#SHAREorg

SHARE
Technology • Connections • Results

The Do's and Dont's of Message Broker Performance [z/OS and Distributed]

David Gorman (gormand@uk.ibm.com)
IBM

14th March 2012
Session Number 10698

Agenda

- What are the main performance costs in message flows?

  …and what are the best practices that will minimize these costs?

- What other performance considerations are there?

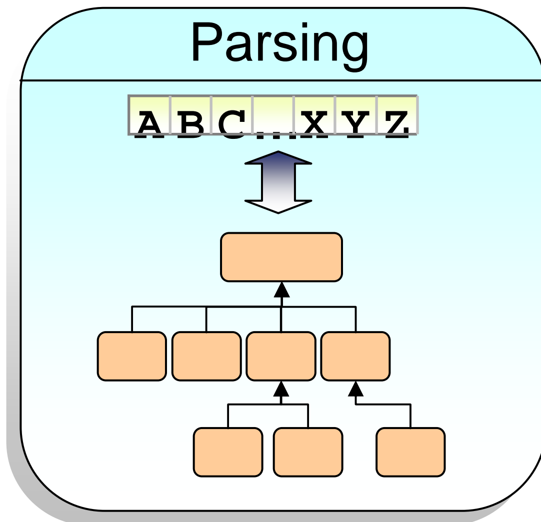
- Understanding your broker's behaviour

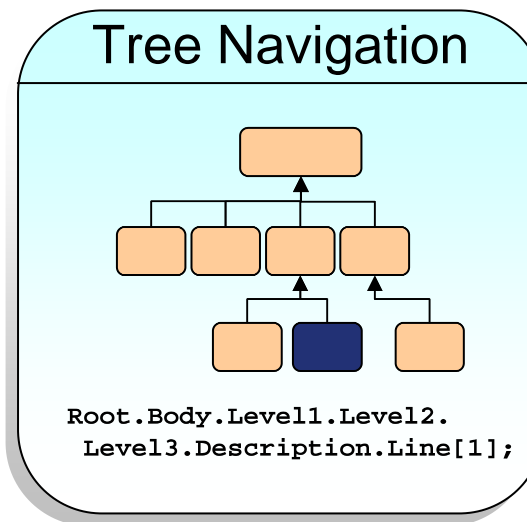
Download this presentation or supportpac **IP04** for a lot more detail. Notes slides are included!

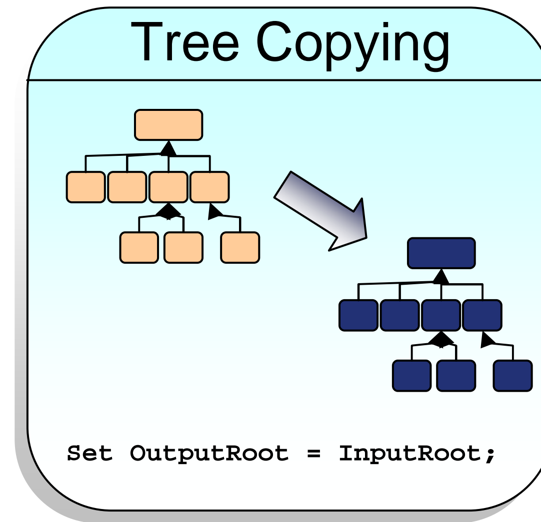
Why should you care about performance?

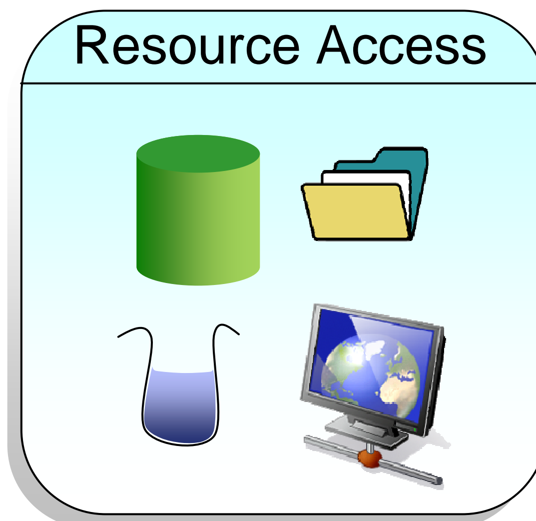
- Optimising your broker's performance can lead to real $$ savings… Time is Money!
  - Lower hardware requirements, lower MIPS
  - Maximised throughput… improved response time
  - Ensure your SLAs are met
- WebSphere Message Broker is a sophisticated product
  - There are often several ways to do things… which is the best?
  - Understanding how the broker works allows you to write more efficient message flows and debug problems more quickly
- It's better to design your message flows to be as efficient as possible from day one
  - Solving bad design later on costs much more
  - Working around bad design can make the problem worse!

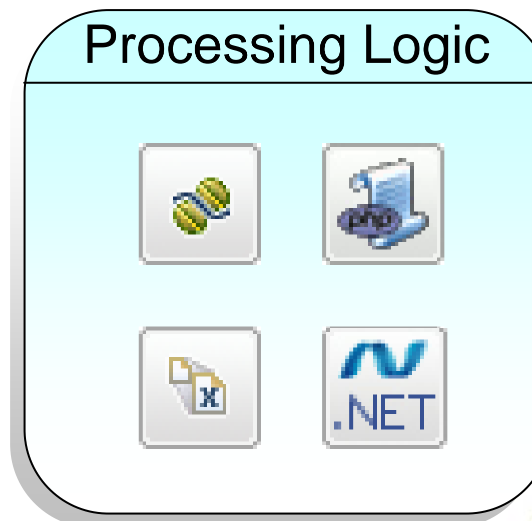
What are the main performance costs in message flows?











Notes (slide 1 of 2)

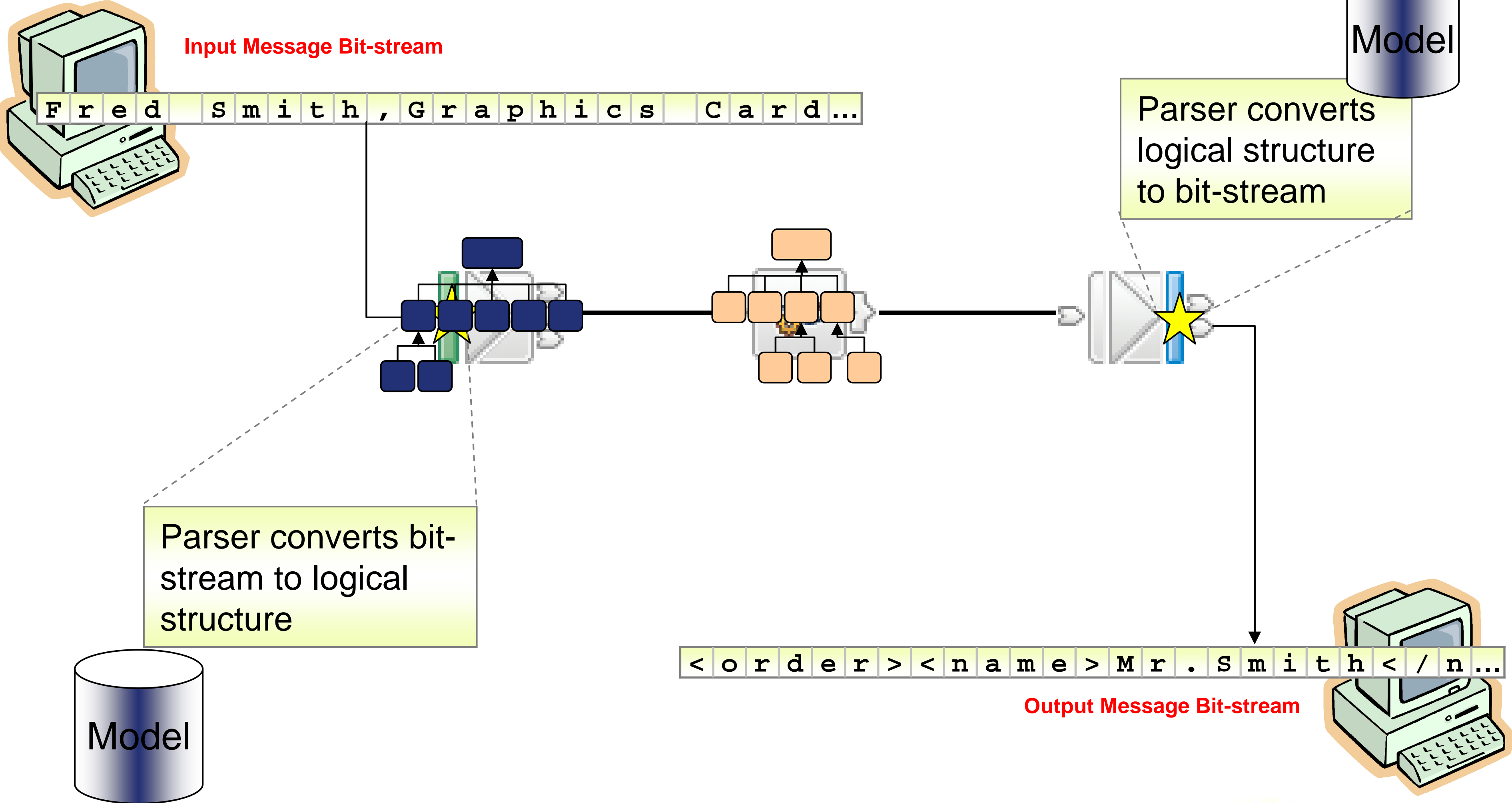
- When this or any other message flow processes messages costs arise. These costs are:
  - Parsing. This has two parts. The processing of incoming messages and the creation of output messages. Before an incoming message can be processed by the nodes or ESQL it must transformed from the sequence of bytes, which is the input message, into a structured object, which is the message tree. Some parsing will take place immediately such as the parsing of the MQMD, some will take place, on demand, as fields in the message payload are referred to within the message flow. The amount of data which needs to be parsed is dependent on the organization of the message and the requirements of the message flow. Not all message flows may require access to all data in a message. When an output message is created the message tree needs to be converted into an actual message. This is a function of the parser. The process of creating the output message is referred to as serialization or flattening of the message tree. The creation of the output message is a simpler process than reading an incoming message. The whole message will be written at once when an output message is created. We will discuss the costs of parsing in more detail later in the presentation.
  - Message/Business Processing. It is possible to code message manipulation or business processing in any one of a number of transformation technologies. These are ESQL, Java, PHP, .NET, Mapping node, XSL and WebSphere TX mappings. It is through these technologies that the input message is processed and the tree for the output message is produced. The cost of running this is dependent on the amount and complexity of the transformation processing that is coded.

Notes (slide 2 of 2)

- Navigation. This is the process of "walking" the message tree to access the elements which are referred to in the ESQL or Java. The cost of navigation is dependent on the complexity and size of the message tree which is in turn dependent on the size and complexity of the input messages and the complexity of the processing within the message flow.
- Tree Copying. This occurs in nodes which are able to change the message tree such as Compute nodes. A copy of the message tree is taken for recovery reasons so that if a compute node makes changes and processing in node incurs or generates an exception the message tree can be recovered to a point earlier in the message flow. Without this a failure in the message flow downstream could have implications for a different path in the message flow. The tree copy is a copy of a structured object and so is relatively expensive. It is not a copy of a sequence of bytes. For this reason it is best to minimize the number of such copies, hence the general recommendation to minimize the number of compute nodes in a message flow. Tree copying does not take place in a Filter node for example since ESQL only references the data in the message tree and does not update it.
- Resources. This is the cost of invoking resource requests such as reading or writing WebSphere MQ messages or making database requests. The extent of these costs is dependent on the number and type of the requests.

- The parsing, navigation and tree copying costs are all associated with the population, manipulation and flattening of the message tree and will be discussed together a little later on.
- Knowing how processing costs are encountered puts you in a better position to make design decisions which minimize those costs. This is what we will cover during the course of the presentation.

