WebSphere Message Broker V6.1

For Solaris on Opteron

Performance report

Version 1.0

May, 2008

Tim Dunn

Rich Bicheno

Ian Hurworth

Adam Rice

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 2008.
This edition applies to *WebSphere Message Broker V6.1 for Solaris on Opteron* and to all subsequent releases and modifications until otherwise indicated in new editions.

Notices

This report is intended for Architects, Systems Programmers, Analysts and Programmers wanting to understand the performance characteristics of WebSphere Message Broker V6.1 for Solaris on Opteron. The information is not intended as the specification of any programming interfaces that are provided by WebSphere MQ or WebSphere Message Broker V6.1 for Solaris on Opteron. It is assumed that the reader is familiar with the concepts and operation of WebSphere Message Broker V6.1.

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.

Summary of Amendments

Date	Changes
09/05/2008	Initial Release

Table of Contents

Introduction

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

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

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

    a. The areas of improvement in performance with WebSphere Message Broker V6.1 when compared with WebSphere Message Broker V6.

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

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

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

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

1. Where applicable the numbers contained in this report are run using 64bit Execution Groups which is the default for V6.1. Note this does not apply to Windows, Linux(Intel) and z/OS.

2. 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.

3. More extensive analysis of product function, including more coverage of different transports.

4. Larger range of message sizes.

5. Where XML Processing is tested the XMLNSC parser is used, this parser has been enhanced in V6.1 and hence we would recommend all users to use or migrate to this parser when processing XML messages.

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 Summary of Tuning Information.

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

Part I

This part contains an overview of the areas of improvement in performance with WebSphere Message Broker V6.1 when compared with WebSphere Message Broker V6.

It contains the following sections:

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

- *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

Improving areas of Message Broker runtime performance has been a specific focus with WebSphere Message Broker V6.1 and as a result there are several improvements in the level of performance when compared with WebSphere Message Broker V6. The improvements come from two sources - updates to existing function and provision of new function.

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

XML Processing

The parser for the XMLNSC domain has been rewritten for WMB V6.1 and offers significant improvements in parsing performance. Any further XML processing improvements are likely to be limited to XMLNSC hence we would recommend all users to use or migrate to this as the XML parser of choice. Below is an illustration to demonstrate the improvements in XML Parsing costs in the XMLNSC domain compared with the XML domain. The flow for this test consisted of a full parse of the incoming message by setting the parameter on the MQ input node to parse immediate. The full message was then propagated out as an MQ Message, no other work was performed. A single copy of the message flow was used.

Msg Size	XML Message Rate	XMLNSC Message Rate	% Improvement
1K	1408.42	3178.81	125.70
4K	1026.28	2088.08	103.46
64K	165.26	203.90	23.38
256K	43.79	51.54	17.70
1024K	11.55	13.68	18.48

The results above are for a message flow whose only real cost is XML parsing and serialisation, hence the extent to which general message flows will benefit will be dependent on the proportion of XML parsing in the flow in relation to other types of processing.

Impact of Trace nodes in Message flows.

The overhead of leaving trace nodes in message flows even when disabled has been significantly reduced in V6.1. Trace nodes can now be left in flows with minimal impact. See the Overheads section of this report for more details.

The improvements in these areas can be obtained by upgrading to WebSphere Message Broker V6.1. **No code or message model changes are required to benefit from the improvements**.

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

XMLNSC Opaque Parsing

In Message Broker V6.1 Opaque parsing has been fully implemented for the XMLNSC domain and integrated into the Toolkit. If you are designing a message flow and you know that certain elements in a message are never referenced by the message flow, you can specify at development time that those elements should be parsed opaquely. This reduces the costs of parsing and writing the message and so helps to improve performance. For guidance on how to use this feature consult the product documentation. The following table shows some of the gains possible comparing message rates per second for a filter test case with and without Opaque Parsing. This test routes a message based on the value found in the last element of the body, only a single copy of the flow was used:

Msg Size	FilterLast XMLNSC	FilterLast Opaque	% Improvement
4K	1872.98	3040.28	62
16K	708.89	1447.55	104
64K	192.21	398.60	107
256K	47.41	107.40	127
1024K	12.07	27.87	131
4096K	3.03	6.43	112
8192K	1.48	3.19	116

XMLNSC Validation

The XMLNSC parser now offers high-performance, standards-compliant XML Schema validation at any point in a message flow. This offers much faster XML validation compared to MRM XML validation. The following table gives an illustration of the relative performance of MRM versus XMLNSC validation when the message is validated using an MQ Input Node, the numbers quoted are messages per second:

Msg Size	MRM Validation	XMLNSC Validation	% Improvement
4K	200.79	1589.57	692
16K	57.58	562.57	877
64K	14.32	132.01	822
256K	3.54	32.60	820
1024K	0.90	8.71	868
4096K	0.22	2.16	880

The Table below shows the results of running a series of use cases in WebSphere Message Broker V6.1. The use cases are briefly described at the end of this section and more fully in Appendix D – Use Case Descriptions. The use cases are largely taken from the samples gallery of WebSphere Message Broker V6.1. The table is intended to give you an idea of the processing capabilities for a range of common scenarios.

Use Case	**Message Size**	**V6.1 Msgs/sec**
Aggregation	4K	415.10
Coordinated Request/Reply	4K	757.19
Large Messaging	10K	788.86
Message Routing	4K	5469.32
SWIFT Message Parse	1K	2541.81
XML Transformation	4K	1837.17
XMLNSC Validation	1K	3816.53
Soap Nodes	1K	632.53
Simple DB Routing	1K	1562.87

Throughput Comparison for Use Cases.

Note:

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

Use Case Outline

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

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

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

- **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.

- **SWIFT Message Parse**
  This demonstrates the use of WebSphere Message Broker to read and parse a SWIFT MT103 message for subsequent processing.

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

- **XMLNSC Validation**
  The XMLNSC Validation sample demonstrates how WebSphere Message Broker's XMLNSC parser can validate XML messages against a schema.

- **SOAP Nodes**
  The SOAP Nodes sample demonstrates the use of the SOAP Input, SOAP Reply and SOAP Request nodes to both provide and consume a simple Web Service.

- **Simplified DB Routing**
  The Simplified Database Routing sample demonstrates how data in a database can be retrieved through a JDBC connection and used to dynamically update message content or be used in the decision on where to route the message.

Additional Information

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

The Web Resources section in the development toolkit of WebSphere Message Broker V6.1 contains links to many additional pieces of information on topics such as Education, Technical Resources and SupportPacs. The Web resources section can be accessed by selecting `Web Resources` from the Help drop down on the development toolkit menu bar.

For additional suggestions consider the following:

- See the announcement letters for
    - IBM WebSphere Message Broker V6.1 which is available at http://www.ibm.com/software/integration/wbimessagebroker/V6
    - IBM WebSphere Message Broker V6.1 for z/OS which is available at http://www.ibm.com/software/integration/wbimessagebroker/V6/zos.html
- IBM WebSphere MQ SupportPacs provide you with a wide range of downloadable code and documentation that complements the WebSphere MQ family of products. Additional performance reports are also available. These are available at http://www.ibm.com/software/integration/support/supportpacs.
- For more information about WebSphere Message Broker V6.1, go to the WebSphere Message Broker Web site. Product documentation is also available. This is available at http:/www.ibm.com/software/integration/wbimessagebroker.
- For more information about WebSphere MQ V6, go to the WebSphere MQ Web site. Product documentation is also available. This is available at http://www.ibm.com/software/integration/wmq .
- For more information about business integration software from IBM go to WebSphere Business Integration Web site. This is available at http://www.ibm.com/software/info1/websphere/index.jsp?tab=products/businessint .
- Get the latest WebSphere Message Broker technical resources at the WebSphere 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 version used was the current development version due to be released later this year. 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.

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)
Rich Bicheno (rich_bicheno@uk.ibm.com)
Ian Hurworth (hurworti@uk.ibm.com)

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

Part II

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

It contains the following sections:

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

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

- *Tuning* which describes the changes made to the default settings for WebSphere Message Broker V6.1 and WebSphere MQ in order to obtain the results detailed in this report.

Measurements for persistent MQ messages will be added to this report at a later date.

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
4K	3K	1K
16K	8K	4K
64K	32K	16K
256K	125K	61K
1024K	498K	241K
4096K	1992K	962K
8192K	3981K	1923K

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
4K	No			
16K	No			
64K	No			
256K	No			
1024K	No			
4096K	No			
8192K	No			

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 4K, 16K, 64K, 256K, 1mb, 4mb and 8mb. 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 is obtained using the calculation:

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

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

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

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

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
1k	No	3772.59	24.00	0.51

Message Parsing and Writing

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

Parsing a Message in the MRM Domain

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

Notes: In these reports we are only quoting numbers for TDS Fixed length format. As can be seen from the WMB V6 performance reports message throughput varied little regardless of format hence we have only quoted this one. If more detail on the differences between the different TDS formats is required consult the V6 performance report.

In addition the performance data for MRM XML has been removed. In WMB V6.1 the XMLNSC parser can now perform XML validation, removing any dependency on MRM XML. Although the MRM XML parser remains in place for compatibility it is deprecated. The XMLNSC is the parser of choice for XML messages. Accordingly only performance data for this XML parser is presented in this report.

