>
                    <Price>22.34</Price>
                    <Quantity>02</Quantity>
                </Item>
                <Item>
                    <Code>06</Code>
                    <Code>07</Code>
                    <Code>01</Code>
                    <Description>Microscope</Description>
                    <Category>Miscellaneous</Category>
                    <Price>36.20</Price>
                    <Quantity>01</Quantity>
                </Item>
                <Balance>81.84</Balance>
                <Currency>Euros</Currency>
            </Invoice>
        </SaleList>
        <Last>Test</Last>
</Parent>
```

Output Message

Two message types exist for the output messages dependent on the test case. These are the Compute and Transform messages.

Compute Message

For compute test cases the balance field for each invoice is validated and the currency is converted into sterling. So there is minor modification of the input message.

The message layout is shown below

```
<Parent>
        <First>1</First>
        <SaleList>
            <Invoice>
                <Initial>K</Initial>
                <Initial>A</Initial>
                <Surname>Braithwaite</Surname>
                <Item>
                    <Code>00</Code>
                    <Code>01</Code>
                    <Code>02</Code>
                    <Description>Twister</Description>
                    <Category>Games</Category>
                    <Price>00.30</Price>
                    <Quantity>01</Quantity>
                </Item>
                <Item>
                    <Code>02</Code>
                    <Code>03</Code>
                    <Code>01</Code>
                    <Description>The Times Newspaper</Description>
                    <Category>Books and Media</Category>
                    <Price>00.20</Price>
                    <Quantity>01</Quantity>
                </Item>
                <Balance>00.50</Balance>
```

```
                <Currency>Sterling</Currency>
            </Invoice>
            <Invoice>
                <Initial>T</Initial>
                <Initial>J</Initial>
                <Surname>Dunnwin</Surname>
                <Item>
                    <Code>04</Code>
                    <Code>05</Code>
                    <Code>01</Code>
                    <Description>The Origin of Species</Description>
                    <Category>Books and Media</Category>
                    <Price>22.34</Price>
                    <Quantity>02</Quantity>
                </Item>
                <Item>
                    <Code>06</Code>
                    <Code>07</Code>
                    <Code>01</Code>
                    <Description>Microscope</Description>
                    <Category>Miscellaneous</Category>