Parsers



SHARE in Atlanta 2012

Parsing

- The means of populating and serializing the tree
  - Can occur whenever the message body is accessed
  - Multiple Parsers available: XMLNSC, DFDL, MRM XML, CWF, TDS, MIME, JMSMap, JMSStream, BLOB, IDOC, RYO
  - Message complexity varies significantly …and so do costs!
- Several ways of minimizing parsing costs
  - Use cheapest parser possible, e.g. XMLNSC for XML parsing, DFDL for non XML.
  - Identify the message type quickly
  - Use Parsing strategies
    - Parsing avoidance
    - Partial parsing
    - Opaque parsing

Notes (slide 1 of 2)

- In order to be able to process a message or piece of data within Message Broker we need to be able to model that data and build a representation of it in memory. Once we have that representation we can transform the message to the required shape, format and protocol using transformation processing such as ESQL or Java. That representation of a sequence of bytes into a structured object is the message tree. It is populated though a process called parsing.
- There are two parts to parsing within Message Broker:
  - The first is the most obvious. It is the reading and interpretation of the input message and recognition of tokens within the input stream. Parsing can be applied to message headers and message body. The extent to which it is applied will depend on the situation. As parsing is an expensive process in CPU terms it is not something that we want to automatically apply to every part of every message that is read by a message flow. In some situations it is not necessary to parse all of an incoming message in order to complete the processing within the message flow. A routing message flow may only need to read a header property such as user identifier or ApplIdentifyData in the MQMD for example. If the input message is very large parsing the whole message could result in a large amount of additional CPU being unnecessarily consumed if that data is not required in the logic of the message flow.
  - The second, is the flattening, or serialisation of a message tree to produce a set of bytes which correspond to the wire protocol of a message in the required output format and protocol.

Notes (slide 2 of 2)

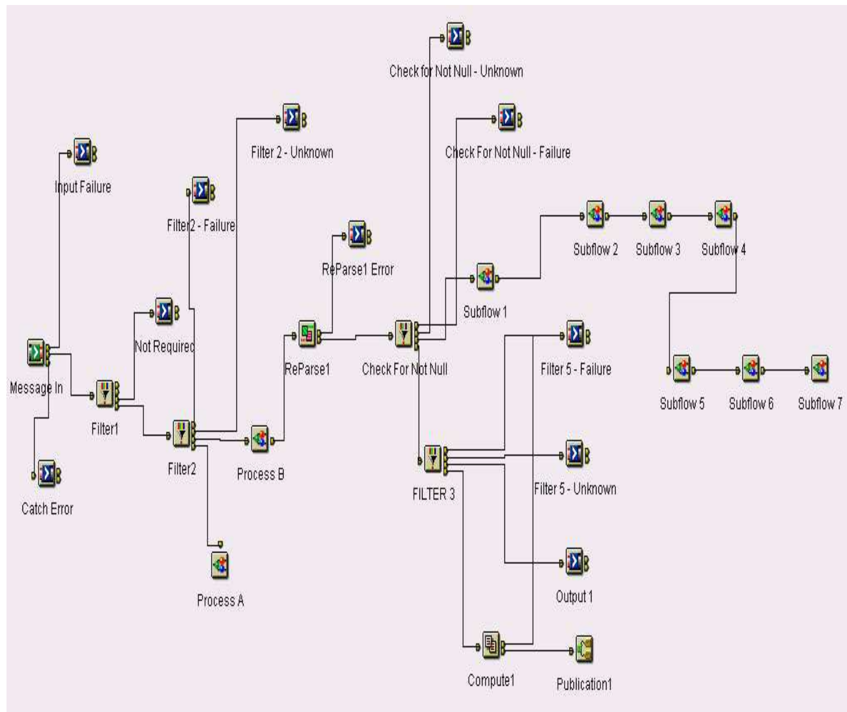
- Message Broker provides a range of parsers. The parsers are named on the foil. The parsers cover different message formats. There is some overlap – XMLNSC and MRM XML are both able to parse a generic XML message but there are also differences. MRM XML can provide validation. XMLNSC does not. Other parsers such as JMSStream are targeted at specific message formats. In this case it is a JMS stream message. Data Format Description Language (DFDL) is a modelling language from the Open Grid Forum, and is the newest parser in Message Broker. It can be used to model non-XML messages.
- Messages vary significantly in their format and the level of complexity which they are able to model. Tagged Delimited string (TDS) messages for example can support nested groups. More complex data structures make parsing costs higher. The parsing costs of different wire formats is different. You are recommended to refer to the performance reports in order to get information on the parsing costs for different types of data.

- Whatever the message format there are a number of techniques that can be employed to reduce the cost of parsing. We will now look at the most common ones. First though we will discuss the reasons for the different cost of parsing.

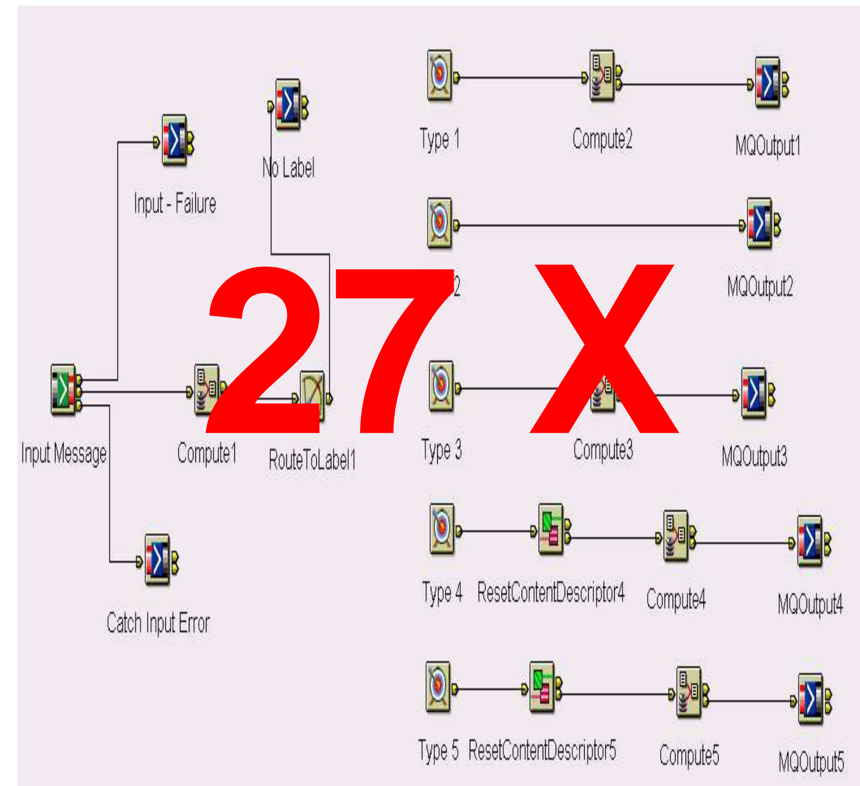
Identifying the message type quickly

SHARE
Technology · Connections · Results

- Avoid multiple parses to find the message type



VS



5 msgs/sec

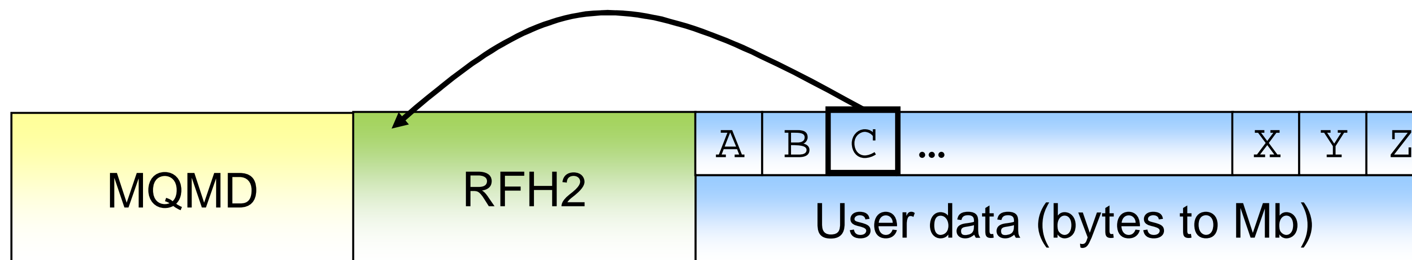
138 msgs/sec

Notes

- It is important to be able to correctly recognise the correct message format and type as quickly as possible. In message flows which process multiple types of message this can be a problem. What often happens is that the message needs to be parsed multiple times in order to ensure that you have the correct format. Dependent on the particular format and the amount of parsing needed this can be expensive.
- This example shows how a message flow which processed multiple types of input message was restructured to provide a substantial increase in message throughput.
- Originally the flow was implemented as a flow with a long critical path. Some of the most popular messages were not processed until late in the message flow resulting in a high overhead. The flow was complex. This was largely due to the variable nature of the incoming messages, which did not have a clearly defined format. Messages had to be parsed multiple times.
- The message flow was restructured. This substantially reduced the cost of processing. The two key features of this implementation were to use a very simple message set initially to parse the messages and the use of the RouteToLabel and Label nodes within the message flow in order to establish processing paths which were specialized for particular types of message.
- Because of the high processing cost of the initial flow it was only possible to process 5 non persistent message per second using one copy of the message flow. With the one copy of the restructured flow it was possible to process 138 persistent messages per second on the same system. This is a 27 times increase in message throughput plus persistent messages where now being processed whereas previously they were non persistent.
- By running five copies of the message flow it was possible to:
  - Process around 800 non persistent messages per second when running with non persistent messages.
  - Process around 550 persistent messages per second when running with persistent messages.

Parser avoidance

- If possible, avoid the need to parse at all!
  - Consider only sending changed data
  - Promote/copy key data structures to MQMD, MQRFH2 or JMS Properties
    - May save having to parse the user data
    - Particularly useful for message routing



Notes

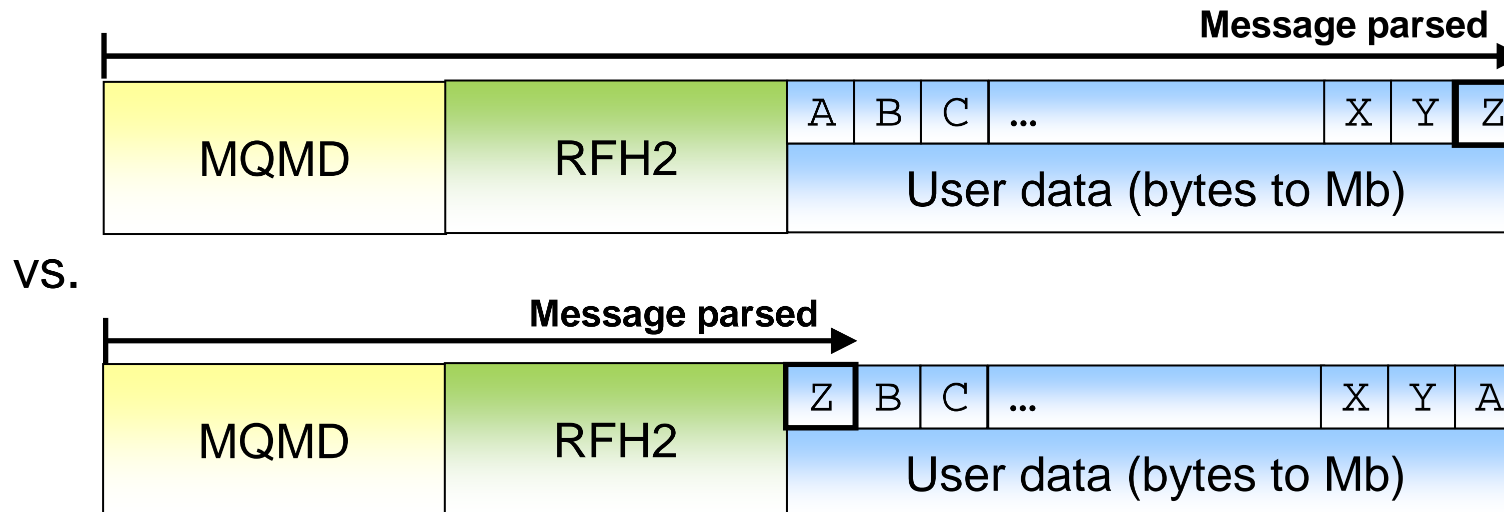
- One very effective technique to reduce the cost of parsing is not to do it. A couple of ways of doing this are discussed in this foil.
- The first approach is to avoid having to parse some parts of the message. Suppose we have a message routing message flow which needs to look at a field in order to make a routing decision. If that field is in the body of the message the body of the incoming message will have to be parsed to get access to it. The processing cost will vary dependent on which field is needed. If it is field A that is OK as it is at the beginning of the body and would be found quickly. If it is field Z then the cost could be quite different, especially if the message is several megabytes in size. A technique to reduce this cost would be to have the application which creates this message copy the field that is needed for routing into a header within the message, say in an MQRFH2 header for an MQ message or as a JMS property if it is a JMS message. If you were to do this it would no longer be necessary to parse the message body so potentially saving a large amount of processing effort. The MQRFH2 or JMS Properties folder would still need to be parsed but this is going to be smaller amount of data. The parsers in this case are also more efficient then the general parser for a message body as the structure of the header is known.