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
4k	No	573.22	15.00	2.09
16k	No	198.40	14.00	5.65
64k	No	56.02	14.00	19.99
256k	No	14.72	14.00	76.09
1024k	No	3.67	14.00	305.18
4096k	No	0.90	14.00	1244.44
8192k	No	0.47	14.00	2382.98

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
4k	No	748.03	15.00	1.60
16k	No	199.96	14.00	5.60
64k	No	52.18	13.00	19.93
256k	No	13.46	13.00	77.29
1024k	No	3.63	13.00	286.50
4096k	No	0.90	13.00	1155.56
8192k	No	0.43	13.00	2418.60

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	445.22	14.00	2.52

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	81.86	13.00	12.71

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	30.30	14.00	36.96

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	47.59	14.00	23.54

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	16.91	14.00	66.23

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
4k	No	395.84	14.33	2.90
16k	No	127.91	14.00	8.76
64k	No	34.53	14.00	32.43
256k	No	8.73	14.00	128.29
1024k	No	2.24	14.00	499.26
4096k	No	0.57	14.00	1964.91
8192k	No	0.27	14.00	4148.15

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
4k	No	526.45	14.33	2.18
16k	No	160.96	14.00	6.96
64k	No	42.74	14.00	26.20
256k	No	11.10	13.67	98.50
1024k	No	2.80	13.00	371.43
4096k	No	0.70	13.67	1561.90
8192k	No	0.34	13.33	3106.80

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 a 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
4k	No	1790.54	19.67	0.88
16k	No	738.21	16.00	1.73
64k	No	199.73	15.00	6.01
256k	No	49.95	14.67	23.49
1024k	No	12.72	14.00	88.05
4096k	No	3.19	14.00	351.10
8192k	No	1.59	15.00	754.72

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
4k	No	1201.62	18.67	1.24
16k	No	414.71	14.00	2.70
64k	No	109.40	14.00	10.24
256k	No	25.14	13.00	41.37
1024k	No	6.66	13.33	160.24
4096k	No	1.61	13.67	679.09
8192k	No	0.80	14.00	1400.00

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
4k	No	1589.57	19.67	0.99
16k	No	562.57	16.00	2.28
64k	No	132.01	15.00	9.09
256k	No	32.60	15.00	36.81
1024k	No	8.71	15.00	137.77
4096k	No	2.16	15.00	556.41
8192k	No	1.07	15.00	1121.50

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
4k	No	1068.24	17.00	1.27
16k	No	329.89	14.00	3.40
64k	No	79.02	13.67	13.84
256k	No	20.30	13.67	53.85
1024k	No	5.37	14.00	208.70
4096k	No	1.30	13.00	800.00
8192k	No	0.64	13.33	1658.03

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
4k	No	730.09	16.67	1.83
16k	No	218.94	13.67	4.99
64k	No	56.48	13.33	18.89
256k	No	14.57	14.00	76.89
1024k	No	3.82	13.33	279.23
4096k	No	0.96	13.33	1114.98
8192k	No	0.46	13.00	2277.37

Opaque Parsing in the XMLNSC Domain

The test in this section illustrates the CPU processing costs of using the XMLNSC domain to 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
4k	No	3040.28	23.33	0.61
16k	No	1447.55	18.00	0.99
64k	No	398.60	17.00	3.41
256k	No	107.40	15.67	11.67
1024k	No	27.87	16.00	45.93
4096k	No	6.43	15.33	190.67
8192k	No	3.19	16.00	401.25

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
4k	No	1432.09	18.00	1.01
16k	No	1300.67	17.00	1.05
64k	No	1005.97	20.67	1.64
256k	No	421.11	24.33	4.62
1024k	No	110.92	17.00	12.26
4096k	No	27.05	15.67	46.34
8192k	No	13.62	16.00	93.98

Updating a row in a Database

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

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

This test identifies the cost of performing a Database UPDATE.

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	1358.32	18.00	1.06
16k	No	1202.19	17.00	1.13
64k	No	948.00	20.33	1.72
256k	No	386.73	20.33	4.21
1024k	No	108.40	16.00	11.81
4096k	No	26.44	15.67	47.40
8192k	No	13.27	15.33	92.46

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. A call is 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.

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	2880.267	21.000	0.58

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
1k	No	6.167	15.667	0.102

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 of the other 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
4k	No	4647.59	26.00	0.45
16k	No	3159.17	25.00	0.63
64k	No	1024.61	21.00	1.64
256k	No	283.52	19.33	5.46
1024k	No	69.08	16.33	18.92
4096k	No	17.47	18.00	82.44
8192k	No	8.55	15.67	146.53

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
4k	No	2605.82	36.67	1.13
16k	No	1831.99	41.33	1.80
64k	No	790.12	50.00	5.06
256k	No	111.15	29.00	20.87
1024k	No	14.27	14.67	82.24

Making an HTTP Request

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

This test shows the overhead of making a HTTP Request from within a message flow running in Message Broker. An MQ Message is written to the input queue. The contents of this message are 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. Note that as the request invokes another flow on the same broker the cost detailed also includes the processing cost of the HTTP Input -> HTTP Reply message flow.

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	801.60	19.67	1.96
16k	No	506.00	17.00	2.69
64k	No	194.22	17.33	7.14
256k	No	56.17	17.00	24.21
1024k	No	14.68	17.00	92.66

File Nodes

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

Note for File Output Node tests it was necessary to insert an MQ Output node after the File Output Node and we measured the rate of messages being drained from the MQ Output node. This was necessary to integrate the test with our automation infrastructure and to avoid 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 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.

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	453.04	27.00	4.77
4k	No	370.13	26.33	5.69
16k	No	211.73	26.67	10.08
64k	No	79.11	26.67	26.97
256k	No	23.01	26.67	92.71
1024k	No	6.06	26.33	347.83
4096k	No	1.53	26.33	1373.91
8192k	No	0.67	26.33	3144.28

Reading and Writing Large Files with Delimited 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 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 rate represents the number of records per second written to the output file.

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.

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	561.76	27.00	3.85
4k	No	549.52	26.67	3.88
16k	No	481.79	27.00	4.48
64k	No	315.87	28.00	7.09
256k	No	113.04	25.67	18.16
1024k	No	32.04	25.00	62.42
4096k	No	5.75	27.00	375.43
8192k	No	2.31	27.33	946.61

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.

Message Domain is set to 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
1k	No	522.65	11.67	1.79
4k	No	383.60	9.67	2.02
16k	No	558.75	16.33	2.34
64k	No	270.85	14.67	4.33
256k	No	73.00	16.67	18.26
1024k	No	24.18	24.67	81.62
4096k	No	5.18	23.67	365.51
8192k	No	3.06	26.00	680.48

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.

Message Domain is set to 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
1k	No	1098.32	15.67	1.14
4k	No	977.11	16.33	1.34
16k	No	821.28	17.33	1.69
64k	No	518.74	20.67	3.19
256k	No	172.55	20.33	9.43
1024k	No	72.22	27.33	30.28
4096k	No	19.44	28.00	115.21
8192k	No	9.19	31.67	275.66

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.

Message Domain is 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
1k	No	1120.81	16.00	1.14
4k	No	1025.25	16.33	1.27
16k	No	742.32	17.00	1.83
64k	No	479.31	19.67	3.28
256k	No	188.88	21.67	9.18
1024k	No	77.09	28.33	29.40
4096k	No	19.73	27.67	112.16
8192k	No	9.28	31.67	273.09

File to MQ

Reading Whole Files and Writing them 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 Whole File. 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 whole files and writing them 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
1k	No	1004.08	28.20	2.25
4k	No	635.24	30.50	3.84
64k	No	456.74	33.83	5.93
256k	No	225.27	53.60	19.03
1024k	No	58.64	66.80	91.13
4096k	No	19.73	70.80	287.12
8192k	No	9.20	66.50	578.32

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.

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 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
1k	No	1225.40	27.33	1.78
4k	No	744.04	27.00	2.90
16k	No	274.65	26.33	7.67
64k	No	95.26	26.33	22.12
256k	No	26.17	27.67	84.59
1024k	No	6.78	28.33	334.48
4096k	No	1.70	27.67	1301.96
8192k	No	0.68	27.33	3215.69

Reading Large Files with Delimited 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.

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. 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 line end to delimit the records and writing the unmodified records out 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
4k	No	1834.76	29.00	1.26
16k	No	1463.07	30.33	1.66
64k	No	670.39	32.00	3.82
256k	No	204.38	59.67	23.36
1024k	No	48.90	61.00	99.80
4096k	No	13.99	74.33	425.17
8192k	No	6.45	67.00	831.44

Reading Large Files with Custom Delimited 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.

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. 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 a custom delimiter to delimit the records and writing the unmodified records out 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
4k	No	1953.09	29.33	1.20
16k	No	1390.85	30.33	1.74
64k	No	726.95	33.67	3.70
256k	No	208.94	63.33	24.25
1024k	No	48.44	62.33	102.94
4096k	No	11.67	60.67	416.00
8192k	No	6.92	70.33	812.71

WebServices (SOAP Nodes)

The tests in this section illustrate the processing cost of receiving, sending and making requests over the SOAP transport. The SOAP nodes are new for WMB V6.1 and make development of SOAP over HTTP flows much easier. The SOAP nodes handle the complexities of the SOAP format but this is at the cost of a runtime performance overhead.

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
1k	No	610.94	14.67	1.92
4k	No	358.78	14.00	3.12
8k	No	241.94	14.00	4.63
16k	No	148.97	13.00	6.98
64k	No	38.64	12.00	24.85
256k	No	9.63	13.00	107.96
1024k	No	1.58	14.00	708.86

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 used in a message flow which acts as a Web service provider. 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 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.