                    <Price>36.20</Price>
                    <Quantity>01</Quantity>
                </Item>
                <Balance>80.88</Balance>
                <Currency>Euros</Currency>
                </Invoice>
            <InvoicesTotal Currency="Sterling">57.116</InvoicesTotal>
        </SaleList>
        <Last>Test</Last>
</Parent>
```

Transform Message

For the transformation test the input message is modified and takes a different layout. For each invoice a statement is created for each customer within a SaleList.

The message layout is shown below.

```
<Parent>
    <SaleList>
        <Statement Type="Monthly" Style="Full">
            <Customer>
                <Initials>KA</Initials>
                <Name>Braithwaite</Name>
                <Balance>00.50</Balance>
            </Customer>
            <Purchases>
                <Article>
                    <Desc>Twister</Desc>
                    <Cost>4.8E-1</Cost>
                    <Qty>01</Qty>
                </Article>
                <Article>
                    <Desc>The Times Newspaper</Desc>
                    <Cost>3.2E-1</Cost>
                    <Qty>01</Qty>
                </Article>
            </Purchases>
            <Amount>8E-1</Amount>
        </Statement>
```

```
        <Statement Type="Monthly" Style="Full">
                <Customer>
                        <Initials>TJ</Initials>
                        <Name>Dunnwin</Name>
                        <Balance>81.84</Balance>
                </Customer>
                <Purchases>
                        <Article>
                                <Desc>The Origin of Species</Desc>
                                <Cost>3.5744E+1</Cost>
                                <Qty>02</Qty>
                        </Article>
                        <Article>
                                <Desc>Microscope</Desc>
                                <Cost>5.792E+1</Cost>
                                <Qty>01</Qty>
                        </Article>
                </Purchases>
                <Amount>1.29408E+2</Amount>
        </Statement>
    </SaleList>
</Parent>
```

Appendix D - Use Case Descriptions

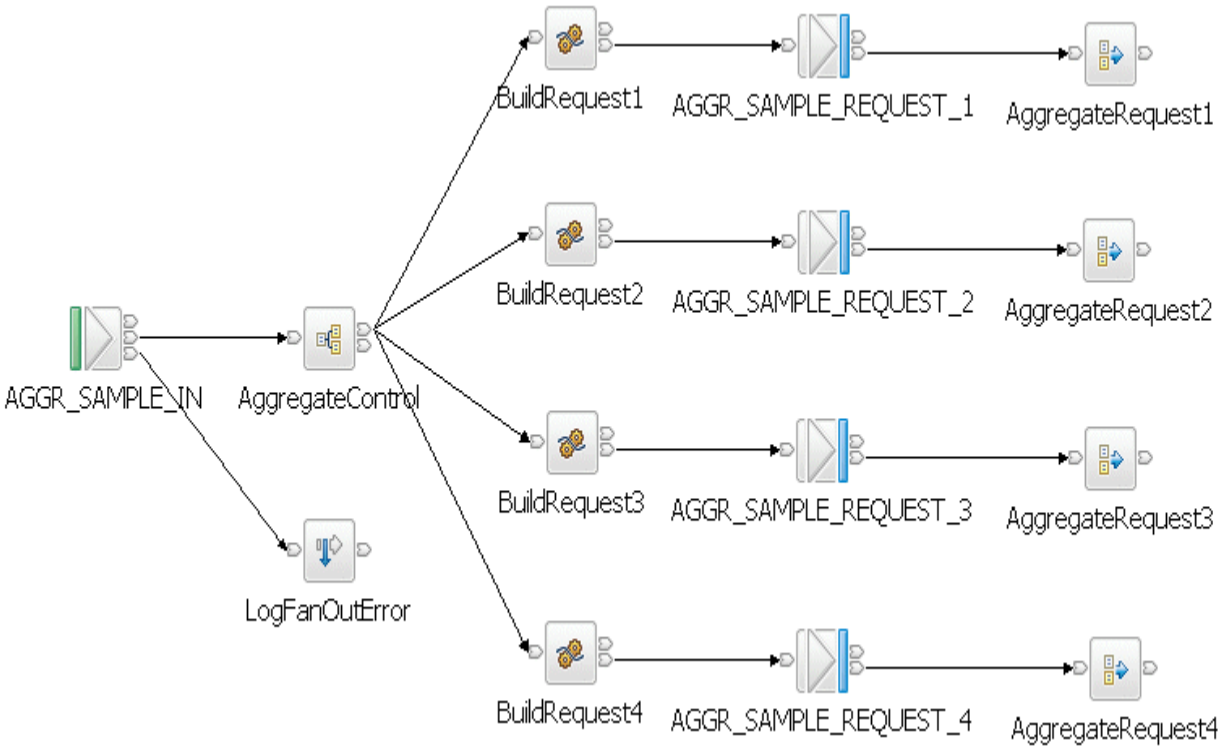
This section contains a description of the processing in each of the use cases which are used to characterise the performance of WebSphere Message Broker V6.

Aggregation

The Aggregation use case demonstrates a simple four-way aggregation operation, using the Aggregate Control, Request, and Reply nodes. It contains three message flows to implement a four-way aggregation: FanOut, RequestReplyApp, and FanIn. This is the type of processing that might be used to invoke four different applications to process a travel booking, one to organise each of the flight, hotel, car and money.

