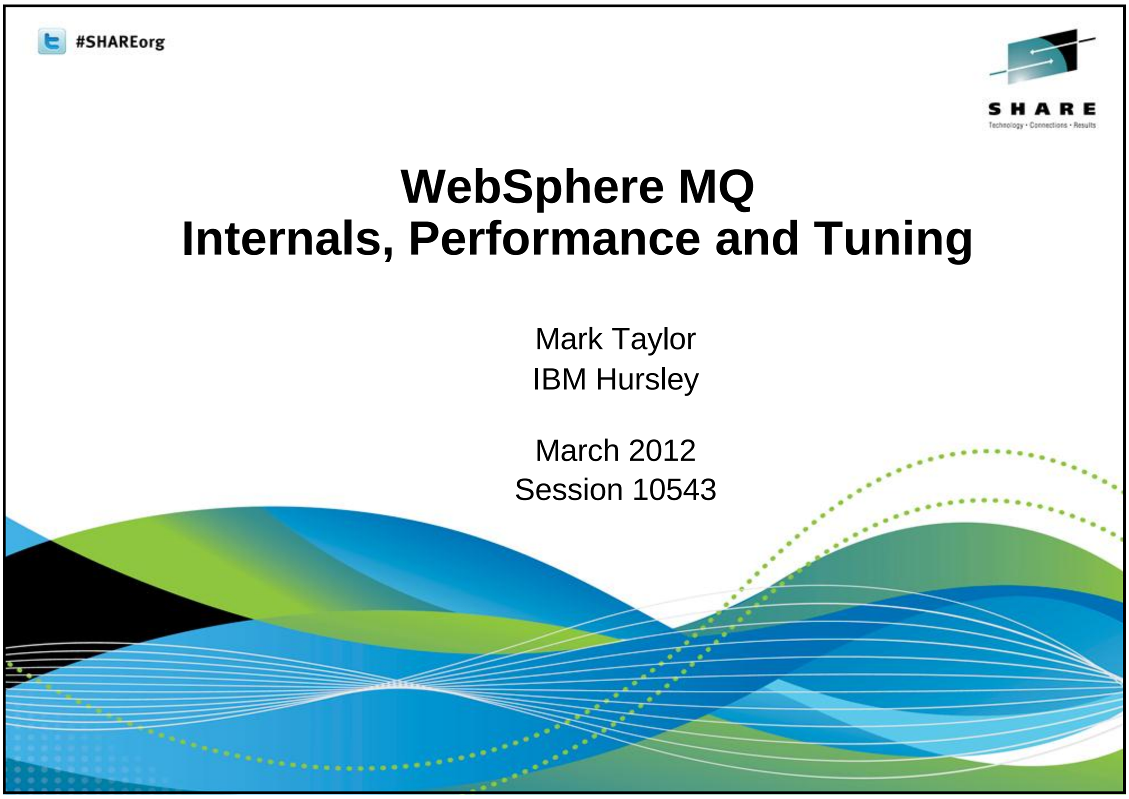
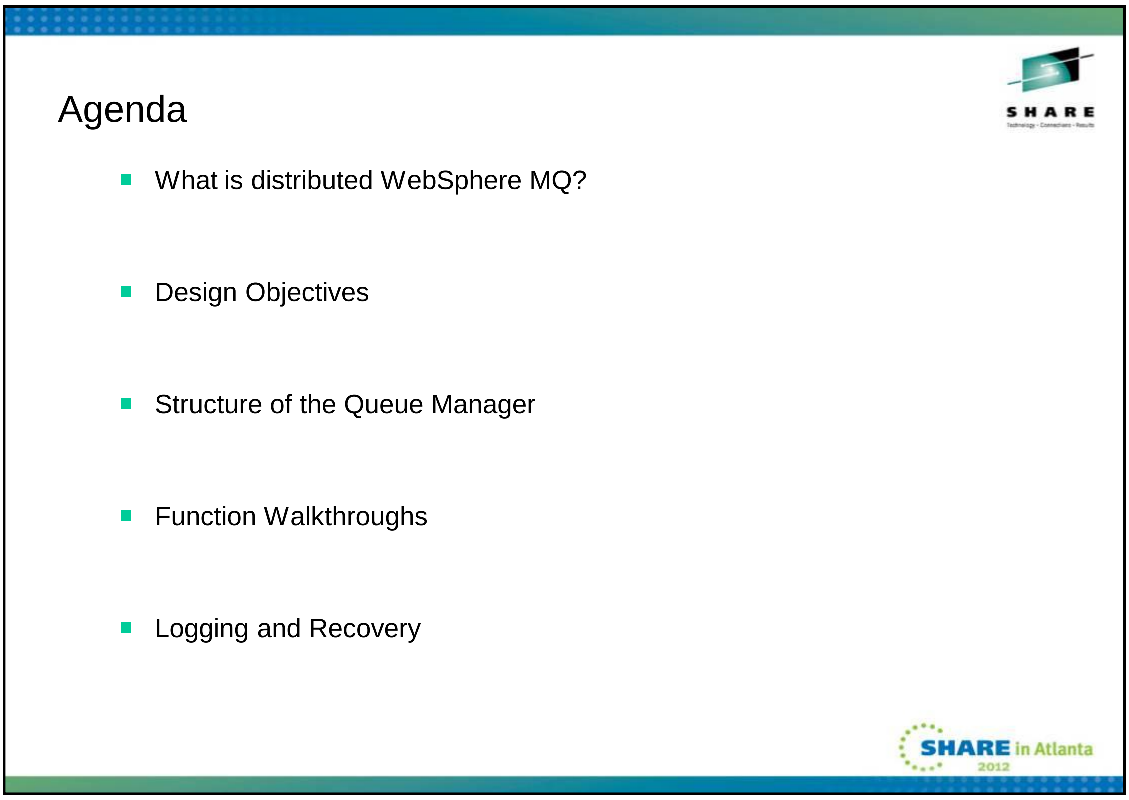


#SHAREorg

WebSphere MQ Internals, Performance and Tuning

Mark Taylor
IBM Hursley

March 2012
Session 10543

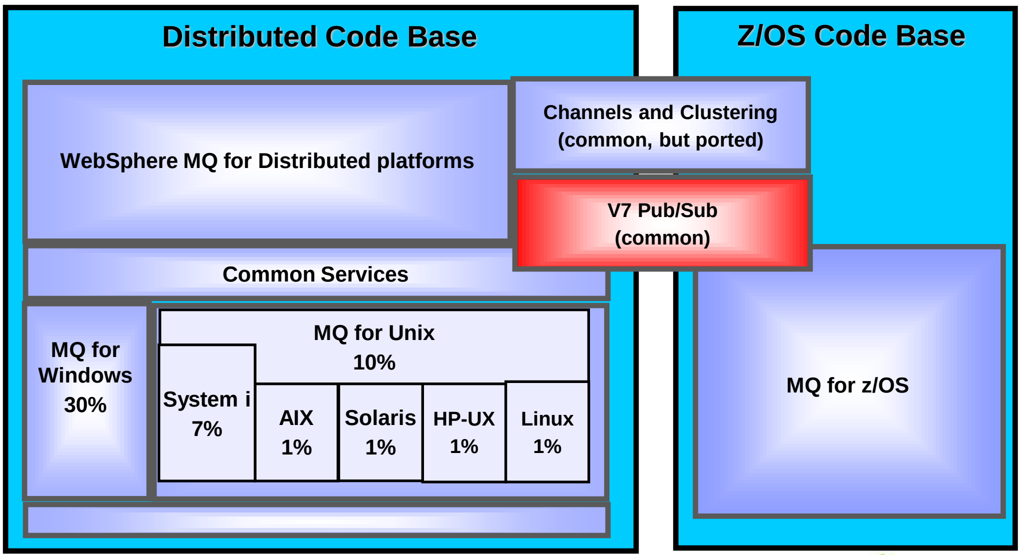


Agenda

- What is distributed WebSphere MQ?
- Design Objectives
- Structure of the Queue Manager
- Function Walkthroughs
- Logging and Recovery

SHARE in Atlanta 2012

What is Distributed MQ?

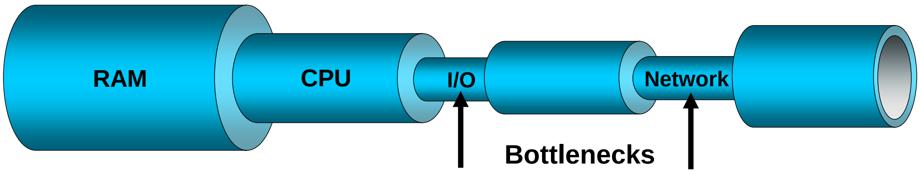


What is Distributed MQ?

NOTES

- There are two core WebSphere MQ code bases developed in Hursley.
- The Distributed version is the code base for range of systems. Its design point is a full function, high performance queue manager with a focus on highly portable code. These are the Distributed queue managers.
  - There is a very high percentage of common code between versions on different operating systems.
  - The code is mainly written in C with special attention to portability. Even the environment to build the executables is portable.
  - Code differences relate mainly to differences in operating system facilities and packaging. On some platforms there is specific code to integrate the user interface with the operating system (eg Windows and System i)
- The z/OS version of MQ is aimed at being a full function, high performance queue manager with maximum exploitation of MVS architecture. Some code, notably the MCAs and clustering, is shared between the implementations.
- Percentages in the diagram represent the approximate proportion of code specific to each version.
- System i is classed as a UNIX platform from the Distributed queue managers point-of-view as it shares many similarities with other UNIX platforms. The platform-specific code here is mostly in dealing with configuration panels, and the use of native logging/journalling features.
- The source code of the Distributed version of MQ is licensed to several partners who use it as the basis of ports of MQ to other operating systems; it is also used by other parts of IBM to further extend the operating system support.
- With V7, there is now truly common code between z/OS and Distributed: previously the “common” code was ported between environments

Performance Bottlenecks

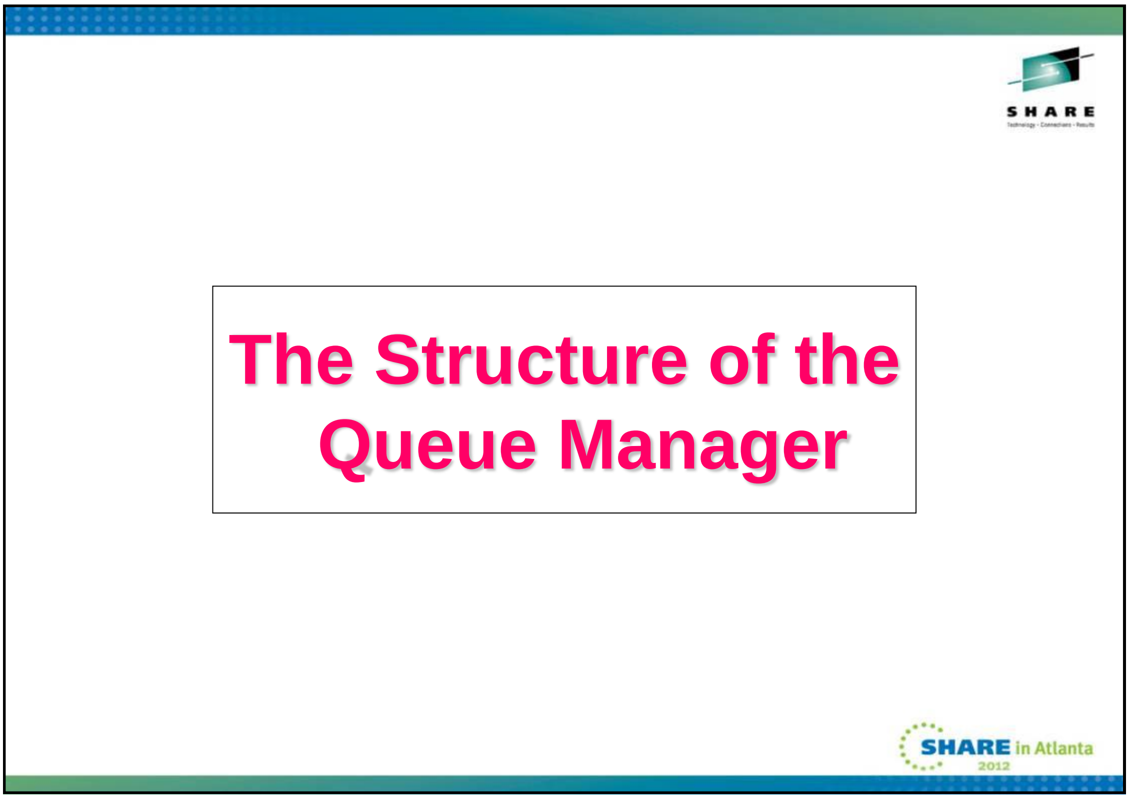


- Business Systems are complex
  - Often no single bottleneck limiting performance
  - Performance can mean different things to different people
  - Throughput
    - Scalability
    - Low resource usage
- Not only limited by physical resources
  - Application design, such as parallelism can have major effect
- Performance Reports (from SupportPac site) show range of scenarios

