WebSphere Message Broker V7.0

For Linux on System x

Performance report

Version 1.0

May, 2010

Tim Dunn

David Gorman

Rob Convery

Oliver Wynn

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".

First Edition, May 2010.
This edition applies to *WebSphere Message Broker V7.0 for Linux on System x* 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 V7.0 for Linux. The information is not intended as the specification of any programming interfaces that are provided by WebSphere MQ or WebSphere Message Broker V7.0 for Linux. It is assumed that the reader is familiar with the concepts and operation of WebSphere Message Broker V7.0.

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 2003, Windows XP, Microsoft Corporation

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

Date	Changes
20/05/2010	Initial Release

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 WMB performance discussed?

Please supply feedback to us at the following e-mail addresses:

Tim Dunn (dunnt@uk.ibm.com)
Dave Gorman (dave_gorman@uk.ibm.com)
Rob Convery (convery@uk.ibm.com)

or use the feedback facility on the SupportPac web page where you obtained this report.

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 three 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 areas of improvement in performance with WebSphere Message Broker V7.0 when compared with WebSphere Message Broker V6.1.

    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. 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.**

3. **Appendices** that contain supplementary information. They are:

    - Appendix A - Measurement Environment
    - Appendix B – Evaluation Method
    - Appendix C – Test Messages
    - Appendix D – Use Case Descriptions
    - Appendix E - Tuning

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

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. In many cases the Hardware, Operating System and prerequisite software are also different making any direct comparisons invalid.

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 *Tuning*.

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 areas of improvement in performance with WebSphere Message Broker V7.0 when compared with WebSphere Message Broker V6.1.

It contains the following sections:

- *Release Highlights* which outlines the main differences in performance when using WebSphere Message Broker V7.0 compared with WebSphere Message Broker V6.1.

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

Release Highlights

Performance Improvements over WebSphere Message Broker V6.1

There have been significant improvements in the performance of WebSphere Message Broker V7.0 in the following areas:

- Message flow deployment
- Toolkit connection times
- Performance analysis

In addition there have been significant enhancements to the Message Broker Toolkit to assist with the rapid diagnosis of runtime performance problems in message flows. It is now possible to very quickly accurately and cheaply diagnose performance problems in a running message flow with a low overhead that is typically no more than 5%.

The throughput capability of the runtime component in WebSphere Message Broker V7.0 is approximately 5% better than that of WebSphere Message Broker V6.1.

Details of the improvements follow.

Message Flow Deployment

The removal of the Configuration Manager in V7.0 has led to a significant improvement in the performance of message flow deployment. Some examples of the improvements are

- The deployment of a message flow to one execution group was measured at 3 seconds; this is less than 25% of the time taken for the same deployment in V6.1. i.e. is less one quarter of the time previously taken.

- The deployment of 10 message flows to each of 20 execution groups was measured at 3.4 seconds per message flow. The total time taken is 23% of the time taken for the same deployment in V6.1.

Toolkit Connection Times

The removal of the Configuration Manager has also led to significant reductions in the time taken for Broker Explorer and the Toolkit to connect to a broker as the figures in the table below show.

- The time taken to connect to a topology of 1 execution group with 1 message flow deployed was measured at 3 seconds; this is 38% of the time taken for the same operation with V6.1.

- The time taken to connect to a topology of 20 execution groups each with 10 message flows deployed was measured at 5 seconds; this is 8% of the time taken for the same operation with V6.1.

The performance improvements mentioned above can be obtained by upgrading to WebSphere Message Broker V7.0. **No code or message model changes are required to benefit from the improvements**.

Fast Low Overhead Performance Analysis

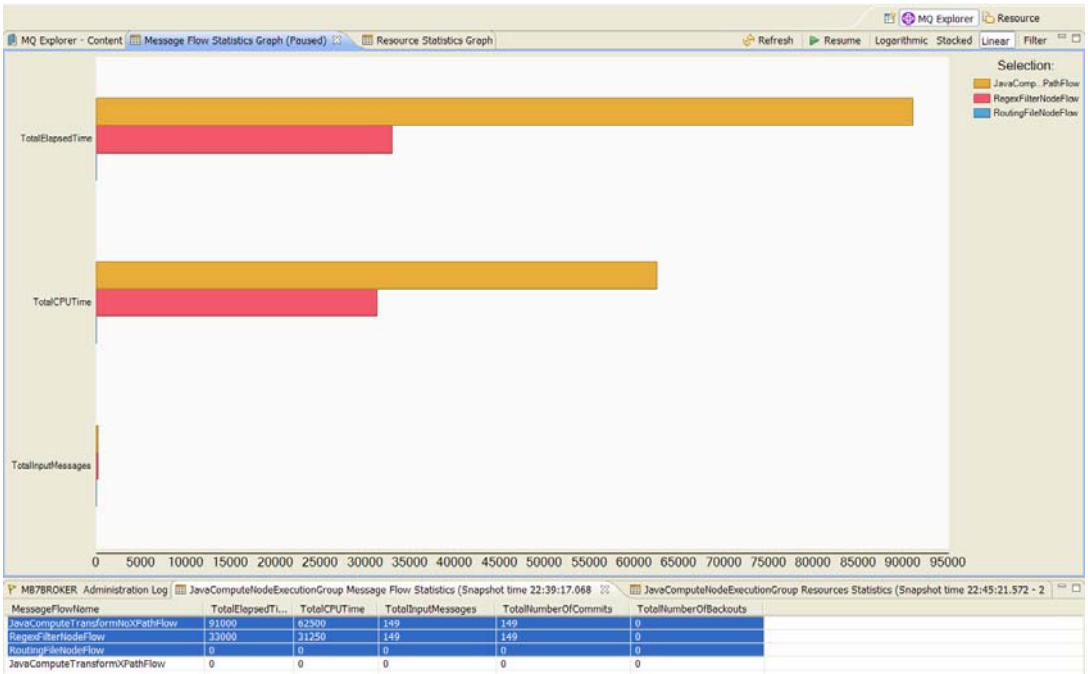
WebSphere Message Broker V7.0 has two new functions in the Broker Explorer to assist with the rapid diagnosis of performance problems in message flows. These are the visualisation of message flow snapshot statistics and resource statistics. The overhead of these functions is very small. Using the XSLT Transformation sample the overhead of the snapshot statistics was measured at 5%. Resource statistics gave a 1% overhead. This is an extremely low overhead given the level of information that is now available.

Message Flow Statistics

Snapshot statistics provide detailed information on the CPU usage and elapsed times of message flows. The data is now available in a graphical and tabular form in the Broker Explorer. The data is updated automatically every 20 seconds once started. It can be paused if needed. There is also the option to copy the data for subsequent analysis into a spreadsheet.

Using this facility it is possible to quickly identify hotspots in message flows for further analysis. This allows effort to be targeted to the correct area straight away and avoids the wasted time that would inevitably result when using a hit and miss approach to problem resolution.

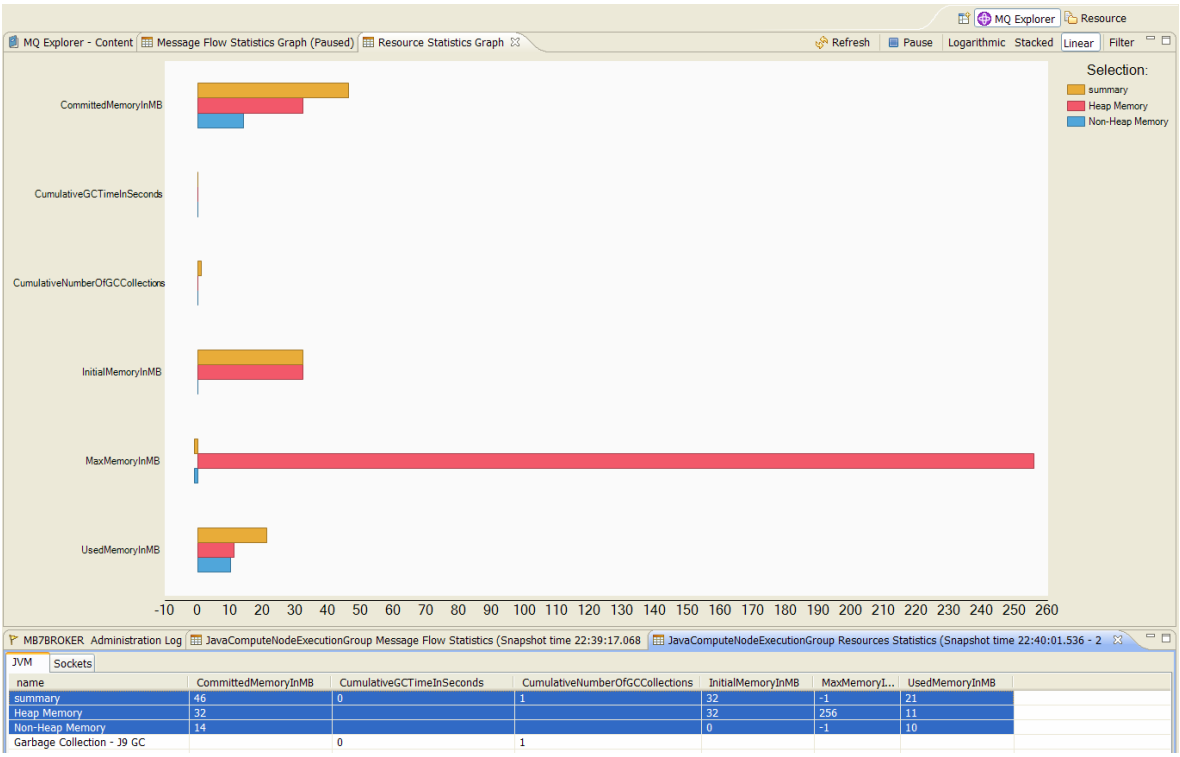
A screenshot of this facility being used is shown below. For more information on this facility consult the product documentation.



Resource Statistics

Message Broker Explorer adds the capability to graphically view information about key broker resources such as the JVM Heap.

A screenshot of the resource statistics facility being used is shown below. For more information on this facility consult the product documentation.



name	CommittedMemoryInMB	CumulativeGCTimeInSeconds	CumulativeNumberOfGCCollections	InitialMemoryInMB	MaxMemoryI...	UsedMemoryInMB
summary	46	0	1	32	-1	21
Heap Memory	32			32	256	11
Non-Heap Memory	14			0	-1	10
Garbage Collection - J9 GC		0	1			

Use Case Throughput

This section illustrates the message throughput that is possible with Message Broker V7.0 for a number of common processing use cases that Message Broker is used for.

You will observe a range of processing rates. The message rate varies with the complexity of the application processing and the size and complexity of the messages as it would for any program.

Use Case Outline

This section contains a brief outline of the tests used and the results for each are presented in the tables below. 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 centre. On receipt at the IT centre 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.

- **File Processing**
  This shows the reading and writing of records stored in files that are read and processed by Message Broker.

- **Message Transformation**
  This shows the transformation of XML messages from one format to another using ESQL which are sent and received over the WebSphere MQ transport:

The following table shows the message rates that were obtained for the different use cases when running on an IBM xSeries X5450 6 with 4 * 3.00 GHz Dual Core processors.

Use Case	Message Size	V7.0 Msgs/sec
Aggregation	20k	410
Coordinated Request/Reply	2k	1337
Data Warehouse	2k	756
Large Messaging	2k	1827
Message Routing	2k	6653
File Processing	2k	1453
Message Transformation	2k	4780

Throughput Comparison for Common Message Broker Use Cases.

Note:

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

Note that there is a range of message rates from 410 to 6653 message per second. These rates reflect the range of complexities for the different use cases shown or put in another way not all use cases are of the same complexity.

When planning a system it is important to understand the complexities of the processing required so that adequate resources can be provided to meet the requirements of the particular situation.

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 V7.0 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 letter for IBM WebSphere Message Broker V7.0 which is available at WebSphere Message Broker V7 Announcement
- 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 V7.0 including a trial edition, 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 V7, go to the WebSphere MQ Web site. Product documentation is also available. This is available at http://www.ibm.com/software/integration/wmqfamily
- 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 Message Broker zone. This is available at http://www.ibm.com/developerworks/websphere/zones/businessintegration/wmb.html
- The MQ,JMS and HTTP transport testing which was run for this report used a tool called the Performance Harness for JMS to generate and consume messages. The tool is useful as a simple way to send and receive messages. The documentation for the tool contains examples of how to run it to send/receive messages. More information on the currently available version can be found here: http://www.alphaworks.ibm.com/tech/perfharness?open&S_TACT=105AGX21&S_CMP=AWRSS.