FanOut Message Flow

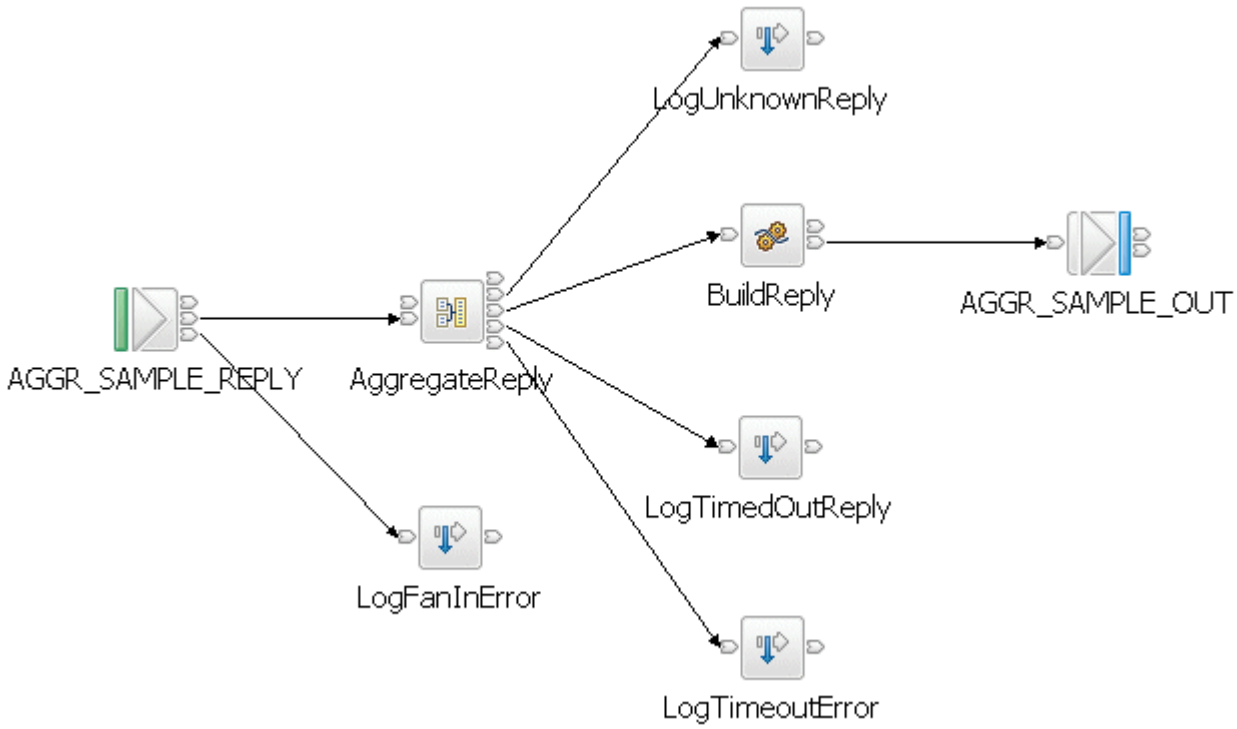
This is the flow that takes the incoming request message, generates four different request messages, sends them out on request/reply, and starts the tracking of the aggregation operation:



RequestReplyApp Message Flow

This message flow simulates the back-end service applications that would normally process the request messages from the aggregation operation. In a real system, these could be other message flows or existing applications. This message flow reads from the same queue that the MQOutput nodes in the FanOut flow write to, and it outputs to the queue that the input node which the FanIn flow reads from - it provides a messaging bridge between the two flows. The messages are put to their reply-to queue (as set by the MQOutput nodes in the FanOut flow).

FanIn Message Flow
This flow receives all the replies from the RequestReplyApp flow, and aggregates them into a single output message. The output message from the Aggregate Reply node cannot be output directly by an MQOutput node without some processing so a Compute node is added to process the data into a format where it can be written out to a queue.



Further information about the Aggregation sample can be found in the Message Brokers section of the Technology samples category which is in the samples gallery of the WebSphere Message Broker development toolkit.

Coordinated Request/Reply

The coordinated request reply sample is based on the scenario of a contemporary and established application communicating through the use of WebSphere MQ messages in a request/reply processing pattern. The contemporary application uses self-defining XML messages and issues a request message. The established application uses Custom Wire Format (CWF) messages. It receives a request message, processes it and delivers a reply message. For the applications to successfully communicate, the message formats must be transformed for both the request and reply messages.

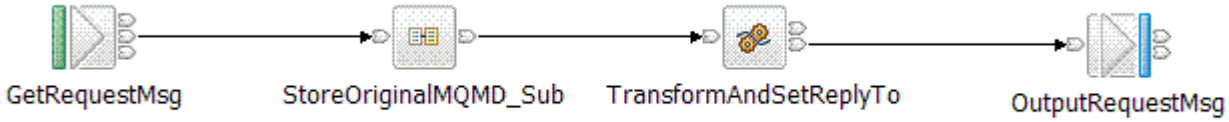
The processing in the sample consists of three message flows and one message set. The message flows are:

Request Message Flow

The request message flow performs the following processing:

- Reads a WebSphere MQ message containing an XML payload.