- A second approach to not parsing data is to not send it in the first place. Where two applications communicate consider sending only the changed data rather than sending the full message. This requires additional complexity in the receiving application but could potentially save significant parsing processing dependent on the situation. This technique also has the benefit of reducing the amount of data to be transmitted across the network.

Partial parsing

- Typically, the Broker parses elements up to and including the required field
  - Elements that are already parsed are not reparsed
- If possible, put important elements nearer the front of the user data

```
Set MyVariable = <Message Element Z>
```



Typical Ratio of CPU Costs	1K msg	16K msg	256K msg
Filter First	1	1	1
Filter Last	1.4	3.4	5.6

Notes (slide 1 of 2)

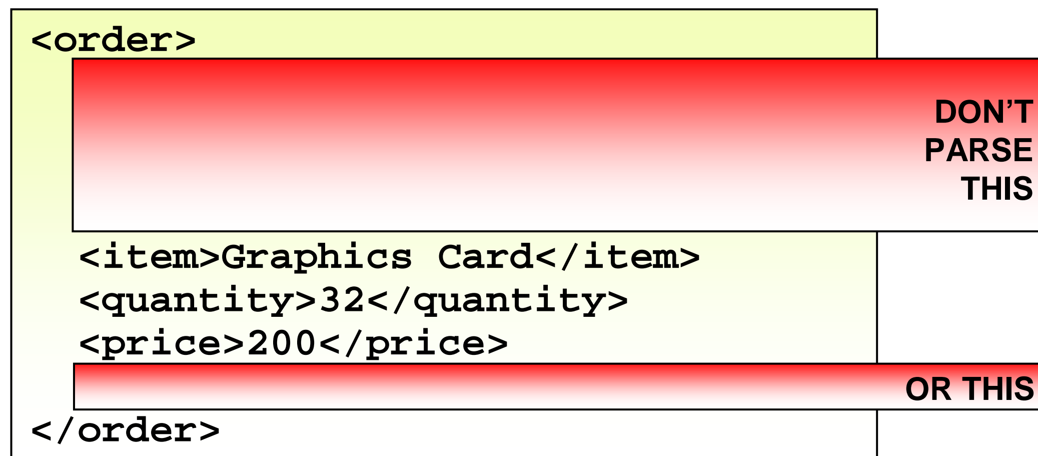
- Given parsing of the body is needed the next technique is to only parse what is needed rather than parse the whole message just in case. The parsers provided with Message Broker all support partial parsing.
- The amount of parsing which needs to be performed will depend on which fields in a message need to accessed and the position of those fields in the message. We have two example here. One with the fields ordered Z to A and the other with them ordered A to Z. Dependent on which field is needed one of the cases will be more efficient than the other. Say we need to access field Z then the first case is best. Where you have influence over message design ensure that information needed for routing for example is placed at the start of the message and not at the end of the message.
- As an illustration of the difference in processing costs consider the table in the foil. The table shows a comparison of processing costs for two versions of a routing message flow when processing several different message sizes.
  - For the filter first test the message flow routes the message based on the first field in the message. For the filter last the routing decision is made based on the last field of the message.
  - The CPU costs of running with each message size have been normalised so that the filter first test for each message represents a cost of 1. The cost of the filter last case is then given as a ratio to the filter first cost for that same message size.
  - For the 1K message we can see that by using the last field in the message to route the CPU cost of the whole message flow was 1.4 times the cost of using the first field in the same message. This represents the additional parsing which needs to be performed to access the last field.
  - For the 16K message, the cost of using the last field in the message had grown to 3.4 times that of the first field in the same message. That is we can only run at one third of the message that we can when looking at the first field of the message.
  - These measurements were taken on a pSeries 570 Power 5 with 8 * 1.5 GHZ processors, using WebSphere Message Broker V6.

Notes (slide 2 of 2)

- For the 256K message, the cost of using the last field in the message was 5.6 times that of the first field in the same message. That is we can only run at one fifth of the message that we can when looking at the first field of the message.
- You can see how processing costs quickly start to grow and this was a simple test case. With larger messages the effect would be even more pronounced. Where the whole message is used in processing this is not an issue as all of the data will need to be parsed at some point. For simple routing cases though the ordering of fields can make a significant difference.
- When using ESQL, and Mapping nodes the field references are typically explicit. That is we have references such as InputRoot.Body.A. Message Broker will only parse as far as the required message field to satisfy that reference. It will stop at the first instance. When using XPath, which is a query language the situation is different. By default an XPath expression will search for all instances of an element in the message which implicitly means a full parse of the message. If you know there is only one element in a message there is the chance to optimise the XPath query to say only retrieve the first instance. See the ESQL and Java coding tips later in the presentation for more information.

Opaque parsing

- Treat elements of an XML document as an unparsed BLOB
- Reduces message tree size and parsing costs
- Cannot reference the sub tree in message flow processing
- Configured on input nodes (“Parser Options” tab)

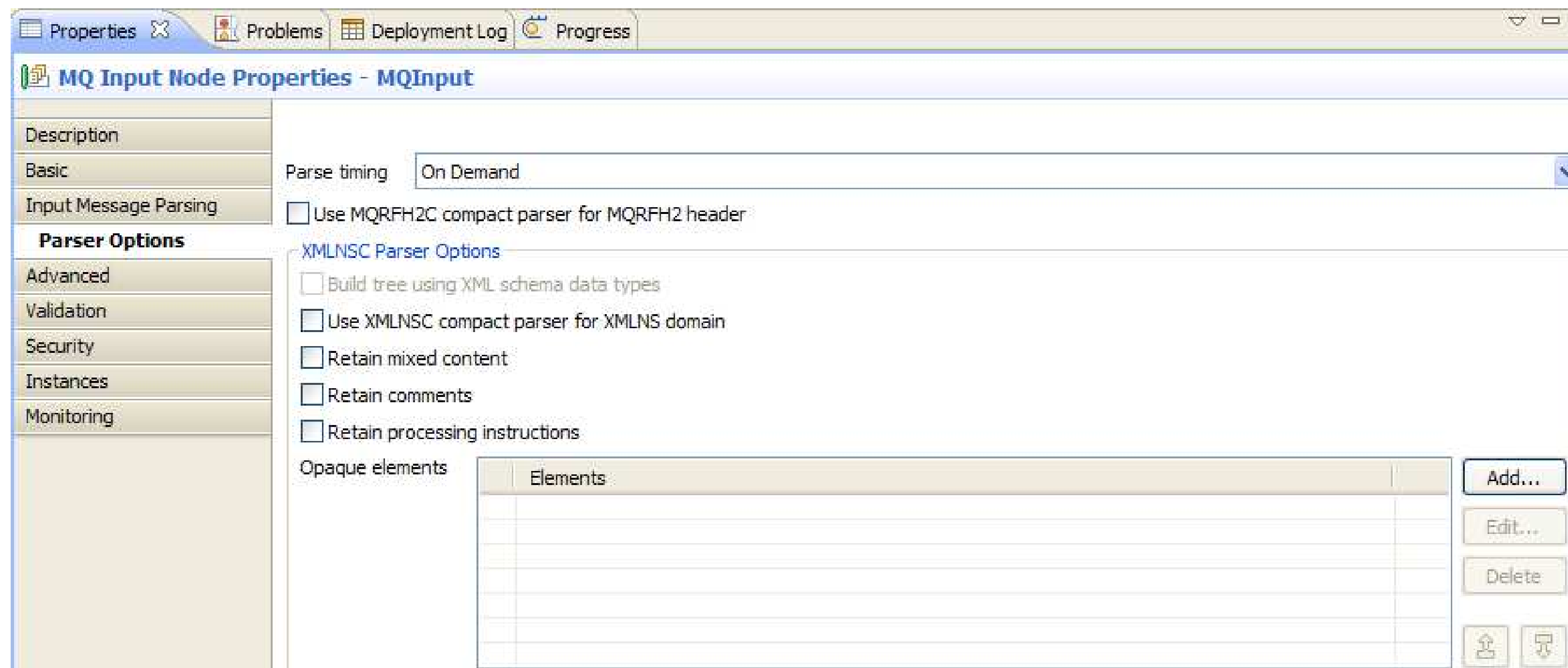


Notes (slides 1 of 2)

- Opaque parsing is a technique that allows the whole of an XML sub tree to placed in the message tree as a single element. The entry in the message tree is the bit stream of the original input message. This technique has two benefits:
  - It reduces the size of the message tree since the XML sub tree is not expanded into the individual elements.
  - The cost of parsing is reduced since less of the input message is expanded as individual elements and added to the message tree.
- Use opaque parsing where you do not need to access the elements of the sub tree, for example you need to copy a portion of the input tree to the output message but may not care about the contents in this particular message flow. You accept the content in the sub folder and have no need to validate or process it in any way.
- Opaque parsing is supported for the XMLNS and XMLNSC domains only.
  - The support for the XMLNS domain was introduced in Message Broker V6 fixpack 1 and was an early version of opaque parsing and as such could be changed in the future.
  - The support for the XMLNSC domain was added in Message Broker V6.1. It has a different interface from the XMLNS domain to specify the element names which are to be opaquely parsed.
- The support for the XMLNS domain continues to be available in its original form. Indeed this is the only way in which the messages in the XMLNS domain can be opaquely parsed. It is not possible to use the same interface that is provided for the XMLNSC to specify element names for the XMLNS domain.
- The following pages of notes explain how opaque parsing is performed for messages in the XMLNS and XMLNSC domains.

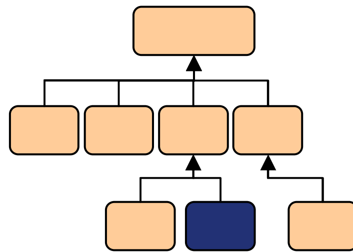
Notes (slides 2 of 2)

- The message domain must be set to XMLNSC.
- The elements in the message which are to be opaquely parsed are specified on the 'Parser Options' page of the input node of the message flow. See the picture below of the MQInput node.
- Enter those element names that you wish to opaquely parse in the 'Opaque Elements' table.
- Be sure not to enable message validation as it will automatically disable opaque parsing. Opaque parsing in this case does not make sense since for validation the whole message must be parsed and validated.
- Opaque parsing for the named elements will occur automatically when the message is parsed.
- It is not currently possible to use the CREATE statement to opaquely parse a message in the XMLNSC domain.