Part II

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

It contains the following sections:

- *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.

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.

Note that the message size quoted is based on the size of the data in XML format hence when the same data is represented at CWF or TDS format the actual size may be significantly less. The sizes for the different formats are shown in the table below:

Msg Size in XML	Msg Size in TDS	Msg Size in CWF
2k	2k	1k
20k	10k	5k
200k	100k	48k
2048K(2mb)	995k	481k
20480K(20mb)	9941k	4807k

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
2k	No			
20k	No			
200k	No			
2mb	No			
20mb	No			
2k	Yes			
20k	Yes			
200k	Yes			
2mb	Yes			
20mb	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 2K, 20K, 200K, 2M and 20M. 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.

CPU ms/msg: Overall CPU cost per message, expressed as CPU milliseconds per message.

The value of CPU ms/msg is obtained using the calculation:

((Number of cores * 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.

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 at least one 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 as only a single copy of the message flow was run.

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.

Sending and Receiving Messages over different Transports

The tests in this section illustrate the processing cost of receiving and sending data over various transports supported by WebSphere Message Broker.

MQ Nodes

The tests in this section illustrate the processing cost of reading and writing MQ Messages.

Reading and writing to an MQ Queue

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

This test shows the overhead of using message broker to move messages from one MQ Queue to another. Also as many other of the tests included in this report use MQ as the transport it can be used to determine how much of the processing incurred is down to using MQ and how much is the routing/transformation or parsing cost.

An MQ Message is placed on the Input Queue where the incoming message is then copied over to the Output Queue. The message contents are treated as a BLOB and are not modified or parsed in anyway.

This test identifies the cost of reading and writing a BLOB message with MQ as the 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
2k	No	7020.09	15.33	0.17
20k	No	2695.23	9.00	0.27
200k	No	261.81	6.00	1.83
2mb	No	26.90	6.00	17.84
20mb	No	2.62	6.00	183.21
2k	Yes	1619.36	8.33	0.41
20k	Yes	934.10	8.00	0.69
200k	Yes	207.76	9.67	3.72
2mb	Yes	24.91	10.00	32.12
20mb	Yes	2.23	9.67	346.27

HTTP Nodes

The tests in this section illustrate the processing cost of reading and writing HTTP Messages.

Reading and Writing messages over the HTTP Transport

This test consists of HTTP Input Node -> HTTP Reply Node.

This test shows the overhead of using message broker to receive and send messages over the HTTP transport. An HTTP bytes Message is written to the broker over HTTP. The incoming message is then written out unmodified back to the client. The message contents are treated as a BLOB and are not modified or parsed in anyway. Note that persistent HTTP connections were used in this test (see Tuning Section for details)

This test identifies the cost of reading and writing a BLOB message with HTTP. 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	6750.23	35.00	0.41
20k	No	2426.24	48.00	1.58
200k	No	315.55	48.00	12.17
2mb	No	31.71	56.67	142.96

Making an HTTP Request

This test consists of two message flows deployed to the same execution group:

Main message flow.

MQ Input Node -> HTTP Request Node -> MQ Output Node

Backend Message Flow

HTTP Input node -> HTTP Reply Node

This test shows the overhead of making a HTTP Request from message broker. An MQ Message is written to the input queue this message is then sent out over HTTP to an empty backend HTTP flow (hosted on the same broker for convenience). The message is sent back to the flow unmodified over HTTP and put to an MQ Queue. The message contents are treated as a BLOB and are not modified or parsed in anyway.

This test identifies the cost of making a HTTP Request. 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	1340.59	17.67	1.05
20k	No	643.69	18.67	2.32
200k	No	110.47	21.33	15.45
2mb	No	11.30	19.33	136.87
20mb	No	0.98	17.00	1387.76

SOAP Nodes

The tests in this section illustrate the processing cost of receiving, sending and making requests over the SOAP transport. The SOAP nodes were introduced in WMB V6.1 and make development of SOAP over HTTP flows much easier. The SOAP nodes handle the complexities of the SOAP format and so you should expect to pay a runtime processing cost for this ease of use over using HTTP nodes.

Receiving and sending messages over the SOAP transport

This test consists of SOAP Input Node -> Compute Node -> SOAP Reply Node.

The message flow is acting as a Web service provider. A SOAP message is received by the broker via the SOAP Input Node. A Compute Node then copies the SOAP request message across to a SOAP response message which is then sent via the SOAP Reply Node.

Note that persistent HTTP connections were used in this test (see Tuning Section for details).

This test identifies the cost of receiving and sending a 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
2k	No	1542.61	16.00	0.83
20k	No	495.40	14.00	2.26
200k	No	54.82	13.00	18.97
2mb	No	4.57	12.00	210.22
20mb	No	0.43	12.00	2232.56

Receiving and sending messages over the SOAP transport with Validation enabled

This test consists of SOAP Input Node -> Compute Node -> SOAP Reply Node.

The SOAP Input and SOAP Reply nodes are are used in a message flow which implements a provider Web service. The SOAP Input node is configured to enable validation (“Content and value”; SOAP Parser Options select “Build tree using XML schema data types”). A SOAP message is received by the broker via the SOAP Input Node, a Compute Node then copies the unmodified SOAP request message across to a SOAP response message which is then sent via the SOAP Reply Node.

Note that persistent HTTP connections were used in this test (see Tuning Section for details).

This test identifies the cost of receiving and sending a SOAP message where the message is validated.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	1218.22	15.67	1.03
20k	No	290.92	14.00	3.85
200k	No	26.77	13.00	38.85
2mb	No	2.40	11.00	366.67
20mb	No	0.23	8.00	2782.61

Receiving and sending messages over the SOAP transport using WS-Addressing

This test consists of SOAP Input Node -> Compute Node -> SOAP Reply Node.

The SOAP Input and SOAP Reply nodes are used in a message flow which acts as a Web services provider. The SOAP Input node is configured to enable WS-Addressing. A SOAP message is received by the broker via the SOAP Input Node. A Compute Node then copies the unmodified SOAP request message across to a SOAP response message which is then sent via the SOAP Reply Node.

Note that persistent HTTP connections were used in this test (see Tuning Section for details).

This test identifies the cost of receiving and sending a SOAP message where WS-Addressing is enabled. 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	942.89	15.67	1.33
20k	No	234.30	14.00	4.78
200k	No	24.47	15.00	49.04
2mb	No	2.39	12.33	412.83
20mb	No	0.23	12.67	4405.80

Receiving and sending messages over the SOAP transport with SwA attachments

This test consists of SOAP Input Node -> Compute Node -> SOAP Reply Node.

The SOAP Input and SOAP Reply nodes are used in a message flow which implements a provider Web service. A SOAP message with an attachment is received by the broker via the SOAP Input Node, a Compute Node then copies the unmodified SOAP request message across to a SOAP response message which is then sent via the SOAP Reply Node. The size of the SOAP message body remains constant and size of the attachment is increased.

Note that persistent HTTP connections were used in this test (see Tuning Section for details).

This test identifies the cost of receiving and sending a SOAP message with SwA attachments.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	1056.46	18.00	1.36
20k	No	818.49	18.00	1.76
200k	No	276.14	22.67	6.57
2mb	No	13.71	10.00	58.34

Making a SOAP Request

This test consists of one (consumer) flow with

SOAP Input Node -> Compute Node -> SOAP Request Node -> SOAP Reply Node

and a backend (provider) message flow which consists of

HTTP Input Node -> HTTP Output Node.

The SOAP Request node is used in a message flow to invoke a Web Service synchronously. A response must be received from the web service before the message flow continues.

A SOAP message is received by the broker via the SOAP Input Node, a Compute Node then copies the SOAP request message across to a SOAP response message, a SOAP Request Node then issues a Web Service request. When the web service has completed a response is sent to the original request via the SOAP Reply Node.

The web service that is invoked synchronously consists of an HTTP message flow, running in the same broker, which returns the request data unmodified.

This test identifies the cost of making a Web Service request via the SOAP Request node. 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	852.12	18.00	1.69
20k	No	369.04	17.67	3.83
200k	No	51.87	17.00	26.22
2mb	No	4.03	14.00	277.69
20mb	No	0.44	12.00	2165.41

Making a SOAP Request with Validation enabled

This test consists of one (consumer) flow with

SOAP Input Node -> Compute Node -> SOAP Request Node -> SOAP Reply Node

and a backend (provider) flow which consists of

HTTP Input Node -> HTTP Output Node.

The SOAP Request node is used in a message flow which calls a Web Service synchronously. This means the node sends a Web Service request and waits for the associated Web Service response to be received before the message flow continues.

The SOAP Request node is configured to enable validation (“Content and value”; SOAP Parser Options select “Build tree using XML schema data types”). A SOAP message is received by the broker via the SOAP Input Node, a Compute Node then copies the unmodified SOAP request message across to a SOAP response message, a SOAP Request Node then makes a Web Service request, the response is then sent via the SOAP Reply Node. The request is returned, unmodified, via a HTTP flow.

This test identifies the cost of making a Web Service request via the SOAP Request node with validation enabled.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	468.92	15.00	2.56
20k	No	125.65	15.67	9.98
200k	No	11.68	14.67	100.49
2mb	No	1.16	13.00	899.14
20mb	No	0.13	13.00	8000.00

SCA Nodes

The tests in this section illustrate the processing cost of using the SCA nodes which allow WebSphere Message Broker to interoperate with WebSphere Process Server. This supports both WebSphere Process Server to WebSphere Message Broker inbound and WebSphere Message Broker to WebSphere Process Server outbound scenarios.

When designing message flows with SCA, you must select a suitable transport (also called a binding). The SCA nodes support the WebSphere MQ transport and WebSphere Broker HTTP transport with SOAP. These tests show the processing costs of SCA over each of these transports.

Receiving and sending inbound messages with SCA over MQ

This test consists of SCA Input Node -> SCA Reply Node

The bindings are set to MQ.

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	5703.47	14.67	0.21
20k	No	2244.52	8.67	0.31
200k	No	218.36	5.67	2.08
2mb	No	22.72	5.67	19.95
20mb	No	2.26	6.00	212.70

Receiving and sending inbound messages with SCA over SOAP

This test consists of SCA Input Node -> Compute Node -> SCA Reply Node

The bindings are set to Web Services and the Compute node modifies the high level tag from request to response.

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	1483.20	16.00	0.86
20k	No	494.67	14.67	2.37
200k	No	54.88	13.00	18.95
2mb	No	4.58	12.00	209.61
20mb	No	0.41	4.67	910.57

Making an Asynchronous outbound request with SCA over MQ

This test consists of two flows deployed to the same Execution Group :

Main Message Flow

MQ Input Node -> SCA Asynchronous Request Node

SCA Asynchronous Response Node -> MQ Output Node

" Backend " Message Flow

MQ Input Node -> Compute Node -> MQ Output Node

The bindings are set to MQ.

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	3756.53	27.00	0.57
20k	No	1649.87	16.33	0.79
200k	No	214.19	8.67	3.24
2mb	No	22.99	8.00	27.84
20mb	No	2.31	8.00	277.06

Making an Asynchronous outbound request with SCA over SOAP

This test consists of two flows deployed to the same Execution Group :

Main Message Flow

MQ Input Node -> SCA Asynchronous Request Node

SCA Asynchronous Response Node -> MQ Output Node

" Backend " Message Flow

SOAP Input Node -> Compute Node -> SOAP Reply Node

The bindings are set to Web Services.

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	531.88	27.67	4.16
20k	No	153.91	29.00	15.07
200k	No	17.36	29.33	135.20
2mb	No	1.83	29.33	1282.33
20mb	No	0.20	27.67	11066.67

Making a Synchronous outbound request with SCA over MQ

This test consists of two flows deployed to the same Execution Group :

Main Message Flow

MQ Input Node -> SCA Request Node -> MQ Output Node

" Backend " Message Flow

MQ Input Node -> Compute Node -> MQ Output Node

The bindings are set to MQ.

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	2261.04	15.67	0.55
20k	No	1560.99	15.67	0.80
200k	No	214.24	9.00	3.36
2mb	No	23.30	8.33	28.61
20mb	No	2.34	8.33	285.31

JMS Nodes

The tests in this section illustrate the processing cost of utilising JMS messages.