- Converts the message into the equivalent CWF format.
- Creates a WebSphere MQ message containing the transformed message.
- Saves the original ReplyToQ and ReplyToQMgr details in a separate WebSphere MQ message for subsequent retrieval by the Reply message flow.
- Sets the ReplyToQ and ReplyToQMgr details to be the input of the Reply message flow.
- Sends the message on to the Backend Reply message flow.

The Request message flow consists of the following nodes:

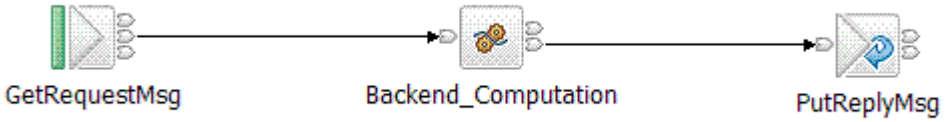


Backend Reply Message Flow

The backend reply message flows performs the following processing:

- Reads a WebSphere MQ message.
- Adds the time the message was modified to the payload of the message.
- Writes a WebSphere MQ message.

The Backend Reply message flow consists of the following nodes:



Reply Message Flow

The reply message flow performs the following processing:

1. Reads a WebSphere MQ message containing a message in CWF format.
2. Converts the message into the equivalent XML format.
3. Obtains the ReplyToQ and ReplyToQ Mgr of the original request message by reading the WebSphere MQ message which was used to store this information in the Request message flow. This is done by using the MQGET node.
4. Creates a WebSphere MQ message containing the transformed message and the retrieved ReplyToQ and ReplyToQMgr values.

The Reply message flow consists of the following nodes:

Further information about the Coordinated Request Reply sample can be found in the Message Brokers section of the Application samples category which is in the samples gallery of the WebSphere Message Broker development toolkit.

Data Warehouse

The Data Warehouse sample demonstrates a scenario in which a message flow is used to perform the archiving of data, such as sales data, into a database. The data is stored for later analysis by another message flow or application.

Because the sales data is analyzed at a later date, the storage of the messages has been organized in a way that makes it easy to select records for specified times. The date and time at which the WebSphere MQ message containing the sales record was written are stored as separate column values when the message is inserted into the database. The database table contains four columns:

- The message data - the payload of the WebSphere MQ message stored as a BLOB.
- The date on which the WebSphere MQ message was created.
- The time when the WebSphere MQ message was created.
- A time stamp created by the database to record the time when the record was inserted.

By storing the data in this way it is possible to retrieve records between specific periods of time, say between the hours of 9:00 a.m. to 12:00 p.m. or 12:01 p.m. and 5:00 p.m. which would allow a comparison of morning and afternoon sales to be made.

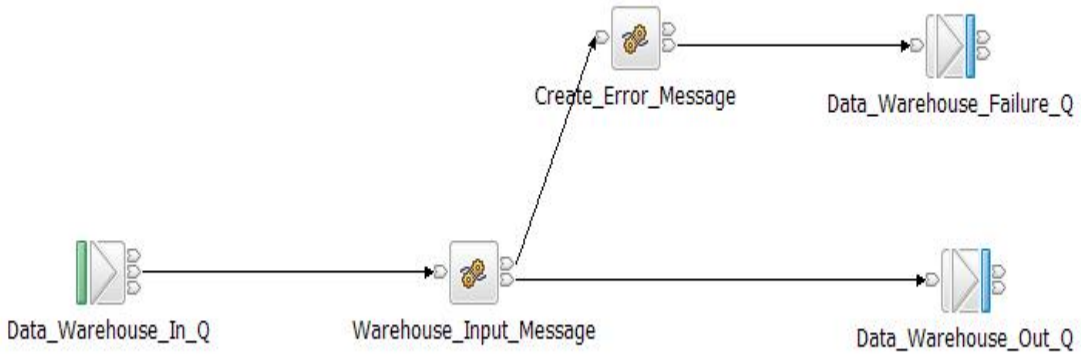
The data archiving is performed by the WarehouseData message flow. This is described below.