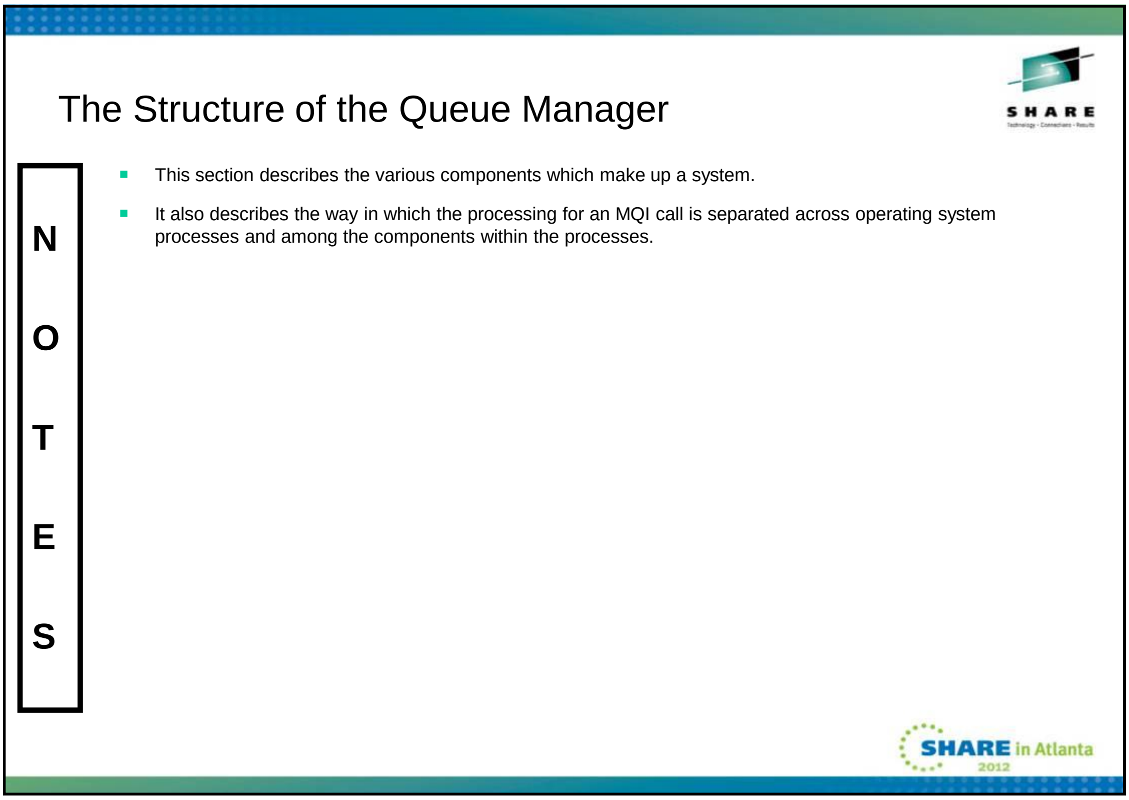
Performance Bottlenecks

N O T E S

- Modern systems are complex and there are many factors which can influence the performance of a system. The hardware resources available to the application as well as the way that application is written all affect the behavior.
- Tuning these environments for maximum performance can be fairly tricky and requires fairly good knowledge of both the application and the underlying system. One of the key points to make is that the simple trial and error approach of changing a value and then measuring may not yield good results. For example, a user could just measure the throughput of messages on the previous foil. They could then double the speed of the disk and re-measure. They would see practically no increase of speed and could wrongly deduce that disk I/O is not a bottleneck.
- Of course throughput is not the only metric that people want to tune for. Sometimes it is more important that a system is scalable or that it uses as little network bandwidth as possible. Make sure you understand what your key goal is before tuning a system. Often increasing one metric will have a detrimental affect on the other.



The Structure of the Queue Manager

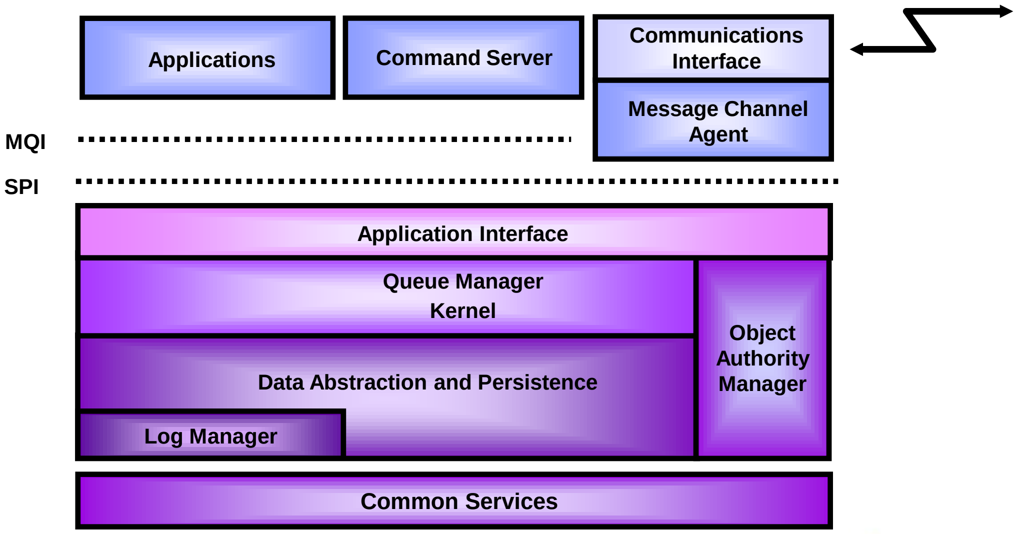


The Structure of the Queue Manager

NOTES

- This section describes the various components which make up a system.
- It also describes the way in which the processing for an MQI call is separated across operating system processes and among the components within the processes.

Queue Manager: Functional View

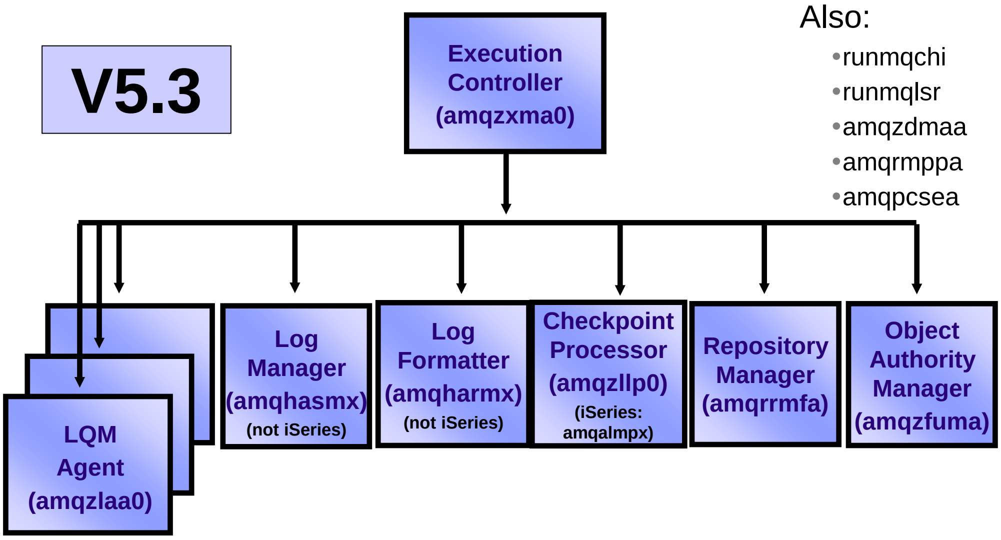


Queue Manager: Functional View

N O T E S

- The queue manager has the following major components:
- **Application Interface** provides the environment and mechanism for execution of MQI calls.
- **Queue Manager Kernel** provides most of the function of the MQI. For example, triggering is implemented here along with message location.
- **Object Authority Manager** provides access control for the queue manager and its resources. It allows specification of which users and groups are permitted to perform which operations against which resources.
- **Data Abstraction and Persistence** provides storage and recovery of the data held by the queue manager.
- **Log Manager** maintains a sequential record of all persistent changes made to the queue manager. This record is used for recovery after a machine crash and for remembering which operations were in which transaction.
- **Message Channel Agents** are special *applications* using the SPI for the majority of their operations. They are concerned with reliable transmission of messages between queue managers.
- **SPI** is a lower-level API similar to MQI but only available to Queue Manager processes offering greater performance and functionality.
- **Command Server** is a special application. It is concerned with processing messages containing commands to manage the queue manager.
- **Common Services** provides a common set of operating system-like services such as storage management, NLS, serialisation and process management. This isolates the Queue Manager from platform differences.

Queue Manager Process Tree



V5.3

Execution Controller (amqzxma0)

LQM Agent (amqzlaa0)

Log Manager (amqhasmx) (not iSeries)

Log Formatter (amqharmx) (not iSeries)

Checkpoint Processor (amqzllp0) (iSeries: amqalmpx)

Repository Manager (amqrrmfa)

Object Authority Manager (amqzfuma)

Also:

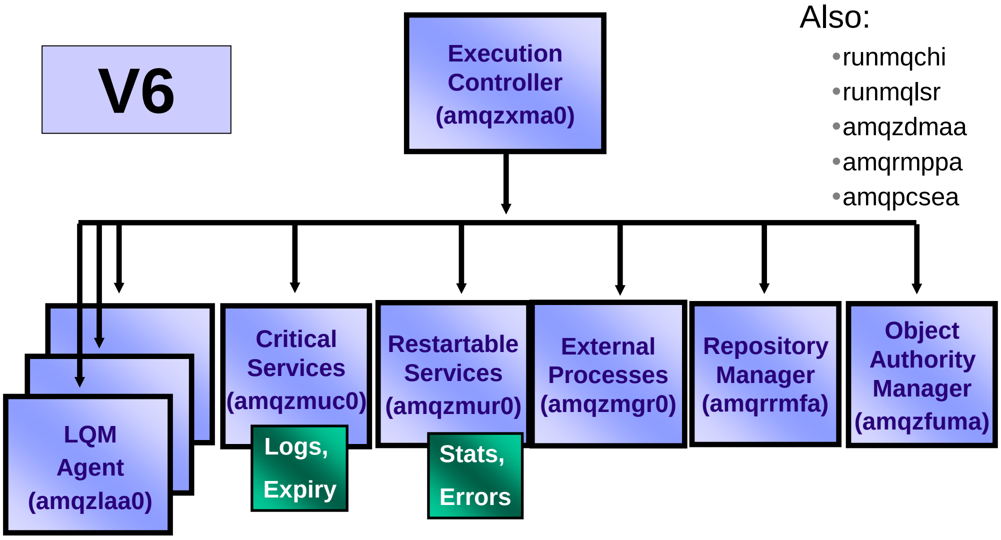
- runmqchi
- runmqlsr
- amqzdmaa
- amqrmppa
- amqpcsea

Queue Manager Process Tree – V5.3

N O T E S

- Although this version of MQ is long out-of-support, it's a good starting point to see the evolution of the product.
- These are the processes you see when a queue manager is running. The example is taken from AIX although Windows and other Unix platforms are similar. The iSeries process structure is slightly different, but still contains many of the same blocks of code and processes.
- The Execution Controller is program amqzxma0. This is the root process of the queue manager and the parent of all of the other processes. It can be thought of as the owner of all of the queue manager's shared resources. It is concerned with managing and monitoring the other queue manager processes and the applications that connect.
- The LQM agents are program amqzlaa0 or amqzlsa0. Agents perform the operations required to process MQI calls on behalf of applications. Nearly all of the code beneath the MQI is actually executed by the agents.
  - The separation of application programs from the queue manager's critical resources protects the queue manager from rogue or malicious applications.
  - The number of agent processes depends on the workload. By default, agents each handle about 60 concurrent connections.
- The log manager is program amqhasmx. All log reading and writing requests go through this process. Communication with the agents is achieved through a set of shared memory buffers.
- If a queue manager is running with linear logging enabled, there will be a log formatter running, This program is amqharmx. Its task is to pre-format log files in time for the log manager to use them. This process is not needed when a qmgr is running a circular log as all the log files are created and initialised when the qmgr is created.
- The checkpoint processor is program amqzllp0. It is concerned with minimising the amount of recovery processing when the queue manager is started.

Queue Manager Process Tree


V6
Execution Controller (amqzxma0)
LQM Agent (amqzlaa0)
Critical Services (amqzmuc0)
Logs, Expiry
Restartable Services (amqzmur0)
Stats, Errors
External Processes (amqzmgr0)
Repository Manager (amqrrmfa)
Object Authority Manager (amqzfuma)

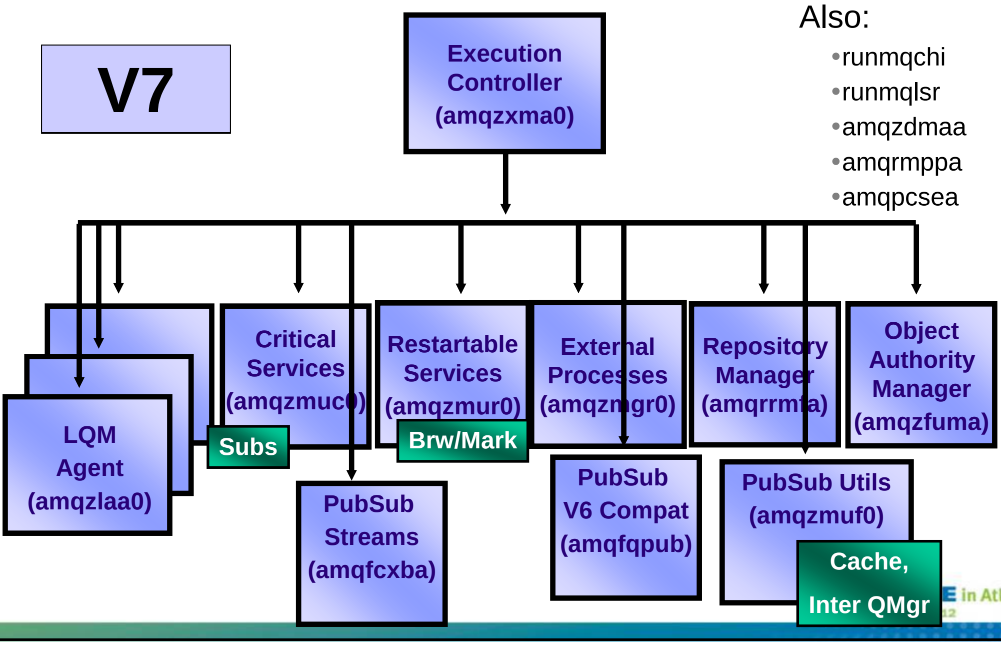
Also:
- runmqchi
- runmqlsr
- amqzdmaa
- amqrmppa
- amqpcsea