Navigation

- The logical tree is walked every time it is evaluated
  - This is not the same as parsing!



```
SET Description =
        Root.Body.Level1.Level2.Level3.Description.Line[1];
```

- Long paths are inefficient
  - Minimise their usage, particularly in loops
  - Use reference variables/pointers (ESQL/Java)
  - Build a smaller message tree if possible
    - Use compact parsers (XMLNSC, MRM XML, RFH2C)
    - Use opaque parsing

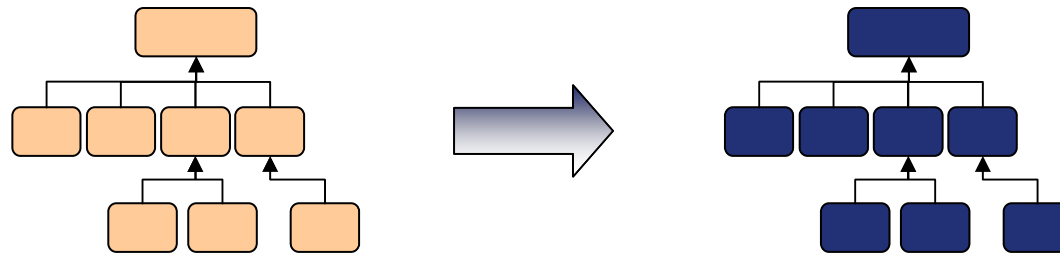
Notes

- Navigation is the process of accessing elements in the message tree. It happens when you refer to elements in the message tree in the Compute, JavaCompute, PHP, .NET and Mapping nodes.
- The cost of accessing elements is not always apparent and is difficult to separate from other processing costs.
- The path to elements is not cached from one statement to another. If you have the ESQL statement

  SET Description = Root.Body.Level1.Level2.Level3.Description.Line[1]; the broker runtime will access the message tree starting at the correlation Root and then move down the tree to Level1, then Level2, then Level3, then Description and finally it will find the first instance of the Line array. If you have this same statement in the next line of your ESQL module exactly the same navigation will take place. There is no cache to the last referenced element in the message tree. This is because access to the tree is intended to be dynamic to take account of the fact that it might change structure from one statement to another. There are techniques available though to reduce the cost of navigation.
- With large message trees the cost of navigation can become significant. There are techniques to reduce which you are recommended to follow:
  - Use the compact parsers (XMLNSC, MRM XML and RFH2C). The compact parsers discard comments and white space in the input message. Dependent on the contents of your messages this may have an effect of not. By comparison the other parsers include data in the original message, so white space and comments would be inserted into the message tree.
  - Use reference variables if using ESQL or reference pointers if using Java. This technique allows you to save a pointer to a specific place in the message tree. Say to Root.Body.Level1.Level2.Level3.Description for example.
- For an example of how to use reference variables and reference pointers see the coding tips given in the Common Problems section.

Message tree copying

```
SET OutputRoot.XML.A = InputRoot.XML.A;
```

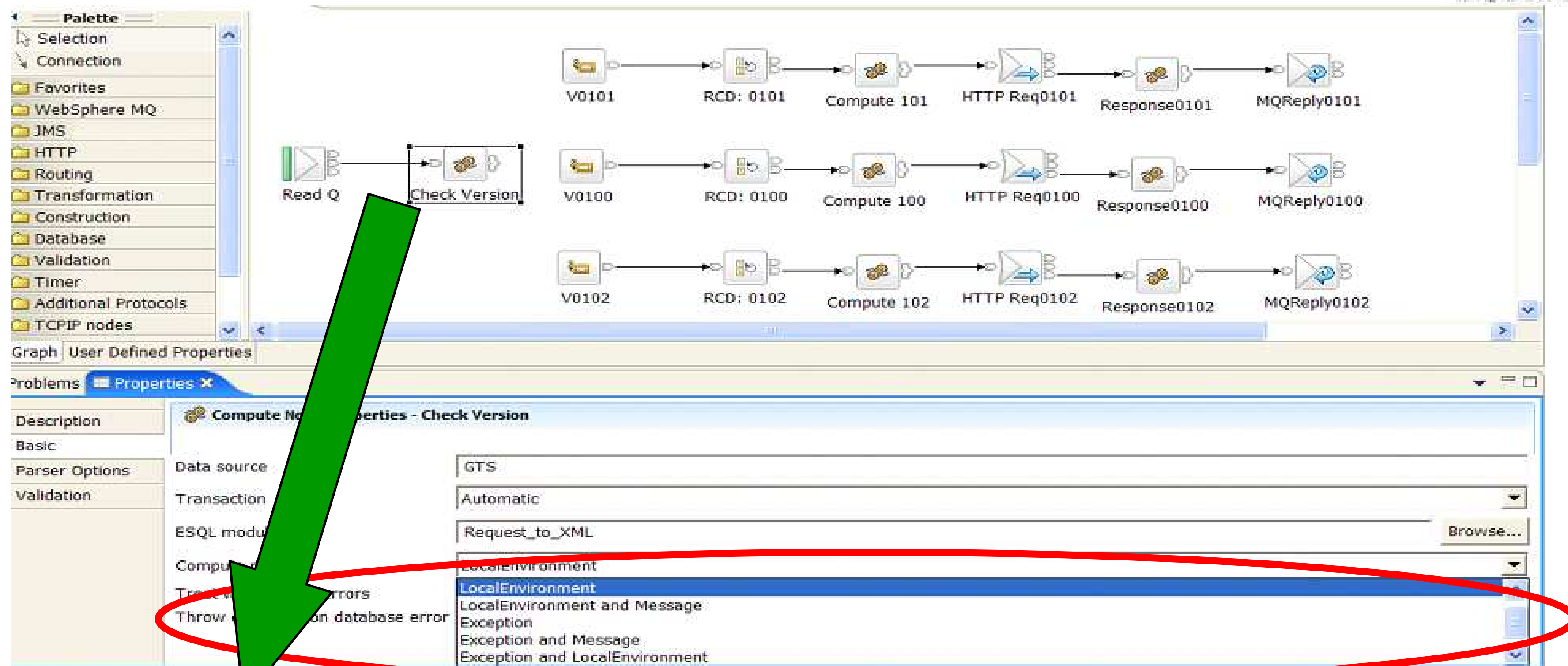


- Message tree copying causes the logical tree to be duplicated in memory… and this is computationally expensive
- Reduce the number of times the tree is copied
  - Reduce the number of *Compute* and *JavaCompute* nodes in a message flow
  - See if “Compute mode” can be set to not include “message”
  - Copy at an appropriate level in the tree (copy once rather than for multiple branch nodes)
  - Copy data to the Environment (although changes are not backed out)
- Minimize the effect of tree copies
  - Produce a smaller message tree (again)!

Notes

- Tree Copying is the process of copying the message tree either in whole or part.
- Copying occurs in nodes which are able to change the message tree such as Compute nodes. A copy of the message tree is taken for recovery reasons so that if a compute node makes changes and processing in node incurs or generates an exception the message tree can be recovered to a point earlier in the message flow. Without this a failure in the message flow downstream could have implications for a different path in the message flow. The tree copy is a copy of a structured object and so is relatively expensive. It is not a copy of a sequence of bytes. For this reason it is best to minimize the number of such copies, hence the general recommendation to minimize the number of compute nodes in a message flow. Tree copying does not take place in a Filter node for example since ESQL only references the data in the message tree and does not update it.
- There are a few ways to reduce the costs of tree copying. These are:
  - Produce a smaller message tree in the first place. A smaller tree will cost less to copy. Ways to achieve this are to use Smaller messages, use Compact Parsers (XMLNSC, MRM XML, RFH2C), use Opaque parsing partial parsing (all discussed previously).
  - Reduce the number of times the whole tree is copied. Reducing the number of compute nodes (ESQL or Java) will help to reduce the number of times that the whole tree needs to be copied. In particular avoid situations where you have one compute node followed immediately by another. In many cases you may need multiple computes across a message flow. What we want to do is to optimise the processing not have to force everything into one compute node.
  - Copy portions of the tree at the branch level if possible rather than copying individual leaf nodes. This will only work where the structure of the source and destination are the same but it is worth doing if possible.
  - Copy data to the Environment correlation (remember it is not backed out) and work with it in Environment. This way the message tree does not have to be copied every time you run a compute node. Environment is a scratchpad area which exists for each invocation of a message flow. The contents of Environment are accessible across the whole of the message flow. Environment is cleared down at the end of each message flow invocation.
    - The first compute node in a message flow can copy InputRoot to Environment. Intermediate nodes then read and update values in the Environment instead of using the InputRoot and OutputRoot correlations as previously.
    - In the final compute node of the message flow OutputRoot must be populated with data from Environment. The MQOutput node will then serialize the message as usual. Serialization will not take place from Environment.
  - Whilst use of the Environment correlation is good from a performance point of view be aware that the any updates made by a node which subsequently generates an exception will remain in place. There is no back-out of changes as there is when a message tree copy was made prior to the node exception

Example of avoiding a tree copy




```
-- Set the version numbers this flow supports
SET Environment.SupportedVersions = '0102 0101 0100';

-- getMessageVersion returns label name for the message version (for example, '0102')
PROPAGATE TO LABEL 'V' || getMessageVersion(Environment, InputRoot, TRUE);

-- PROPAGATE already passed the message; do not propagate to "Out"
RETURN FALSE;
```

Notes

- This foil shows one case where it was possible to avoid a tree copy. You may be able to use this same technique in other situations.
- The purpose of the message flow is to read MQ messages, determine the message type and route to the appropriate type of processing. There are three legs of processing, each starting with a Label node (V0100, V0101 & V0102).
- The compute node Check_Version issues a PROPAGATE to the appropriate label node dependent on a field in the incoming message. The message version information is obtained from the incoming message by a function called getMessageVersion. The ESQL on the foil shows how this is done.
- As processing at this point is limited to routing only there is no need for Check_Version to modify the incoming message tree and as such a value of LocalEnvironment is sufficient for the Compute Mode of the node. Had a value of LocalEnvironment and Message been selected then we would have needed to copy the message tree which would have increased processing costs, unnecessarily in this case.

Resource Access

- **MQ**
  - Tune the QM and associated QM's, logs, buffer sizes
  - Consider running as a trusted application (but be aware of the risks)
  - Avoid unnecessary use of persistent messages, although always use persistent messages as part of a co-ordinated transaction
  - Use fast storage if using persistent messages
  - http://www.ibm.com/developerworks/websphere/library/techarticles/0712_dunn/0712_dunn.html

- **JMS**
  - Follow MQ tuning if using MQ JMS
  - Follow provider instructions
  - http://www.ibm.com/developerworks/websphere/library/techarticles/0604_bicheno/0604_bicheno.html

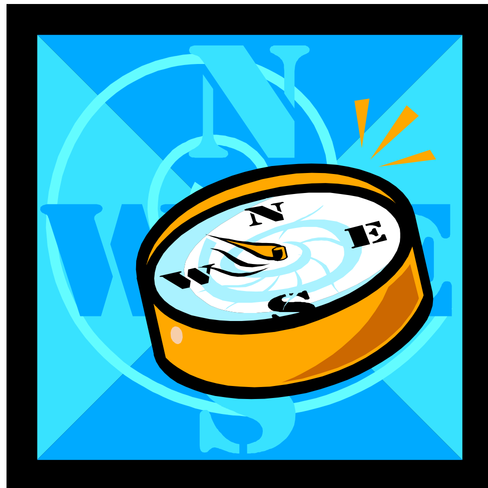
- **SOAP/HTTP**
  - Use persistent HTTP connections
  - Use embedded execution group-level listener for HTTP nodes (V7.0.0.1)
  - http://www.ibm.com/developerworks/websphere/library/techarticles/0608_braithwaite/0608_braithwaite.html

- **File**
  - Use cheapest delimiting option
  - Use fast storage

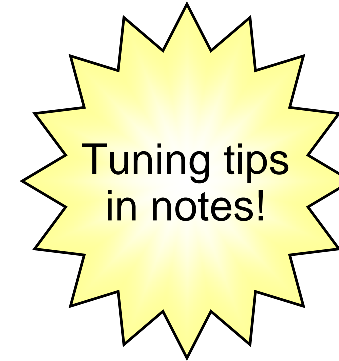
Notes

- There are a wide variety of ways of reading data into Message Broker and sending it out once it is has been processed. The most common methods are MQ messages, JMS messages, HTTP and files. Collectively for this discussion lets refer to them as methods of transport although we recognise that is pushing it where file, VSAM and database are concerned.
- Different transports have different reliability characteristics. An MQ persistent message has assured delivery. It will be delivered once and once only. MQ non persistent messages are also pretty reliable but are not assured. If there is a failure of machine or power outage for example data will be lost with these non persistent messages. Similarly data received over a raw TCP/IP session will be unreliable. With WebSphere MQ Real-time messages can be lost if there is a buffer overrun on the client or server for example.
- In some situations we are happy to accept the limitations of the transport in return for its wide availability or low cost for example. In others we may need to make good the shortcoming of the transport. One example of this is the use of a WebSphere MQ persistent message (or database insert) to reliably capture coming over a TCP/IP connection as the TCP/IP session is unreliable – hey this is why MQ is popular because of the value it builds on an unreliable protocol.
- As soon as data is captured in this way the speed of processing is changed. We now start to work at the speed of MQ persistent messages or the rate at which data can be inserted into a database. This can result in a significant drop in processing dependent on how well the queue manager or database are tuned.
- In some cases it is possible to reduce the impact of reliably capturing data if you are prepared to accept some additional risk. If it is acceptable to have a small window of potential loss you can save the data to an MQ persistent message or database asynchronously. This will reduce the impact of such processing on the critical path. This is a decision only you can make. It may well vary by case by case dependent on the value of the data being processed.
- It is important to think how data will be received into Message Broker. Will data be received one message at a time or will it arrive in a batch. What will be involved in session connection set-up and tear-down costs for each message or record that is received by the message flow. Thinking of it in MQ terms will there be an MQCONN, MQOPEN, MQPUT, MQCMT and MQDISC by the application for each message sent to the message flow. Lets hope not as this will generate lot of additional processing for the broker queue manager [assuming the application is local to the broker queue manager]. In the HTTP world you want to ensure that persistent sessions (not to be confused with MQ persistence) are used for example so that session set-up and tear-down costs are minimised. Make sure that you understand the sessions management behaviour of the transport that is being used and that you know how to tune the transport for maximum efficiency.
- Message Broker give you the ability to easily switch transport protocols, so data could come in over MQ and leave the message flow in a file or be sent as a JMS message to any JMS 1.1 provider. In your processing you should aim to keep a separation between the transport used and the data being sent over that transport. This will make it easier to use different transports in the future.