WarehouseData Message Flow

The WarehouseData message flow performs the following processing.

1. Reads a WebSphere MQ message containing an XML payload. The payload contains the data to be archived.
2. Converts a portion of the message tree to a BLOB ready for insertion into the database.
3. Inserts the message BLOB along with the date and time at which the WebSphere MQ message was written into a database.
4. Sends a WebSphere MQ confirmation message to signal successful insertion of the message into the database.

The WarehouseData message flow consists of the following nodes:



Further information about the Data Warehouse sample can be found in the Message Brokers section of the Application samples category which is in the samples gallery of the WebSphere Message Broker development toolkit.

Large Messaging

The Large Messaging sample is a sample based on the scenario of end-of-day processing of sales data. Messages recording the details of sales through the day are batched together in the store for transmission to the IT center. On receipt at the IT center the batched messages are split back out into their constituent parts for subsequent processing.

This splitting is achieved using a WebSphere Message Broker message flow. Each of the individual messages representing a sale has the same structure.

The input and output messages in this sample are implemented as self-defining XML messages for simplicity. Other message formats could easily be used.

Each input message consists of three parts:

- A header containing a count of the number of repetitions of the repeating SaleList structure that follows.
- The body that contains the repetitions of the repeating SaleList structure.
- The trailer that contains the time the message was processed.

The aim of the processing in this sample is to write each of the instances of the SaleList structure as a separate WebSphere MQ message while minimizing overall memory requirements.

The message flow implements a memory saving technique through the use of a mutable message tree.

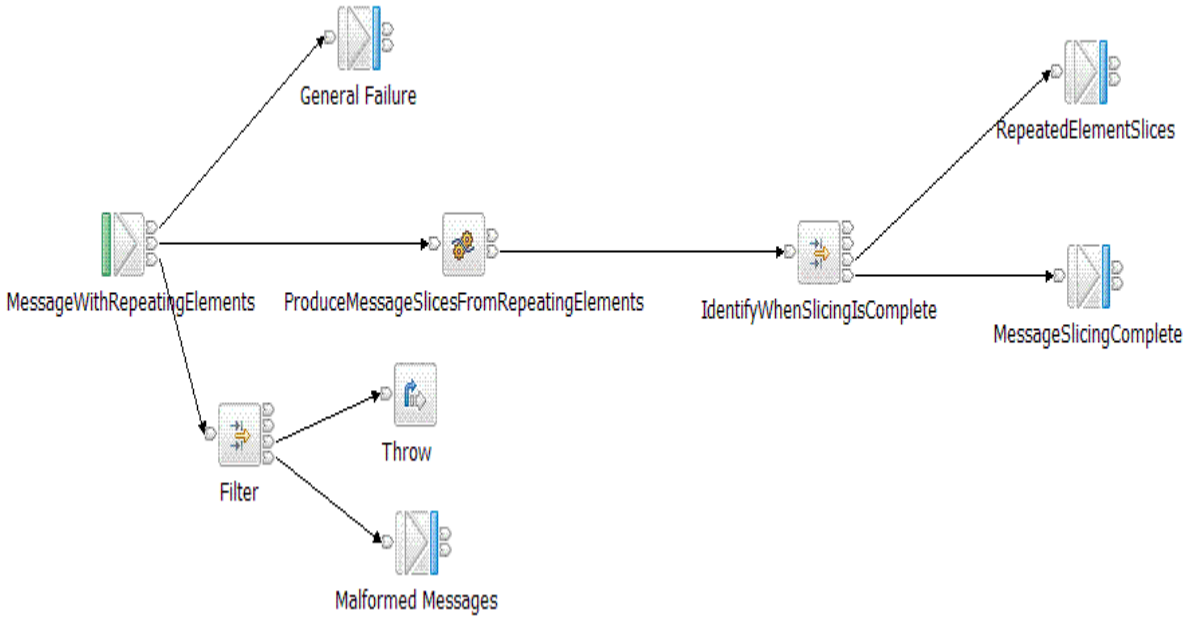
The processing in the sample consists of one message flow. The processing it performs is described below.