Queue Manager Process Tree – V6

NOTES

- V6 brought new asynchronous activities that did not warrant extra processes
  - Would have been too many
  - Three processes now run various tasks or services as THREADS
  - All queue manager platforms are multi-threaded – was not always the case
- amqzmgr0
  - Controls the traditional external processes such as command server, listener
  - Also controls processes defined as SERVICES
- amqzmuc0
  - Hosts internal services which are fundamental to the health of the queue manager
  - Failures in this process result in queue manager termination
  - Logger, checkpoint, formatter,
  - New function: Message Expiry scanner – approximately every 5 minutes
- amqzmur0
  - Hosts internal services considered not fundamental to queue manager health
  - This process can be restarted in the event of a failure
  - Error logging task and the statistics task

Queue Manager Process Tree


V7
Execution Controller (amqzxma0)
LQM Agent (amqzlaa0)
Subs
Critical Services (amqzmuc0)
PubSub Streams (amqfcxba)
Restartable Services (amqzmur0)
Brw/Mark
External Processes (amqzmgr0)
PubSub V6 Compat (amqfqpub)
Repository Manager (amqrrmfa)
PubSub Utils (amqzmuf0)
Cache, Inter QMgr
Object Authority Manager (amqzfuma)

Also:

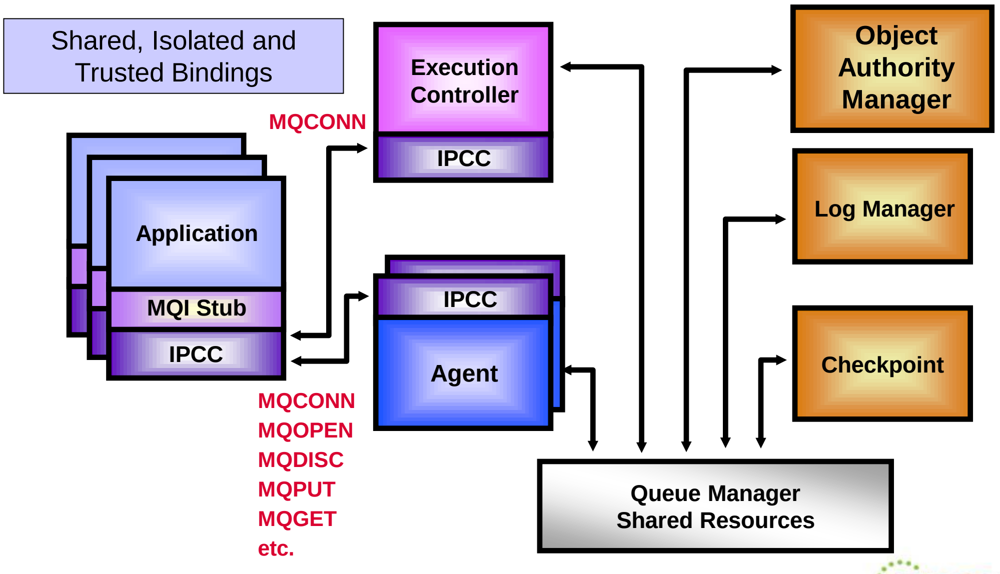
- runmqchi
- runmqlsr
- amqzdmaa
- amqrmppa
- amqpcsea

Queue Manager Process Tree – V7

NOTES

- V7 has new processes to handle publish/subscribe operations
- amqfcxba, amqfqpub
  - Provides compatibility with V6 queued pub/sub processing for streams
- amqzmuf0
  - Pub/Sub Utility Container
  - Cache management
  - Inter-queue manager pub/sub daemon
- Other tasks added to existing controllers
  - Browse/Mark scanner (restartable)
  - Durable Subscription manager (critical)
- Look in error log to see them starting
- These processes will continue to change in future

Queue Manager: Process Model



Shared, Isolated and Trusted Bindings

Execution Controller

IPCC

MQCONN

Application

MQI Stub

IPCC

IPCC

Agent

MQCONN
MQOPEN
MQDISC
MQPUT
MQGET
etc.

Object Authority Manager

Log Manager

Checkpoint

Queue Manager Shared Resources

Queue Manager: Process Model

N

O

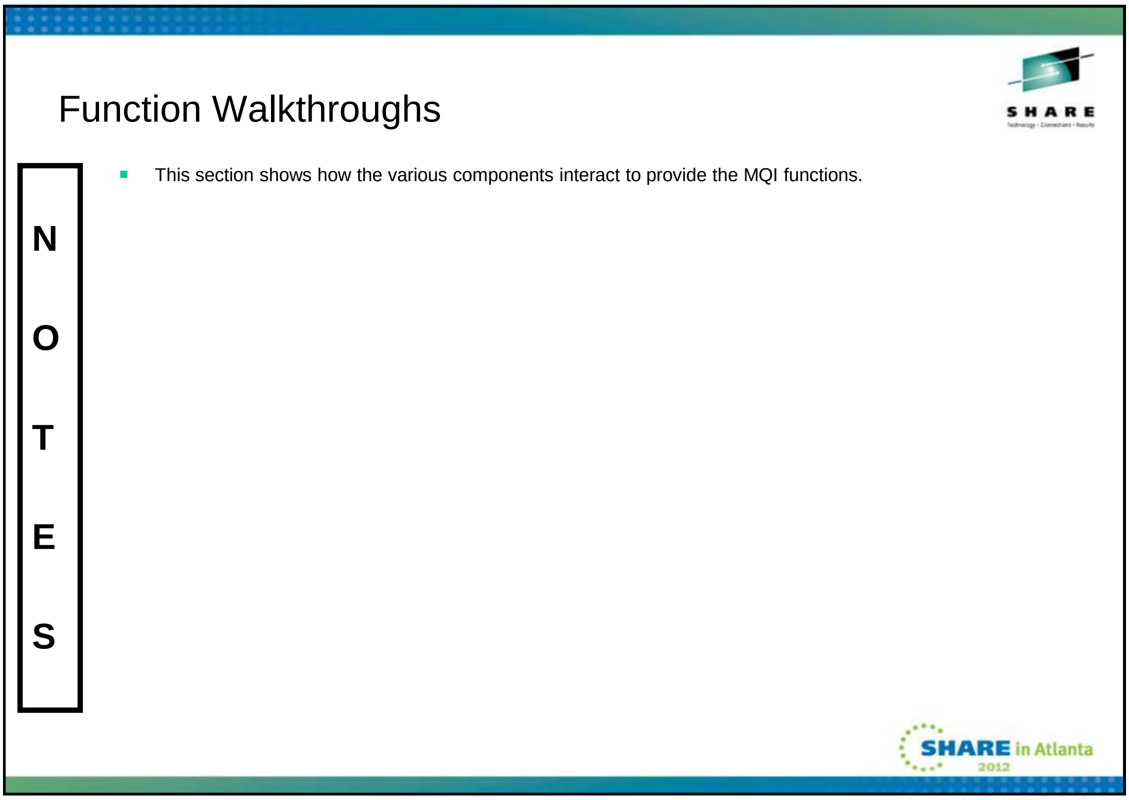
T

E

S

- This diagram shows the processes in terms of their interactions.
- The application communicates with the Execution Controller when it needs an agent to talk to. The EC is responsible for managing the agent processes. It monitors the agents and their associated applications.
- The Application Interface is split into two parts:
  - The MQI Application Stub is bound with the application code. It packages MQ requests and passes them to the agent process using the IPCC.
  - The Inter-Process Communication Component (IPCC) provides a message-passing interface between the MQI applications, the agents and the EC.
- The application communicates with its agent process via the IPCC. The agent process performs the MQI calls on the application's behalf. The IPCC exchanges between the application and agent are synchronous request-reply exchanges.
- The processes within the queue manager share information using shared memory. The other queue manager tasks such as the log manager and the checkpoint process also share queue manager information in this way.
- The IPCC is implemented with several different options: the normal mechanism uses shared memory, which provides for reasonable isolation with reasonable performance. Isolated bindings use Unix-domain sockets, giving greater isolation but slower operations. Applications using shared bindings can inhibit restart of a queue manager if they are not terminated. Trusted bindings give the best performance (particularly for non-persistent operations) but can lead to internal corruption if the application runs rogue.

Function Walkthroughs

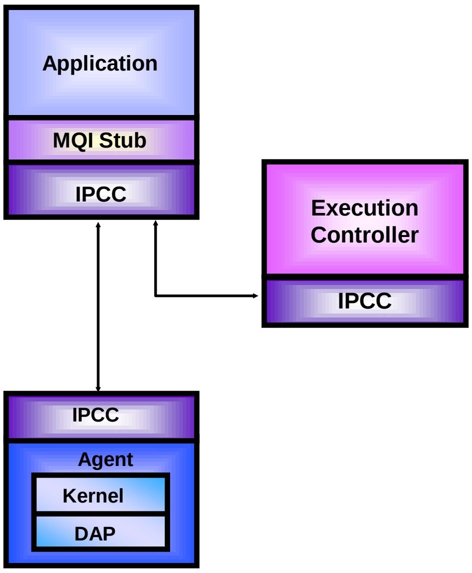


Function Walkthroughs

NOTES

- This section shows how the various components interact to provide the MQI functions.

MQCONN



Application (MQI Stub)
- Verify parameters and handles
- Construct a Connect message
- Call API crossing exit
- Send a message to the EC

Execution Controller
- Choose an agent or start a new one
- Construct a reply IPCC message
- Return reply to application

Application (MQI Stub)
- Receive the reply
- Construct an IPCC message
- Send the message to the agent

Agent
- Check App's permission to connect
- Allocate and assign agent resources
- Send IPCC reply back to application

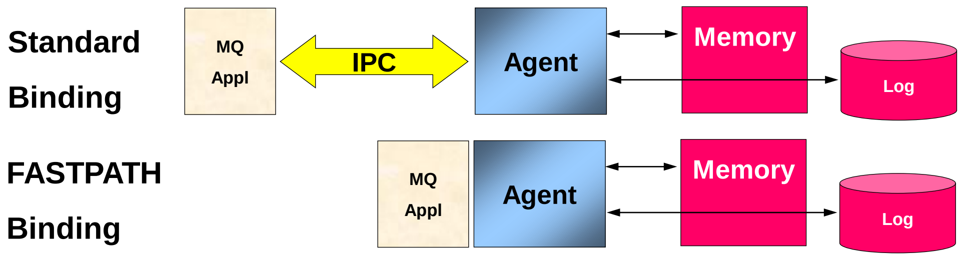
Application (MQI Stub)
- Receive reply and call API crossing exit
- Return HCONN

MQCONN

NOTES

- MQCONN is different to most calls in that the application communicates directly with the Execution Controller. The Execution Controller owns and manages the agent processes. When an application tries to make a connection, the EC decides whether to start a new agent, to start a thread in an existing agent or to reuse an existing agent which has just been released by another application. It will also create an IPCC link for the application and agent to use to communicate if a new agent/thread is to be created.
- When the application issues MQCONN (not a client connect) the application stub which is bound to the application does basic parameter checking. This is limited to checks which can be performed without access to protected queue manager resources. For example, the stub can check to see if the application is already connected and the queue manager requested exists on the machine.
- The parameters to MQCONN are bundled up into a Connect message. This is then sent across to the EC using the IPCC. The EC selects or starts a new agent and returns the details to the application stub.
- If a new thread is to be created in the agent (the EC tells the application if it is) the application stub sends a Start Thread message to the agent using the IPCC. The agent receives the message and associates itself with the application. A thread will be started if the agent is running on an operating system where multiple threads can be used in the agent. The application stub then sends a connection message to this thread.
- Otherwise, an existing agent thread is to be used and the application stub sends a connection message directly to the thread.
- The Kernel checks that the application is authorised to connect. It creates a connection handle which the application will use on all future calls.
- When the IPCC reply message is received in the application stub, it is unpacked and the output parameters are returned to the application.
- FastPath applications bypass most of the IPCC processing at the expense of integrity

Performance Implications: Connection Binding



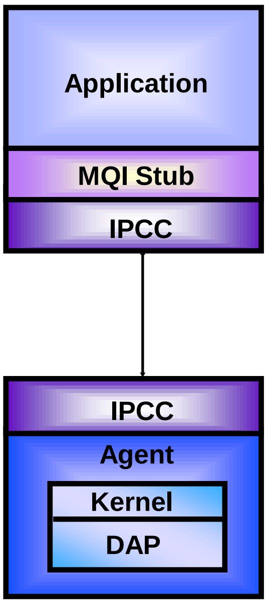
- Fastpath binding removes inter-process communications
  - Implies that the application is 'trusted'
  - MQCONNX option MQCNO_FASTPATH_BINDING
  - Application failure can corrupt queue manager
- Primary benefit is for non-persistent message processing
  - Use for MCAs, Broker
    - - 30% CPU saving

Performance Implications: Connection Binding

NOTES

- Two of the major overheads in the processing path for MQ are the Inter-Process Communication component and the I/O subsystem.
- For non-persistent messages, the I/O subsystem is rarely used. Therefore there is substantial benefit to be gained from by-passing the IPC component. This is what the Trusted Binding provides.
- Depending upon the efficiency of the IPC component for a particular platform, the use of a Trusted Binding will provide anything up to an 3 times reduction in the pathlength for non-persistent message processing.
- There is a price to pay for this improvement in pathlength. The Standard Binding for applications provides separation of user code and MQ code (via the IPC component). The actual Queue Manager code runs in a separate process from the application, known as an agent process (AMQZLAA0). Using standard binding it is not possible for a user application to corrupt queue manager internal control blocks or queue data. This will NOT be the case when a Trusted Binding is used, and this implies that ONLY applications which are fully tested and are known to be reliable should use the Trusted Binding.
- The Trusted Binding applies to the application process and will also apply to persistent message processing. However, the performance improvements are not so great as the major bottleneck for persistent messages is the I/O subsystem.
- Use for Channel programs
  - MQ_CONNECT_TYPE=FASTPATH env variable
  - qm.ini under the Channels: section
    - MQIBindType=FASTPATH
  - Do not issue Stop Channel mode(TERMINATE)
  - Exit code
- Write exits so that Channels and Broker can run trusted