Receiving and sending JMS Messages

This test consists of JMSInput Node -> JMSOutput Node

The JMSInput Node acts as a JMS Receiver on an MQ JMS Queue.

The JMS Output Node acts as a JMS Sender and sends the same message to the same JMS Provider.

For this test the JMS Provider is the broker WMQ Queue Manager.

This test uses a JMS Bytes message.

This test identifies the cost of receiving and sending a JMS Bytes Message with a 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
2k	No	1417.25	18.33	1.03
20k	No	1021.48	20.67	1.62
200k	No	227.88	25.33	8.89
2mb	No	23.47	25.00	85.23
20mb	No	2.30	19.67	684.06

JMS to MQ Protocol conversion

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

The JMSInput Node acts as a JMS Receiver on an MQ JMS Queue.

For this test the JMS Provider is the broker WMQ Queue.

This test uses a JMS Bytes message.

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.

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
2k	No	1407.06	7.67	0.44
20k	No	926.81	10.67	0.92
200k	No	190.17	14.33	6.03
2mb	No	22.41	21.00	74.97
20mb	No	2.30	14.67	510.14

MQ to JMS Protocol conversion

This test consists of
MQ Input Node -> MQJMSTransform Node -> JMSOutput Node

The JMS Output Node acts as a JMS Sender to an MQ JMS Queue.

For this test the JMS Provider is broker WMQ Queue Manager.

Within the MQJMSTransform node the tree built from the MQ input message is converted to one suitable for the JMS transport. A JMS Bytes output message is written.

This test identifies the cost of converting a MQ Message to a JMS 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
2k	No	1957.74	9.67	0.40
20k	No	1312.59	11.67	0.71
200k	No	205.44	12.00	4.67
2mb	No	22.88	16.00	55.95
20mb	No	2.29	12.67	442.50

File Nodes

The tests in this section illustrate the processing cost of reading and writing Files with the File Nodes.

Note for File Output Node tests it was necessary to insert an MQ Output node after the File Output Node. By counting these messages it was possible to determine the rate at which data was being written. This gave a technique that allowed the counting to be done whilst still using the existing automation infrastructure. It avoided contamination of results by having a separate application running locally on the Server counting files and impacting file system operation.

For File Input Node tests a Perl script was run locally to keep the input folder topped up to ensure there was always files to be processed by the broker. Hence these results will include the overhead of this Perl script in the CPU results. In most cases this was at most 5% CPU.

File to File

Reading and Writing Whole Files

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

Message Domain is set to BLOB. The File Input Node is configured to set the Record Detection to Whole File, files are deleted once processed. The File Output Node is configured to set the Record Definition to Whole File. For every file written an MQ message is propagated to the Compute node, the headers and body are not copied and an empty message is sent to the MQ Output node. The queue is drained and the rate is reported.

This test identifies the cost of reading and writing whole Files.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	1208.73	15.67	1.04
20k	No	893.11	12.33	1.10
200k	No	209.95	9.00	3.43
2mb	No	52.33	5.00	7.64
20mb	No	3.79	19.00	401.41

Reading Whole Files and Merging into a Single File

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

Message Domain is set to BLOB. The File Input Node is configured to set the Record Detection to Whole File, files are deleted once processed. The File Output Node is configured to set the Record Definition to unmodified data. For every record written to the file an MQ message is propagated to the Compute node, the headers and body are not copied and an empty message is sent to the MQ Output node. The queue is drained and the rate is reported.

This test identifies the cost of reading whole Files and writing the contents as records to a single large file.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	1331.04	15.67	0.94
20k	No	1197.60	15.33	1.02
200k	No	234.70	10.33	3.52
2mb	No	48.89	6.67	10.91
20mb	No	3.29	11.33	275.58

Reading and Writing Large Files with Delimited Records

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

Message Domain is set to BLOB. The File Input Node is configured to set the Record Detection to Delimited and the Delimiter set to DOS or UNIX line end. Input files are deleted once processed. The File Output Node is configured to set the Record Definition to unmodified data. For every record written an MQ message is propagated to the Compute node, the headers and body are not copied and an empty message is sent to the MQ Output node. The queue is drained and the rate is reported.

This test identifies the cost of reading records from a large file using the line end to delimit the records and writing the unmodified records to another large file.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	1953.56	14.33	0.59
20k	No	1390.63	15.00	0.86
200k	No	298.04	19.33	5.19
2mb	No	29.91	12.00	32.10
20mb	No	0.56	13.33	1916.17

MQ to File

Writing MQ Messages out as Whole Files

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

The Message Domain had a value of BLOB on the MQ Input Node. The File Output Node is configured to set the Record Definition to Whole File. This means that we write a whole file for every MQ message received. The contents of the MQ message are written as the contents of the file.

For every file written an MQ message is propagated to the Compute node, the headers and body are not copied and an empty message is sent to the MQ Output node. The queue is drained and the rate is reported, this rate represents the number of files per second written to the output folder.

This test identifies the cost of reading MQ messages and writing each message as a whole file. 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	2313.14	15.67	0.54
20k	No	2016.41	18.33	0.73
200k	No	256.10	16.00	5.00
2mb	No	26.92	10.00	29.72
20mb	No	2.51	10.00	318.73

Writing MQ Messages as Delimited Records in Large Files

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

The output file was closed when it reached 1GB of data regardless of record size. This was achieved by using the first compute node to count the number of MQ messages received and to check the message size, when 1GB of data was written it sent a message to the Finish File terminal of the File Output Node.

The Message Domain had a value of BLOB on the MQ Input node. The File Output Node is configured to set the Record Definition to Delimited data and the Delimiter is set to the Broker System Line End.

For every record written to the file an MQ message is propagated to the Compute node, the headers and body are not copied and an empty message is sent to the MQ Output node. The queue is drained and the rate is reported this rate represents the number of records per second written to the output file.

This test identifies the cost of reading MQ Messages and writing each one as a record in a large file and delimiting the record with a line end character. 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	2703.49	15.67	0.46
20k	No	2108.41	17.67	0.67
200k	No	460.96	26.33	4.57
2mb	No	45.59	17.00	29.83
20mb	No	4.00	18.00	360.00

Writing MQ Messages as Records in Large Files

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

The output file was closed when it reached 1GB of data regardless of record size. This was achieved by using the first compute node to count the number of records read and to check the record size, when 1GB of data was written it sent a message to the Finish File terminal of the File Output Node.

The Message Domain was set to BLOB on the MQ Input node. The File Output Node is configured to set the Record Definition to Unmodified data.

For every record written to the file an MQ message is propagated to the last Compute node, the headers and body are not copied and an empty message is sent to the MQ Output node. The queue is drained and the rate is reported, this rate represents the number of records per second written to the output file.

This test identifies the cost of reading MQ Messages and writing each one as a record in a large file without any delimiter. 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	2512.43	15.67	0.50
20k	No	2115.51	17.67	0.67
200k	No	458.78	27.00	4.71
2mb	No	46.92	17.67	30.12
20mb	No	4.80	21.33	355.56

File to MQ

Reading Large Files with XML Parsed Record Sequence Records and Writing as MQ Messages

This test consists of File Input Node -> MQ Output Node

For this test we had multiple large input files (each was ~100MB) in the input folder. Each file contained records of the appropriate size. For example for the 4K test the 100MB file contained 256000 records. Each record was a valid XML document.

The Message Domain had a value of XMLNSC. The File Input Node is configured to set the Record Detection to Parsed Record Sequence using the XMLNSC parser. Input files are deleted once processed.

The MQ Output node writes out the record contents as an MQ Message. The output queue was drained and the message rate reported, this represents the number of records per second processed.

This test identifies the cost of reading records from a large file using the XMLNSC parser to delimit the records and writing records as MQ Messages. 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	2703.36	13.33	0.39
20k	No	652.91	13.67	1.67
200k	No	75.17	13.67	14.55
2mb	No	6.75	13.67	15.18
20mb	No	0.70	14.00	1600.00

Reading Large Files with Delimited Records and Writing as MQ Messages

This test consists of File Input Node -> MQ Output Node

Message Domain is set to BLOB. The File Input Node is configured to set the Record Detection to Delimited and the Delimiter set to DOS or UNIX line end. Input files are deleted once processed. The MQ Output node writes out the file contents as an MQ Message.

This test identifies the cost of reading records from a large file using the line end to delimit the records and writing the unmodified records out as MQ Messages.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	5000.21	16.33	0.26
20k	No	3400.11	19.00	0.45
200k	No	384.01	16.33	3.40
2mb	No	40.86	16.00	31.33
20mb	No	3.59	17.00	379.18

Reading and Writing Large Files with Custom Delimited Records

This test consists of File Input Node -> MQ Output Node

Message Domain is set to BLOB. The File Input Node is configured to set the Record Detection to Delimited and the Delimiter set to a custom postfix 4 character delimiter. Input files are deleted once processed. The MQ Output node writes out the file contents as an MQ Message.

This test identifies the cost of reading records from a large file using a custom delimiter to delimit the records and writing the unmodified records out as MQ Messages.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	4975.24	16.33	0.26
20k	No	3586.79	19.00	0.42
200k	No	449.79	16.67	2.96
2mb	No	44.27	18.67	33.73
20mb	No	4.48	19.33	345.50

TCPIP Nodes

The tests in this section illustrate the processing cost of using the TCPIP nodes.

Receiving and sending messages over TCPIP using Fixed Length record detection

This test consists of TCPIPServerInput Node -> TCPIPServerOutput Node

This test illustrates the cost of using the Fixed Length record detection in the TCPIPServerInput node. The incoming message is received by the TCPServerInput node and the response sent from the TCPIPServerOutput. The messages processed are in the XMLNSC domain and record detection was set to Fixed Length on the TCPIPServerInput node.

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	2237.05	15.67	0.56
20k	No	1791.30	16.33	0.73

Receiving and sending messages over TCPIP using Parsed Record Sequence record detection

This test consists of TCPIPServerInput Node -> TCPIPServerOutput Node

This test illustrates the cost of using parsed record sequence detection in the TCPIPServerInput node. The incoming message is received by the TCPServerInput node and the response sent from the TCPIPServerOutput. The messages processed are in the XMLNSC domain and record detection was set to Parsed Record Sequence on the TCPIPServerInput node.

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	1612.28	15.00	0.74
20k	No	555.18	14.00	2.02

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.

The input and output messages are processed using the XMLNSC domain.

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
2k	No	5942.13	15.67	0.21
20k	No	3590.16	17.00	0.38
200k	No	514.75	9.00	1.40
2mb	No	55.16	8.33	12.09
20mb	No	5.11	7.00	109.59
2k	Yes	1490.07	9.67	0.52
20k	Yes	1155.63	10.00	0.69
200k	Yes	385.74	11.00	2.28
2mb	Yes	44.20	10.67	19.31
20mb	Yes	3.00	7.00	186.67

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.

Notes: In this report figures only figures for TDS Fixed length format are given. Previous measurements showed that message throughput varied little regardless of format hence the decision to only report the one type. If more detail on the differences between the different TDS formats is required consult the WebSphere Message Broker V6 performance reports

Parsing a Tagged Delimited String, Fixed Length Input Message

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

The input message is processed with the TDS domain.

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
2k	No	1676.04	13.33	0.64
20k	No	317.15	13.00	3.28
200k	No	32.82	13.00	31.68
2mb	No	3.14	13.00	330.86
20mb	No	0.31	13.00	3354.84
2k	Yes	713.97	13.00	1.46
20k	Yes	255.69	12.00	3.75
200k	Yes	30.88	12.00	31.09
2mb	Yes	3.20	12.00	300.00
20mb	Yes	0.32	13.00	3250.00

Parsing a Custom Wire Format Input Message

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

The input message is processed with the CWF domain.

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
2k	No	2278.68	12.00	0.42
20k	No	494.67	13.00	2.10
200k	No	53.62	12.67	18.90
2mb	No	4.96	12.00	193.68
2k	Yes	750.55	9.00	0.96
20k	Yes	402.95	4.00	0.79
200k	Yes	51.73	12.00	18.56
2mb	Yes	4.98	12.00	192.77

Parsing a Comma Separated Value Input Message using Data Patterns

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

The input message is processed with the TDS domain.

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 XMLNSC 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	1095.33	9.00	0.66
1k	Yes	568.14	11.33	1.60

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

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

The input message is processed with the TDS domain.