Processing Logic



- Various options available
  - Java
  - ESQL
  - PHP
  - .NET
  - Mapping
  - XSL
  - WebSphere TX (as a separate product)



- Performance characteristics of the different options is rarely an issue
  - Choose an option based on skills, ease-of-use and suitability for the scenario
- Think carefully before mixing transformation options in the same message flow

Notes

- There are a number of different transformation technologies available with Message Broker V7. Theses are:
  - ESQL, an SQL like syntax, which runs in nodes such as the Compute, Filter and Database.
  - Java which runs in a JavaCompute node.
  - The Mapping node. The mapping capability was rewritten in V6. It gives a spreadsheet style format in which you can specify the connection of source and destination fields along with some additional processing.
  - Extensible Stylesheets (XSL) running in an XML Transformation node. This provides a good opportunity to reuse an existing asset.
  - PHP which runs in the PHPCompute node.
  - The .NET compute node routes or transforms messages by using any Common Language Runtime (CLR) compliant .NET programming language, such as C#, Visual Basic (VB), F# and C++/CLI (Common Language Infrastructure).
  - The WTX node which runs a WTX map to parse the message.
- As there is a choice of transformation technology this offers significant flexibility. All of these technologies can be used within the same message flow if needed.

- Different technologies will suit different projects and different requirements.

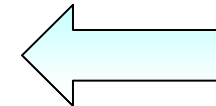
- Factors which you may want to take into account when making a decision about the use of transformation technology are:
  - The pool of development skills which is available. If developers are already skilled in the use of ESQL you wish continue using this as the transformation technology of choice. If you have a strong Java skills then the JavaCompute node may be more relevant, strong C# skills will make the .NET node more relevant, rather than having to teach your programmers ESQL.
  - The ease with which you could teach developers a new development language. If you have many developers it may be not be viable to teach all of the ESQL for example and you may only educate a few and have them develop common functions or procedures.
  - The skill of the person. Is the person an established developer, or is the mapping node more suitable.
  - Asset re use. If you are an existing user of Message Broker you may have large amounts of ESQL already coded which you wish to re-use. You may also have any number of resources that you wish to re-use such as Java classes for key business processing, stylesheets, PHP scripts, .NET applications or WTX mappings. You want to use these as the core of message flow processing and extend the processing with one of the other technologies.
  - Performance is rarely an issue.
  - Think carefully before combining a lot of different transformation technologies within a path of execution of a message flow. ESQL, Java, PHP, .NET and Mapping node will happily intermix but there is an overhead in moving to/from other technologies such as XSL as they do not access the message directly. The message has to be serialised when it is passed to those other technologies and then the results parsed to build a new message tree.

Agenda

- What are the main performance costs in message flows?

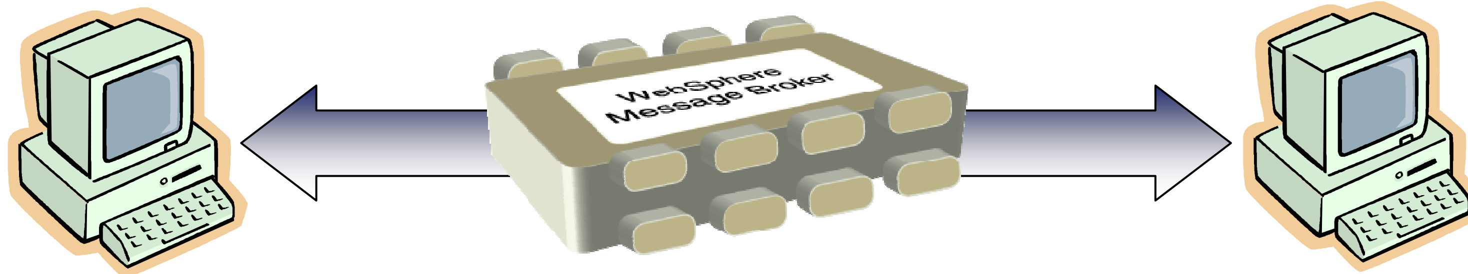
  …and what are the best practices that will minimize these costs?

- What other performance considerations are there?



- Understanding your broker's behaviour

Consider the end-to-end flow of data

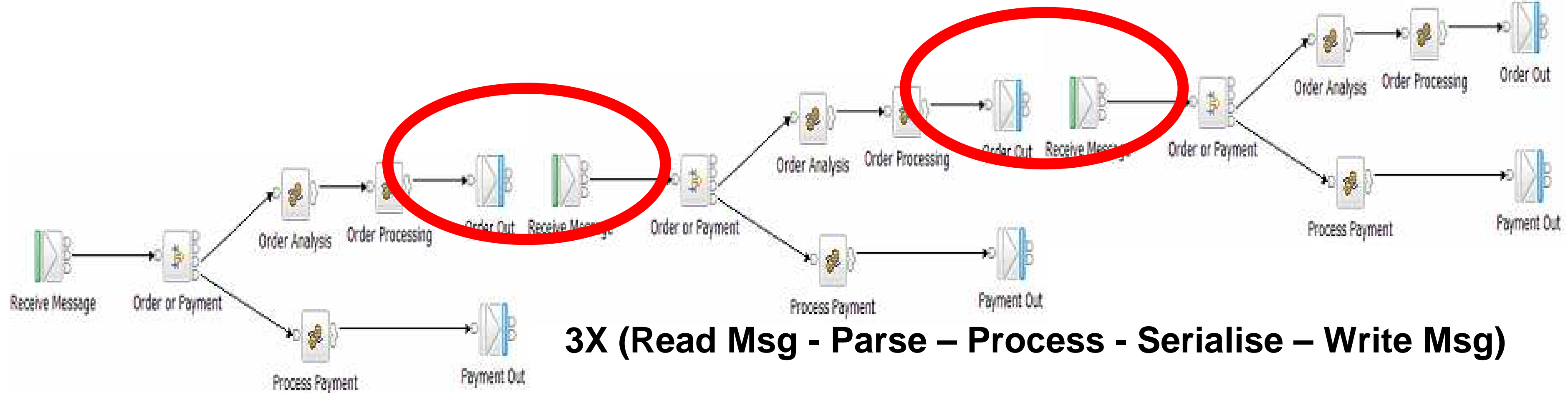


- Interaction between multiple applications can be expensive!
  - Store and Forward
  - Serialise … De-serialise

- Some questions to ponder:
  - Where is data being generated? Where is data being consumed?
  - What applications are interacting in the system?
  - What are the interactions between each of these applications?

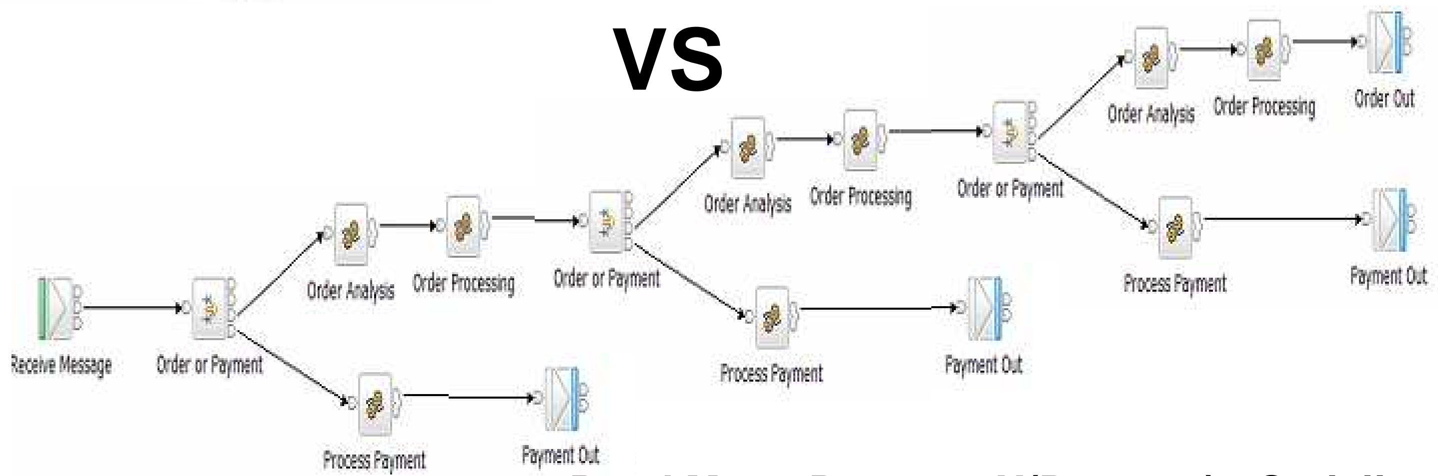
- And as data passes through the broker:
  - Where are messages arriving? Where are they leaving?
  - Are multiple message flows being invoked?
    - Request/Reply?
    - Flow-to-flow?

Consider the sequencing of message flows



3X (Read Msg - Parse – Process - Serialise – Write Msg)

VS



Read Msg - Parse – 3X(Process) - Serialise – Write Msg

Notes (slide 1 of 2)

- It is important to think about the structure of your message flows and to think about how they will process incoming data. Will one message flow process multiple message types or will it process a single type only. Where a unique message flow is produced for each different type of message it is referred to as a specific flow. An alternative approach is to create message flows which are capable of processing multiple message types. There may be several such flows, each processing a different group of messages. We call this a generic flow.
- There are advantages and disadvantages for both specific and generic flows.
  - With specific flows there are likely to be many different message flows. In fact as many message flows as there are types of input message. This does increase the management overhead which is a disadvantage. A benefit of this approach though is that processing can be optimized for the message type. The message flow does not have to spend time trying to determine the type of message before then processing it.
  - With the generic flow there is a processing cost over and above the specific processing required for that type of message. This is the processing required to determine which of the possible message types a particular message is. Generic flows can typically end up parsing an input message multiple times. Initially to determine the type and then subsequently to perform the specific processing for that type of message. A benefit of the generic flow is that there are likely to be fewer of them as each is able to process multiple message types. This makes management easier. Generic flows tend to be more complex in nature. It is relatively easy to add processing for another type if message though. With generic message flows it is important to ensure that the most common message types are processed first. If the distribution of message types being processed changes over time this could mean processing progressively becomes less efficient. You need to take account of this possibility in you message flow design. Use of the RouteToLabel node can help with this problem.
- There are advantages to both the specific and generic approaches. From a message throughput point of view it is better to implement specific flows. From a management and operation point of view it is better to use generic flows. Which approach you choose will depend on what is important in your own situation.
- To bring clarity to your message flow processing consider using the RouteToLabel node. It can help add structure and be effective in separating different processing paths with a message flow. Without flows can become long and unwieldly resulting in high processing costs for those messages with have a long processing path. See the two examples on the foil. Both perform the same processing but have different structures. One is clearly more readable than the other.
- There are several ways of achieving code reuse with Message Broker. They are subflows, ESQL procedures and functions, invoking external Java classes and calling a database stored procedure. There are some performance implications of using the different techniques which we will briefly look at.