MQOPEN of a queue



Application

MQI Stub

IPCC

IPCC

Agent

Kernel

DAP

Application (MQI Stub)

- **Verify hConn**
- **Call API crossing exit**
- **Verify parameter addressability**
- **Place parameters in an Open message**
- **Send an IPCC message to the agent**

Application (MQI Stub)

- **Receive reply**
- **Call API crossing exit**
- **Return HOBJ**

Agent

Application interface

- **Verify open parameters**

Kernel

- **Verify operation validity**
- **Resolve target – including cluster lookup**
- **Check permissions on the queue**

DAP

- **Load the queue ready for gets and puts if required**
  - **This is the part that can use system resources**

Kernel

- **Generate handle to object for application**
- **Generate responses and event messages**

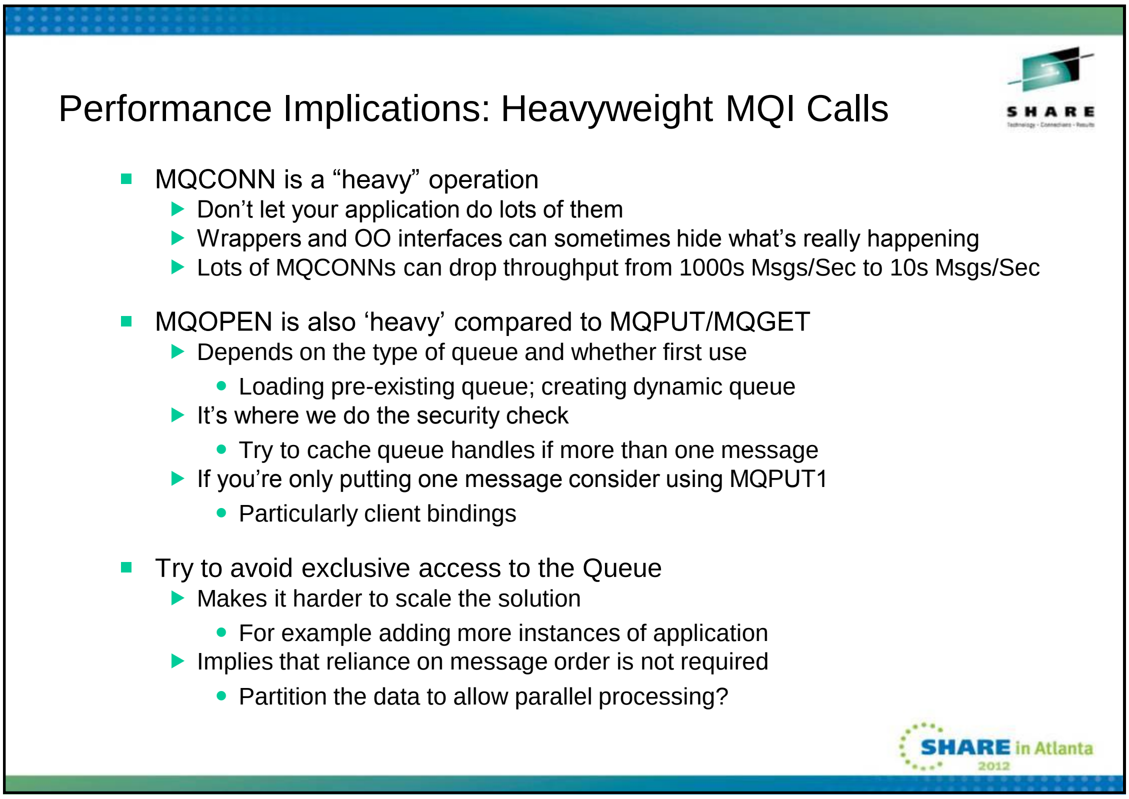
IPCC

- **Send reply back to application**

MQOPEN of a queue

NOTES

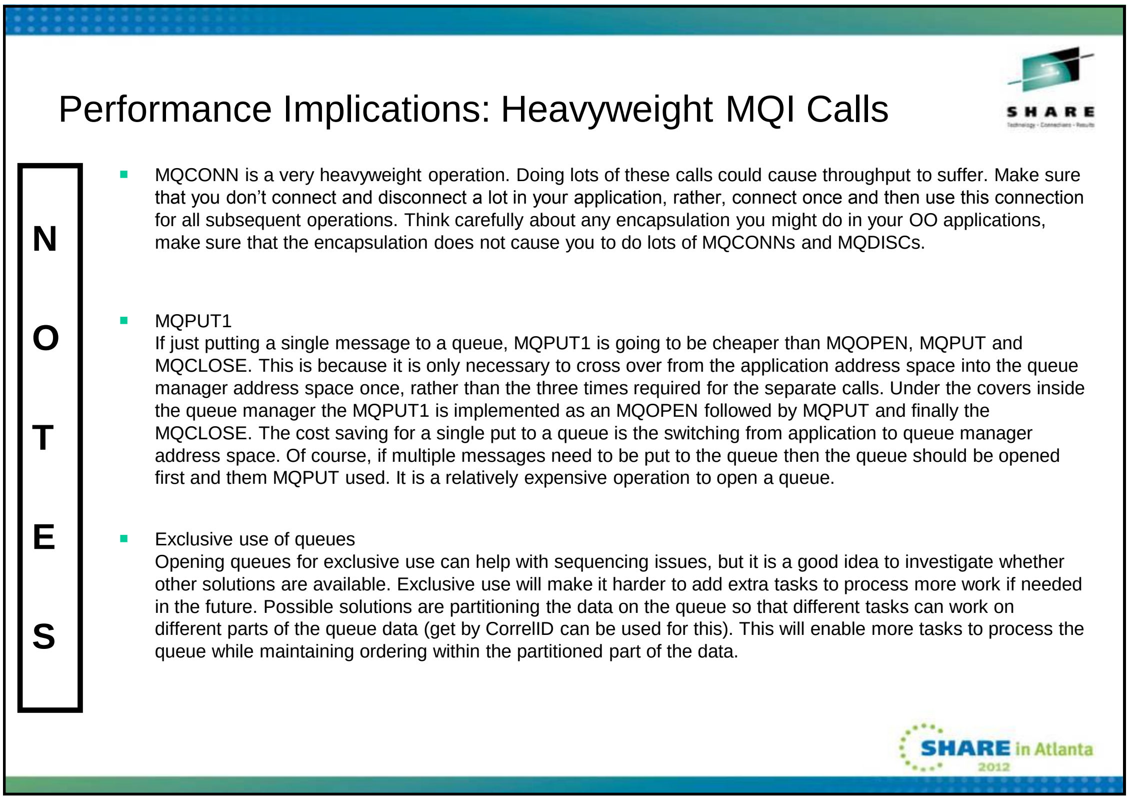
- The MQI application stub first does basic parameter checking. This is limited to checks which can be performed without access to protected queue manager resources.
- The parameters to MQOPEN are bundled up into an Open message. This is then sent across to the agent using the IPCC.
- The agent thread dedicated to this connection in the meantime has been waiting for a message. Periodically, it checks that the application is still alive so that cleanup can be performed if it ends without disconnecting.
- The application interface checks the syntax of the MQOPEN request.
- The kernel verifies the operation for validity. Many aspects will already have been verified, but some can only be checked at this stage. The kernel resolves the name of the target queue. The queue name supplied by the caller may be a local, remote or alias queue and might be a model queue being used to create a dynamic queue. The target queue may be a normal local queue or a transmission queue if the messages are destined for another queue manager. If it's a queue that is part of a cluster then some resolution of the target (to the cluster transmit queue) will be done here; final resolution to a specific queue manager may be done depending on the MQOO options.
- The kernel sorts this lot out and opens the appropriate underlying queue. Whilst doing this, the kernel also checks that the requester of the operation is actually authorised to perform it. It calls the OAM to perform these checks.
- The DAP performs the operations needed to make the physical (local) queue available. This is termed loading the queue. It involves opening the file containing the underlying message data and allocating the shared memory buffers and other shared resources necessary for the queue to be used. Of course, if the queue is already loaded, this work can be avoided.
- Finally, the Kernel creates the 'handle' which the application will use to access the queue.
- When the IPCC reply message is received in the application stub, it is unpacked, the API crossing exit is called again, and the output parameters are returned to the application.



Performance Implications: Heavyweight MQI Calls

- MQCONN is a “heavy” operation
  - Don’t let your application do lots of them
  - Wrappers and OO interfaces can sometimes hide what’s really happening
  - Lots of MQCONNs can drop throughput from 1000s Msgs/Sec to 10s Msgs/Sec
- MQOPEN is also ‘heavy’ compared to MQPUT/MQGET
  - Depends on the type of queue and whether first use
    - Loading pre-existing queue; creating dynamic queue
  - It’s where we do the security check
    - Try to cache queue handles if more than one message
  - If you’re only putting one message consider using MQPUT1
    - Particularly client bindings
- Try to avoid exclusive access to the Queue
  - Makes it harder to scale the solution
    - For example adding more instances of application
  - Implies that reliance on message order is not required
    - Partition the data to allow parallel processing?

SHARE in Atlanta 2012



Performance Implications: Heavyweight MQI Calls

N

O

T

E

S

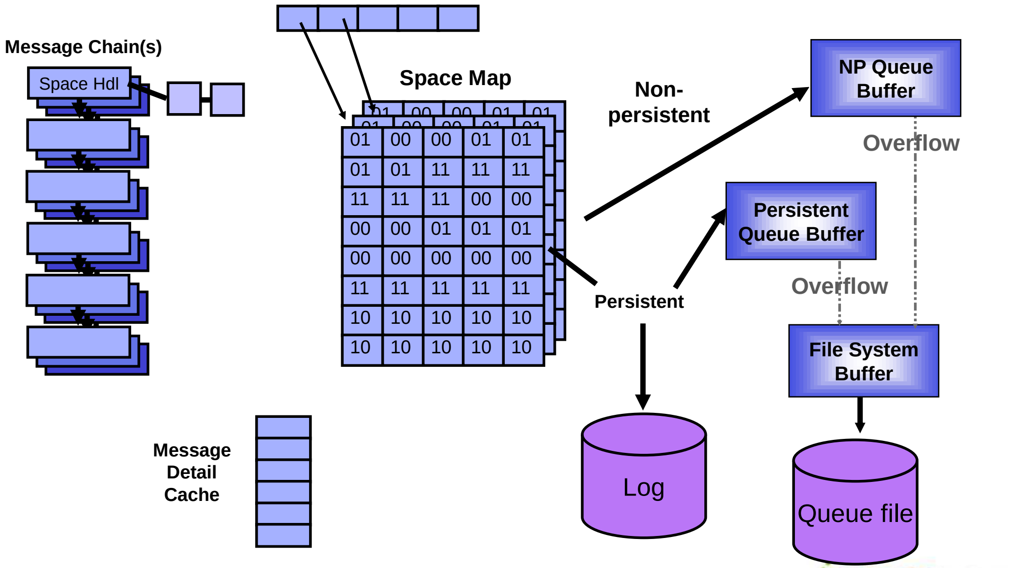
- MQCONN is a very heavyweight operation. Doing lots of these calls could cause throughput to suffer. Make sure that you don’t connect and disconnect a lot in your application, rather, connect once and then use this connection for all subsequent operations. Think carefully about any encapsulation you might do in your OO applications, make sure that the encapsulation does not cause you to do lots of MQCONNs and MQDISCs.

- MQPUT1
  If just putting a single message to a queue, MQPUT1 is going to be cheaper than MQOPEN, MQPUT and MQCLOSE. This is because it is only necessary to cross over from the application address space into the queue manager address space once, rather than the three times required for the separate calls. Under the covers inside the queue manager the MQPUT1 is implemented as an MQOPEN followed by MQPUT and finally the MQCLOSE. The cost saving for a single put to a queue is the switching from application to queue manager address space. Of course, if multiple messages need to be put to the queue then the queue should be opened first and them MQPUT used. It is a relatively expensive operation to open a queue.

- Exclusive use of queues
  Opening queues for exclusive use can help with sequencing issues, but it is a good idea to investigate whether other solutions are available. Exclusive use will make it harder to add extra tasks to process more work if needed in the future. Possible solutions are partitioning the data on the queue so that different tasks can work on different parts of the queue data (get by CorrelID can be used for this). This will enable more tasks to process the queue while maintaining ordering within the partitioned part of the data.