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	302.29	8.00	2.12
4k	No	280.17	8.67	2.47
8k	No	219.79	11.00	4.00
16k	No	139.10	13.33	7.67
64k	No	27.60	12.00	34.78
256k	No	5.48	10.00	145.99
1024k	No	1.03	11.00	854.37

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 service 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 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
1k	No	386.80	15.33	3.17
4k	No	255.10	13.00	4.08
8k	No	166.32	14.00	6.73
16k	No	100.99	14.00	11.09
64k	No	27.44	13.00	37.91
256k	No	6.82	14.00	164.14
1024k	No	1.01	18.33	1447.37

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 acts as a Web service provider. A SOAP message with an attachment is received by the broker via the SOAP Input Node, a Compute Node then copies the SOAP request message and the SOAP attachment across to a SOAP response message which is then sent via the SOAP Reply Node. The size of the SOAP message body remains constant (1K) 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.

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

Attachment Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1k	No	380.24	14.80	3.11
4k	No	365.93	15.00	3.28
8k	No	342.64	15.00	3.50
16k	No	306.49	15.00	3.92
64k	No	201.56	16.00	6.35
256k	No	57.14	12.00	16.80
1024k	No	3.85	4.00	83.03

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) 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 complete 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
1k	No	353.33	17.00	3.85
4k	No	264.06	15.00	4.54
8k	No	182.91	15.00	6.56
16k	No	118.73	15.00	10.11
64k	No	33.80	15.00	35.50
256k	No	8.48	13.00	122.64
1024k	No	1.42	14.33	805.62

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 to call 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 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.

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 with validation 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
1k	No	260.57	12.00	3.68
4k	No	154.33	15.67	8.12
8k	No	108.27	15.00	11.08
16k	No	67.80	14.00	16.52
64k	No	18.04	13.00	57.64
256k	No	3.93	11.00	223.73
1024k	No	0.81	13.00	1283.95

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 was the WMQ Queue Manager on which the broker was running.

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
4k	No	860.76	17.67	1.64
16k	No	639.60	16.00	2.00
64k	No	313.02	16.00	4.09
256k	No	104.88	17.00	12.97
1024k	No	27.24	17.00	49.92
4096k	No	6.78	17.00	200.69
8192k	No	2.92	18.33	501.71

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 was the WMQ Queue Manager on which the broker was running.

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
4k	No	952.32	18.33	1.54
16k	No	717.38	17.00	1.90
64k	No	414.18	19.33	3.73
256k	No	155.19	21.00	10.83
1024k	No	44.44	21.00	37.80
4096k	No	10.58	21.00	158.84
8192k	No	4.25	23.00	433.28

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 was the WMQ Queue Manager on which the broker was running.

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
4k	No	1542.59	19.67	1.02
16k	No	1155.83	18.00	1.25
64k	No	590.91	21.00	2.84
256k	No	220.37	24.67	8.95
1024k	No	61.80	25.00	32.36
4096k	No	15.84	24.67	124.55
8192k	No	7.86	26.00	264.74

Routing and Transformation Logic

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

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

Using ESQL

The tests in this section illustrate the processing costs of using ESQL for different routing and transformation operations. 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
4k	No	3699.08	26.33	0.57
16k	No	2856.21	26.00	0.73
64k	No	757.11	18.00	1.90
256k	No	292.48	21.00	5.74
1024k	No	70.97	16.67	18.79
4096k	No	17.44	18.00	82.55
8192k	No	8.67	17.67	163.08

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
4k	No	1872.98	19.67	0.84
16k	No	708.89	16.00	1.81
64k	No	192.21	15.00	6.24
256k	No	47.41	15.00	25.31
1024k	No	12.07	15.00	99.42
4096k	No	3.03	15.00	396.04
8192k	No	1.48	15.67	846.85

Computation on an Input Message using the XMLNSC Parser

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

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

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

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	653.27	16.67	2.04
16k	No	191.41	14.00	5.85
64k	No	48.75	14.00	22.97
256k	No	11.84	14.00	94.57
1024k	No	2.30	14.00	486.96

Manipulation of an Input Message using the XMLNSC Parser

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

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

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

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	635.82	16.00	2.01
16k	No	187.87	13.00	5.54
64k	No	46.18	13.33	23.10
256k	No	9.04	14.00	123.85
1024k	No	1.00	13.67	1093.33

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
4k	No	2710.25	23.67	0.70
16k	No	2125.39	24.00	0.90
64k	No	1135.52	27.67	1.95
256k	No	292.55	22.33	6.11
1024k	No	72.91	19.00	20.85
4096k	No	17.53	18.00	82.14
8192k	No	8.74	18.00	164.70

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
4k	No	1610.66	19.33	0.96
16k	No	601.99	15.67	2.08
64k	No	160.40	14.00	6.98
256k	No	41.47	14.00	27.01
1024k	No	10.79	14.67	108.78
4096k	No	2.69	14.00	416.87
8192k	No	1.33	14.00	842.11

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
4k	No	2940.25	24.67	0.67
16k	No	2288.41	23.33	0.82
64k	No	977.43	22.67	1.86
256k	No	301.42	23.00	6.10
1024k	No	73.16	19.00	20.78
4096k	No	17.47	18.33	83.97
8192k	No	8.73	18.00	164.89

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
4k	No	1624.58	19.00	0.94
16k	No	623.65	15.67	2.01
64k	No	159.68	14.00	7.01
256k	No	41.96	14.67	27.97
1024k	No	10.74	14.67	109.22
4096k	No	2.63	14.33	435.99
8192k	No	1.29	14.00	868.22

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

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

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

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

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	440.08	16.00	2.91
16k	No	130.68	13.00	7.96
64k	No	34.01	13.33	31.36
256k	No	8.68	14.00	129.03
1024k	No	2.18	13.67	501.53
4096k	No	0.53	13.67	2062.89
8192k	No	0.27	14.00	4148.15

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

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

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

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

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	265.43	15.67	4.72
16k	No	73.37	14.00	15.27
64k	No	18.69	14.00	59.93
256k	No	4.71	14.00	237.79
1024k	No	1.20	14.00	933.33
4096k	No	0.30	14.00	3733.33
8192k	No	0.13	14.00	8615.38

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

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

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

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

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	554.85	16.67	2.40
16k	No	153.33	14.00	7.30
64k	No	39.73	14.00	28.19
256k	No	10.02	14.00	111.78
1024k	No	2.53	14.00	442.69
4096k	No	0.62	14.00	1806.45
8192k	No	0.30	14.00	3733.33

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
4k	No	1261.42	17.67	1.12
16k	No	887.56	17.67	1.59
64k	No	701.59	23.00	2.62
256k	No	301.51	25.33	6.72
1024k	No	71.44	18.33	20.53
4096k	No	17.46	17.33	79.43
8192k	No	8.62	17.67	163.90

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
4k	No	2302.84	20.33	0.71
16k	No	2092.38	22.00	0.84
64k	No	782.76	23.00	2.35
256k	No	296.99	20.00	5.39
1024k	No	70.45	16.67	18.93
4096k	No	17.42	17.67	81.12
8192k	No	8.59	17.33	161.43

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.

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	284.78	12.33	3.46
16k	No	247.88	18.33	5.92
64k	No	87.33	16.00	14.66
256k	No	25.45	15.67	49.24
1024k	No	6.26	15.33	196.06
4096k	No	1.50	16.00	853.33
8192k	No	0.78	16.33	1682.40

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 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. 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.

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	406.74	21.00	4.13
16k	No	155.50	12.00	6.17
64k	No	82.27	17.33	16.86
256k	No	22.79	17.33	60.85
1024k	No	6.57	17.00	207.11
4096k	No	1.57	17.67	902.13
8192k	No	0.77	18.00	1878.26

Using XSLT

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

Computation on an Input Message

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

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

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

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	323.68	14.00	3.46
16k	No	143.89	14.33	7.97
64k	No	44.59	14.00	25.12
256k	No	11.63	14.00	96.28
1024k	No	3.60	16.00	355.56
4096k	No	0.82	16.00	1560.98
8192k	No	0.38	16.33	3438.60

Manipulation of an Input Message

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

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

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

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

Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
4k	No	386.57	16.33	3.38
16k	No	177.48	14.00	6.31
64k	No	59.71	14.00	18.76
256k	No	16.12	14.00	69.46
1024k	No	5.00	16.00	256.00
4096k	No	1.09	17.00	1247.71
8192k	No	0.50	16.67	2666.67

Publish Subscribe

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

Topic Based Publish/Subscribe using MQ Messages with 1 Subscriber

This test consists of MQInput node -> Publication node.

A publisher publishes a message on a single topic. The test is run with a single subscriber. The subscriber is registered to receive messages on the single topic. The subscribers queue is drained and the message rate reported, this rate represents the total number of messages per second delivered to all subscribers, this does not include published messages.

This test identifies the cost of using the Publication node for a single publisher and a single subscriber, single topic and a single copy of the message flow when using 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
4k	No	796.65	10.33	1.04
16k	No	677.29	11.33	1.34
64k	No	270.06	8.67	2.57
256k	No	104.82	8.67	6.61
1024k	No	28.01	7.00	19.99
4096k	No	6.98	7.33	84.09
8192k	No	3.45	7.00	162.48

Topic Based Publish/Subscribe using MQ Messages with 10 Subscribers

This test consists of MQInput node -> Publication node.

A publisher publishes a message on a single topic. The test is run with 10 subscribers. The subscribers are registered to receive messages on the single topic and messages for all subscribers are stored on the same MQ Queue. The subscribers queue is drained and the message rate reported, this rate represents the total number of messages per second delivered to all subscribers, this does not include published messages.

This test identifies the cost of using the Publication node for a single publisher and 10 subscribers, single topic and a single copy of the message flow when using 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
4k	No	4533.49	20.00	0.35
16k	No	3043.10	17.67	0.46
64k	No	1108.19	16.33	1.18
256k	No	262.93	20.67	6.29
1024k	No	79.72	23.00	23.08
4096k	No	21.59	27.33	101.28
8192k	No	10.63	27.67	208.15

Topic based Publish/Subscribe using MQ Messages with 100 Subscribers

This test consists of MQInput node -> Publication node.