Notes (slide 2 of 2)

- Subflows are a facility to encourage the reuse of code. Subflows can be embedded into message flows or deployed separately in v8. There are no additional nodes inserted into the message flow as a result of using subflows. However be aware of implicitly adding extra nodes into a message flow as a result of using subflows. in some situations compute nodes are added to subflows to perform marshalling of data from one part of the message tree into a known place in the message tree so that the data can be processed by the message flow. The result may then be copied to another part of the message tree before the subflow completes. This approach can easily lead to the addition of two compute nodes, each of which performs a tree copy. In such cases the subflow facilitates the reuse of logic but unwittingly adds an additional processing head. Consider the use of ESQL procedures instead for code reuse. They provide a good reuse facility and the potential additional overhead of extra Compute nodes is avoided.
- Existing Java classes can be invoked from within a message flow using either ESQL or Java. The same is true for .NET applications which can also be called from ESQL in V8.
  - From ESQL it is possible to declare an external function or procedure which specifies the name of the Java routine to be invoked.. Parameters (in,out, in-out) can be passed to the Java routine.
  - From within a Java compute node the class can be invoked as any other Java class. You will need to ensure that the Java class file is accessible to the broker runtime for this to work.
- A database stored procedure can be called from within ESQL by using an external procedure. It is also possible to invoke a stored procedure using a PASSTHRU statement but this is not recommended from a performance point of view.
- The cost of calling these external routines will vary and be dependent on what is being called (Java VS Stored procedure) and the number and type of parameters. The performance reports give the CPU cost of calling different procedure types and you can compare the cost of using these methods to other processing which you might do.
- There are no hard and fast rules about whether to call an external procedure or not. It will very much depend on what code you want to access. It is important to bear in mind the cost of making the call against the function that is being executed. Is it really worth it if the external procedure is only a few lines of code. It might easier to re-write the code and make it available locally within the message flow.
- Database stored procedures can be a useful technique for reducing the size of the result set that is returned to a message flow from a database query. Especially where the result set from an SQL statement might contain hundreds or thousands of rows otherwise. The stored procedure can perform an additional refinement of the results. The stored procedure is close to the data and using it to refine results is a more efficient way of reducing the volume of data. Otherwise data has to be returned to the message flow only to be potentially discarded immediately.

Consider the use of patterns (V7)

- Patterns provide top-down, parameterized connectivity of common use cases
  - e.g. Web Service façades, Message oriented processing, Queue to File…
- They help describe best practice for efficient message flows
  - There are also quick to create and less prone to errors
- Write your own patterns too!
  - Help new message flow developers come on board more quickly

Broker Application Development - Pattern Specification - WebSphere Message Broker Toolkit - Message Broker

File Edit Navigate Search Project Run Window Help

Broker Development | Patterns Explorer | Pattern specification

- Patterns
  - Application Integration
    - SAP
      - MQ one-way (IDoc)
  - File Processing
    - Record Distribution
      - MQ one-way
  - Matt Patterns
    - MyPattern
  - Message-based Integration
    - Message Correlator
      - MQ request-response with persistence
      - MQ request-response without persistence
    - Message Splitter
      - MQ one-way (XML)
  - Service Enablement
    - Service Access
      - MQ one-way
    - Service Facade
      - MQ one-way with acknowledgment
      - MQ request-response
  - Service Virtualization
    - Service Proxy
      - Static endpoint

Outline | Data Sourc | Tasks | Brokers

- Brokers
  - MB7BROKER
    - default

View Pattern Specification

View information about the selected pattern and click the "Create New Instance" button or click here to start using a pattern.

Service Proxy: static endpoint pattern

Use this pattern to provide decoupling between Web service requesters and Web service providers b
virtual service that is bound directly to the target service provider.

Use this pattern:

- When you do not want the client to access the service directly, either because the address of tl
  the service is located might change, or because it must be hidden to control access to the servi
- To support service access over HTTP or HTTPS
- When you want to provide logging of all requests to the service provider, or when you want to p
  handling capabilities without making changes to the service provider

Solution

The solution is to implement a message flow that provides a proxy for the provider Web service and
logging and error handling capabilities.

Service Requester | Service Proxy: static endpoint | Set provider address | Service Provider

Consider your operational environment

- Network topology
  - Distance between broker and data

- What hardware do I need?
  - Including high availability and disaster recovery requirements
  - IBM can help you!

- Transactional behaviour
  - Early availability of messages vs. backout
  - Transactions necessitate log I/O – can change focus from CPU to I/O
  - Message batching (commit count)

- What are your performance requirements?
  - Now… and in five years…

Consider how your message flows will scale

- How many **execution groups**?
  - As processes, execution groups provide isolation
  - Higher memory requirement than threads
  - Rules of thumb:
    - One or two execution groups per application
    - Allocate all instances needed for a message flow over those execution groups
    - Assign heavy resource users (memory) to specialist execution groups

- How many **message flow instances**?
  - Message flow instances provide thread level separation
  - Lower memory requirement than execution groups
  - Potential to use shared variables
  - Rules of thumb:
    - There is no pre-set magic number
    - What message rate do you require?
    - Try scaling up the instances to see the effect; to scale effectively, message flows should be efficient, CPU bound, have minimal I/O and have no affinities or serialisation.

Notes (slide 1 of 10)

- So far we have focused on the message and contents of the message flow. It is important to have an efficient message flow but this is not the whole story. It is also important to have an efficient environment for the message flow to run in.
- We collectively refer to this as the broker environment. It includes:
  - Configuration of the broker
  - Configuration of the broker queue manager
  - The topology of the environment in which the messages are processed
  - Business database configuration
  - Hardware
  - Software
- It is important that each of these components is well configured and tuned. When this occurs we can get the best possible message throughput for the resources which have been allocated.
- We will now briefly look at each of these aspects. The aim is to outline those aspects which you should be aware of and further investigate rather than to provide full guidance in this presentation.

Notes (slide 2 of 10)

- As with other MQ applications it is possible to run a broker as a trusted application on Windows, Solaris, HP-UX and AIX. The benefit of running in trusted mode is that the cost of an application communicating with the queue manager is reduced and MQ operations are processed more efficiently. It is thus possible to achieve a higher level of messaging with a trusted application. However when running in trusted mode it is possible for a poorly written application to crash and potentially corrupt the queue manager. The likelihood of this happening will depend on user code, such as a plugin node, running in the message flow and the extent to which it has been fully tested. Some users are happy to accept the risk in return for the performance benefits. Others are not.
- Using the MBExplorer it is possible to increase the number of instances of a message flow which are running. Similarly you can increase the number of execution groups and assign message flows to those execution groups. This gives the potential to increase message throughput as there are more copies of the message flow. There is no hard and fast rule as to how many copies of a message flow to run. For guidelines look at the details in the Additional Information section
- Tuning is available for non MQ transports. For example:
  - When using HTTP you can tune the number of threads in the HTTP listener
  - When using JMS nodes follow the tuning advice for the JMS provider that you are connecting to.
- As the broker is dependent on the broker queue manager to get and put MQ messages the performance of this component is an important component of overall performance.
- When using persistent messages it is important to ensure the queue manager log is efficient. Review log buffer settings and the speed of the device on which the queue manager log is located.
- It is possible to run the WebSphere queue manager channels and listener as trusted applications. This can help reduce CPU consumption and so allow greater throughput.
- In practice the broker queue manager is likely to be connected to other queue managers. It is important to examine the configuration of these other queue managers as well in order to ensure that they are well tuned.

Notes (slide 3 of 10)

- The input messages for a message flow do not magically appear on the input queue and disappear once they have been written to the output queue by the message flow. The messages must some how be transported to the broker input queue and moved away from the output queue to a consuming application.
- Let us look at the processing involved to process a message in a message flow passing between two applications with different queue manager configurations.
  - Where the communicating applications are local to the broker, that this they use the same queue manager as the broker, processing is optimized. Messages do not have to move over a queue manager to queue manager channel. When messages are non persistent no commit processing is required. When messages are persistent 3 commits are required in order to pass one message between the two applications. This involves a single queue manager.
  - When the communicating applications are remote from the broker queue manager, messages passed between the applications must now travel over two queue manager to queue manager channels (one on the outward, another on the return). When messages are non persistent no commit processing is required. When messages are persistent 7 commits are required in order to pass one message between the two applications. This involves two queue managers.
- If the number of queue managers between the applications and broker where to increase so would the number of commits required to process a message. Processing would also become dependent on a greater number of queue managers.
- Where possible keep the applications and broker as close as possible in order to reduce the overall overhead.
- Where multiple queue managers are involved in the processing of messages it is important to make sure that all are optimally configured and tuned.

Notes (slide 4 of 10)

- The extent to which business data is used will be entirely dependent on the business requirements of the message flow. With simple routing flows it is unlikely that there will be any database access. With complex transformations there might be involved database processing. This could involve a mixture of read, update, insert and delete activity.
- It is important to ensure that the database manager is well tuned. Knowing the type of activity which is being issued against a database can help when tuning the database manager as you can focus tuning on the particular type of activity which is used.
- Where there is update, insert or delete activity the performance of the database log will be a factor in overall performance. Where there is a large amount of read activity it is important to review buffer allocations and the use of table indices for example.
- It might be appropriate to review the design of the database tables which are accessed from within a message flow. It may be possible to combine tables and so reduce the number of reads which are made for example. All the normal database design rules should be considered.
- Where a message flow only reads data from a table, consider using a read only view of that table. This reduces the amount of locking within the database manager and reduces the processing cost of the read.

Notes (slide 5 of 10)

- Allocating the correct resources to the broker is a very important implementation step.
- Starve a CPU bound message flow of CPU and you simply will not get the throughput. Similarly neglect I/O configuration and processing could be significantly delayed by logging activities for example.
- It is important to understand the needs of the message flows which you have created. Determine whether they are CPU bound or I/O bound.
- Knowing whether a message flow is CPU bound or I/O bound is key to knowing how to increase message throughput. It is no good adding processors to a system which is I/O bound. It is not going to increase message throughput. Using disks which are much faster such as a solid state disk or SAN with a fast-write non-volatile cache will have a beneficial effect though.
- As we saw at the beginning Message Broker has the potential to use multiple processors by running multiple message flows as well as multiple copies of those message flows. By understanding the characteristics of the message flow it is possible to make the best of that potential.
- In many message flows parsing and manipulation of messages is common. This is CPU intensive activity. As such message throughput is sensitive to processor speed. Brokers will benefit from the use of faster processors. As a general rule it would be better to allocate fewer faster processors than many slower processors.
- We have talked about ensuring that the broker and its associated components are well tuned and it is important to emphasize this. The default settings are not optimized for any particular configuration. They are designed to allow the product to function not to perform to its peak.
- It is important to ensure that software levels are as current as possible. The latest levels of software such as Message Broker and WebSphere MQ offer the best levels of performance.

Notes (slide 6 of 10)

- Avoid use of array subscripts [ ] on the right hand side of expressions – use reference variables instead. See below. Note also the use of LASTMOVE function to control loop execution.

```
DECLARE myref REFERENCE TO OutputRoot.XML.Invoice.Purchases.Item[1];
-- Continue processing for each item in the array
WHILE LASTMOVE(myref)=TRUE DO
-- Add 1 to each item in the array
SET myref = myref + 1;
-- Move the dynamic reference to the next item in the array
MOVE myref NEXTSIBLING;
END WHILE;
```

- Use reference variables rather than long correlation names such as InputRoot.MRM.A.B.C.D.E. Find a part of the correlation name that is used frequently so that you get maximum reuse in the ESQL. You may end up using multiple different reference variables throughout the ESQL. You do not want to reference variable pointing to InputRoot for example. It needs to be deeper than that.

```
DECLARE myref REFERENCE TO InputRoot.MRM.A.B.C;
SET VALUE = myRef.D.E;
```

- Avoid use of EVAL, it is very expensive!
- Avoid use of CARDINALITY in a loop e.g. `while ( I < CARDINALITY (InputRoot.MRM.A.B.C[]))`. This can be a problem with large arrays where the cost of evaluating CARDINALITY is expensive and as the array is large we also iterate around the loop more often. See use of reference variables and LASTMOVE above.
- Combine ESQL into the minimum number of compute nodes possible. Fewer compute nodes means less tree copying.
- Use new FORMAT clause (in CAST function) where possible to perform data and time formatting.
- Limit use of shared variables to a small number of entires (tens of entries) when using an array of ROW variables or order in probability of usage (current implementation is not indexed so performance can degrade with higher numbers of entries).
- For efficient code re-use consider using ESQL modules and schema rather than subflows. What often happens with subflows is that people add extra compute nodes to perform initialisation and finalisation for the processing that is done in the subflow. The extra compute nodes result in extra message tree copying which is relatively expensive as it is a copy of a structured object.