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	159.66	13.00	6.51
7k	Yes	148.41	10.00	5.39

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.

The input and output message are processed with the TDS domain.

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	60.71	13.00	17.13
7k	Yes	57.40	10.33	14.40

Parsing an HL7 Input Message using the Tagged Delimited String Parser

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

The input message is processed with the TDS domain.

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 HL7 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
1k	No	87.59	13.00	11.87
1k	Yes	82.06	12.00	11.70

Parsing and Writing a HL7 Input Message using the Tagged Delimited String Parser

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

The input and output message are processed with the TDS domain.

Within the compute node the Envelope within the incoming HL7 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 HL7 message and serializing it again using the Tagged Delimited String format. A single implementation of this message was used which was approximately 1K 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
1k	No	32.77	13.00	31.74
1k	Yes	31.20	12.00	30.77

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, Fixed Length Output Message

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

The input message is processed with the XMLNSC domain. The output message is processed using the TDS domain.

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
2k	No	1467.88	14.00	0.76
20k	No	235.72	13.00	4.41
200k	No	23.09	13.00	45.04
2mb	No	2.10	13.00	495.24
20mb	No	0.21	13.00	4952.38
2k	Yes	663.10	2.00	0.24
20k	Yes	199.52	7.00	2.81
200k	Yes	21.98	12.67	46.10
2mb	Yes	2.02	12.00	475.25
20mb	Yes	0.21	13.00	4952.38

Writing a Custom Wire Format Output Message

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

The input message is processed with the XMLNSC domain. The output message is processed using the CWF domain.

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. 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	2009.44	14.00	0.56
20k	No	338.39	13.00	3.07
200k	No	32.68	12.67	31.01
2mb	No	2.85	12.67	355.97
20mb	No	0.28	13.00	3714.29
2k	Yes	855.37	11.00	1.03
20k	Yes	278.82	12.00	3.44
200k	Yes	30.99	12.00	30.98
2mb	Yes	2.80	13.00	371.43
20mb	Yes	0.29	13.00	3586.21

Parsing Messages in the XMLNSC Domain

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

Parsing an 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 XML input message. As there is no message body on the output message 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
2k	No	4827.63	17.33	0.29
20k	No	1208.53	14.00	0.93
200k	No	129.12	14.00	8.67
2mb	No	12.67	14.00	88.40
20mb	No	1.26	14.00	891.25
2k	Yes	1257.38	9.00	0.57
20k	Yes	511.08	11.33	1.77
200k	Yes	119.22	10.00	6.71
2mb	Yes	11.88	15.00	101.04
20mb	Yes	1.23	15.00	972.97

Writing a Message in the XMLNSC Domain

The test in this section illustrates the CPU processing costs of using the XMLNSC 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 XMLNSC 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 modified 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
2k	No	3624.00	15.33	0.34
20k	No	771.37	14.00	1.45
200k	No	80.14	14.00	13.97
2mb	No	7.12	14.00	157.30
20mb	No	0.72	14.00	1555.56
2k	Yes	1009.58	9.00	0.71
20k	Yes	355.50	10.00	2.25
200k	Yes	64.20	11.00	13.71
2mb	Yes	6.52	11.00	134.90
20mb	Yes	0.62	12.67	1634.41

Validation in the XMLNSC Domain

The test in this section illustrates the CPU processing costs of using the XMLNSC domain to validate an xml message. This is the processing associated with taking a message and validating it against an associated XML Schema.

Validating an XML Message on Input

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

The input node is set to validate the message contents and value. This causes a full parse and validation of the incoming message which is then written unmodified as the payload of the output message.

This test identifies the cost of validating a XML input message in the input node and writing an unmodified 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
2k	No	4133.79	16.00	0.31
20k	No	706.77	13.20	1.49
200k	No	55.87	14.00	20.05
2k	Yes	1146.57	11.00	0.77
20k	Yes	338.01	12.20	2.89
200k	Yes	57.61	13.00	18.05

Validating an XML Message mid flow

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

The validate node is set to validate the message contents and value. This causes a full parse and validation of the incoming message which is then written unmodified as the payload of the output message.

This test identifies the cost of validating a XML input message using the validate node to do this mid flow and writing an unmodified 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
2k	No	2965.34	16.00	0.43
20k	No	500.18	14.00	2.24
200k	No	99.30	13.20	10.63
2mb	No	29.91	13.00	34.77
20mb	No	1.83	13.00	569.55
2k	Yes	897.14	13.00	1.16
20k	Yes	278.9	14.00	0.82
200k	Yes	43.43	12.20	22.47
2mb	Yes	4.13	13.80	267.05
20mb	Yes	0.75	14.00	1485.41

Validating an XML Message on Output

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

The output node is set to validate the message contents and value. This causes a full parse and validation of the incoming message which is then written unmodified as the payload of the output message.

This test identifies the cost of validating a XML input message using the MQ Output Node on output from the flow and writing an unmodified 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
2k	No	2049.57	1.00	0.04
20k	No	278.71	12.00	3.44
200k	No	26.72	12.00	35.93
2mb	No	3.09	13.00	336.79
20mb	No	0.31	13.00	3398.69
2k	Yes	777.52	11.00	1.13
20k	Yes	234.83	11.20	3.82
200k	Yes	29.17	12.80	35.11
2mb	Yes	2.94	13.00	353.50
20mb	Yes	0.29	13.20	3666.67

Opaque Parsing in the XMLNSC Domain

The test in this section illustrates the CPU processing costs of using the XMLNSC domain opaquely parse an XML message. For an explanation of opaque parsing please see the product documentation.

Filtering on the last element of an XML Message using Opaque Parsing on the XML Body

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

The MQ Input node is set to parse the repeating SalesList Elements of the XML message opaquely. Within the filter node the last element of the incoming message is examined, this last element is outside of a sales list element. The result is always set to be true and thus the message is propagated to the MQ Output node.

This test identifies the cost of filtering on an element at the end of a message using the XMLNSC parser with opaque parsing. Almost the entire body of the message will be opaquely parsed. 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	7626.21	17.33	0.18
20k	No	2687.80	15.00	0.45
200k	No	256.24	13.67	4.27
2mb	No	26.98	14.00	41.51
20mb	No	2.61	13.33	408.68
2k	Yes	1217.06	8.00	0.53
20k	Yes	743.91	8.67	0.93
200k	Yes	144.93	12.00	6.62
2mb	Yes	18.56	15.00	64.67
20mb	Yes	1.10	9.00	654.55

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 and PHPCompute). 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 Database Route and Route Nodes

The tests in this section illustrate the processing costs of using the new Database Route and Route Nodes for routing operations.

Using Database Route Node to Route an Incoming Message Based on Data in a Database

This test consists of MQ Input Node -> Database Route Node -> MQ Output Node.

The MQ Input node is set to use the XMLNSC domain. Within the Database Route node a query is performed to obtain a single piece of data from the Database. This data is used to route the message to an output queue. The lookup result is always set to be true and thus the message is propagated to the MQ Output node.

This test identifies the cost of using the Database Route Node to route a 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
2k	No	3208.58	17.67	0.44
20k	No	2200.03	18.33	0.67
200k	No	259.49	8.00	2.47
2mb	No	26.79	6.00	17.92
20mb	No	2.62	6.00	182.97
2k	Yes	1084.43	11.00	0.81
20k	Yes	656.53	9.67	1.18
200k	Yes	196.02	10.00	4.08
2mb	Yes	22.31	10.00	35.86
20mb	Yes	2.00	9.00	359.40

Using Route Node to Route an Incoming Message Based on Data in the Incoming Message

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

The MQ Input node is set to use the XMLNSC domain. Within the Route node the first element of the message is queried and the message routed based on this value. The lookup result is always set to be true and thus the message is propagated to the MQ Output node.

This test identifies the cost of using the Route Node to route a message 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
2k	No	5982.41	19.33	0.26
20k	No	2686.62	14.33	0.43
200k	No	261.49	6.67	2.04
2mb	No	27.01	6.67	19.75
20mb	No	2.62	6.00	183.21
2k	Yes	1259.12	10.00	0.64
20k	Yes	755.14	8.00	0.85
200k	Yes	208.89	10.33	3.96
2mb	Yes	23.75	10.00	33.69
20mb	Yes	2.13	10.00	375.00

Using ESQL

The tests in this section illustrate the processing costs of using ESQL for different routing and transformation operations. For more details on the impact of using specific ESQL functions such as ROW or EVAL see the Message Broker V6.0 reports which cover these aspects of ESQL in more detail.

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 MQ Output 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
2k	No	7631.71	17.67	0.19
20k	No	2668.19	11.00	0.33
200k	No	255.65	6.00	1.88
2mb	No	26.83	6.33	18.88
20mb	No	2.59	5.67	175.03
2k	Yes	1575.02	10.00	0.51
20k	Yes	876.14	8.67	0.79
200k	Yes	204.46	9.00	3.52
2mb	Yes	23.13	10.00	34.58
20mb	Yes	1.91	8.00	335.08

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 MQ Output 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
2k	No	5937.05	19.00	0.26
20k	No	1266.67	15.67	0.99
200k	No	130.95	14.00	8.55
2mb	No	12.33	14.00	90.81
20mb	No	1.23	14.00	910.57
2k	Yes	1174.98	9.33	0.64
20k	Yes	463.39	10.67	1.84
200k	Yes	85.16	12.67	11.90
2mb	Yes	10.56	14.33	108.62
20mb	Yes	0.99	15.00	1212.12

Transformation 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 transformation 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
2k	No	4780.17	16.67	0.28
20k	No	1147.96	14.00	0.98
200k	No	131.31	14.00	8.53
2mb	No	12.42	14.00	90.18
20mb	No	1.28	14.00	875.00
2k	Yes	1293.23	11.00	0.68
20k	Yes	640.17	12.67	1.58
200k	Yes	124.31	15.00	9.65
2mb	Yes	12.33	15.00	97.30
20mb	Yes	1.27	15.00	944.88

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 MQ Output 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
2k	No	4494.85	14.00	0.25
20k	No	1922.60	10.67	0.44
200k	No	219.29	5.33	1.95
2mb	No	23.10	5.67	19.62
20mb	No	2.37	5.00	169.01
2k	Yes	1382.38	10.00	0.58
20k	Yes	862.93	9.33	0.87
200k	Yes	198.30	9.00	3.63
2mb	Yes	21.40	8.33	31.15
20mb	Yes	2.10	9.67	368.25

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
2k	No	4148.68	16.00	0.31
20k	No	1067.36	15.00	1.12
200k	No	125.06	14.00	8.96
2mb	No	11.71	14.00	95.64
20mb	No	1.19	14.00	941.18
2k	Yes	1218.86	11.00	0.72
20k	Yes	431.87	2.00	0.37
200k	Yes	80.94	8.00	7.91
2mb	Yes	10.03	11.33	90.37
20mb	Yes	0.98	12.00	982.94

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
2k	No	4465.44	14.33	0.26
20k	No	1913.91	10.67	0.45
200k	No	220.14	5.00	1.82
2mb	No	23.39	6.00	20.52
20mb	No	2.36	5.00	169.73
2k	Yes	1358.92	9.67	0.57
20k	Yes	874.08	7.00	0.64
200k	Yes	189.89	7.00	2.95
2mb	Yes	21.52	9.67	35.94
20mb	Yes	1.87	9.33	400.00

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
2k	No	4385.72	16.00	0.29
20k	No	1090.19	15.00	1.10
200k	No	123.84	14.00	9.04
2mb	No	11.19	13.67	97.71
20mb	No	1.17	14.00	957.26
2k	Yes	1147.52	10.00	0.70
20k	Yes	436.18	10.33	1.90
200k	Yes	86.73	11.33	10.45
2mb	Yes	9.81	12.33	100.58
20mb	Yes	0.93	13.33	1142.86

Transformation 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 transformation. 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	1204.00	14.00	0.93
20k	No	176.02	13.00	5.91
200k	No	17.79	13.00	58.46
2mb	No	1.67	13.00	622.75
20mb	No	0.17	13.00	6117.65
2k	Yes	663.78	13.00	1.57
20k	Yes	138.42	9.00	5.20
200k	Yes	16.73	12.00	57.37
2mb	Yes	1.63	12.00	588.96
20mb	Yes	0.16	12.67	6468.09

Transformation 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 transformation.