A publisher publishes a message on a single topic. The test is run with 100 subscribers. The subscribers are registered to receive messages on the single topic.

This test identifies the cost of using the Publication node for a single publisher and 100 subscribers, single topic and a single copy of the message flow when using 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
4k	No	7766.63	24.00	0.25
16k	No	3266.71	22.67	0.56
64k	No	704.12	15.00	1.70
256k	No	190.12	16.67	7.01
1024k	No	75.93	21.67	22.83
4096k	No	21.90	26.33	96.19
8192k	No	10.61	27.00	203.58

Scaling Message Throughput

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

Using Additional Instances

Message Broker allows the use of additional instances of the flow to be run, these instances map on to threads running within the Broker execution group process on distributed platforms and to Task Control Blocks (TCB's) within the Brokers address space on z/OS. This test consists of running the XSLT Transform Sample with a varying number of instances of the message flow in a single execution group.

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

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

The results of running this test are given in the table below. They indicate the results of scaling with Instances with non persistent messages

Instances	Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1	4k	no	382.69	15.33	3.21
2	4k	no	763.92	27.00	2.83
3	4k	no	1110.53	40.00	2.88
4	4k	no	1400.02	53.67	3.07
5	4k	no	1616.30	67.67	3.35
6	4k	no	1782.60	80.67	3.62
7	4k	no	1836.70	87.00	3.79
8	4k	no	1939.37	95.33	3.93
9	4k	no	1948.79	94.33	3.87

As the number of instances of the message flow was increased so did the message rate. With nine instances of the message flow it was possible to achieve over five times the message rate that was achieved with a single instance and CPU utilisation went from 15% to a peak of 95% showing that it was possible to virtually utilise the whole of the machine running this message flow. The message rate also substantially increased from the initial 382/second to 1948/second.

It is worth noting that in addition to the nine instances of the message flow there was also the Message Broker queue manager listener process which is used to receive messages from the client machine. In a busy system this process alone is capable of fully using one processor.

The number of instances required to achieve a given message rate will vary dependent on the message flow, configuration and the hardware on which the runs take place. You are recommended to determine the optimum number of instances to use for each message flow individually through experimentation with a varying number of instances.

From these measurements we can see that use of additional instances can be an effective mechanism for increasing message throughput and allowing a machine to be fully utilised. The CPU usage and message rate were increased substantially over the initial position.

Using Mulitple Execution Groups

Message Broker allows the use of multiple execution groups to be run, these map on to Operating System processes on distributed platforms and to Address Spaces on z/OS. This test consists of running the XSLT Transform Sample with a varying number of execution groups.

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

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

The results of running this test with are given in the table below. They indicate the results of scaling with execution groups with non persistent messages.

Execution Groups	Msg Size	Persistent	Message Rate (Msgs/sec)	% CPU Busy	CPU ms/msg
1	4k	no	382.69	15.33	2.88
2	4k	no	797.14	29.33	2.94
3	4k	no	1142.67	43.33	3.03
4	4k	no	1411.90	57.33	3.25
5	4k	no	1476.81	72.33	3.92
6	4k	no	1696.18	84.00	3.96
7	4k	no	1781.17	94.00	4.22
8	4k	no	1837.17	100.00	4.35

As the number of execution groups running one copy of the message flow was increased so did the message rate. With eight copies of the message flow it was possible to achieve around five times the message rate that was achieved with a single instance and CPU utilisation went from 15% to a peak of 100% showing that it was possible to utilise the whole of the machine running this message flow. The message rate also substantially increased from the initial 382/second to 1837/second.

It is worth noting that in addition to the eight copies of the message flow there was also the Message Broker queue manager listener process which is used to receive messages from the client machine. In a busy system this process alone is capable of fully using one processor.

The number of copies of the message flow required to achieve a given message rate will vary dependent on the message flow, configuration and the hardware on which the runs take place. You are recommended to determine the optimum number of copies to use for each message flow individually through experimentation with a varying number of copies.

From these measurements we can see that use of multiple copies of the message over multiple execution groups can be an effective mechanism for increasing message throughput and allowing a machine to be fully utilised. The CPU usage and message rate were increased substantially over the initial position.

Overheads

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

Using the Accounting and Statistics Plug-in with Message Broker Explorer

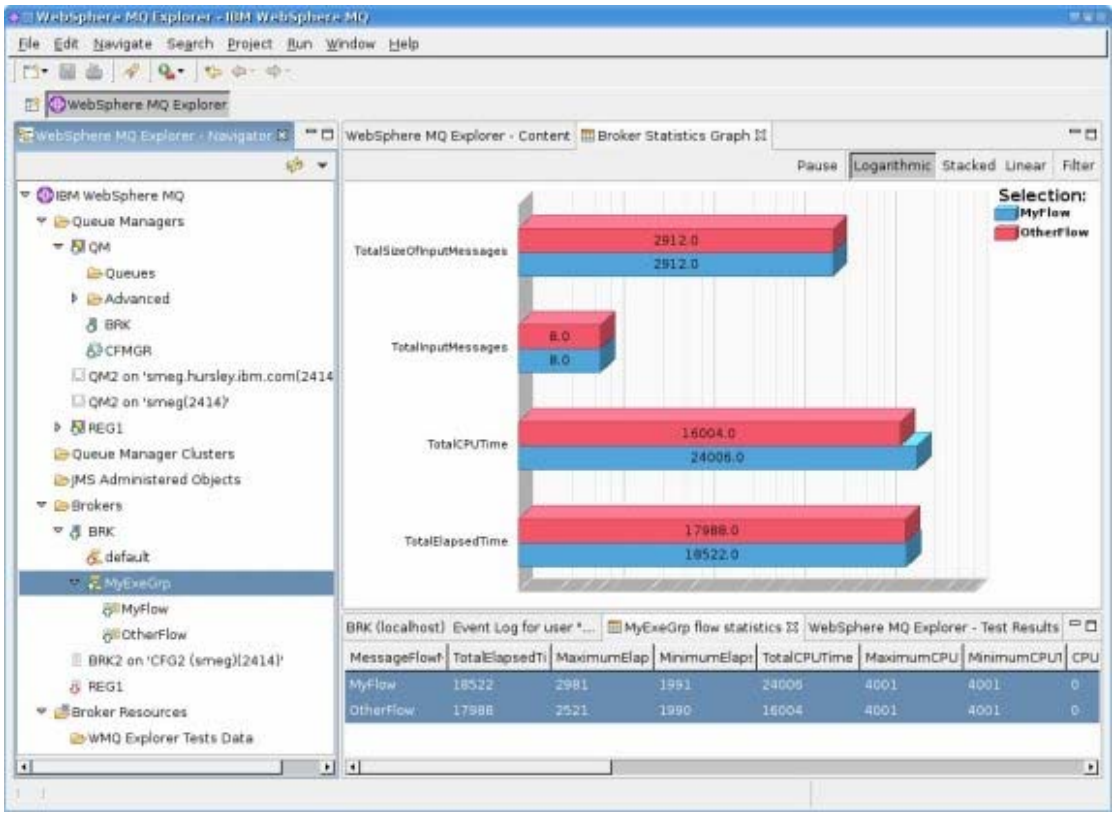
Whilst the accounting and statistics provided in broker can be accessed in a variety of ways we would recommend using this Support Pac.

The Message Broker Explorer Support Pac (IS02) is available from here:

http://www-1.ibm.com/support/docview.wss?rs=171&uid=swg24012457&loc=en_US&cs=utf-8&lang=en

This Support Pac now includes the new WebSphere Message Broker Accounting and Statistics facility which can help you monitor message flow performance and resource usage of a broker or execution group at the message flow, node, or terminal level, and can thus be a tremendous help in solving performance and resource utilization problems.

An example screen shot of output at the execution group level is shown below:



Collecting accounting and statistics data from a broker affects its performance, so you may want to restrict its use to development and test systems. The performance overhead depends on the level of data collected and the complexity of the message flow. For example, running the Large Messaging Sample shipped with WebSphere Message Broker V6.1 using a 4K message with basic thread level and advanced node level accounting activated resulted in a 6.2% processing overhead.

Further information on this utility can be found here:
http://www.ibm.com/developerworks/websphere/library/techarticles/0710_piatek/0710_piatek.html

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 a 20% reduction in message throughput. This reflects the CPU and I/O overhead of writing user trace. With debug trace the overhead will be even higher as debug trace is more extensive.

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

In WebSphere Message Broker V6.1 the overhead of leaving trace nodes in your message flow has now been reduced. Trace nodes can also now be enabled/disabled easily using the 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 V6.1. 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
4k	No	585.08	17.33	2.37

Impact of Running a Flow with a Trace Node Turned On

This Test consists of the Large Message Sample flow as shipped with WMB V6.1. 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
4k	No	202.79	14.33	5.65

This indicates that the overhead of having the trace node on is 65%.

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
4k	No	568.00	15.00	2.11

This indicates that overhead of running with trace nodes disabled for this flow is 2.9%. 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://publib.boulder.ibm.com/epubs/pdf/c3468660.pdf

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 2 GHz or faster processor. For production a multi-processor machine is recommended. For development 2GB+ of memory is recommended. For 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 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 ps command was used to get these results with the following flags "ps -p <Dataflowengine Pid> -o vsz".

Use Case	Message Size	Virtual Memory Peak After Processing Messages	Real Memory Peak After Processing Messages
Aggregation	4K	526.148	230.236
Coordinated Request/Reply	4K	527.776	231.848
Large Messaging	10K	500.524	211.384
Message Routing	4K	519.196	223.672
SWIFT Message Parse	1K	630.276	331.848
XML Transformation	4K	692.624	406.468
XMLNSC Validation	1K	588.672	256.136
SOAP Nodes	1K	574.292	286.228
Simple DB Routing	1K	660.484	362.42

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

Note: The methodology used to get this data is different to that used for the V6.0 performance reports.