Large Messaging Message Flow

The large messaging message flow performs the following processing:

1. Reads a WebSphere MQ message containing an XML payload under transactional control.
2. Formats a WebSphere MQ message for each instance of the SaleList structure.
3. Writes the WebSphere MQ messages to the output queue.
4. Produces a WebSphere MQ message to signal completion of the processing when the final element has been processed.

The Large Messaging message flow consists of the following nodes:



Further information about the Large Messaging sample can be found in the Message Brokers section of the Application samples category which is in the samples gallery of the WebSphere Message Broker development toolkit.

Message Routing

The message routing sample shows how a database table can be used to store routing information which a message flow can then use to route messages to WebSphere MQSeries queues. This uses function which is new in WebSphere Message Broker V6.

The message routing sample shows how to implement a routing table, using shared variables, to route messages in a message flow. Two versions of the message flow were used in these evaluations. One using a database was run as the WebSphere Business Integration Message Broker V5 test case and the second using the routing table implemented using shared variables was run as the WebSphere Message Broker V6 test case.

The processing in the message flows is described below:

Routing_using_database_table Message Flow
The message flow performs the following processing:

1. Reads a WebSphere MQ message containing an XML payload under transactional control.
2. Creates a destination list based on data in a database table and then routes the message to the entries in the destination list. Note this involves a read to the database for every message processed.
3. Produces a WebSphere MQ output message. The destination of the message is specified in the destination list.

This version of the message flow was used for the WebSphere Business Integration Message Broker V5 measurements.

Routing_using_memory_cache Message Flow
The message flow performs the following processing:

1. Reads a WebSphere MQ message containing an XML payload under transactional control.
2. Creates a destination list based on data which is held in shared variables.
3. Produces a WebSphere MQ output message. The destination of the message is specified in the destination list.

This version of the message flow was used for the WebSphere Message Broker V6 measurements.

Further information about the Message Routing sample can be found in the Message Brokers section of the Application samples category which is in the samples gallery of the WebSphere Message Broker development toolkit.

SWIFT Message Parse

The processing of SWIFT messages is a common requirement for financial institutions. The parsing of the messages is achieved using the MRM parser with a message format of Tagged Delimited String (TDS).

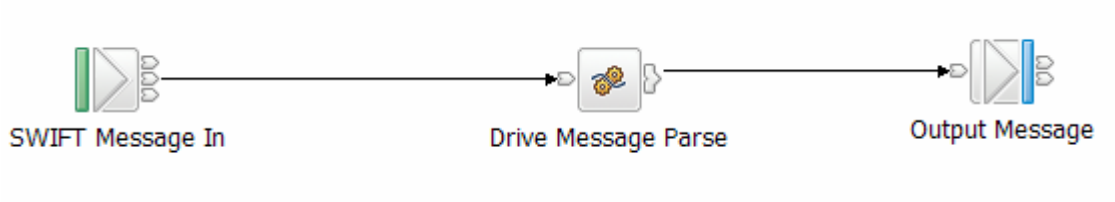
The processing in this test consists of a full parse of a SWFIT MT543 message format.

SWIFT Message Parsing Message Flow

The processing in the SWIFT Message Parse message flow consists of the following:

1. Reads a WebSphere MQ message containing a SWIFT MT543 message in tagged delimited string format.
2. Accesses the last element in the input message.
3. Produces a WebSphere MQ message to signal completion of the processing.

The SWIFT Message Parse processing consists of the following nodes:



The output message is a minimal WebSphere MQ message.

Feedback

This report and other tools that are produced by the performance group are produced in order to help you understand the performance characteristics of WebSphere Message Broker and to assist you with sizing.

It is important that the reports and tools are effective in what they do and it is very useful to have feedback on the content and style of the information which is produced. Your comments, both positive and negative, are therefore welcome.

Your answers to the following questions are particularly interesting:

- What are your most common performance questions?
- Do the reports provide what is needed?
- Is there any other performance information which is required to help you do your job?
- Would you like to see any other aspects of WBIMB performance discussed?

Please supply feedback to Tim Dunn (dunnt@uk.ibm.com) or use the feedback facility on the SupportPac web page where you obtained this report.