SHARE in Atlanta 2012

In the Depths of a Queue



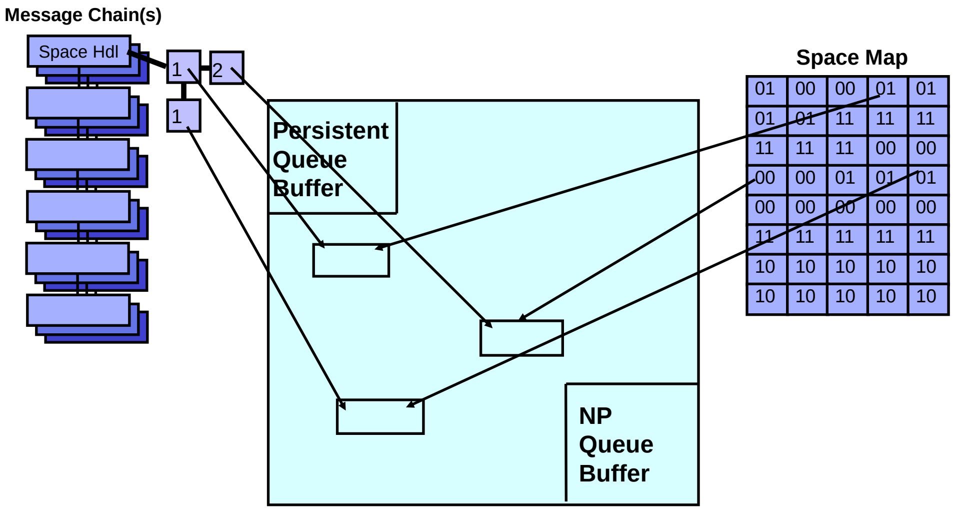
01	00	00	01	01
01	01	11	11	11
11	11	11	00	00
00	00	01	01	01
00	00	00	00	00
11	11	11	11	11
10	10	10	10	10
10	10	10	10	10

In the Depths of a Queue

N O T E S

- A queue is loaded into memory in the following structures, taking up around 60k in current versions (down from 300k in earlier versions). Session specific data is stored separately for queue browse cursors etc.
- Message Chains: Each message on a queue has an entry in a message chain. All messages, persistent and non-persistent, committed and uncommitted, appear in one of the message chains. There are actually 10 message chains - one for each message priority. The message chain is a linked list of 32 byte "space handles" made up of a hash of the message id and correl id, message expiry time, flags, the location of the message head in the queue. If the messages is fragmented, the handles link to space handles for the other parts of the message.
  - The width of the hash for msgid/correlid was doubled in MQ V6 on 64-bit queue managers; those systems also build better indexes for searching by correlid
- Message Details Cache: This is a table of selected message attributes for the 512 most recently used messages. This optimises access for messages which don't stay on the queue very long. It contains details of the message ID, Correl ID etc.
- Space Map: A map is kept to manage the space in the queue buffers and queue file. The queue is split up into blocks of 512 bytes which contain messages or parts of a message. 2 bits are used to represent each block, with different values to indicate if a block is Free (10), Free and allocated (11), contains NP data (00), or contains persistent data (01).
- Non-Persistent and Persistent Queue Buffers, and the Queue File: Messages are stored in shared memory buffers by preference. If the buffers overflow, they are written to the file system buffer but we never perform a synchronous disk write for an NP message. The buffers default to 64k, and 128k for the NP and P buffers respectively, doubled on the 64-bit queue managers.
- Log: Whenever a persistent message is put or got, at least one log record is written. If the message is put or got outside syncpoint, the log record will be written synchronously. If it is done inside syncpoint, a synchronous write is not required until commit or rollback.
  - For a transaction containing only non-persistent message operations, we don't write any log records at all.
  - There is an exception to the rule about writing log records for persistent messages - one scenario allows us to pass messages directly between applications without any I/O provided the messages are not part of a transaction (outside syncpoint).
- Additional tables are maintained so that segmented and grouped messages can be recognised and retrieved when flags such as MQGMO_COMPLETE_MSG are issued.

Views of the Queue



01	00	00	01	01
01	01	11	11	11
11	11	11	00	00
00	00	01	01	01
00	00	00	00	00
11	11	11	11	11
10	10	10	10	10
10	10	10	10	10

Views of the Queue

NOTES

- Ultimately the message chains and the space map are referring to the same storage. The space handles are referring to the storage which is in use and allows the queue manager to store and retrieve messages. The space map allows the queue manager to keep track of storage which is not in use.
- The queue file is split into 512 byte blocks. If a message is larger than 512 bytes it will be fragmented across different 512 byte blocks. A chain of space handles is created in the message chains to indicate where all parts of the message are held. These space handles may not be adjacent to each other in the queue buffer - the next available free block will be allocated. On the slide message 1 takes up 2 blocks, indicating by the two space handles marked '1'.
- When an MQGET of a persistent message is performed the space handle of the message just got will be placed into the log. There is no need to describe the data removed. We only keep track of which parts of storage became free. The queue buffer must therefore remain in a consistent state with the logs otherwise if we came to undo the MQGET we might overwrite data. Therefore before we can undo any operations we must ensure the log and the queue buffer are synchronized.
- Queues are unloaded in two phases: the first phase occurs at the first checkpoint after the last open handle to a queue is closed, the second phase then occurs if the queue remains unreferenced to the next checkpoint. Shrinking of the queue file occurs during the first of these two phases of unload. Checkpoints are typically taken every 10,000 recoverable (i.e persistent) operations. If all message operations are for non-persistent messages then checkpoints could be very infrequent.
  - Older versions of MQ were VERY conservative about releasing queue space back to the OS; V6.0 is more aggressive in releasing unused queue space. Originally MQ would will release unused queue space if the queue is idle (no puts or gets to the queue) for 5 consecutive checkpoints, or if the queue is empty at the time of the checkpoint for 20 consecutive checkpoints.
  - From MQ V6.0, we compare the actual size of the queue file with the required size of the queue file every time that queue is checkpointed and truncates the queue file if it is oversized by both 1% and 16KB (regardles of how many open handles reference the queue or the current qdepth, or the number of puts and gets since the last checkpoint).
- A queue is checkpointed when a checkpoint occurs AND a recoverable (i.e.persistent) update to the queue has been made since the last checkpoint. MQ will always truncate the queue to a minimal size when a CLEAR QL command is issued.

Tuning Queue Buffers

- Increasing buffers can improve performance
  - More information can be kept in memory, without flushing to disk
- But no documented external mechanism to do it
- SupportPac MS0P (Cat2) includes “Qtune” program
- To display and modify buffer sizes
  - Some sizes are given as “QMgr default”, which can change by release
- Also can set the maximum qFile size
- For MQ V6, V7 and V7.1

```
c:\> java -jar qtune.jar -d c:\mqm\qmgrs\QMA\queues\SYSTEM!DEFAULT!LOCAL!QUEUE

File c:\mqm\qmgrs\QMA\queues\SYSTEM!DEFAULT!LOCAL!QUEUE\q
Stored npBuff     = 64 kB
Stored pBuff      = QMgr default
Stored maxQSize   = 2,097,151 MB
```

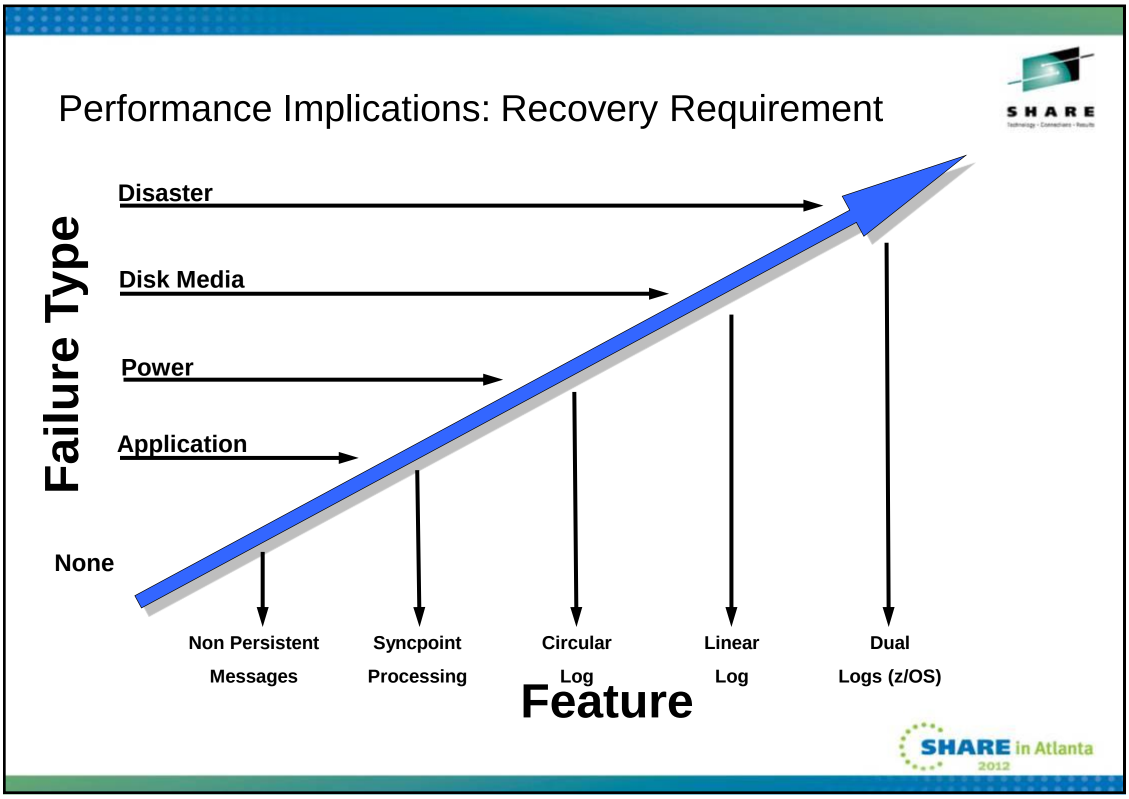
SHARE in Atlanta 2012

Tuning Queue Buffers

NOTES

- SupportPac MS0P contains a number of utilities. Many are extensions to the MQ Explorer, but there is also the qtune program to modify the queue buffer sizes.
- The queue manager must be stopped when changing these buffers, but you can display current values without stopping it.
- Remember that the buffers take up memory while the queue is opened, so do not over-size every queue on the queue manager.

SHARE in Atlanta 2012



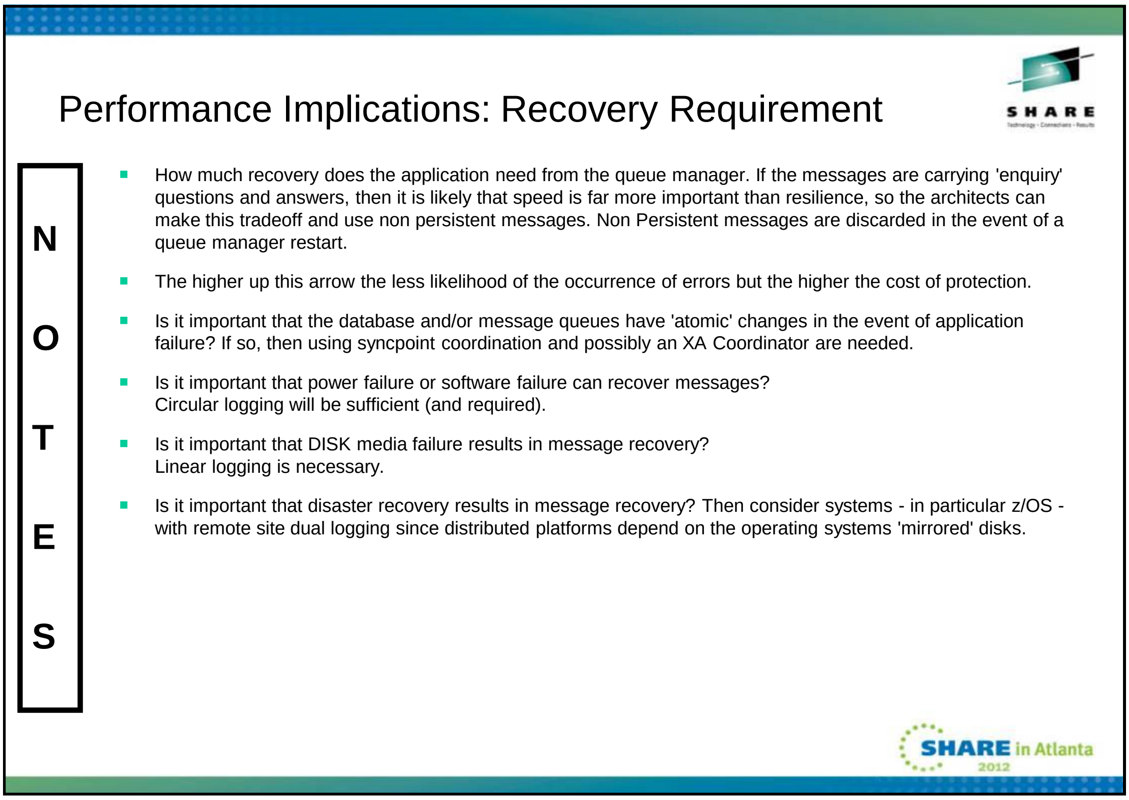
Performance Implications: Recovery Requirement

Failure Type

- Disaster
- Disk Media
- Power
- Application
- None

Feature

- Non Persistent Messages
- Syncpoint Processing
- Circular Log
- Linear Log
- Dual Logs (z/OS)



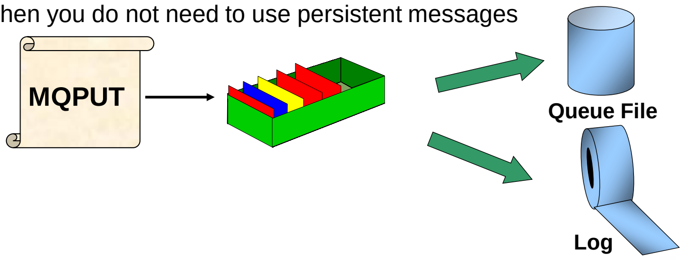
Performance Implications: Recovery Requirement

N O T E S

- How much recovery does the application need from the queue manager. If the messages are carrying 'enquiry' questions and answers, then it is likely that speed is far more important than resilience, so the architects can make this tradeoff and use non persistent messages. Non Persistent messages are discarded in the event of a queue manager restart.
- The higher up this arrow the less likelihood of the occurrence of errors but the higher the cost of protection.
- Is it important that the database and/or message queues have 'atomic' changes in the event of application failure? If so, then using syncpoint coordination and possibly an XA Coordinator are needed.
- Is it important that power failure or software failure can recover messages?
  Circular logging will be sufficient (and required).
- Is it important that DISK media failure results in message recovery?
  Linear logging is necessary.
- Is it important that disaster recovery results in message recovery? Then consider systems - in particular z/OS - with remote site dual logging since distributed platforms depend on the operating systems 'mirrored' disks.