Tuning

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

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

Message Broker

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

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

2. The Max heap size of the The WebSphere Message Broker Java Virtual Machine (JVM) (in which much of the publish subscribe code is executed) was set to 512MB. For the non Publish Subscribe tests the default value of 256MB was used.

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 -1.

- For HTTP and SOAP Request Node tests the SocketConnectionManager was set to use persistent connections by setting MaxKeepAlives was set to -1.

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 100MB for the input and output queues used in the tests. This value is the maximum supported and was used because in most test messages of up to 8MB 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 10
   - LogSecondaryFiles was set to 1

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

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

5. 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. All machines used the operating system default values.

Database

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

Miscellaneous

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

- Locate the log of any WebSphere MQ queue manager through which persistent messages pass on a dedicated disk.

- Locate the WebSphere MQ queue manager log on a very fast disk such as one with a non-volatile fast write cache. Such disks are consistently capable of I/O times of 1ms compared with a time of 6 ms for a 10,000 RPM SCSI disk. When using a disk with a fast write cache it is essential that it has a non-volatile capability as the log data is critical to the integrity of your queue manager.

- Locate the log of the Message Broker database on a dedicated disk.

- Locate the log of the Message Broker database on a very fast disk such as one with a non-volatile fast write cache.

- When performing BLOB inserts to a database locate the data portion of the database on a very fast disk such as one with a non-volatile fast write cache. BLOB I/O is not buffered by a database such as DB2 and is written to disk immediately.

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:

http://www-1.ibm.com/support/docview.wss?uid=swg24006518

The Performance Harness

The current development version of IBM Performance Harness for JMS was used to drive all of the tests contained within this report. This version is likely to be released later this year. Although this tool was primarily developed as a performance driver for JMS but it also has the ability to send and receive WebSphere MQ messages and in the development version has been extended to drive HTTP messages.

The documentation for the tool contains examples of how to run it to send/receive messages to/from a JMS Provider and MQ.

The current version of the tool can be downloaded from:

http://www.alphaworks.ibm.com/tech/perfharness

This tool is Java-based and so can be run on any platform that supports Java and has the MQ Client available.

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:

- A Sun Fire V40z with 4 * 2.2 GHz dual-core AMD Opteron processors
- Three 73 GB SCSI hard drives formatted with UFS.
- 32 GB RAM
- Broadcom NetXtreme Gigabit Ethernet

The software consisted of

- SunOS 5.10
- WebSphere MQ V6.02
- WebSphere Message Broker V6.1.0.1
- DB2 V9.1

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 2 * 3.6 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 2003
- WebSphere MQ V6.01
- IBM Java 1.5

Network Configuration

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

Appendix B - Evaluation Method

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

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

1. Point to Point Message Processing. This configuration tested these transports:
    a. MQ
    b. JMS
    c. HTTP
    d. SOAP
2. Publish Subscribe Message Processing
3. 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 read 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.

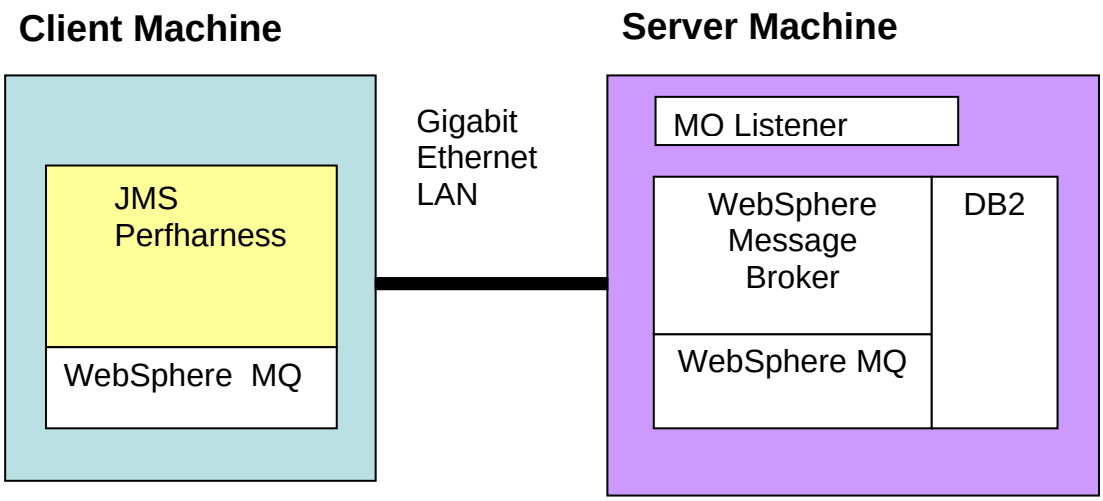
There was a single client machine.

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.

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

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

Publish Subscribe Testing

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

Message Generation and Consumption

The following modules of the Performance Harness for JMS were used in this testing

- jms.r11.Publisher – for sending JMS Messages
- jms.r11.Receiver – for receiving JMS Messages

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

At the start of the tests the relevant number of non durable subscriptions were registered with the broker on the single topic. Each subscription registered the same queue name for messages to be delivered to, hence all messages for all subscriptions were delivered to the same queue.

The receiving client was then started to drain the subscriber queue, the publishing client was then started. Messages were transmitted from the client machines to the server machine using the WebSphere MQ JMS Messages.

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

In all of the tests it was verified that all publications were delivered to subscribers without any loss of messages.

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

Publishers

The JMS Publisher sent non-persistent and persistent publications dependent on the test case. A transacted JMS Session was used for persistent messages. The publisher produced publications at a constant rate, i.e. a fixed number of publications per second.

Subscribers

The JMS application draining the subscriber queue used non transacted sessions for non persistent tests but used transacted sessions for persistent tests. A single topic was used for all tests and so all subscribers were subscribed to the same topic. This meant that for every message published a copy was received on the subscriber queue each subscription.

The same machine configuration was used as point to point testing i.e. 1 Client machine and one Server machine. The publisher and subscriber applications were run on the same machine in different JVMs.

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 prepopulated 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

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

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

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

For a configuration consisting of one publisher and 10 subscribers where the publication rate was 10 messages/second the total message rate is 10 * 10 = 100 messages second.

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

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

Appendix C - Test Messages

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

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

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

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

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

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

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

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.

Compute Message

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

The message layout is shown below

```
<Parent>
        <First>1</First>
        <SaleList>
                <Invoice>
                        <Initial>K</Initial>
                        <Initial>A</Initial>
                        <Surname>Braithwaite</Surname>
                        <Item>
                                <Code>00</Code>
                                <Code>01</Code>
                                <Code>02</Code>
                                <Description>Twister</Description>
                                <Category>Games</Category>
                                <Price>00.30</Price>
                                <Quantity>01</Quantity>
                        </Item>
                        <Item>
                                <Code>02</Code>
                                <Code>03</Code>
                                <Code>01</Code>
```

```
                        <Description>The Times Newspaper</Description>
                        <Category>Books and Media</Category>
                        <Price>00.20</Price>
                        <Quantity>01</Quantity>
                    </Item>
                    <Balance>00.50</Balance>
                    <Currency>Sterling</Currency>
                </Invoice>
                <Invoice>
                    <Initial>T</Initial>
                    <Initial>J</Initial>
                    <Surname>Dunnwin</Surname>
                    <Item>
                        <Code>04</Code>
                        <Code>05</Code>
                        <Code>01</Code>
                        <Description>The Origin of Species</Description>
                        <Category>Books and Media</Category>
                        <Price>22.34</Price>
                        <Quantity>02</Quantity>
                    </Item>
                    <Item>
                        <Code>06</Code>
                        <Code>07</Code>
                        <Code>01</Code>
                        <Description>Microscope</Description>
                        <Category>Miscellaneous</Category>
                        <Price>36.20</Price>
                        <Quantity>01</Quantity>
                    </Item>
                    <Balance>80.88</Balance>
                    <Currency>Euros</Currency>
                    </Invoice>
                <InvoicesTotal Currency="Sterling">57.116</InvoicesTotal>
        </SaleList>
        <Last>Test</Last>
</Parent>
```

Transform Message

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

The message layout is shown below.