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	1840.76	15.00	0.65
20k	No	282.39	13.00	3.68
200k	No	27.57	13.00	37.73
2mb	No	2.51	13.00	414.34
20mb	No	0.27	13.00	3851.85
2k	Yes	823.84	12.33	1.20
20k	Yes	234.58	6.00	2.05
200k	Yes	25.70	12.67	39.43
2mb	Yes	2.44	13.00	425.65
20mb	Yes	0.24	14.00	4602.74

Using PHP

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

Copying a message using the PHPCompute node

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

The PHPCompute node uses the MessageBrokerCopyTransform annotation which instructs the PHPCompute node to copy the input message and pass the copied message to the evaluate method. No further modifications are made to the output message.

This test identifies the cost of using the PHPCompute node scripting 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
2k	No	4116.57	20.00	0.39
20k	No	2648.08	18.00	0.54
200k	No	263.34	9.00	2.73
2mb	No	26.54	7.00	21.10
20mb	No	2.61	6.00	183.91
2k	Yes	1198.37	11.67	0.78
20k	Yes	762.81	10.00	1.05
200k	Yes	206.41	10.00	3.88
2mb	Yes	24.26	11.00	36.28
20mb	Yes	2.22	10.00	360.36

Transformation of an Output Message using the PHPCompute node

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

The PHPCompute node uses the MessageBrokerSimpleTransform annotation 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 PHPCompute node transformation (scripting) 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
2k	No	472.01	15.67	2.66
20k	No	62.90	17.33	22.05
200k	No	6.40	18.67	233.33
2mb	No	0.65	20.00	2474.23
2k	Yes	448.54	16.67	2.97
20k	Yes	63.99	17.67	22.09
200k	Yes	6.49	19.33	238.32
2mb	Yes	0.66	20.33	2477.16

Using XSLT

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

Transformation 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 transformation. 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	1654.32	17.67	0.85
20k	No	338.79	17.33	4.09
200k	No	39.02	17.00	34.85
2mb	No	3.71	18.00	387.79
20mb	No	0.38	16.33	3438.60
2k	Yes	730.32	12.00	1.31
20k	Yes	272.49	15.67	4.60
200k	Yes	34.24	15.00	35.04
2mb	Yes	3.42	16.33	381.69
20mb	Yes	0.37	15.67	3387.39

Using the Collector Node

The tests in this section illustrate the processing costs of using the Collector node for combining incoming messages. To allow for comparisons between collector tests the compute node used for all tests in this section is the same i.e. processing costs of this part in the flow is the same in all tests.

Collecting Messages from Several Inputs Based on Number of Messages

This test consists of

2 MQ Input Nodes -> Collector Node -> Java Compute Node -> MQ Output Node.

The two MQ Input nodes each propagate messages to the collector node. In the collector node a collection is defined as being 1 input message from each of the two input terminals. The Collector node Persistence mode is set to Non-Persistent. Which means that the messages are stored on the Collector node's queues Non-Persistently. Once this collection is satisfied it is propagated to the Java Compute Node which copies the entire message from the first input terminal to the output message. Then one field from the message received on the second terminal is retrieved from the input message and used to add a new field to the out going message. The message is then sent to an MQ Output node.

This test identifies the cost of using the collector node to collect 2 messages Non-Persistently from different inputs and then update one of them with a field from the other message.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	1395.71	25.00	1.43
20k	No	608.21	19.67	2.59
200k	No	74.05	15.33	16.56
2mb	No	7.06	16.00	181.39
20mb	No	0.69	15.00	1739.13

The results in the table below show the cost of running with the Collector node Persistence mode set to Persistent and using persistent transacted MQ messages to drive the flow. Running with a Collector node persistence mode set to persistent means that the messages are stored on the Collector node's queues as MQ persistent messages.

This test identifies the cost of using the collector node to collect 2 messages persistently from different inputs and then update one of them with a field from the other 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
2k	Yes	728.45	21.67	2.38
20k	Yes	480.18	19.67	3.28
200k	Yes	74.20	17.00	18.33
2mb	Yes	7.00	17.00	194.29
20mb	Yes	0.70	17.00	1942.86

Collecting Messages from Several Inputs Based on Number of Messages with a Correlation Pattern

This test consists of 2 MQ Input Nodes -> Collector Node -> Java Compute Node -> MQ Output Node.

The two MQ Input nodes all propagate messages to the collector node. In the collector node a collection is defined as being one input message from each of the two input terminals and also a correlation path to look at the first customer surname in the message. Hence messages with the same customer name are put in the collection. The Collector node Persistence mode is set to Non-Persistent. This means that the messages are stored on the Collector node's queues Non-Persistently. Once the collection is satisfied it is propagated to the Java Compute Node which copies the entire message from the first input terminal to the output message. Then one field from the message received on the second terminal is retrieved from the input message and used to add a new field to the out going message. The message is then sent to an MQ Output node. All input messages had the same matching name.

This test identifies the cost of using the collector node to collect 2 messages Non-Persistently from different inputs which have a matching surname and then update one of them with a field from the other message.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	994.22	27.33	2.20
20k	No	561.06	21.67	3.09
200k	No	73.72	15.33	16.64
2mb	No	7.08	16.00	180.79
20mb	No	0.69	15.33	1777.78

The results in the table below show the cost of running with the Collector node Persistence mode set to Persistent and using persistent transacted MQ messages to drive the flow. Running with a Collector node persistence mode set to persistent means that the messages are stored on the Collector node's queues as MQ persistent messages.

This test identifies the cost of using the collector node to collect 2 messages Non-Persistently from different inputs which have a matching surname and then update one of them with a field from the other 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
2k	Yes	656.92	23.67	2.88
20k	Yes	428.24	20.67	3.86
200k	Yes	71.72	17.00	18.96
2mb	Yes	7.07	17.00	192.45
20mb	Yes	0.68	16.67	1960.78

Using the Sequence Node

The tests in this section illustrate the processing costs of using the Sequence node to generate a sequence number.

Incrementing Sequence numbers for each Input Message

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

The Sequence node is used to increment the sequence number for each new message (which are all part of the same sequence group) and to store the sequence number in the LocalEnvironment ($OutputLocalEnvironment/Sequence/Number) using a literal start and end of sequence definition.

This test identifies the cost of using the Sequence node to generate sequence numbers.

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	5355.01	20.00	0.30
20k	No	2702.80	16.33	0.48
200k	No	259.26	7.00	2.16
2mb	No	26.55	6.00	18.08
20mb	No	2.53	6.00	189.47
2k	Yes	1276.60	10.33	0.65
20k	Yes	827.14	9.33	0.90
200k	Yes	209.76	9.67	3.69
2mb	Yes	24.70	10.00	32.39
20mb	Yes	2.34	10.00	342.37

Using the IMS Node

The tests in this section illustrate the processing costs of using the IMS node to send synchronous requests to an IMS system.

Using the IMS node to make synchronous requests

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

The test illustrates the cost of synchronously invoking the IMS request node.

An MQ input message is received to initiate the message flow. The IMSRequest node then synchronously invokes a transaction on a remote IMS system (on z/OS).

The transaction did not involve the updating of any resources on the IMS system.

The commit mode (SEND_THEN_COMMIT) and sync level (CONFIRM) were set on the IMSRequest node. 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	582.06	4.00	0.55
1k	Yes	585.82	6.00	0.82

Business-level Monitoring

The tests in this section illustrate the processing costs associated with using Business-level Monitoring. The message flow is configured to emit event messages that can be used to support transaction monitoring and auditing, and business process monitoring.

Emitting one event with header information on Transaction start

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

The MQInput node is configured to emit an event on Transaction start where the message contains information about the source of the event, the time of the event, and the reason for the event. The event does not include the message bit stream in the event payload.

This test identifies the cost of emitting a single event with only the event header information. 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	5218.93	17.33	0.27
20k	No	2686.11	15.33	0.46
200k	No	260.00	7.00	2.15
2mb	No	26.75	6.00	17.94
20mb	No	2.61	6.00	183.67
2k	Yes	1296.80	10.33	0.64
20k	Yes	835.62	9.67	0.93
200k	Yes	205.07	9.33	3.64
2mb	Yes	24.22	10.00	33.03
20mb	Yes	2.28	10.00	351.39

Emitting one event with a single selected element

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

The MQInput node is configured to emit an event on Transaction start where the message contains information about the source of the event, the time of the event, and the reason for the event. In addition the event emitted also uses an XPath query take one field from the message tree and add this to the event payload.

The XPath query used is the expression ($Body/Parent [1]/First[1]) thereby ensuring that a full parse of the message is not driven.

This test identifies the cost of emitting a single event with a simple XPath expression.
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	3914.14	17.33	0.35
20k	No	2679.92	14.00	0.42
200k	No	260.09	6.67	2.05
2mb	No	26.98	6.00	17.79
20mb	No	2.65	6.00	180.90
2k	Yes	1129.68	11.00	0.78
20k	Yes	760.68	9.67	1.02
200k	Yes	184.27	9.00	3.91
2mb	Yes	24.26	10.00	32.98
20mb	Yes	2.31	10.33	357.86

Emitting one event plus the full message tree

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

The MQInput node is configured to emit an event on Transaction start where the message contains information about the source of the event, the time of the event, and the reason for the event. In addition the event emitted also uses an XPath query to select all the fields from the message tree and add these to the event payload.

The XPath query uses an expression ($Body/Parent) whereby all the fields in the message tree are parsed.

This test identifies the cost of emitting a single event with a full message parse.
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	2513.04	14.67	0.47
20k	No	663.81	14.00	1.69
200k	No	72.21	14.00	15.51
2mb	No	6.00	14.00	186.67
20mb	No	0.57	14.00	1964.91
2k	Yes	843.02	9.67	0.92
20k	Yes	342.38	12.00	2.80
200k	Yes	44.21	9.67	17.49
2mb	Yes	5.43	13.00	191.41
20mb	Yes	0.51	14.00	2196.08

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.

The input and output message are processed with the XMLNSC domain.

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.
The results are not cached in the flow, so a lookup is performed for each message. The volume of data in the database was small and so this represents the best case.

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
2k	No	2794.16	15.33	0.44
20k	No	2702.61	17.67	0.52
200k	No	562.38	11.00	1.56
2mb	No	53.81	8.00	11.89
20mb	No	5.10	7.00	109.80
2k	Yes	1064.89	11.00	0.83
20k	Yes	935.39	10.33	0.88
200k	Yes	360.22	12.00	2.67
2mb	Yes	43.91	10.67	19.43
20mb	Yes	2.98	7.00	188.13

Reading from a Database using the Database Retrieve Node

This test consists of MQ Input Node -> Database Retrieve Node -> MQ Output Node.

The input and output message are processed with the XMLNSC domain.

Within the DB Retrieve node the copy message box is ticked to copy over the input message contents to the outgoing message. The DB is queried to obtain a piece of data from the Database. This data is used to create an element in the output message.

This test identifies the cost of using the Database Retrieve Node to retrieve a piece of data from the database and insert it into the outgoing message. The test performs a similar function to reading from a database as in the case above but achieves it using a different

node and involves no programming. 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	2617.04	18.00	0.55
20k	No	2360.78	19.00	0.64
200k	No	261.05	8.33	2.55
2mb	No	26.70	6.00	17.98
20mb	No	2.66	6.00	180.68
2k	Yes	1061.04	12.00	0.90
20k	Yes	720.75	10.33	1.15
200k	Yes	198.23	10.00	4.04
2mb	Yes	22.97	11.00	38.32
20mb	Yes	1.92	9.00	374.35

Inserting into a Database

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

The input and output message are processed with the XMLNSC domain.

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
2k	No	1441.20	9.00	0.50
20k	No	1396.53	10.33	0.59
200k	No	562.93	11.00	1.56
2mb	No	54.08	8.00	11.84
20mb	No	5.14	7.00	109.02
2k	Yes	847.29	8.67	0.82
20k	Yes	712.43	8.00	0.90
200k	Yes	298.61	10.00	2.68
2mb	Yes	36.41	8.00	17.58
20mb	Yes	2.82	6.33	179.67

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.
The following tests are shown as examples of these kinds of processing with WMB V6.1 but the cost will vary substantially with the number of parameters passed into the external procedure. The WMB V6.0 performance reports contains more detailed analysis on the cost of different parameters and can be referred to if required.