Performance Implications: Persistence

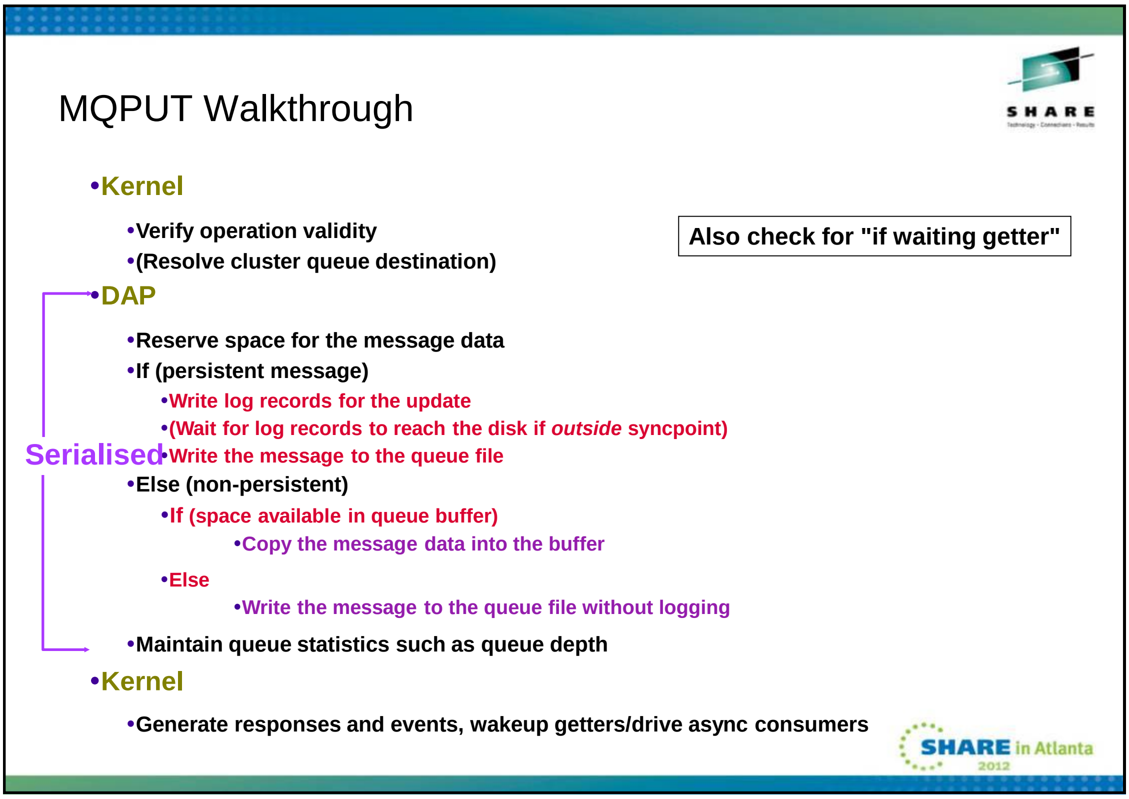
- Log bandwidth is going to restrict throughput
  - Put the log files on the fastest disks you have
- Persistent messages are the main things requiring recovery after an outage
  - Can significantly affect restart times
- Why use persistence?
  - False assumption that persistence is for "important" data and nonpersistent for when you don't care
  - The real reason for persistent messages is to reduce application complexity
  - With persistent, apps do not need logic to detect and deal with lost messages
  - If your app (or operational procedures) can detect and deal with lost messages, then you do not need to use persistent messages



Performance Implications: Persistence

N
O
T
E
S

- If persistent messages are used then the maximum rate that messages can be put is typically going to be limited by the logging bandwidth available. This is normally the over riding factor as to the throughput available when using persistent messages.
- As persistent messages need to be made available when a queue manager is restarted, they may need to be recovered if there has been a failure (could be queue manager or system etc). The persistent workload that has been done is the main key as to how long it is going to take to restart the queue manager after a failure. There are other factors involved which include the frequency of checkpoints etc, but ultimately it all comes down to the fact that persistent messages have been used. If there has been a failure then no recovery is required on non-persistent messages, the pages that contained them are simply marked as not used.
- If your application (or operational procedures) can detect and deal with lost messages, then you do not need to use persistent messages.
- Consider:
  - A bank may have sophisticated cross checking in its applications and in its procedures to ensure no transactions are lost or repeated.
  - An airline might assume (reasonably) that a passenger who does not get a response to a reservation request within a few seconds will check what happened and if necessary repeat or cancel the reservation.
- In both cases the messages are important but the justification for persistence may be weak.



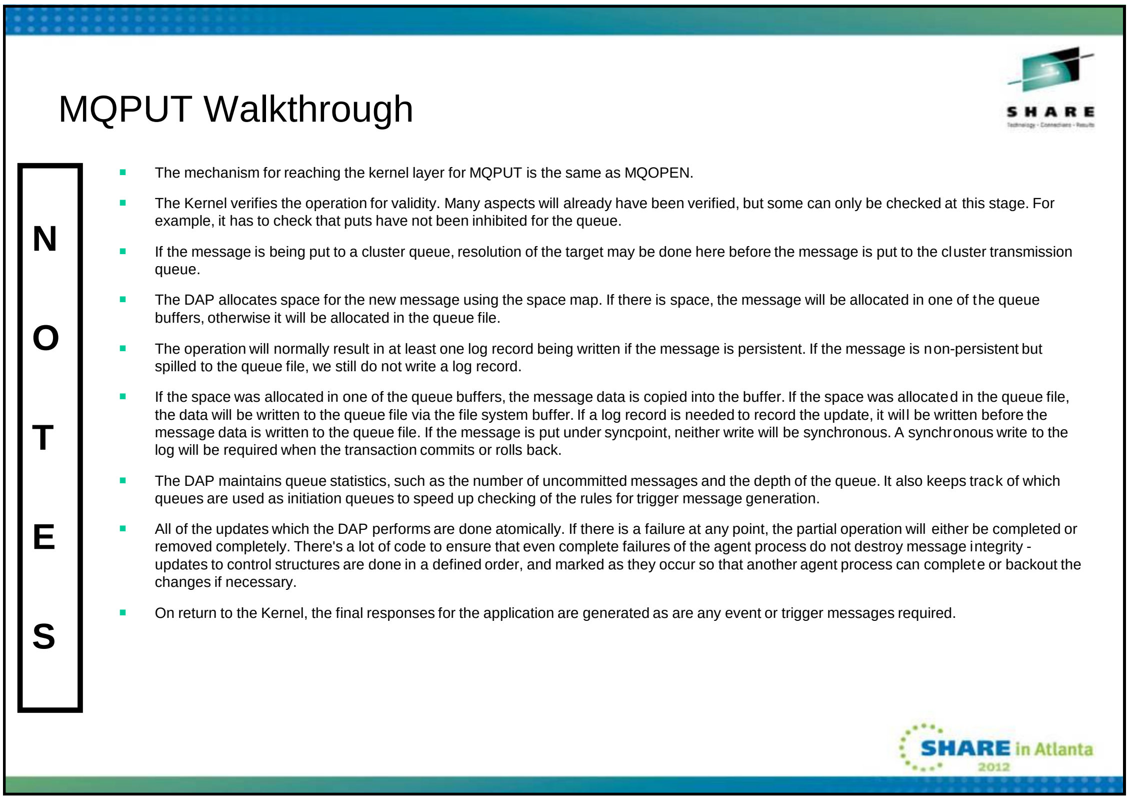
MQPUT Walkthrough

- Kernel
  - Verify operation validity
  - (Resolve cluster queue destination)
- DAP
  - Reserve space for the message data
  - If (persistent message)
    - Write log records for the update
    - (Wait for log records to reach the disk if *outside* syncpoint)
    - Write the message to the queue file
  - Else (non-persistent)
    - If (space available in queue buffer)
      - Copy the message data into the buffer
    - Else
      - Write the message to the queue file without logging
  - Maintain queue statistics such as queue depth
- Kernel
  - Generate responses and events, wakeup getters/drive async consumers

Serialised

Also check for "if waiting getter"

SHARE in Atlanta 2012



MQPUT Walkthrough

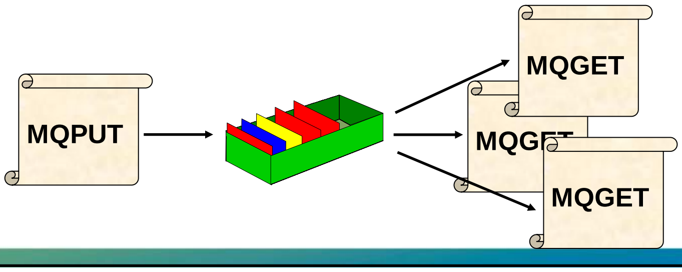
NOTES

- The mechanism for reaching the kernel layer for MQPUT is the same as MQOPEN.
- The Kernel verifies the operation for validity. Many aspects will already have been verified, but some can only be checked at this stage. For example, it has to check that puts have not been inhibited for the queue.
- If the message is being put to a cluster queue, resolution of the target may be done here before the message is put to the cluster transmission queue.
- The DAP allocates space for the new message using the space map. If there is space, the message will be allocated in one of the queue buffers, otherwise it will be allocated in the queue file.
- The operation will normally result in at least one log record being written if the message is persistent. If the message is non-persistent but spilled to the queue file, we still do not write a log record.
- If the space was allocated in one of the queue buffers, the message data is copied into the buffer. If the space was allocated in the queue file, the data will be written to the queue file via the file system buffer. If a log record is needed to record the update, it will be written before the message data is written to the queue file. If the message is put under syncpoint, neither write will be synchronous. A synchronous write to the log will be required when the transaction commits or rolls back.
- The DAP maintains queue statistics, such as the number of uncommitted messages and the depth of the queue. It also keeps track of which queues are used as initiation queues to speed up checking of the rules for trigger message generation.
- All of the updates which the DAP performs are done atomically. If there is a failure at any point, the partial operation will either be completed or removed completely. There's a lot of code to ensure that even complete failures of the agent process do not destroy message integrity - updates to control structures are done in a defined order, and marked as they occur so that another agent process can complete or backout the changes if necessary.
- On return to the Kernel, the final responses for the application are generated as are any event or trigger messages required.

SHARE in Atlanta 2012

Put to a waiting getter

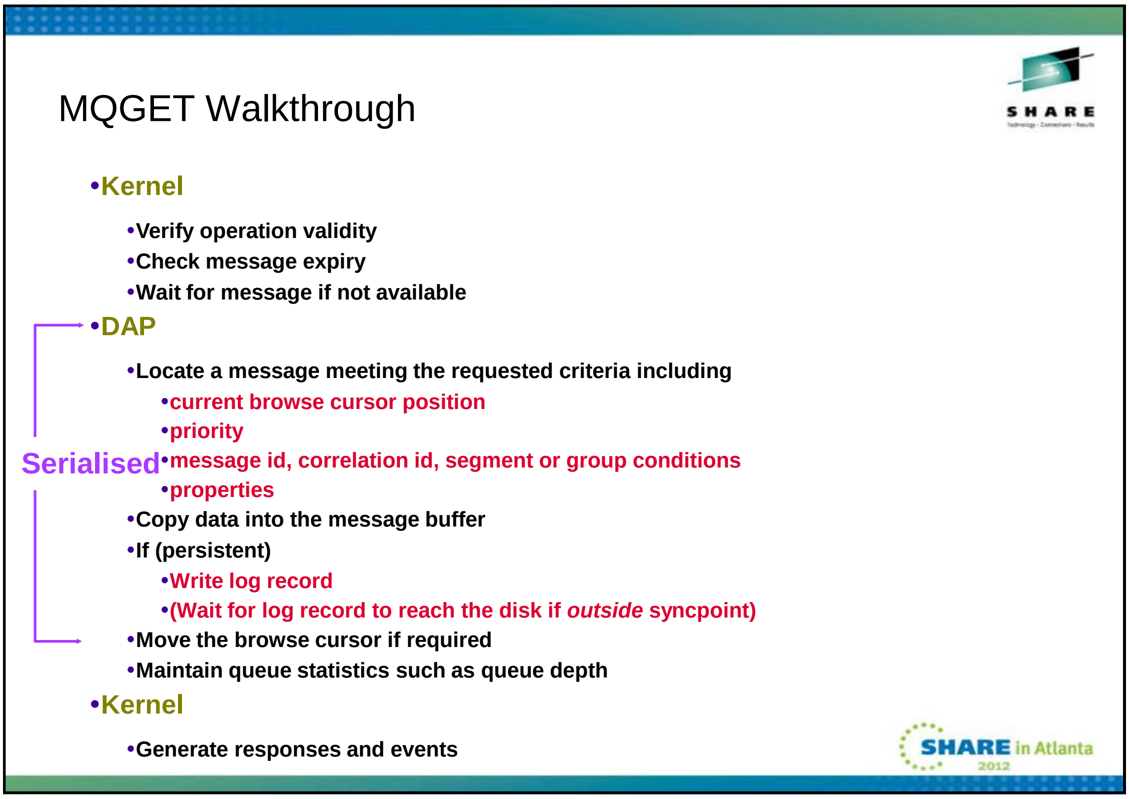
- MQPUT most efficient if there is getting application waiting
  - Having multiple applications processing queue increases percentage
- Only for out of syncpoint messages
  - Both persistent and non-persistent
- No queuing required
  - Removes a lot of processing of placing the message onto the queue
- Significantly reduces CPU cost and improves throughput