Notes (slide 7 of 10)

- Avoid use of PASSTHRU with a CALL statement to invoke a stored procedure. Use the "CREATE PROCEDURE ... EXTERNAL ..." and "CALL ..." commands instead
- If you do have to use PASSTHRU use host variables (parameter markers) when using the statement. This allows the dynamic SQL statement to be reused within DB2 [assuming statement caching is active]
- Declare and Initialize ESQL variables in same statement
  - Example:
    - *Declare InMessageFmt CHAR;*
    - *Set InMessageFmt = 'SWIFT';*
    - *DECLARE C INTEGER;*
    - *Set C = CARDINALITY(InputRoot.*[]);*
  - Better to use:
    - *Declare InMessageFmt CHAR 'SWIFT';*
    - *DECLARE C INTEGER CARDINALITY(InputRoot.*[])*
- Initialize MRM output message fields using MRM null/default rather than ESQL
  - Messageset.mset > CWF>Policy for missing elements
    - Use null value
  - Cobol import
    - Create null values for all fields if initial values not set in Cobol
  - Avoids statements like
    - `Set OutputRoot.MRM.THUVTILL.IDTRANSIN.NRTRANS = '';`
- For frequently used complex structures consider pre-building and storing in a ROW variable which is also a SHARED variable (and so accessible by multiple copies of the message flow). The pre-built structure can be copied across during the processing of each message so saving processing time in creating the structure from scratch eace time. For an example of ROW and SHARED variable use see the message flow `Routing_using_memory_cache` which is part of the Message Routing sample in the WebSphere Message Broker Toolkit samples gallery.

Notes (slide 8 of 10)

- **Follow applicable points from ESQL like reducing the number of compute nodes**
- **Store intermediate references when building / navigating trees**

Instead of

```
MbMessage newEnv = new MbMessage(env);
newEnv.getRootElement().createElementAsFirstChild(MbElement.TYPE_NAME,"Destination", null);
newEnv.getRootElement().getFirstChild().createElementAsFirstChild(MbElement.TYPE_NAME, "MQDestinationList", null);
newEnv.getRootElement().getFirstChild().getFirstChild()
createElementAsFirstChild(MbElement.TYPE_NAME,"DestinationData", null);
```

Store references as follows:

```
MbMessage newEnv = new MbMessage(env);
MbElement destination = newEnv.getRootElement().createElementAsFirstChild(MbElement.TYPE_NAME,"Destination", null);
MbElement mqDestinationList = destination.createElementAsFirstChild(MbElement.TYPE_NAME, "MQDestinationList", null);
mqDestinationList.createElementAsFirstChild(MbElement.TYPE_NAME,"DestinationData", null);
```

Notes (slide 9 of 10)

- Avoid concatenating java.lang.String objects as this is very expensive since it (internally) involves creating a new String object for each concatenation. It is better to use the StringBuffer class:

Instead of:

```
keyforCache = hostSystem + CommonFunctions.separator + sourceQueueValue +
      CommonFunctions.separator + smiKey + CommonFunctions.separator + newElement;
```

Use this:

```
StringBuffer keyforCacheBuf = new StringBuffer();
keyforCacheBuf.append(hostSystem);
keyforCacheBuf.append(CommonFunctions.separator);
keyforCacheBuf.append(sourceQueueValue);
keyforCacheBuf.append(CommonFunctions.separator);
keyforCacheBuf.append(smiKey);
keyforCacheBuf.append(CommonFunctions.separator);
keyforCacheBuf.append(newElement);
keyforCache = keyforCacheBuf.toString();
```

Notes (slide 10 of 10)

- Although XPath is a powerful statement from a development point of view it can have an unforeseen performance impact in that it will force a full parse of the input message. By default it will retrieve all instances of the search argument. If you know there is only one element or you only want the first then there is an optimization to allow this to happen. The benefit of using this is that it can save a lot of additional, unnecessary parsing.

- The XPath evaluation in Message Broker is written specifically for the product and has a couple of performance optimizations that are worth noting.

- XPath performance tips:
  - Use /aaa[1] if you just want the first one
    - It will stop searching when it finds it
  - Avoid descendant-or-self (//) if possible
    - It traverses (and parses) the whole message
    - Use /descendant::aaa[1] instead of (//aaa)[1]

Message Flow Deployment Algorithm

How many Execution Groups should I have?
How many Additional Instances should I add?

Execution Groups	Additional Instances
• Results in a new process/address-space • Increased memory requirement • Multiple threads including management • Operational simplicity • Gives process level separation • Scales across multiple EGs	• Results in more processing threads • Low(er) memory requirement • Thread level separation • Can share data between threads • Scales across multiple EGs

Recommended Usage

- Check resource constraints on system
  - How much memory available?
  - How many CPUs?
- Start low (1 EG, No additional instances)
- Group applications in a single EG
- Assign heavy resource users to their own EG
- Increment EGs and additional instances one at a time
  - Keep checking memory and CPU on machine
- Don't assume configuration will work the same on different machines
  - Different memory and number of CPUs

Ultimately have to balance
Resource
Manageability
Availability

SHARE Technology · Connections · Results

SHARE in Atlanta 2012

Notes (slide 1 of 2)

- Within a broker it is possible to have one or more execution groups.
- Execution group is a broker term and is a container in which one or more message flows can run.
- The implementation of an execution group varies with platform. On Windows and UNIX platforms it is implemented as an operating system process. On z/OS it is implemented as an address space.
- The execution group provides separation at the operating system level. If you have message flows which you must separate for some reason, you can achieve this by assigning them to different execution groups.
- An individual message flow runs as an operating system thread on the Windows and UNIX platforms and as a Task Control Block (TCB) on z/OS.
- It is possible to run more than one copy of a message flow in an execution group, in which case there will be multiple threads or TCBs running the same message flow. Equally you can run a message flow in more than one execution group. In which case there will be one or more threads or TCBs running the message flow in each of the processes or address spaces to which the message flow has been assigned.
- A significant benefit of using Message Broker is that the threading model is provided as standard. The message flow developer does not need to explicitly provide code in order to cope with the fact that multiple copies of the message flow might be run.
- How many copies and where they are run is an operational issue and not a development one. This provides significant flexibility.
- The ability to use multiple threads or TCBs and also to replicate this over multiple processes or address spaces means that there is excellent potential to use and exploit multiprocessor machines.

Notes (slide 2 of 2)

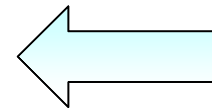
- In testing and production you will need to decide the which message flows are assigned to which execution groups and how many execution groups are allocated. There are a number of possible approaches: Allocate all of the message flows in to one or two execution groups: Have an execution group for each message flow; Split message flows by project and so on. Before considering what the best approach is, lets look at some of the characteristics of execution groups and additional instances.
- Execution Group:
  - Each execution group is an operating system process or address space in which one or more message flows run. Each new execution group will result in an additional process or address space being run.
  - An execution group might typically require ~150MB of memory to start and load executables. Its size after initialisation will then depend on the requirements of the message flows which run within it. Each additional execution group which is allocated will need a minimum of another ~150MB so you can see that using a high number of execution groups is likely to use a large amount of memory quickly.
  - As well as the threads needed to run a message flow, an execution group also has a number of additional threads which are use to perform broker management functions. So allocating many execution groups will result in more management threads being allocated overall.
  - An execution group provides process level separation between different applications. This can be useful for separating applications.
- Additional Instance:
  - Each new additional instance results in one more thread being allocated in an existing execution group. As such the overhead is low. There will be some additional memory requirements but this will be significantly less than the ~150MB that is typically needed to accommodate a new execution group.
  - Message flows running as instances in a single execution group can access common variables called shared variables. This gives a cheap way to access low volumes of common data. The same is not true of messages flows assigned to different execution groups.
- When assigning message flows to execution groups there is no fixed algorithm imposed by Message Broker. There is significant flexibility in the way in which the assignment can be managed. Some users allocate high numbers of execution groups and assign a few message flows to each. We have seen several instance of 100+ execution groups being used. Others allocate many message flows to a small number of execution groups.
- A key factor in deciding how many execution groups to allocate is going to be how much free memory you have available on the broker machine.
- A common allocation pattern is to have one or possibly two execution groups per project and to then assign all of the message flows for that project to those execution groups. By using two execution groups you build in some additional availability should one execution fail. Where you have message flows which require large amounts of memory in order to execute, perhaps because the input messages are very large or have many elements to them, then you are recommended to keep all instances of those message flows to a small number of execution groups rather than allocate over a large number of execution groups [It is not uncommon for an execution group to use 1GB+ of memory.]. Otherwise every execution group could require a large amount of memory as soon as one of the large messages was processed by a flow in that execution group. Total memory usage would rise significantly in this case and could be many gigabytes.
- As with most things this is about achieving a balance between the flexibility you need against the level of resources (and so cost) which have to be assigned to achieve the required processing

Agenda

- What are the main performance costs in message flows?

  …and what are the best practices that will minimize these costs?

- What other performance considerations are there?

- Understanding your broker's behaviour 

Some tools to understand your broker's behaviour

- *PerfHarness* – Drive realistic loads through the broker
  - http://www.alphaworks.ibm.com/tech/perfharness
- OS Tools
  - Run your message flow under load and determine the limiting factor.
  - Is your message flow CPU, memory or I/O bound?
  - e.g. "perfmon" (Windows), "vmstat" or "top" (Linux/UNIX) or "SDSF" (z/OS).
  - This information will help you understand the likely impact of scaling (e.g. additional instances), faster storage and faster networks
- Third Party Tools
  - *RFHUtil* – useful for sending/receiving MQ messages and customising all headers
  - *NetTool* – useful for testing HTTP/SOAP
  - *Filemon* – Windows tool to show which files are in use by which processes
  - *Java Health Center* – to diagnose issues in Java nodes
- MQ Explorer
  - Queue Manager administration
  - Useful to monitor queue depths during tests
- Message Broker Explorer
  - Understand what is running
  - Offload WS-Security processing onto XI50 appliance
  - performance and resource statistics

Broker performance statistics

Statistics
User Trace All Flows
Trace Nodes All Flows
Service Trace
Flow Debug Port

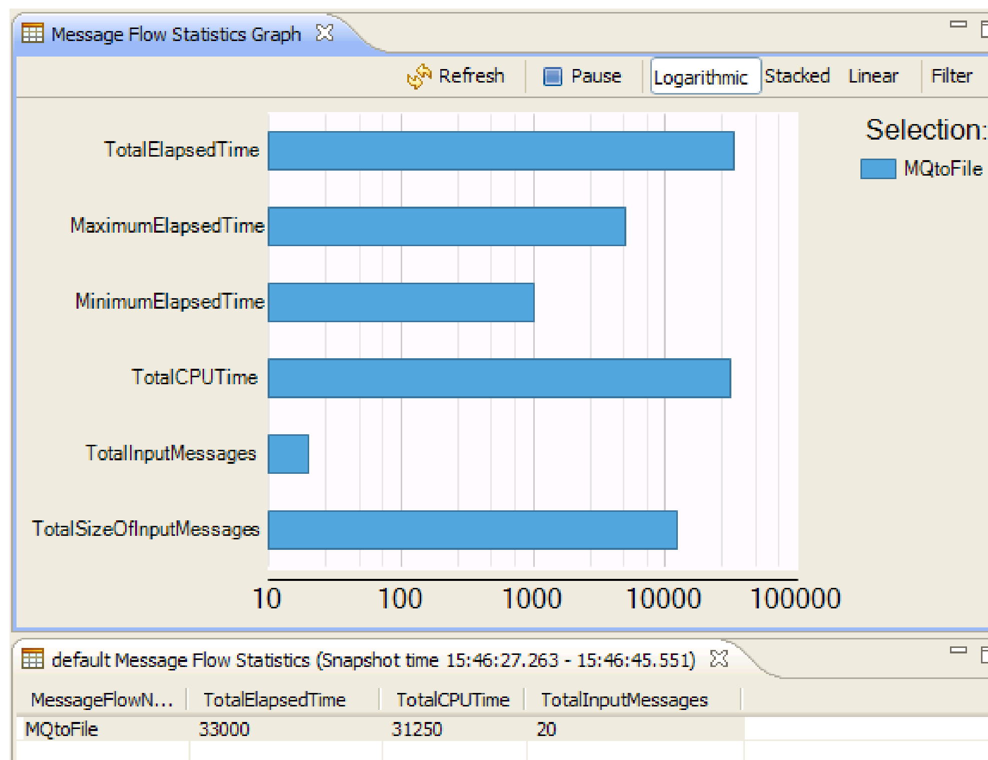
Start Message Flow Statistics
Stop Message Flow Statistics
Open Statistics Views
Start Resource Statistics
Stop Resource Statistics
Open Resource Views