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. 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	4924.877	16.00	0.00
2k	Yes	739.757	6.00	0.00

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. 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	9.323	13.00	0.06
2k	Yes	9.157	11.33	0.05

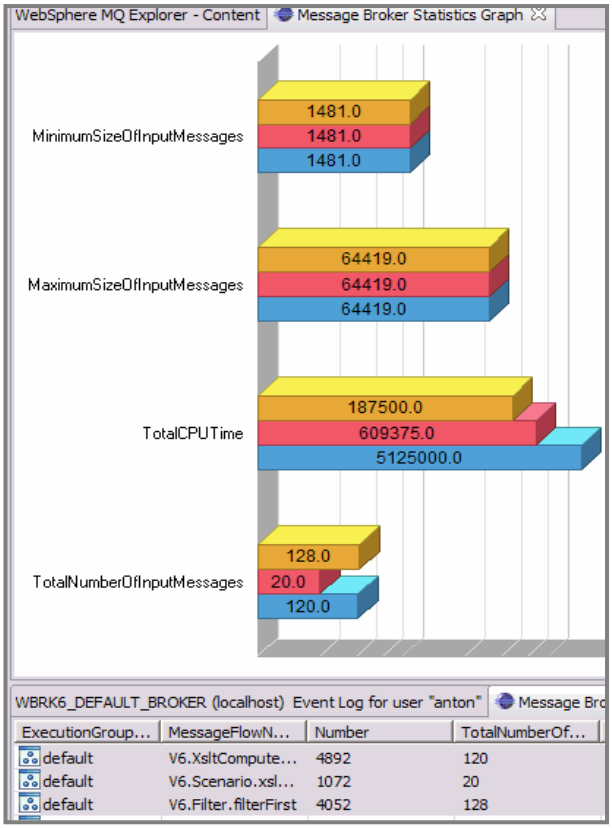
Overheads

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

Message Flow Execution Statistics

Message Broker V7.0 provides a new function called the Broker Explorer. Within the Broker Explorer is the capability to start the collection of snapshot statistics for named message flows. The collected data can be displayed in both a graphical and tabular format in the Broker Explorer. The collected data is automatically updated every 20 seconds.

The picture below shows an example of the data display feature.



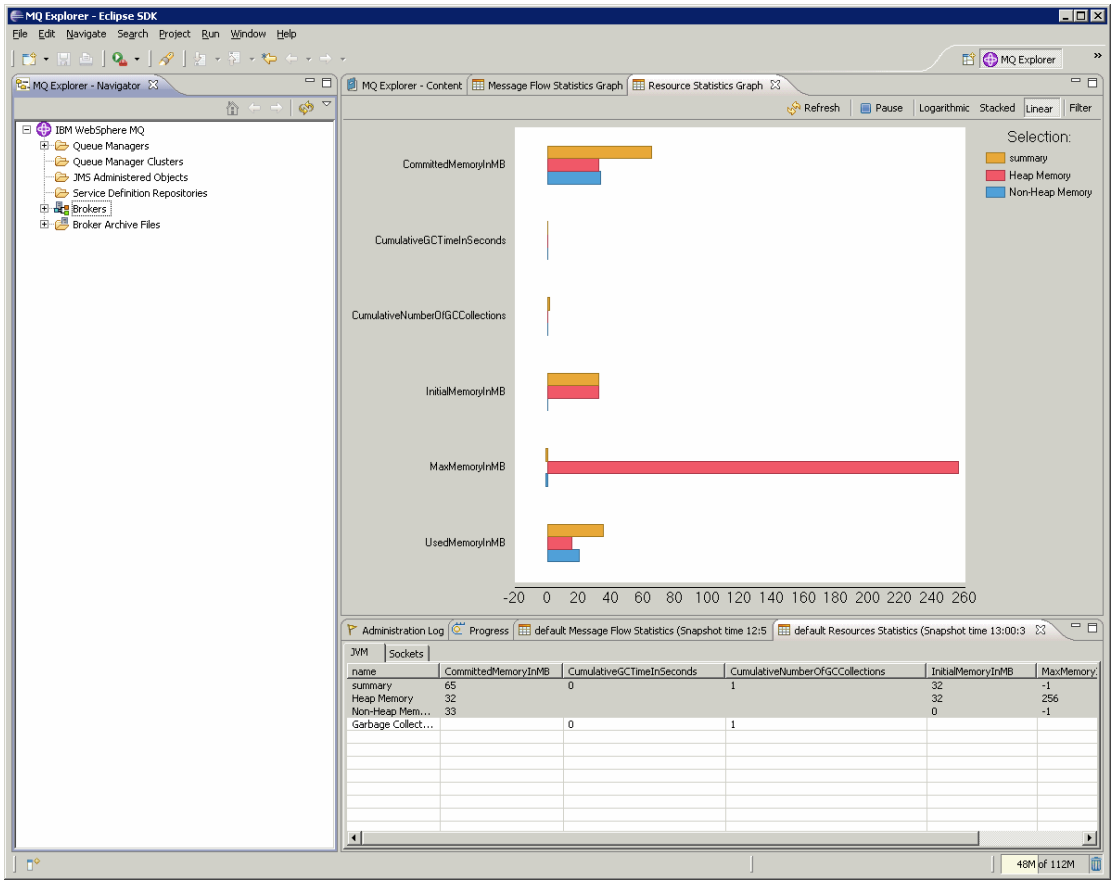
The overhead of using this facility was measured when running the XSLT Transform Sample processing a 2K input message. The processing overhead was measured at 5%. This is a very small overhead for the quality of the data collected.

Resource Statistics

Message Broker Explorer adds the capability to graphically view information about key broker resources such as JVM Heap Sizes and Sockets.

Resource statistics complement the accounting and statistics data that you can collect on message flows, which are also available in Broker Explorer.

A screenshot of the resource statistics facility being used is shown below. For more information on this facility consult the product documentation



The overhead of using this facility was measured when running the XSLT Transform Sample processing a 2K input message. The processing overhead was measured at 1%. This is a very small overhead for the quality of the data collected.

Using Trace and Trace Nodes

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 4K message size there was an 13% reduction in the 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, approx 24%.

You are strongly recommended not to use WebSphere Message Broker trace in a production system.

Since WebSphere Message Broker V6.1 the overhead of leaving trace nodes in a message flow has been reduced. Trace nodes can also now be enabled/disabled easily using the Message Broker Toolkit.

The following tests show the overhead associated with using trace nodes.

Running a Flow with no Trace Nodes

This test consists of the Large Message Sample flow as shipped with WMB V7.0. It indicates the throughput of the flow before any trace nodes are added for comparison.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	1827.43	16.00	0.70

Impact of Running a Flow with a Trace Node Turned On

This Test consists of the Large Message Sample flow as shipped with WMB V7.0. The flow has been modified to add a trace node after the compute node. The trace node writes out to a file the Root of the message tree for every message.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	453.2	14.00	2.78

This indicates that the overhead of having the trace node on.

Impact of Running a Flow with a Trace Node Turned Off

This test is identical to the one above but the Trace node has been disabled.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
2k	No	1783.6	16.00	0.72

This indicates that overhead of running with trace nodes disabled for this flow is 2.4%. This is a small overhead and you may well feel it is worth the cost in order to have the trace nodes embedded in the message with the ability to turn them on with a single command. That is without any need to change the message flow.

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 and the runtime can be found in the Installation Guide which can be downloaded here:

http://www.ibm.com/software/integration/wbimessagebroker/requirements/#BrokersV70

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 improved performance use a 3 GHz or faster processor. For test and production a multi-processor machine is recommended. For development 2GB+ of memory is recommended. For test, QA and production a suggested minimum would be 8GB of memory.

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 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 using pslist for windows and the ps command for other platforms.

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 in MB used by an execution group for the message flow after it has processed a number of messages and the size has 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.

The command used to determine memory usage was "ps -p <Dataflowengine Pid> -o vsz,sz”

Use Case	Message Size	Virtual Memory Peak After Processing Messages (MB)	Real Memory Peak After Processing Messages (MB)
Aggregation	2k	977	244
Coordinated Request/Reply	2k	943	236
Data Warehouse	2k	815	204
Large Messaging	2k	841	210
Message Routing	2k	818	204
XML Transformation	2k	828	207

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

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 x5450 with 4 * 3.0 GHz Dual Core processors giving 8 cores in total.
- One 300 GB SAS hard drive
- SAN comprising
    - Two IBM SAN32M-2 (2026-432) 4Gbps 32-port switches
    - IBM DS6800 (1750-511) with
        - Two drawers of 73GB 15K RPM fibre channel drives configured as RAID-5
        - Eight 2Gbps SAN ports
- 33 GB RAM
- 1 GB Ethernet Cards

The software consisted of:

Red Hat Enterprise Linux Server release 5.3 (Kernel 2.6.18-128.4.1.el5)

- WebSphere MQ V7.0.1
- WebSphere Message Broker V7.0
- DB2 V9.5

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 335 with 2 * 3.06 GHZ Xeon processors
- Two 34 GB SCSI hard drives formatted with NTFS
- 4 GB RAM
- 1 GB Ethernet card

The software consisted of:

- Microsoft Windows 2000 Service Pack 4
- WebSphere MQ V7.0.1
- IBM Java 1.6

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.

Two 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. This configuration tested these transports:
    a. MQ
    b. JMS
    c. HTTP
    d. SOAP
2. File 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. This approach was used for MQ, JMS, HTTP and SOAP messages.

Message Generation and Consumption

The Performance Harness for JMS, a multi threaded WebSphere MQ Client program written in Java was used to generate input messages for the test case being run and to consume the output messages. The following PerfHarness modules were used for point to point testing:

- mqjava.Requestor – for MQ Messages
- http.Requestor – for Sending SOAP and HTTP messages
- jms.r11.Requestor – for sending and receiving JMS Messages

Differences between the transport testing are detailed below:

MQ

Both persistent and non persistent messages MQ Messages were generated from this program. Persistent messages were sent as part of a transaction which was committed after every message.

Sufficient threads (typically 20) 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.

Each thread sent a message and then immediately went to receive a reply on the output queue. 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. Once a thread received a reply it sent another message. The message content was the same for all threads and all messages.

JMS

The tool sent non persistent JMS Bytes messages to a JMS Destination. The connection factory for the client used the MQ Client transport to send messages. This destination was mapped to an MQ Queue on the Brokers Queue Manager. The JMS Input node was configured to read from this queue, the connection factory for the nodes used the MQ Bindings transport for connection. The flow then placed the reply message on another MQJMS queue on output where the client could then receive the reply.

Sufficient threads (typically 20) 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.

Each thread sent a message and then immediately went to receive a reply on the output queue. 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 JMS correlation identifiers to limit consumption of a message to the thread which created it. Once a thread received a reply it sent another message. The message content was the same for all threads and all messages.

SOAP and HTTP

The tool sends predefined SOAP and HTTP Messages that it reads from files. The tool sent the messages to broker using persistent HTTP connections, this means that each thread reused the same TCPIP socket for each request. Each client thread had its own TCPIP socket connection to send/receive data.

Sufficient threads (typically 20) were run in the multi threaded client to ensure that there were always messages to be processed. This is important when measuring message throughput.

As per the HTTP request/reply protocol each thread sent a message and then immediately went to receive a reply on socket. Once a thread received a reply it sent another message. The message content was the same for all threads and all messages.

See the Performance Harness section in this report for more information on this tool.

Machine Configuration

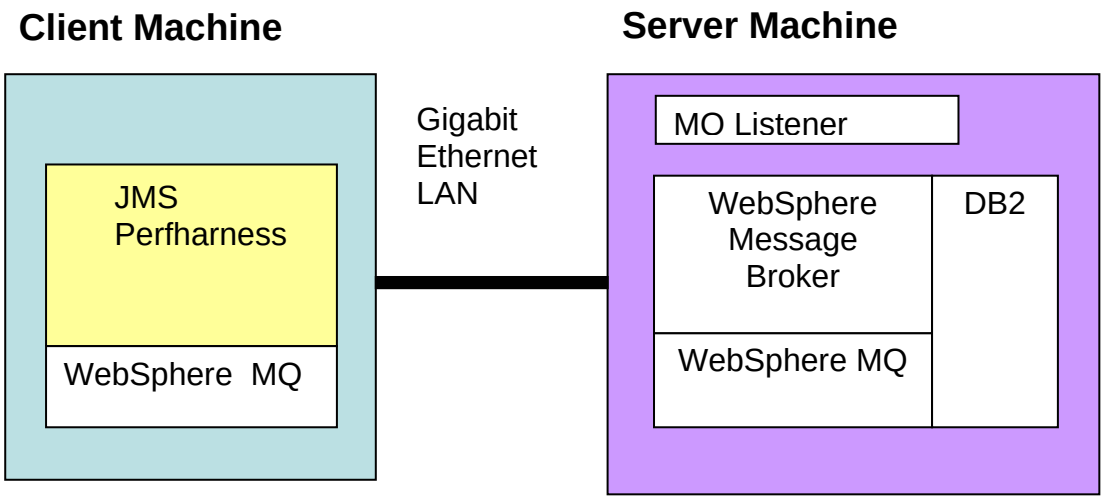
The Performance Harness for JMS was used to generate and consume messages for the message flows was and run on a dedicated machine, the Client Machine. The Message Broker, its dedicated WebSphere MQ queue manager and broker database were all located on a dedicated machine, the Server Machine.