Put to a waiting getter

NOTES

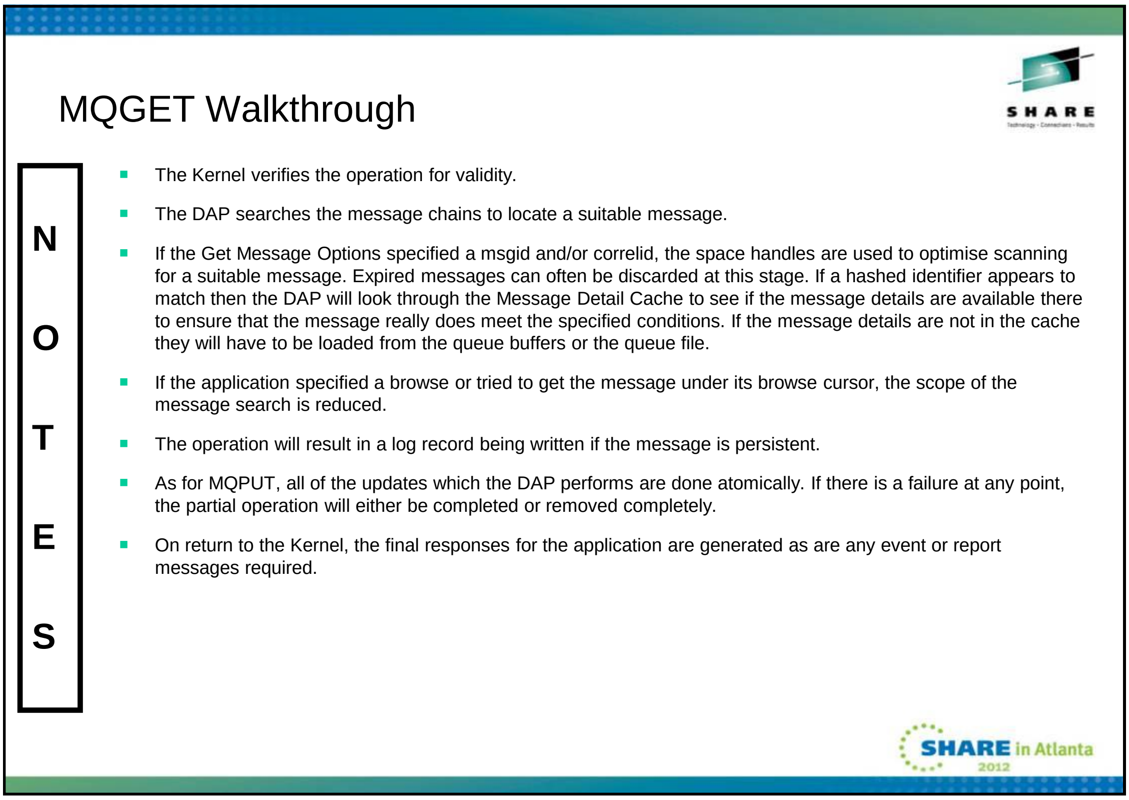
- "Put to a waiting getter" (aka I/O-avoidance) is a technique whereby a message may not actually be put onto a queue if there is an application already waiting to get the message. Certain conditions must be satisfied for this to occur, in particular the putter and getter must be processing the message outside syncpoint control (and on z/OS the message must also be non-persistent). If these conditions are met then the message will be transferred from the putter's buffer into the getter's buffer without actually touching the MQ queue. This removes a lot of processing involved in putting the message on the queue and therefore leads to increased throughput and lower CPU costs.
- When in "put to waiting getter" mode the Queue Manager will try to keep one thread 'hot'.
  - Distributed always tries to keep one thread 'hot'
- You should not expect to see "even" workload distribution between applications when they are all getting from the same queue



MQGET Walkthrough

- Kernel
  - Verify operation validity
  - Check message expiry
  - Wait for message if not available
- DAP
  - Locate a message meeting the requested criteria including
    - current browse cursor position
    - priority
    - message id, correlation id, segment or group conditions
    - properties
  - Copy data into the message buffer
  - If (persistent)
    - Write log record
    - (Wait for log record to reach the disk if *outside* syncpoint)
  - Move the browse cursor if required
  - Maintain queue statistics such as queue depth
- Kernel
  - Generate responses and events

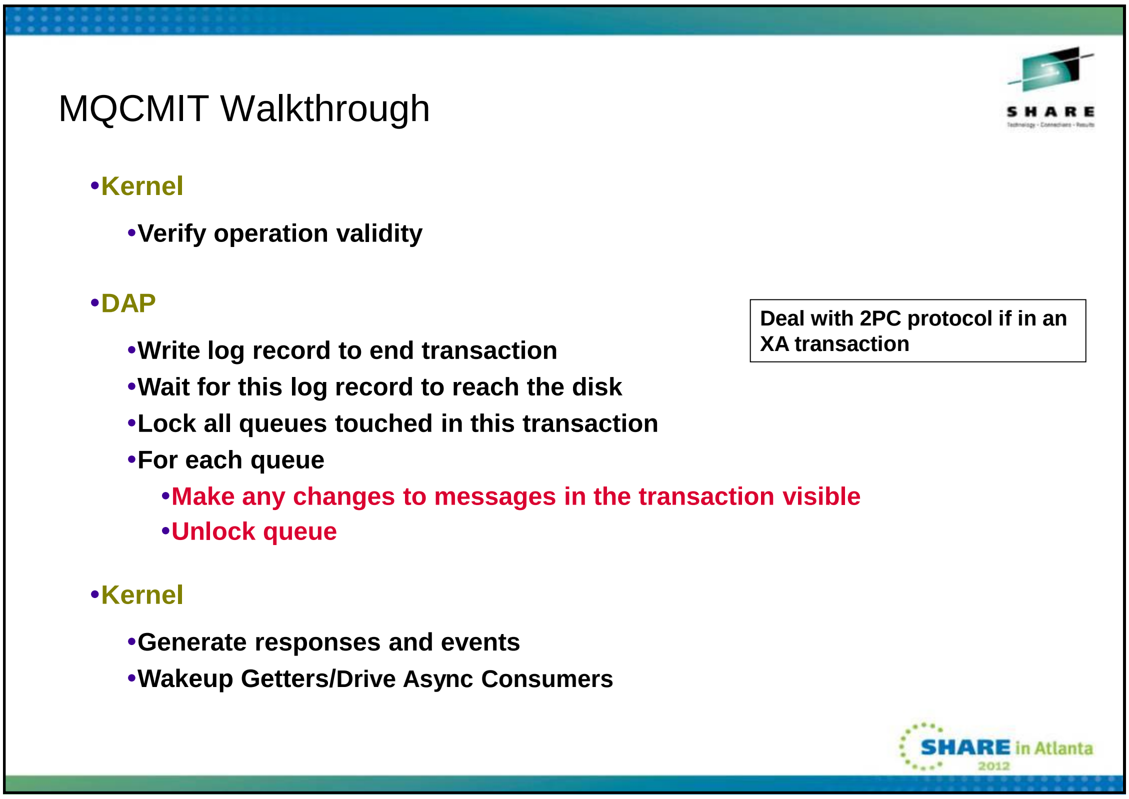
Serialised



MQGET Walkthrough

NOTES

- The Kernel verifies the operation for validity.
- The DAP searches the message chains to locate a suitable message.
- If the Get Message Options specified a msgid and/or correlid, the space handles are used to optimise scanning for a suitable message. Expired messages can often be discarded at this stage. If a hashed identifier appears to match then the DAP will look through the Message Detail Cache to see if the message details are available there to ensure that the message really does meet the specified conditions. If the message details are not in the cache they will have to be loaded from the queue buffers or the queue file.
- If the application specified a browse or tried to get the message under its browse cursor, the scope of the message search is reduced.
- The operation will result in a log record being written if the message is persistent.
- As for MQPUT, all of the updates which the DAP performs are done atomically. If there is a failure at any point, the partial operation will either be completed or removed completely.
- On return to the Kernel, the final responses for the application are generated as are any event or report messages required.

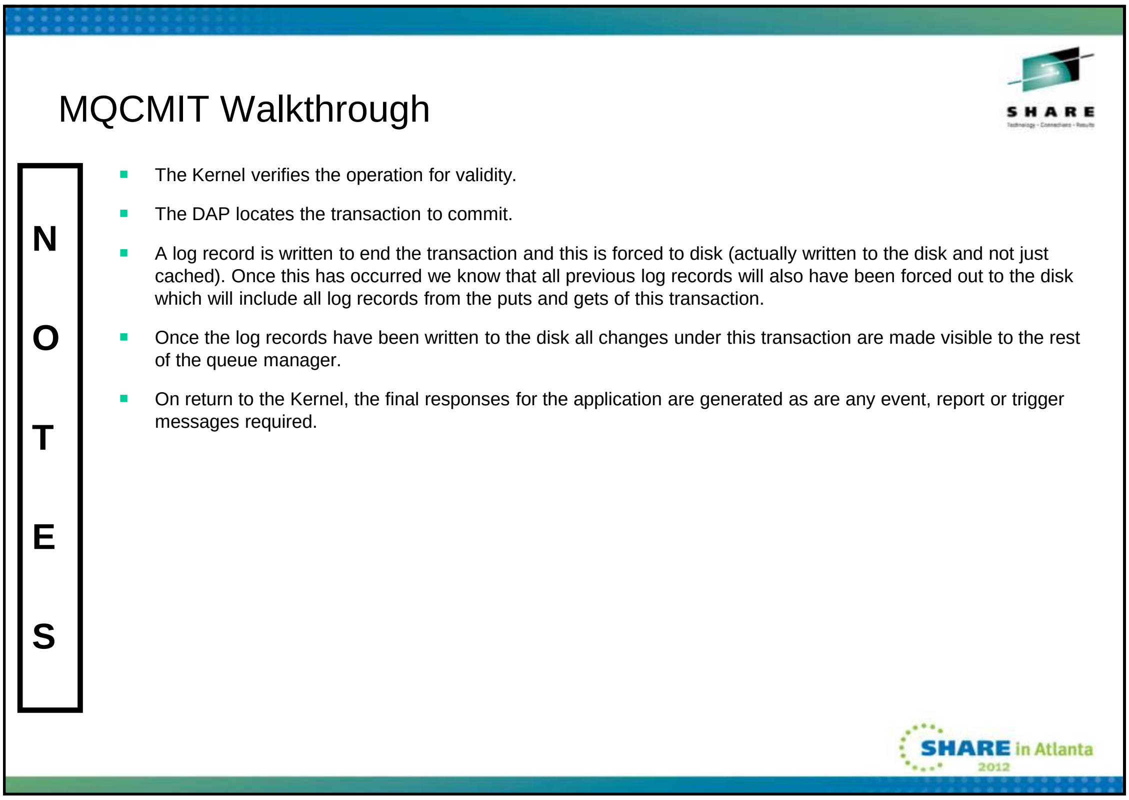


MQCMIT Walkthrough

- Kernel
  - Verify operation validity
- DAP
  - Write log record to end transaction
  - Wait for this log record to reach the disk
  - Lock all queues touched in this transaction
  - For each queue
    - Make any changes to messages in the transaction visible
    - Unlock queue
- Kernel
  - Generate responses and events
  - Wakeup Getters/Drive Async Consumers

Deal with 2PC protocol if in an XA transaction

SHARE in Atlanta 2012



MQCMIT Walkthrough

NOTES

- The Kernel verifies the operation for validity.
- The DAP locates the transaction to commit.
- A log record is written to end the transaction and this is forced to disk (actually written to the disk and not just cached). Once this has occurred we know that all previous log records will also have been forced out to the disk which will include all log records from the puts and gets of this transaction.
- Once the log records have been written to the disk all changes under this transaction are made visible to the rest of the queue manager.
- On return to the Kernel, the final responses for the application are generated as are any event, report or trigger messages required.

SHARE in Atlanta 2012

Performance Implications: Syncpoint

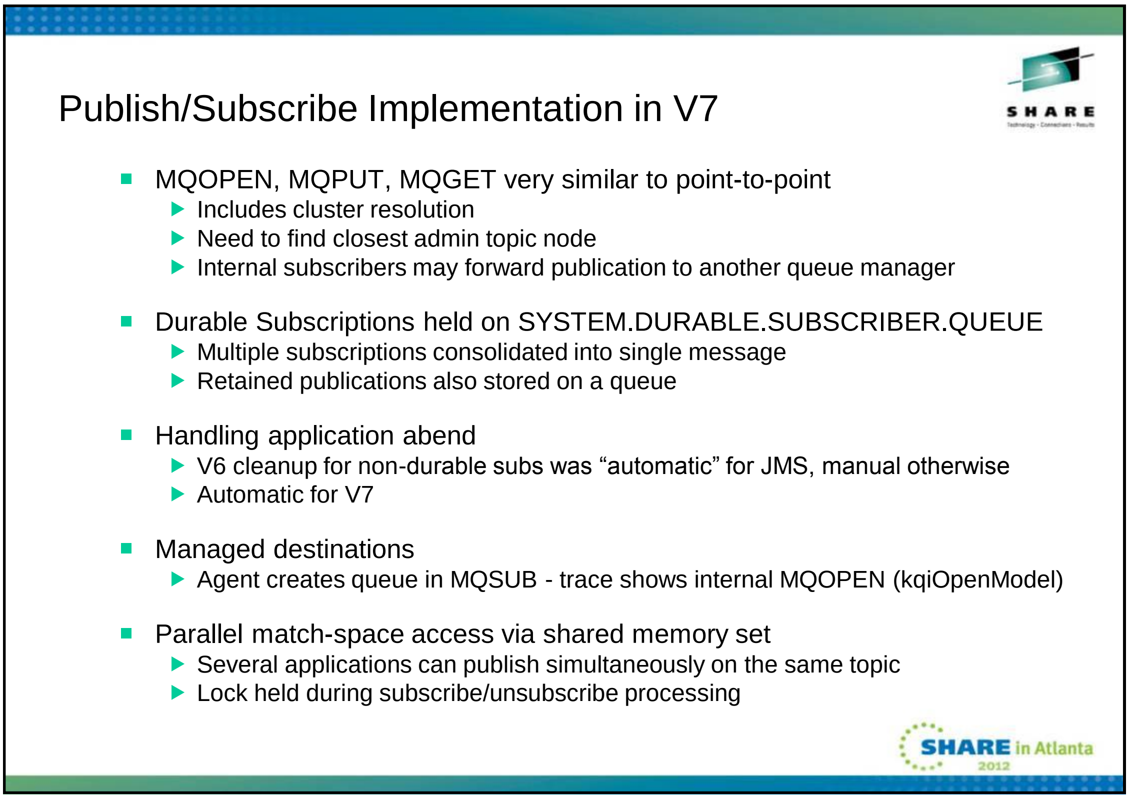
- Do you need it?
  - Yes, when a set of work needs to either all be performed, or all not performed
- Maximum Size of UOW can be limited
  - QMGR MAXUMSGS parm
  - Set to sensible value to avoid runaway applications
- Make sure you keep the size of your UOWs small
  - Don’t forget to end the UOW
- Cheaper to process in syncpoint for persistent messages
  - Up to a point, not huge UOWs
  - Log not forced after every MQPUT/MQGET
- Useful even when only a single message inside syncpoint
  - And running multiple parallel applications