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

```
                        <Desc>The Times Newspaper</Desc>
                        <Cost>3.2E-1</Cost>
                        <Qty>01</Qty>
                    </Article>
                </Purchases>
                <Amount>8E-1</Amount>
            </Statement>
            <Statement Type="Monthly" Style="Full">
                <Customer>
                    <Initials>TJ</Initials>
                    <Name>Dunnwin</Name>
                    <Balance>81.84</Balance>
                </Customer>
                <Purchases>
                    <Article>
                        <Desc>The Origin of Species</Desc>
                        <Cost>3.5744E+1</Cost>
                        <Qty>02</Qty>
                    </Article>
                    <Article>
                        <Desc>Microscope</Desc>
                        <Cost>5.792E+1</Cost>
                        <Qty>01</Qty>
                    </Article>
                </Purchases>
                <Amount>1.29408E+2</Amount>
            </Statement>
    </SaleList>
</Parent>
```

Appendix D - Use Case Descriptions

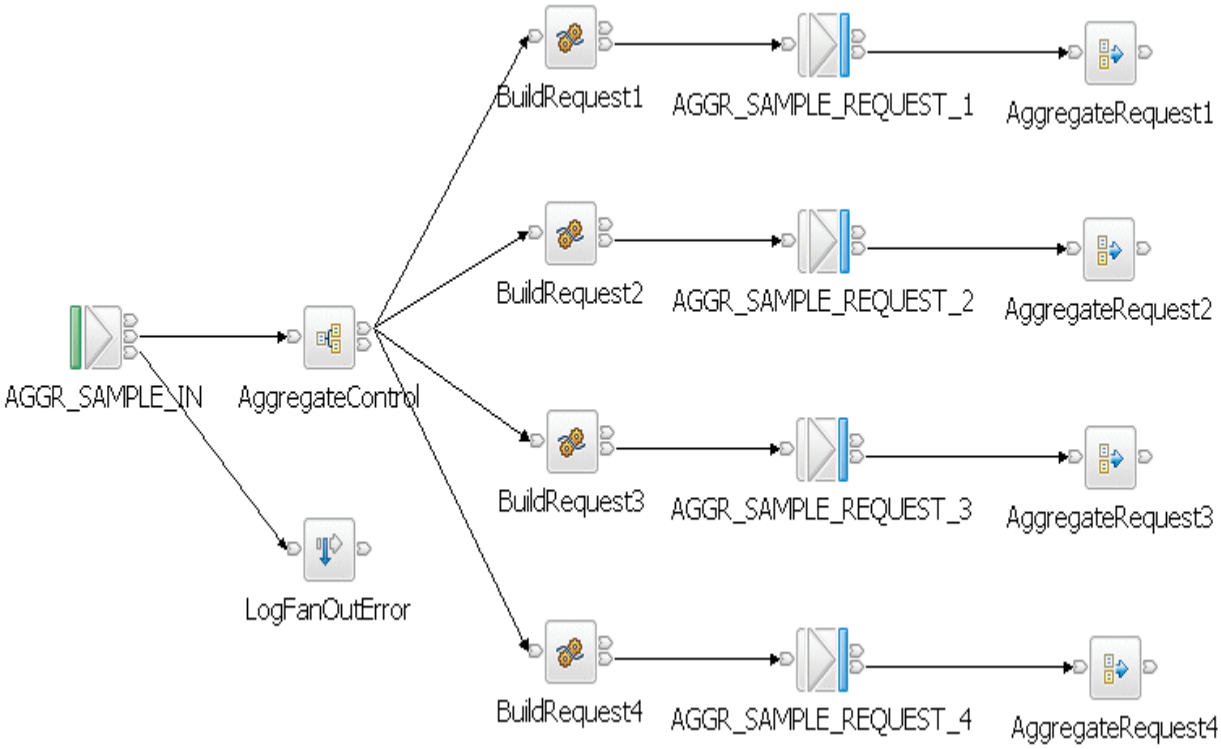
This section contains a description of the processing in each of the use cases which are used to characterise the performance of WebSphere Message Broker V6. All of these use cases are shipped as samples in WebSphere Message Broker V6.1. See the samples gallery for more information.

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

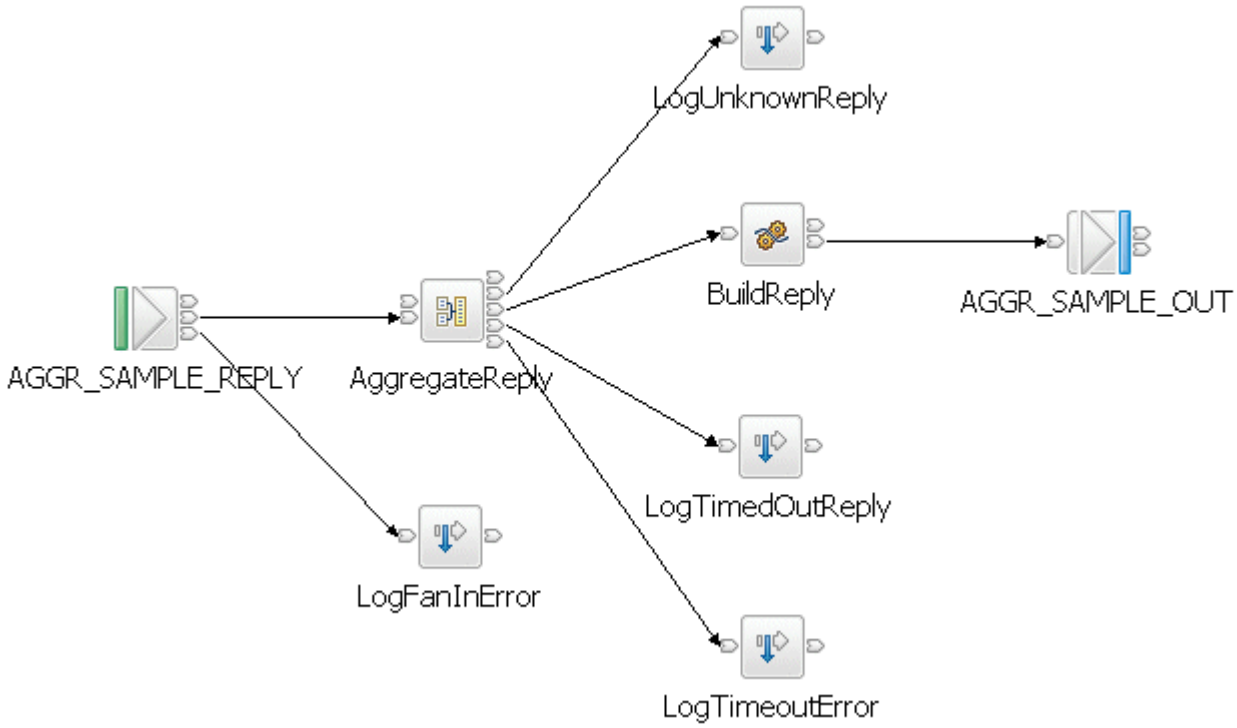
This is the flow that takes the incoming request message, generates four different request messages, sends them out on request/reply, and starts the tracking of the aggregation operation:



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

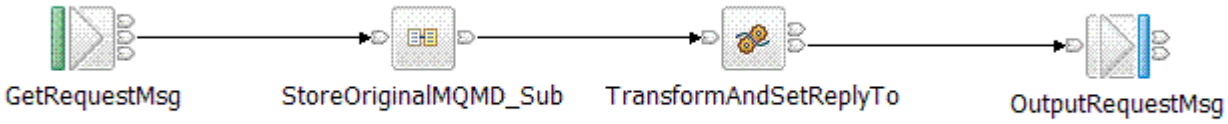
The coordinated request reply sample is based on the scenario of a contemporary and established application communicating through the use of WebSphere MQ messages in a request/reply processing pattern. The contemporary application uses self-defining XML messages and issues a request message. The established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

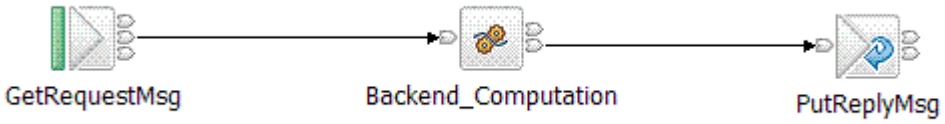
The Request message flow consists of the following nodes:



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.

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.

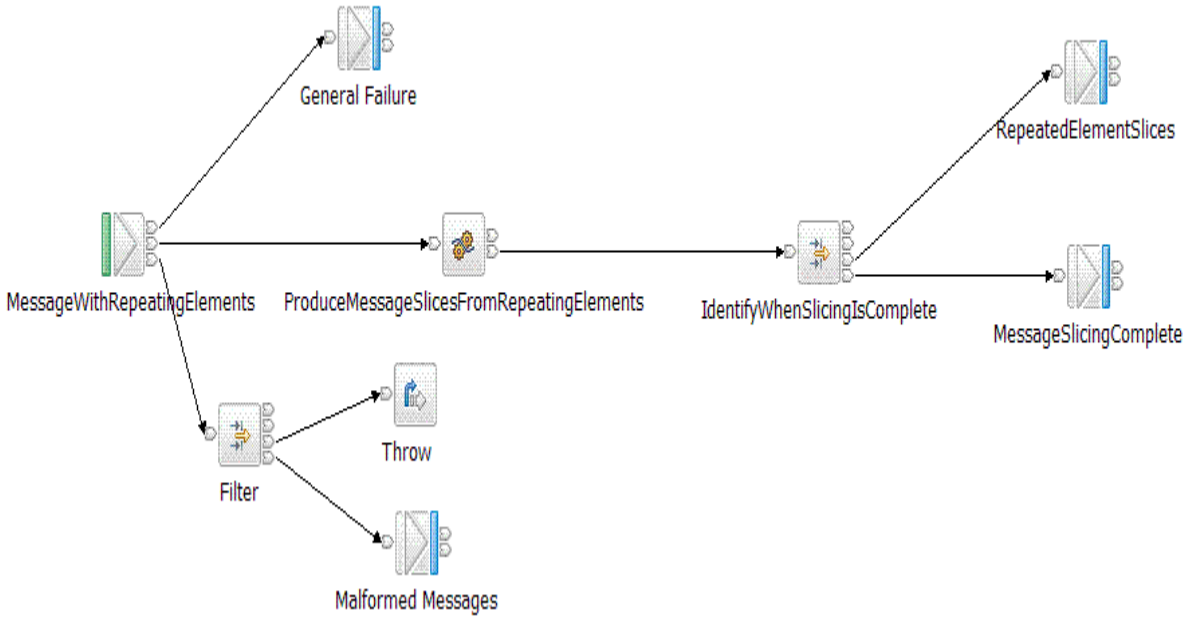
The processing in the sample consists of one message flow. The processing it performs is described below.

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.