There was a single client machine.

For MQ and JMS based Tests 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.

Messages were transmitted from the client machines to the server machine using the WebSphere MQ transport or SOAP/HTTP or JMS depending on the test type.

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.

File Processing

This section describes how messages were generated and consumed for tests which used the file nodes.

Message Generation and Consumption

For tests using the file input node the directory was pre-populated with a number of files for the test. During the test a Perl script was run locally on the broker machine which monitored the number of files and copied more files to the input directory if required. This ensured that there were always files to be processed, this is important when measuring throughput. If the flow was File to MQ then we used our normal performance harness tool to drain the output queue and quoted the message rate.

For tests using a file output node we added an MQ Output node after it so for each file processed we received an MQ Message to a queue. We then drained this queue using the performance harness tool and quoted the message rate. Also a compute node was added between the file and MQ Output Nodes to stop the message body being copied to keep the overhead of writing out the MQ Message to a minimum. Where the output node was writing many records to a single file the flow was modified to add an extra compute node before the output node. This compute node stored the message/record size of the first message and then used this to work out how many messages/records were allowed before the output file reached 1GB. The compute node incremented a counter for each message/record and once the counter indicated that 1GB of records had been written to the file a message was sent to the Finish File terminal to close the file. This meant for any test the output file did not exceed 1GB.

Where the test involved MQ to File processing we used the mqjava.Requestor module as per our normal point to point testing to drive the flow and measure results.

The same machine configuration was used as point to point testing i.e. one Client machine and one Server machine.

Reported Message Rates

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 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 Messages

This section details the types of input messages used in the report.

General Input Messages

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

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>
```

SOAP Input Message and WSDL

Below is the input message and WSDL used for the SOAP Nodes tests:

```
<soapenv:Envelope
              xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:tns="http://WssSale.miwssoap.broker.mqst.ibm.com"
              xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <soapenv:Header>
    <wsa:Action>SummerSale</wsa:Action>
    <wsa:MessageID>uuid:515704D6-0111-4000-E000-82267F000001</wsa:MessageID>
  </soapenv:Header>
  <soapenv:Body>
    <tns:SaleRequest>
      <SaleEnvelope>
        <Header>
          <SaleListCount>1</SaleListCount>
        </Header>
        <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>
      <Trailer>
        <CompletionTime>12.00.00</CompletionTime>
      </Trailer>
    </SaleEnvelope>
  </tns:SaleRequest>
 </soapenv:Body>
</soapenv:Envelope>


<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://WssSale.miwssoap.broker.mqst.ibm.com"
  xmlns:tns="http://WssSale.miwssoap.broker.mqst.ibm.com"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl">
  <wsdl:types>
    <xsd:schema
          targetNamespace="http://WssSale.miwssoap.broker.mqst.ibm.com"
          xmlns:tns="http://WssSale.miwssoap.broker.mqst.ibm.com"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="SaleRequest" type="tns:RootMessage"/>
      <xsd:element name="SaleResponse" type="tns:RootMessage"/>
      <xsd:complexType name="RootMessage">
        <xsd:sequence>
```

```
            <xsd:element name="SaleEnvelope">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="Header" type="tns:Header"/>
                  <xsd:element maxOccurs="unbounded" name="SaleList" type="tns:SaleList"/>
                  <xsd:element name="Trailer" type="tns:Trailer"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="SaleList">
          <xsd:sequence>
            <xsd:element maxOccurs="2" minOccurs="2" name="Invoice" type="tns:Invoice"/>
          </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="Invoice">
          <xsd:sequence>
            <xsd:element maxOccurs="2" minOccurs="2" name="Initial" type="xsd:string"/>
            <xsd:element name="Surname" type="xsd:string"/>
            <xsd:element maxOccurs="2" minOccurs="2" name="Item" type="tns:Item"/>
            <xsd:element name="Balance" type="xsd:float"/>
            <xsd:element name="Currency" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="Item">
          <xsd:sequence>
            <xsd:element maxOccurs="3" minOccurs="3" name="Code" type="xsd:string"/>
            <xsd:element name="Description" type="xsd:string"/>
            <xsd:element name="Category" type="xsd:string"/>
            <xsd:element name="Price" type="xsd:float"/>
            <xsd:element name="Quantity" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="Header">
          <xsd:sequence>
            <xsd:element name="SaleListCount" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="Trailer">
          <xsd:sequence>
            <xsd:element name="CompletionTime" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:schema>
    </wsdl:types>
    <wsdl:message name="SaleRequest">
      <wsdl:part element="tns:SaleRequest" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="SaleResponse">
      <wsdl:part element="tns:SaleResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:portType name="WssSale">
      <wsdl:operation name="Sale">
        <wsdl:input  message="tns:SaleRequest" name="SaleRequest"

wsaw:Action="http://WssSale.miwssoap.broker.mqst.ibm.com/WssSale/services/WssSale/SaleRequest"/>
        <wsdl:output message="tns:SaleResponse" name="SaleResponse"

wsaw:Action="http://WssSale.miwssoap.broker.mqst.ibm.com/WssSale/services/WssSale/SaleResponse"/>
```

```
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="WssSaleSoapBinding" type="tns:WssSale">
    <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="Sale">
      <wsdlsoap:operation soapAction="SummerSale"/>
      <wsdl:input name="SaleRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="SaleResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="WssSaleService">
    <wsdl:port binding="tns:WssSaleSoapBinding" name="WssSale">
      <wsdlsoap:address location="http://localhost:9081/WssSale/services/WssSale"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Output Message

For those tests that modified the message one of two message formats was used for the output messages dependent on the test case. These are the Compute and Transform messages.

Transformation Message

For the message 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

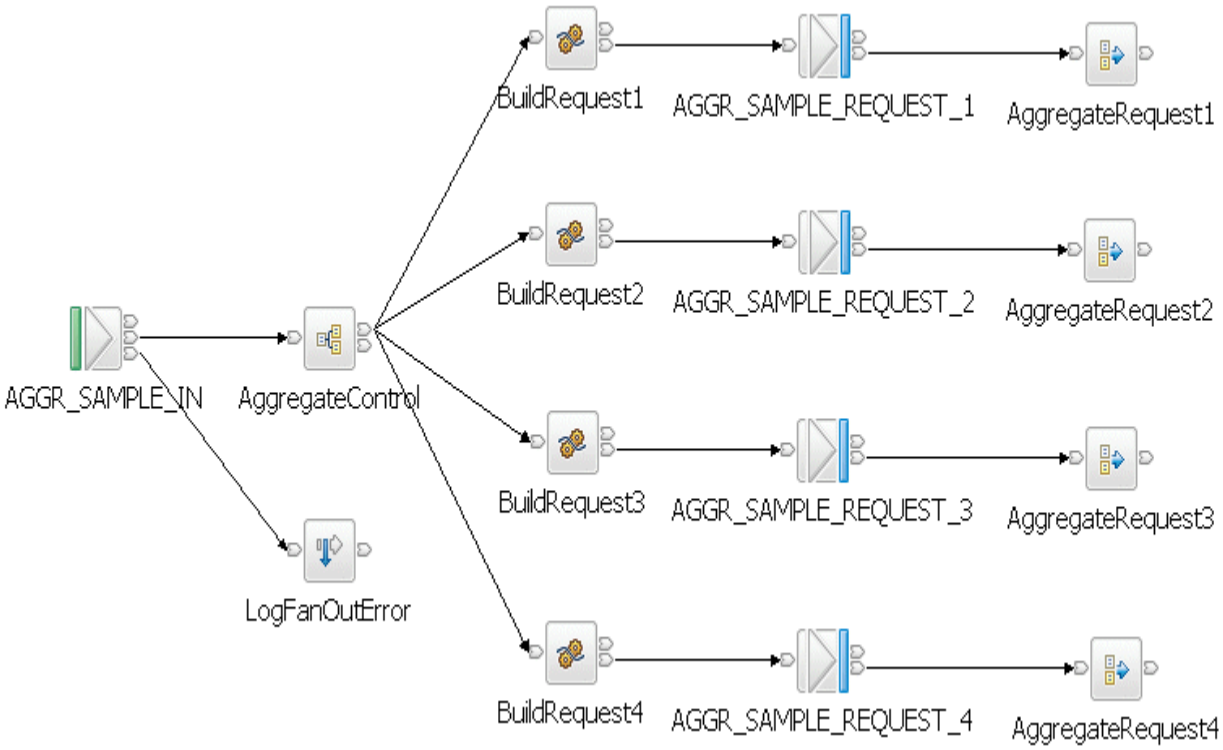
This section contains a description of the processing in each of the use cases which are used to characterise the performance of WebSphere Message Broker V7. All of these use cases are shipped as samples in WebSphere Message Broker V7. See the samples gallery for more information.

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

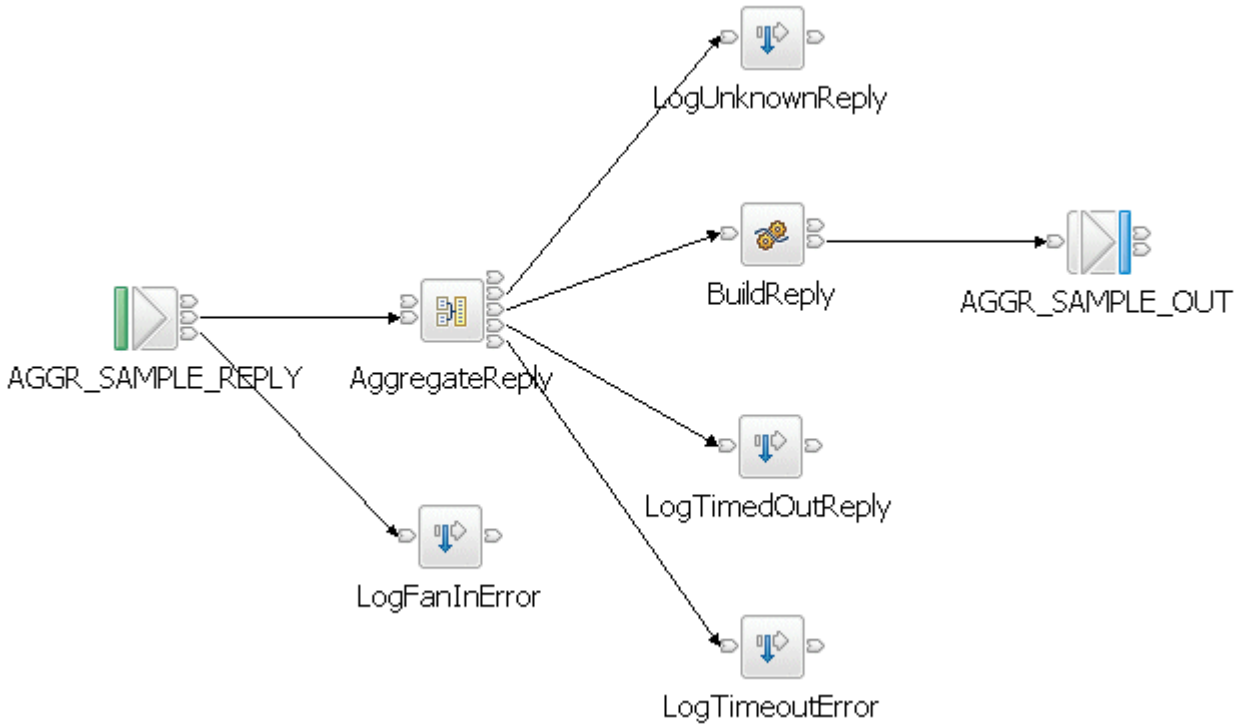
This is the flow that takes the incoming request message, generates four different request messages, sends them out on request/reply, and starts the tracking of the aggregation operation:



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

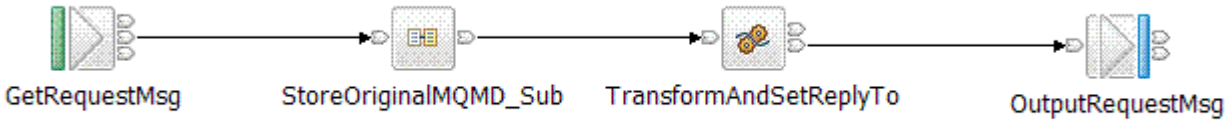
The coordinated request reply sample is based on the scenario of a contemporary and established application communicating through the use of WebSphere MQ messages in a request/reply processing pattern. The contemporary application uses self-defining XML messages and issues a request message. The established 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.