MQ Explorer - Content | testbrk Administration Log

Execution Group default

Warning: Performance Impact - Statistics Reporting active

- The Message Broker Explorer enables you to start/stop message flow statistics on the broker, and view the output.
- New in V7 (although supportpac IS02 available for V6.1)
- Warnings are displayed advising there may be a performance impact (typically ~3%)



Broker resource statistics

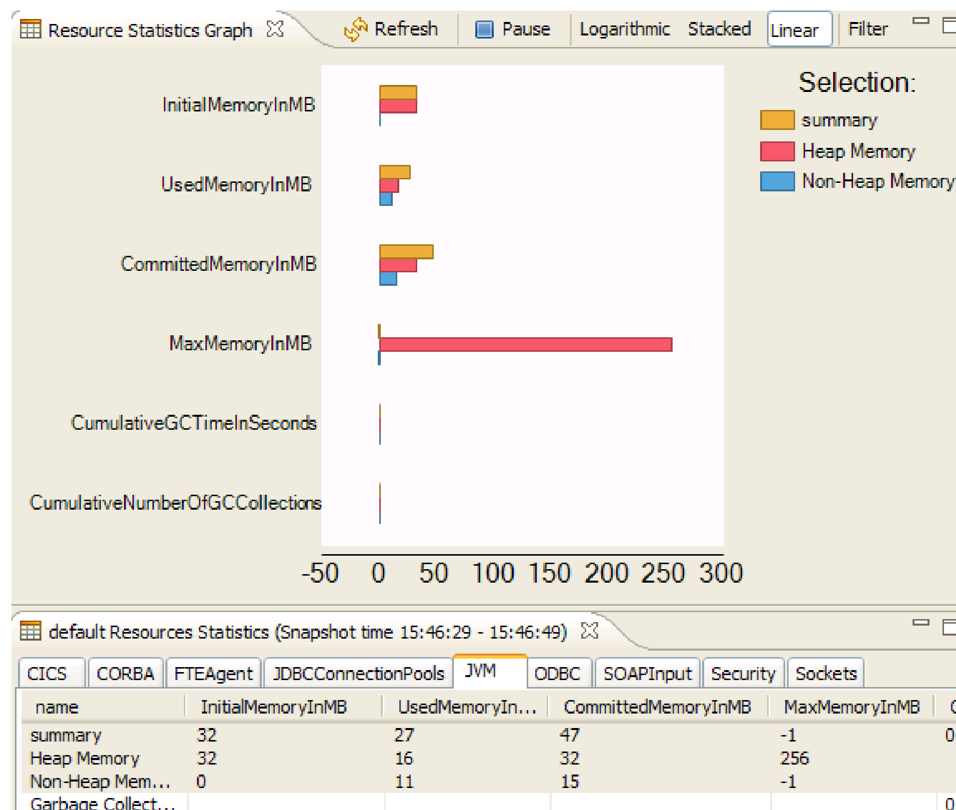
Statistics
User Trace All Flows
Trace Nodes All Flows
Service Trace
Flow Debug Port

Start Message Flow Statistics
Stop Message Flow Statistics
Open Statistics Views
Start Resource Statistics
Stop Resource Statistics
Open Resource Views

MQ Explorer - Content
testbrk Administration Log

Execution Group default

Warning: Performance Impact - Statistics Reporting active



name	InitialMemoryInMB	UsedMemoryIn...	CommittedMemoryInMB	MaxMemoryInMB	C
summary	32	27	47	-1	0
Heap Memory	32	16	32	256	
Non-Heap Mem...	0	11	15	-1	
Garbage Collect...					0

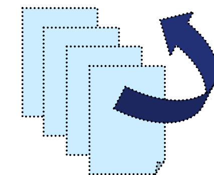
- The Message Broker Explorer enables you to start/stop resource statistics on the broker, and view the output.
- New in V7, more resource managers added in V8!
- Warnings are displayed advising there may be a performance impact (typically ~1%)

Activity Log (V8)

- **New Activity Logging Allows users to understand what a message flow is doing**
  - Complements current extensive product trace by providing end-user oriented trace
  - Can be used by developers, but target is operators and administrators
  - Doesn't require detailed product knowledge to understand behaviour
  - Provides qualitative measure of behaviour
- **End-user oriented with external resource lifecycle**
  - Focus on easily understood actions & resources
  - "GET message queue X", "Update DB table Z"…
  - Complements quantitative resource statistics

	Message…	Timestamp	Message Summary
i	BIP12001I	17-Jun-2011 10:10:50.85…	Connected to JMS provider 'WebSphere_MQ
i	BIP12002I	17-Jun-2011 10:10:50.85…	Created a 'Transaction_None' session for JMS provider 'WebSphere_MQ
i	BIP12004I	17-Jun-2011 10:10:50.93…	Created JMS producer for destination 'ASYNCREQUESTQ
i	BIP12007I	17-Jun-2011 10:10:50.93…	Sent a JMS message to queue 'ASYNCREQUESTQ
i	BIP12004I	17-Jun-2011 10:10:50.52…	Created JMS producer for destination 'ASYNCRECEIVEQ
⊗	BIP12014E	17-Jun-2011 13:47:51.65…	Failed to send message to 'ASYNCRECEIVEQ'
i	BIP12001I	17-Jun-2011 13:47:54.99…	Connected to JMS provider 'WebSphere_MQ
i	BIP12004I	17-Jun-2011 13:47:55.00…	Created JMS producer for destination 'ASYNCRECEIVEQ

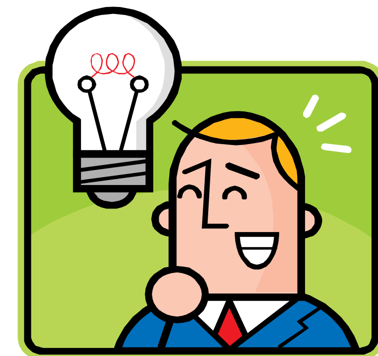
- **Flow & resource logging**
  - User can observe all events for a given flow
    - e.g. "GET MQ message", "Send IDOC to SAP", "Commit transaction"…
  - Users can focus on individual resource manager if required
    - e.g. SAP connectivity lost, SAP IDOC processed
  - Use event filters to create custom activity log
    - e.g. capture all activity on JMS queue REQ1 and C:D node CDN1
  - Progressive implementation as with resource statistics, starting with JMS, C:D and SAP resources
- **Comprehensive Reporting Options**
  - Reporting via MB Explorer, log files and programmable management (CMP API)
  - Extensive filtering & search options, also includes save data to CSV file for later analysis



- **Log Rotation facilities**
  - Rotate resource log file when reaches using size or time interval

Testing and Optimising Message Flows

- Run driver application (e.g. *PerfHarness*) on a machine separate to broker machine
- Test flows in isolation
- Use real data where possible
- Avoid pre-loading queues
- Use a controlled environment – access, code, software, hardware
- Take a baseline before any changes are made
- Make ONE change at a time.
- Where possible retest after each change to measure impact
- Start with a single flow instance/EG and then increase to measure scaling and processing characteristics of your flows



Performance Reports

- See http://www-1.ibm.com/support/docview.wss?uid=swg27007150
- Consists of 2 Sections:
  - High level release highlights and use case throughput numbers
    - Use to see what new improvements there are
    - Use to see what kind of rates are achievable in common scenarios
    - Numbers in this section can be replicated in your environment using product samples and perfharness
  - Detailed low level metrics (nodes/parsers)
    - Use to compare parsers
    - Use to compare persistent vs non-persistent
    - Use to compare transports
    - Use to compare transformation options
    - Use to see scaling and overhead characteristics
    - Shows tuning and testing setup
- Please send us your feedback!

Message Broker V8 – Performance highlights

- Message parsing and serialisation
  - The new DFDL (Data Format Description Language) parser and serialiser, has excellent performance characteristics. We have measured improvements of up to 70% compared to existing MRM technology, with a typical improvement of 50%.
- Mapping
  - The new Mapping node allows the user to visually map and transform data from source to target. It has excellent performance characteristics, and is a viable option for performance sensitive transformations. We have measured some tests performing close to optimised programmatic transformations ESQL, Java and .Net, with the typical measurement being 50%.
- Message Broker on AIX
  - Specific performance enhancements for AIX on Power. These include optimisations to internal string handling and changes to MALLOC options. These have resulted in improvements of 10% across a range of performance tests.
- Resource Statistics
  - More resource managers now emitting resource statistical information with minimal overhead. These now include: Dot Net App domains, CICS, DotNet GC, CORBA, ConnectDirect, FTEAgent, FTP, File, JDBCConnectionPools, JMS, JVM, ODBC, Parsers, SOAPInput, Security, Sockets, TCPIPClientNodes and TCPIPServerNodes. All of these can be visualized in the WebSphere Message Broker Explorer.
- Activity Log
  - Activity logs is a new feature to help users understand what their message flows are doing by providing a high-level overview of how the broker runtime interacts with external resources. Collection is always on and overheads are negligible.

Summary

- What are the main performance costs in message flows?
  …and what are the best practices that will minimize these costs?
- What other performance considerations are there?
- Understanding your broker's behaviour

For more information, including links to additional material,

download a copy of this presentation or IP04!

Notes

- All Performance Reports are available at
  - http://www.ibm.com/support/docview.wss?rs=171&uid=swg27007150&loc=en_US&cs=utf-8&lang=en

- developerWorks - WebSphere Message Broker Home Page
  - http://www.ibm.com/developerworks/websphere/zones/businessintegration/wmb.html

- Some developerWorks Articles
  - What's New in WebSphere Message Broker V7.0
  - Determining How Many Copies of a Message Flow Should Run Within WBI Message Broker V5
  - How to Estimate Message Throughput For an IBM WebSphere MQ Integrator V2.1 Message Flow
  - Optimising DB2 UDB for use with WebSphere Business Integration Message Broker V5
  - How to monitor system and WebSphere Business Integration Message Broker V5 resource use
  - The Value of WebSphere Message Broker V6 on z/OS
  - JMSTransport Nodes in WebSphere Message Broker V6
  - Connecting the JMS Transport Nodes for WebSphere Message Broker v6 to Popular JMS Providers
  - HTTP transport nodes in WebSphere Message Broker V6
  - Testing and troubleshooting message flows with WebSphere Message Broker Test Client

This was session 10698 - The rest of the week ……

	Monday	Tuesday	Wednesday	Thursday	Friday
08:00			Free MQ! - MQ Clients and what you can do with them.	MQ Performance and Tuning on distributed	
09:30		The MQ API for dummies - the basics	The Dark Side of Monitoring MQ - SMF 115 and 116 record reading and interpretation	The even darker arts of SMF	CICS Programs Using WMQ V7 Verbs
11:00		Putting the web into WebSphere MQ: A look at Web 2.0 technologies The Doctor is in. Hands-on Lab and Lots of Help with the MQ Family	Message Broker administration	The Do's and Don'ts of z/OS Queue Manager Performance	
12:15		WebSphere MQ: Highly scalable publish subscribe environments		MQ & DB2 – MQ Verbs in DB2 & Q-Replication	
01:30	WebSphere MQ 101: Introduction to the world's leading messaging provider	What's new in WebSphere Message Broker V8.0	The Do's and Don'ts of Message Broker Performance	Diagnosing problems for MQ	
03:00	WebSphere Message Broker 101: The Swiss army knife for application integration	What's new in WebSphere MQ V7.1	WebSphere MQ Security - with V7.1 updates	Diagnosing problems for Message Broker	
04:30	Introduction to the WebSphere MQ Product Family - including what's new in the family products	Under the hood of Message Broker on z/OS - WLM, SMF and more	MQ Java zero to hero	Shared Q including Shared Message Data Sets	
06:00			For your eyes only - WebSphere MQ Advanced Message Security	MQ Q-Box - Open Microphone to ask the experts questions	