SWIFT Message Parse

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

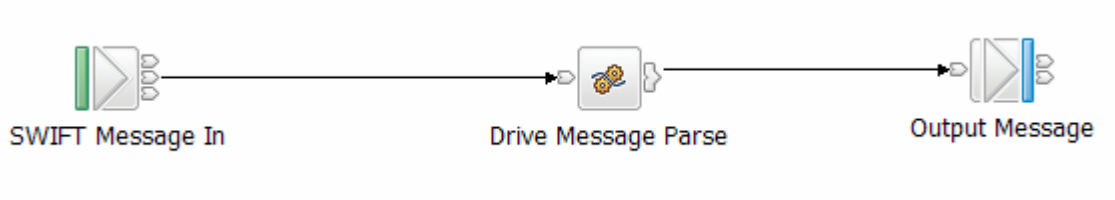
The processing in this test consists of a full parse of a SWFIT MT103 message format.

SWIFT Message Parsing Message Flow

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

1. Reads a WebSphere MQ message containing a SWIFT MT103 message in tagged delimited string format.
2. Accesses the last element in the input message.
3. Produces a WebSphere MQ message to signal completion of the processing.

The SWIFT Message Parse processing consists of the following nodes:



The output message is a minimal WebSphere MQ message.

XML Transformation

The XMLT sample shows how an XML message can be transformed into a different layout using an XMLTransformation node and a XSL stylesheet. This type of processing could be performed in any situation where the layout of a message needs to be changed to suit the requirements of different application.

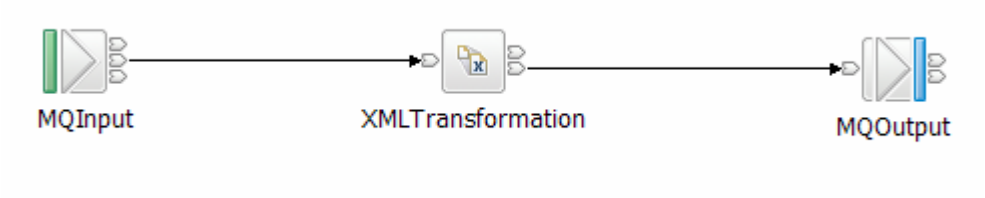
This technology provides the ability to use an XSL stylesheet in a new way, using it as part of a message flow.

XSL Transformation Message Flow

The processing in the message flow consisted of the following:

1. Reads a WebSphere MQ message containing an XML payload.
2. Invoke the XSL Stylesheet transformation.
3. Write an MQOutput message containing the modified message.

The following figure shows the XSL Transformation message flow:



The XSL stylesheet used in the processing was as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<Parent>
<xsl:for-each select="/Parent/SaleList">
<SaleList>
<xsl:for-each select="Invoice">
<xsl:if test="not(contains(Surname,'Shop'))">
<Statement>
<xsl:attribute name="Type">Monthly</xsl:attribute>
<xsl:attribute name="Style">Full</xsl:attribute>
<Customer>
<Initials>
<xsl:for-each select="Initial">
<xsl:value-of select="."/>
</xsl:for-each>
</Initials>
<Name><xsl:value-of select="Surname"/></Name>
<Balance><xsl:value-of select="Balance"/></Balance>
</Customer>
<Purchases>
<xsl:for-each select="Item">
<Article>
<Desc><xsl:value-of select="Description"/></Desc>
<Cost><xsl:value-of select='format-number((number(Price)*1.6),"####.##")'/></Cost>
<Qty><xsl:value-of select="Quantity"/></Qty>
</Article>
</xsl:for-each>
</Purchases>
<Amount>
<xsl:attribute name="Currency">
<xsl:value-of select="Currency" />
</xsl:attribute>
<xsl:call-template name="sumSales">
<xsl:with-param name="list" select="Item"/>
</xsl:call-template>
</Amount>
</Statement>
</xsl:if>
</xsl:for-each>
</SaleList>
</xsl:for-each>
</Parent>
</xsl:template>
<xsl:template name="sumSales">
<xsl:param name="list" />
<xsl:param name="result"  select="0"/>
<xsl:choose>
<xsl:when test="$list">
<xsl:call-template name="sumSales">
<xsl:with-param name="list"
select="$list[position()!=1]"/>
<xsl:with-param name="result"
select="$result + number($list[1]/Price)*number($list[1]/Quantity)*1.6"/>
</xsl:call-template>
</xsl:when>
<xsl:otherwise>
<xsl:value-of select='format-number(number($result),"####.##")'/>
</xsl:otherwise>
```

</xsl:choose>
</xsl:template>
</xsl:stylesheet>

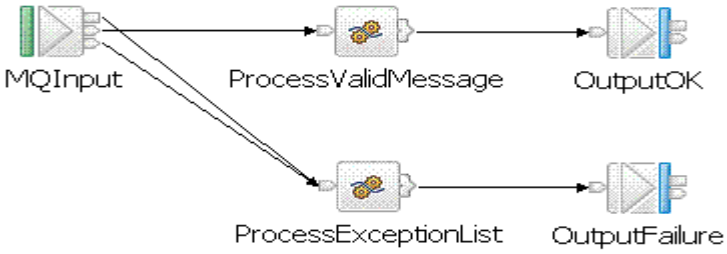
The message being transformed was the same format as that described in the Section Input Message.

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

XMLNSC Validation

The XMLNSC Validation sample demonstrates how WebSphere Message Broker's XMLNSC parser can validate XML messages against a schema.

Message Flow
The processing in the sample consists of one message flow called XMLNSCVALIDATIONMF1. The following figure shows the XMLNSCVALIDATIONMF1 message flow:



The XMLNSCVALIDATIONMF1 message flow is in the message flow project called XMLNSCVALIDATIONMF.

The message flow performs the following tasks:

- Takes an XML message in through an MQInput node from the WebSphere MQ queue XMLNSCVALIDATIONMF1.IN.
- The MQInput node has the following properties defined on the:
  Input Message Parsing tab:
  - **Message domain** is set to XMLNSC.
  - **Message set** is set to XMLNSCVALIDATIONMS.

  Parser Options tab:
  - **Parse Timing** is set to Complete.
  - **Build tree using XML schema data types** is checked.

  Input Message Parsing tab:
  - **Validation** is set to Content and Value.
  - **Failure action** is set to Exception.
- The **Out** terminal of the MQInput node is wired to a Compute node called ProcessValidMessage. The Compute node is associated with ESQL that copies InputRoot to OutputRoot.
- The **Out** terminal of the Compute node is wired to an MQOutput node called OutputOK. The MQOutput node puts messages to queue XMLNSCVALIDATIONMF1.OUT.
- The **Catch** and **Failure** terminals of the MQInput node are wired to a second Compute node called ProcessExceptionList. This Compute node copies the relevant part of the ExceptionList to OutputRoot.

- The **Out** terminal of this Compute node is wired to an MQOutput node called OutputFailure. This MQOutput node puts messages to queue XMLNSCVALIDATIONMF1.FAIL.

Message set

Message set XMLNSCVALIDATIONMS specifies runtime domain XMLNSC and contains an XML schema called SampleMessageValidation.xsd. A message definition file has been created from this schema. The message set contains the following:

- One message called SaleListMessage. The SaleListMessage is based on the message that is used in many of the other samples.
- Several constraints have been added to the XML schema for this message to demonstrate various types of validation failure. These are:
  - Value constraints:
    - Surname has maxLength set to 20.
    - Currency has valid values 'GBP', 'Sterling', 'USD', 'EUR', 'Euros' and 'JPY'.
    - Quantity has minInclusive 1 and maxInclusive 100.
    - Code has a pattern of 2 alphanumeric characters.
  - Changed elements and attributes:
    - Element CompletionTime has type of xsd:string changed to xsd:dateTime.
    - An optional attribute 'Title' to Header.
    - A mandatory attribute 'occurrence' to SaleList.
    - An optional attribute 'guaranteed' to SaleList.
    - A mandatory attribute 'isLast' to Trailer.

Test messages

The following test messages are used in this sample for the performance tests:

- ValidMessage.xml

Simplified Database Routing

The Simplified Database Routing sample is based on the scenario of an employee management processing system. It demonstrates how to use some of the features provided by WebSphere Message Broker.

The Simplified Database Routing sample demonstrates how you can design applications that process self-defining XML messages. XML messages are self-defining because each piece of data is prefixed by a tag name or an attribute name. An XML message definition is in the message itself and is not held anywhere else. Therefore, you can run a message flow using an XML message without the aid of an external message set.

If you use self-defining XML messages, you can program the message flow to manipulate messages, as you can if you use an external message set to hold message format information. However, without a message set, you must code all the format information in ESQL in the nodes of the message flow, and you cannot use the full range of built-in nodes. In this sample the message format is transformed through the use of the DatabaseRetrieve node, where either ESQL or XPath message processing transformation languages may be used to specify the out going message format.

The Simplified Database Routing sample performs the following actions:

- Uses a Route node to bypass an unnecessary step in the flow. This node tests if certain employee data is in its input message, and only performs a database lookup operation if the data is missing.
- Uses a DatabaseRetrieve node to add information, acquired from a database query, into its input message to form a new output message. The query is performed using a key value expected in the original input message.
- Uses a DatabaseRoute node to dynamically route one or more copies of its input message down different flow legs based on user provided expressions applied to values retrieved from a specified database query, also specified on this node.

- Based on routing decisions made by the DatabaseRoute node, it may generate a reply message to confirm that a given employee should be ordered a clock in recognition of ten years service to the company.
- Based on routing decisions made by the DatabaseRoute node, it may generate a reply message to categorise that a given employee is senior in age to their respective manager.
- Based on routing decisions made by the DatabaseRoute node, it may generate a reply message to confirm that neither of the above two conditions apply to a given employee.
- Based on a query 'key not found condition' arising in the DatabaseRoute node, it may generate a reply message to indicate an empty result set was produced by the query, based on the search information provided in the node's input message.

The message flow
The Simplified Database Routing sample includes the following message flow:

- SimplifiedDBRouting dynamically routes messages and retrieves information about an employee, based on the the details specified in the usedbretrievenode, usedbretrievenode_nofiltermatch, bypassdbretrievenode and bypassdbretrievenode_keynotfound test client input messages.
  For more information, read about message flows in the WebSphere Message Broker documentation.

The messages
The Simplified Database Routing sample processes self-defining, or generic, XML messages. A self-defining XML message carries the information about its content and structure within the message in the form of a document that adheres to the XML specification. A definition of the message is not held anywhere else. When the message flow receives the message. The message is identified by the XMLNSC parser rather than the deprecated generic XML parser, and the message is parsed according to the XML definitions contained within the message itself.
For the performance tests we reused the following input message:

- One bypassdbretrievenode input message: The message already contains information about a specific employee, originally providing all required employee details. The employee satisfies all routing conditions set against an employee's details at the end of the flow.
-

The database
The Simplified Database Routing sample has one database called SROUTEDB. The message flow in the sample directly access SROUTEDB, which contains two database tables called DEPARTMENT and EMPLOYEE.

Note that the Simplified Database Routing sample is driven at its inputs and outputs. In between the inputs and outputs are database interactions. This makes the sample a slightly unusual scenario. In most WebSphere Message Broker applications the true application data resides in external databases because WebSphere Message Broker is usually the integration middleware between a client and a back-end database (which often resides on some sort of existing server platform).
This does not detract from the usefulness of the sample. If you want, you could remove the database and make it external. The message flow would then have to be split in two:

- The first part would prepare queries and send them out as messages. You would then need a WebSphere MQ or WebSphere Message Broker adapter at the database server to enable the message to access the back-end databases.
- The second part would catch the response from the back-end databases via its adapter and return a response to the requester (client).

If done carefully, such a system would be indistinguishable (to the client) from the sample as supplied here, in which database access is internal.

The WebSphere MQ queues
The Simplified Database Routing sample message flow interacts with WebSphere MQ local queues. The queues are defined on the WebSphere MQ queue manager that hosts the

broker on which the message flow runs. For more information about the WebSphere MQ queues in the Simplified Database Routing sample see the product documentation.

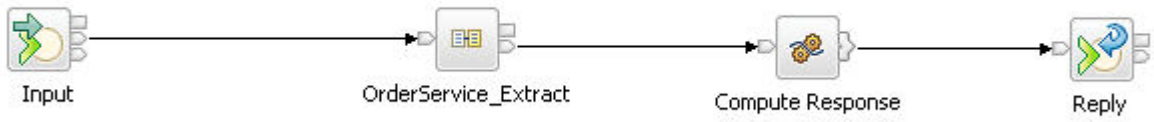
SOAP Nodes

The SOAP Nodes sample shows how the SOAP Input, Reply and Request nodes can be used to both provide and consume a Web Service.
The starting point for the sample was a WSDL file that defines a simplified ordering service. The web service always returns a response indicating the part order is available; an option for extending the web service might be to use a Database node to query a stock database.
This WSDL file was used in conjunction with the "Start from WSDL and/or XSD files" wizard to create the message set, and two message flows that constitute this sample.
The SOAP Nodes sample demonstrates the following tasks:

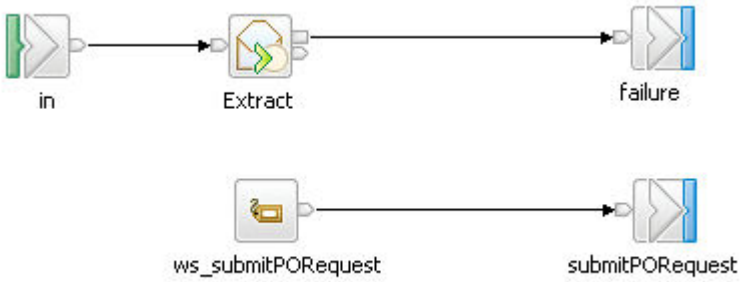
- How to provide a Web Service using SOAP Input and SOAP Reply nodes
- How to consume a Web Service using a SOAP Request node

The message flows

The sample uses two message flows. One provides a Web Service and one consumes a Web Service. These are described below.
The following figure shows the Web Service provider message flow:

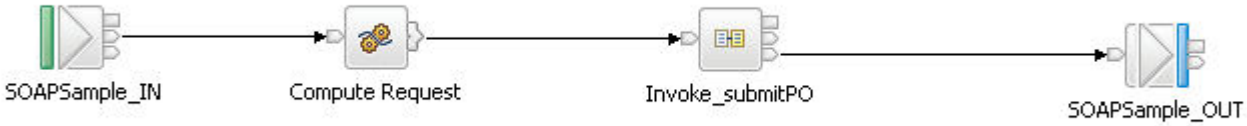


The SOAP Input node recieves incoming SOAP messages and, if they are valid, passes them down the message flow to the OrderService_Extract subflow. The OrderService_Extract subflow is created by the "Start from WSDL and/or XSD files" wizard and looks like this:

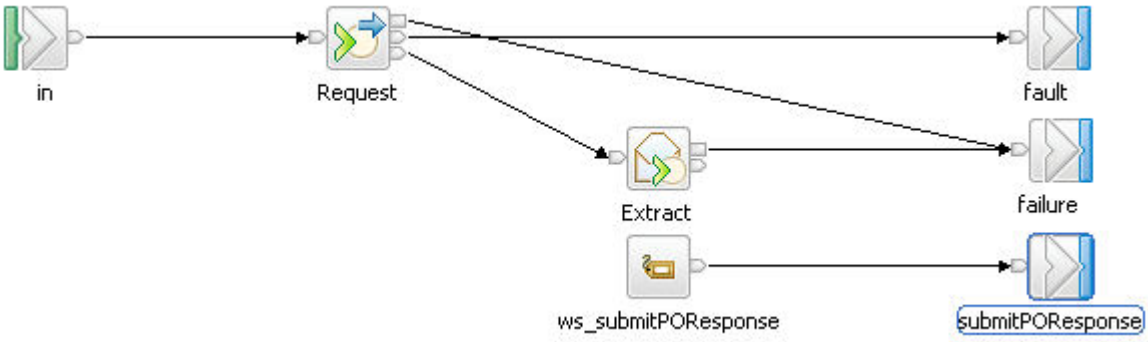


The SOAP Extract node takes a SOAP message and removes the SOAP envelope. In this sample, this leaves an XML message which can now be used in the XMLNSC domain in nodes such as the Mapping node or the Compute node. The SOAP Extract node then routes the message to a label based on the Web Service operation that is being invoked. In this sample, only one operation is used. If the WSDL used as a starting point has multiple operations, then the "Start from WSDL and/or XSD files" gives the option of implementing more than one operation, and this subflow would have multiple labels, which translate to multiple output terminals in the parent message flow.
Upon exiting the subflow and returning to the main provider message flow, the XMLNSC message enters a Compute node. It is now possible to reference the original SOAP body as XML and perform ESQL operations on any of the incoming data. However, in this sample, we ignore the incoming message and instead simply create some valid XML data from scratch, rather than using any of the incoming data. This data is then sent to the SOAP Reply node which constructs a suitable

SOAP message to return to the Web Service consumer that initiated the Web Service call.
The following figure shows the Web Service consumer message flow:



This Web Service consumer flow is initiated by an MQ message arriving on the queue monitored by the MQ Input node. In this sample, this is an XML message in the XMLNSC domain. This message then enters a Compute node where the incoming data is used to create the XML data we want to send to the Web Service. The message is then passed down the flow to the Invoke_submitPO subflow. The Invoke_submitPO subflow is created by the "Start from WSDL and/or XSD files" wizard and looks like this:



The SOAP Request node takes the incoming XML data and uses it to build a valid SOAP message to send to the Web Service defined by the node properties. Assuming a valid response is recieved, it is passed to a SOAP Extract node which removes the SOAP envelope from the reponse, returning the message to the XMLNSC domain. At this point the message is routed to the ws_submitPOResponse label and exits the subflow.
Back in the main consumer flow, the message is then sent to an MQ Output node which writes the XML data to the specified MQ queue.

The messages

The web client message flow is driven by an MQ message. A test client file is supplied for running the sample using the following XML message:

```
<OrderMessage>
  <localElement>
    <FirstName>Message</FirstName>
    <LastName>Broker</LastName>
    <Street>IBM</Street>
    <City>IBM</City>
    <ZipCode>IBM</ZipCode>
    <PartNumber>Some Part</PartNumber>
    <Quantity>1</Quantity>
  </localElement>
</OrderMessage>
```

The valid format for a web service request message and response message is shown in an edited schema extract below:

```
<xsd:element name="submitPORequest">
...
        <xsd:complexType>
          <xsd:sequence>
```

```
            <xsd:element name="partNo"
type="xsd:string"/>
            <xsd:element name="partQuantity"
type="xsd:int"/>
            <xsd:element name="personName">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="firstName"
type="xsd:string"/>
                  <xsd:element name="lastName"
type="xsd:string"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="address">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="street"
type="xsd:string"/>
                  <xsd:element name="city"
type="xsd:string"/>
                  <xsd:element name="zipCode"
type="xsd:string"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
</xsd:element>
<xsd:element name="submitPOResponse">
...
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="orderStatus"
type="xsd:string"/>
            <xsd:element name="orderAmt" type="xsd:int"/>
            <xsd:element name="partNo"
type="xsd:string"/>
            <xsd:element name="partQuantity"
type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```