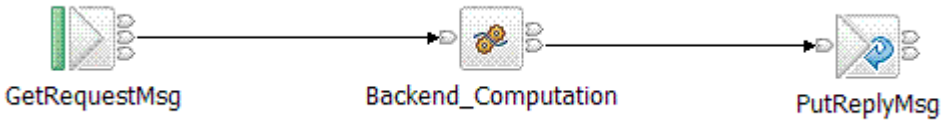
The Request message flow consists of the following nodes:



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.

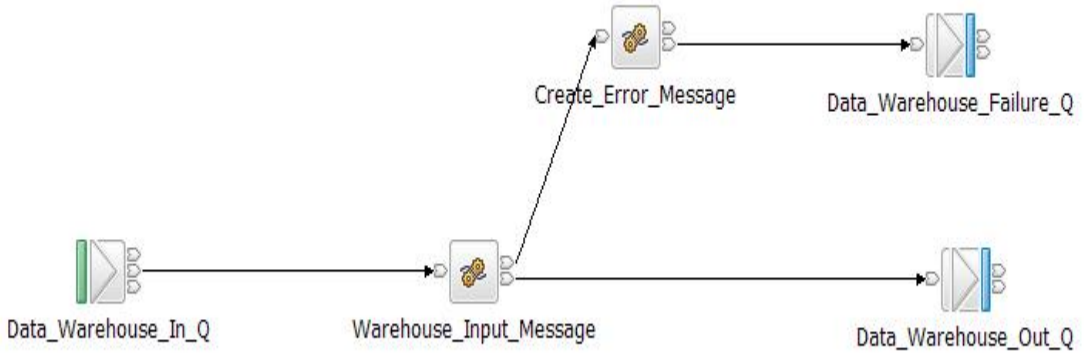
The data archiving is performed by the WarehouseData message flow. This is described below.

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 centre. On receipt at the IT centre 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.

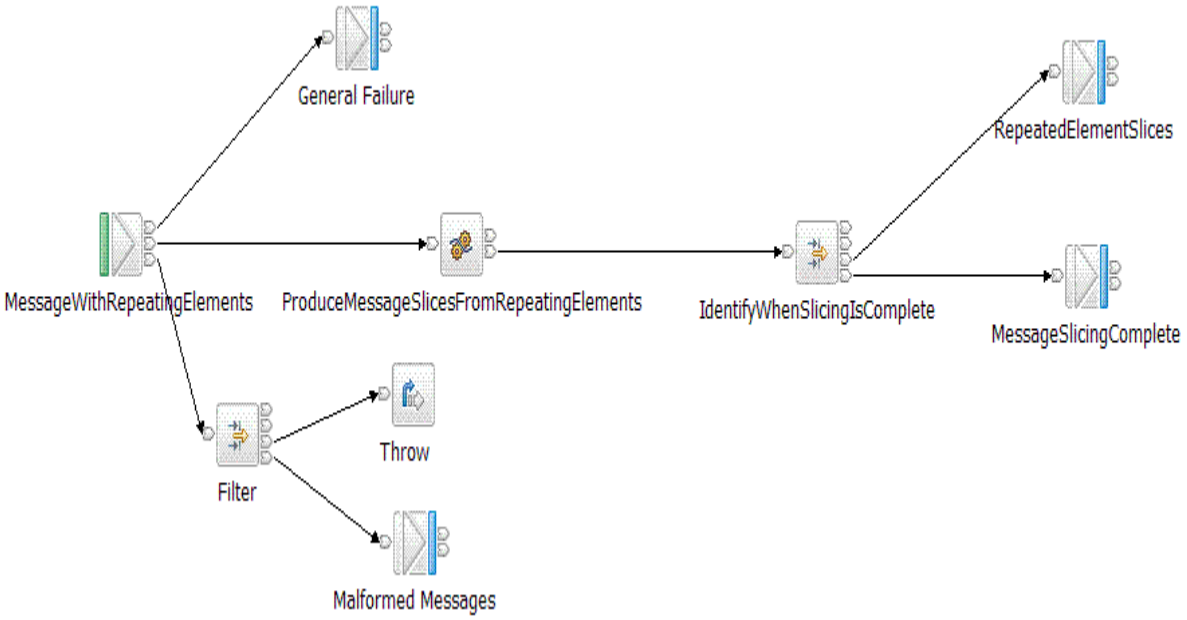
The processing in the sample consists of one message flow. The processing it performs is described below.

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 MQ queues.

The message routing sample shows how to implement a routing table, using shared variables, to route messages in a message flow. This test is using the routing table implemented using shared variables.

The processing in the message flows is described below:

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.

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.

Transformation using ESQL

The transformation using ESQL use case is based on processing of sales data. At the time of sale the customer name, the code for the product, a description of the product, its category, the unit price and quantity purchased are recorded. Each customer may purchase several items.

Subsequently a statement is produced for each customer and it is the production of the statement that is performed in this use case. The processing results in a restructuring of the original message.

The messages used (input and output) are self-defining XML messages. Each message with sales data consists of three parts:

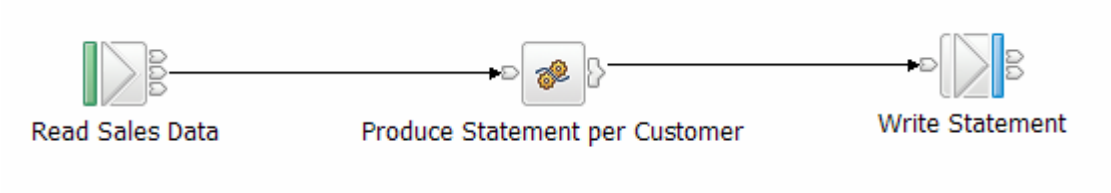
- A header containing a count of the number of repetitions of the repeating SaleList structure that follows.
- The body that contains the repetitions of the repeating SaleList structure.
- The trailer that contains the time the message was processed.

The production of the statement for each customer within a SaleList is achieved with a single message flow, the Transformation with ESQL Message Flow.

Transformation with ESQL Message Flow

The message flow performs the following processing:

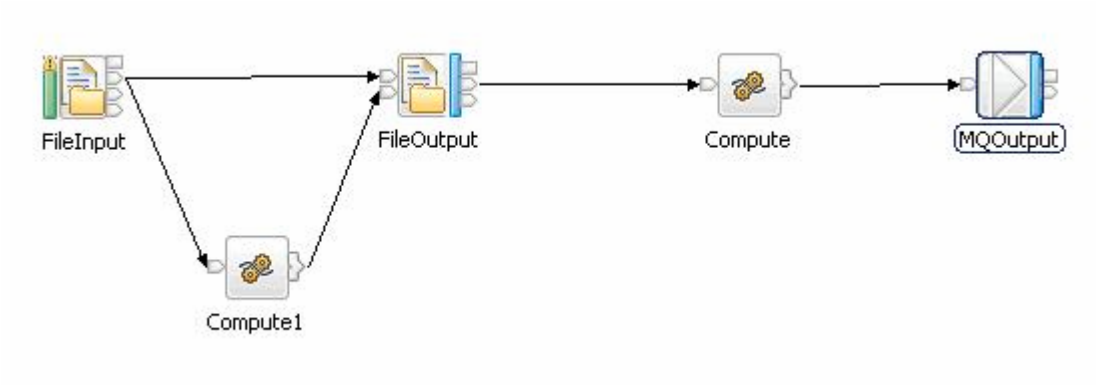
1. Reads a WebSphere MQ message containing an XML payload under transactional control.
2. The input message is parsed and an invoice produced for each customer. This is achieved with a single Compute node containing ESQL.
3. Produces a WebSphere MQ output message containing an XML payload under transactional control.



File to File

Reading and Writing Large Files with XML Parsed Record Sequence Records

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



For this test we had multiple large input files (each was ~100MB) in the input folder. Each file contained records of the appropriate size. For example for the 4K test the 100MB file contained 256000 records. Each record was a valid XML document.

The output file was closed when it reached 1GB of data regardless of record size. This was achieved by using the first compute node to count the number of records read and to check the record size, when 1GB of data was written it sent a message to the Finish File terminal of the File Output Node.

Message Domain is set to XMLNSC. The File Input Node is configured to set the Record Detection to Parsed Record Sequence using the XMLNSC parser. Input files are deleted once processed. The File Output Node is configured to set the Record Definition to unmodified. For every record written an MQ message is propagated to the Compute node, the headers and body are not copied and an empty message is sent to the MQ Output node. The queue is drained and the rate is reported. This rate represents the number of records per second written to the output file.

This test identifies the cost of reading records in from files using the XMLNSC parser to delimit the records and writing records to another large file.

Appendix E – Tuning

This appendix describes the tuning that was applied to WebSphere Message Broker, WebSphere MQ and DB2.

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. The broker ran as a trusted WebSphere MQ application. This was achieved by use of the '-t' flag on broker creation (with the `mqsicreatebroker` command) and by ensuring that the environment variable MQ_CONNECT_TYPE=FASTPATH was present in the environment in which the broker was started. NOTE: The reader should be aware that there is a potential integrity exposure to the Message Broker queue manager as the level of isolation between the Message Broker and queue is reduced. This is where the improved performance comes from.

2. 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 for SOAP and HTTP Tests:

- The clients sending data to the broker were configured to use persistent HTTP connections i.e. MaxKeepAlives was set to 0

- For HTTP and SOAP Request Node tests the SocketConnectionManager was set to use persistent connections by setting MaxKeepAlives to 0.

To set these values consult the documentation for the mqsichangeproperties command.

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 and DefaultPQBufferSize was increased to a value of 50MB for the input and output queues used in the tests. This value was used because in many cases test messages of up to 20MB were used. When using smaller messages all of the time, a smaller value is likely to be more appropriate.

2. Given the use of persistent messages in the tests the following MQ log parameters were modified:
   - LogBufferPages was set to 4096
   - LogFilePages was set to 65535
   - LogType was set to circular
   - LogPrimaryFiles was set to 15
   - LogSecondaryFiles was set to 1

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

4. The following values were set for the TCP stanza in the queue manager ini file:
   - SndBuffSize=70000
   - RcvBuffSize=70000
   - RcvSndBuffSize=70000
   - RcvRcvBuffSize=70000
   - Blocking=YES

5. 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.

6. The WebSphere MQ queue manager log was located on SAN with a non-volatile fast write cache used for the disk on which the log was located. 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

For further information on MQ tuning see this article:

http://www.ibm.com/developerworks/websphere/library/techarticles/0712_dunn/0712_dunn.html

TCP/IP

No specific tuning was performed for TCP/IP

Database

The database used in the performance tests was modified from the default in the following way:

1. The TCP/IP loopback adapter was used for the database.

2. The database data and log files were placed on a dedicated file system that was located on a SAN with fast write non volatile cache. This was done to minimise I/O times and improve the capacity of the log.

3. The database was modified using these commands:
    - db2 update db cfg for userdb using logprimary 10
    - db2 update db cfg for userdb using logfilsiz 250000
    - db2 update db cfg for userdb using logbufsz 4096

Additional Tuning Information

In order to obtain the maximum message rate for your implementation it is important that you understand the 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.

There is also a Support Pac, IP04, which covers the main design decisions when building message flows. It is available at
http://www.ibm.com/support/docview.wss?uid=swg24006518