Performance Implications: Syncpoint

NOTES

- The size of a UOW (number of messages in it) can be controlled via the MAXUMSGS queue manager parameter. This however has no impact on the duration of the UOW, it simply controls the maximum number of messages that a UOW can contain. It prevents runaway applications

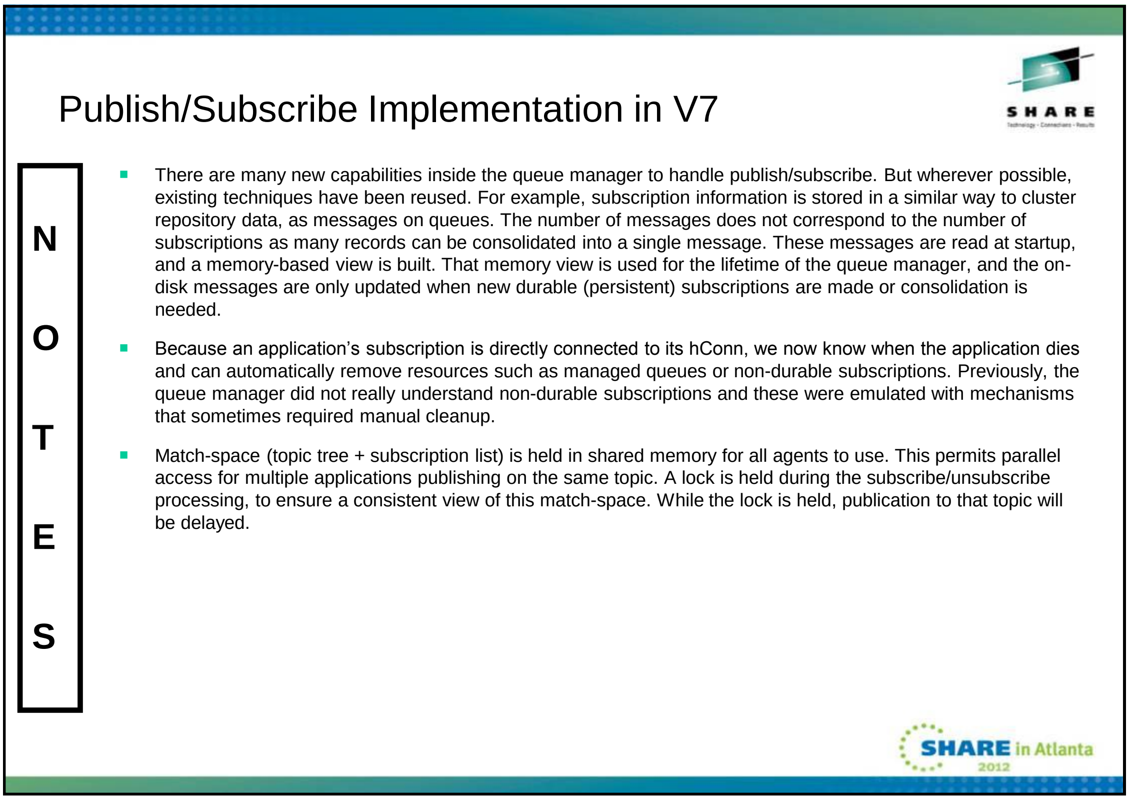
- It can be considerably cheaper to process multiple persistent messages inside syncpoint rather than processing them outside syncpoint. This is because if persistent messages are being used outside of syncpoint, it is necessary to force them to the log as soon as they are put, to ensure that they are available if a failure occurs. If they are processed inside syncpoint it is only necessary to force the log when the UOW is committed. This means that we will spend less time waiting for the pages to be forced out to disk. In effect the cost of forcing the UOW out to disk is shared between all of the messages put and got, rather than each one having to bear the cost. Syncpoint should not be considered as an ‘overhead’.



Publish/Subscribe Implementation in V7

- MQOPEN, MQPUT, MQGET very similar to point-to-point
  - Includes cluster resolution
  - Need to find closest admin topic node
  - Internal subscribers may forward publication to another queue manager
- Durable Subscriptions held on SYSTEM.DURABLE.SUBSCRIBER.QUEUE
  - Multiple subscriptions consolidated into single message
  - Retained publications also stored on a queue
- Handling application abend
  - V6 cleanup for non-durable subs was “automatic” for JMS, manual otherwise
  - Automatic for V7
- Managed destinations
  - Agent creates queue in MQSUB - trace shows internal MQOPEN (kqiOpenModel)
- Parallel match-space access via shared memory set
  - Several applications can publish simultaneously on the same topic
  - Lock held during subscribe/unsubscribe processing

SHARE in Atlanta 2012



Publish/Subscribe Implementation in V7

NOTES

- There are many new capabilities inside the queue manager to handle publish/subscribe. But wherever possible, existing techniques have been reused. For example, subscription information is stored in a similar way to cluster repository data, as messages on queues. The number of messages does not correspond to the number of subscriptions as many records can be consolidated into a single message. These messages are read at startup, and a memory-based view is built. That memory view is used for the lifetime of the queue manager, and the on-disk messages are only updated when new durable (persistent) subscriptions are made or consolidation is needed.
- Because an application’s subscription is directly connected to its hConn, we now know when the application dies and can automatically remove resources such as managed queues or non-durable subscriptions. Previously, the queue manager did not really understand non-durable subscriptions and these were emulated with mechanisms that sometimes required manual cleanup.
- Match-space (topic tree + subscription list) is held in shared memory for all agents to use. This permits parallel access for multiple applications publishing on the same topic. A lock is held during the subscribe/unsubscribe processing, to ensure a consistent view of this match-space. While the lock is held, publication to that topic will be delayed.

SHARE in Atlanta 2012

Message Processing in V7

- Persistent pubs switch to non-persistent-ish for non-durable subscriptions
  - Does not change the reliability level
  - Messages are not logged, but they keep the “persistent” flag
  - Improves performance
- Properties stored as part of the message
  - Logged for persistence, rcdmqimg etc
  - Written to disk in either RFH2 or an “internal” format
  - Converted to application-required format during MQGET
- Selectors on queues can cause all messages to be browsed
  - Queue lock may be held during selection

Message Processing in V7

NOTES

- A non-durable subscriber is permitted to miss messages if it abends, so there is no point in doing a full hardening of those messages. But they do need to know if the message was originally a persistent message, in case they want to forward it unchanged.

Channels

Channels

N

O

T

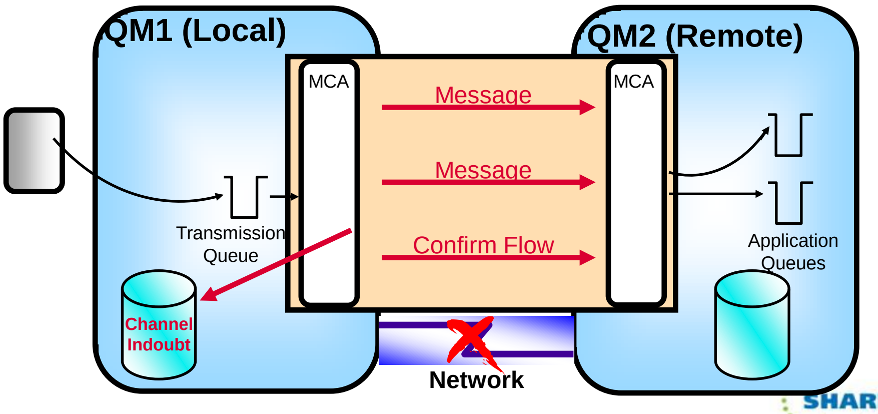
E

S

- How they work

Internal Resources

- Channel synchronisation uses ScratchPads
  - The SYNCQ holds channel status across restarts
  - A small area of data which can be part of 2-phase commit processing
  - Channel sync also uses file AMQRSYNA.DAT as an index into the scratchpads
  - Messages in an in-doubt batch cannot be reallocated by clustering algorithm



Internal Resources

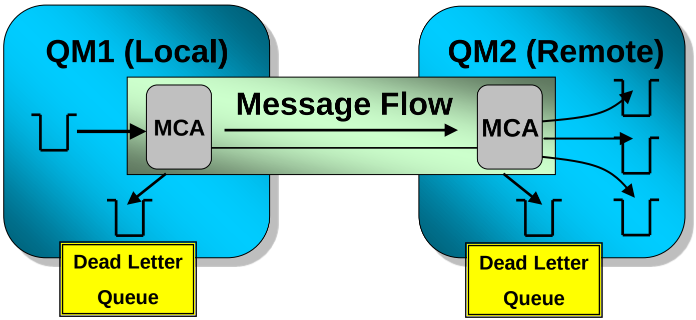
NOTES

- Here we can see the two ends of the channel transferring a batch of messages. Note how the MCA's write data to disk at the end of the batch. This is only done for recoverable batches. The data written contains the transaction id of the transaction used at the sending end to retrieve all the messages. Once the sender end has issued a 'confirm' flow to its partner it is 'indoubt' until it receives a response. In other words, the sender channel is not sure whether the messages have been delivered successfully or not. If there is a communications failure during this phase then the channel will end indoubt. When it reconnects to it's partner it will notice from the data store that it was indoubt with its partner and so will ask the other channel whether the last batch of messages were delivered or not. Using the answer the partner sends, the channel can decide whether to commit or rollback the messages on the transmission queue.

- This synchronisation data is viewable by issuing a DIS CHSTATUS(*) SAVED command. The values displayed should be the same at both ends of the channel.

- Note that if the channel is restarted when it is indoubt it will automatically resolve the indoubt. However, it can only do this if it is talking to the same partner. If the channel attributes are changed or a different Queue Manager takes over the IP address or a different channel serving the same transmission queue is started then the channel will end immediately with message saying that it is still indoubt with a different Queue Manager. The user must start the channel directing it at the correct Queue Manager or resolve the indoubt manually by issuing the RESOLVE CHANNEL command. Note that in this case the user should use the output from DIS CHS(*) SAVED to ensure that the correct action COMMIT or BACKOUT is chosen.

Channel Protocol



- Send from XmitQ until Batchsize/limit or Empty & Batchint expired.
- Store 'indoubt' record and send 'End-of-Batch' to Remote MCA.
- Remote MCA updates Batch Sequence, MQCMIT, sends ack
- Local MCA updates Batch Sequence number and issues MQCMIT.
- Pipeline Length =2 provides additional thread that will start processing next batch after 'End-of-Batch' sent to Remote MCA

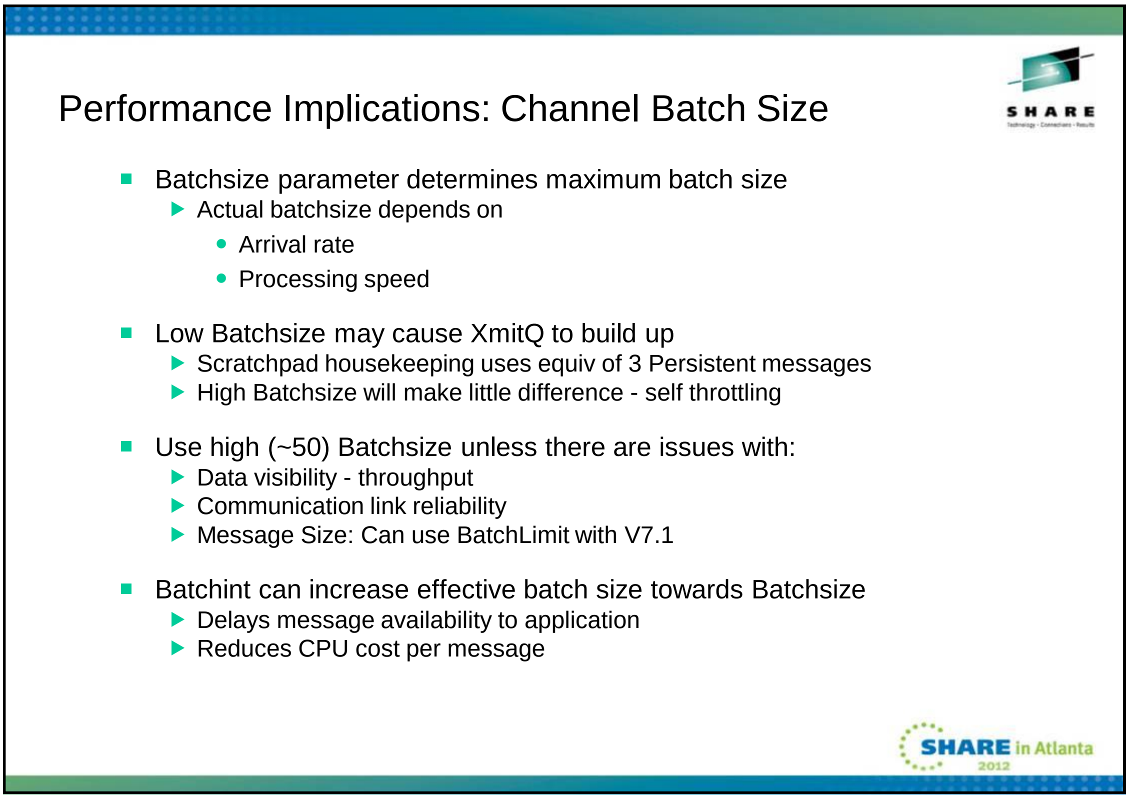
Channel Protocol

NOTES

- The channel operation conforms to a quite simple model:

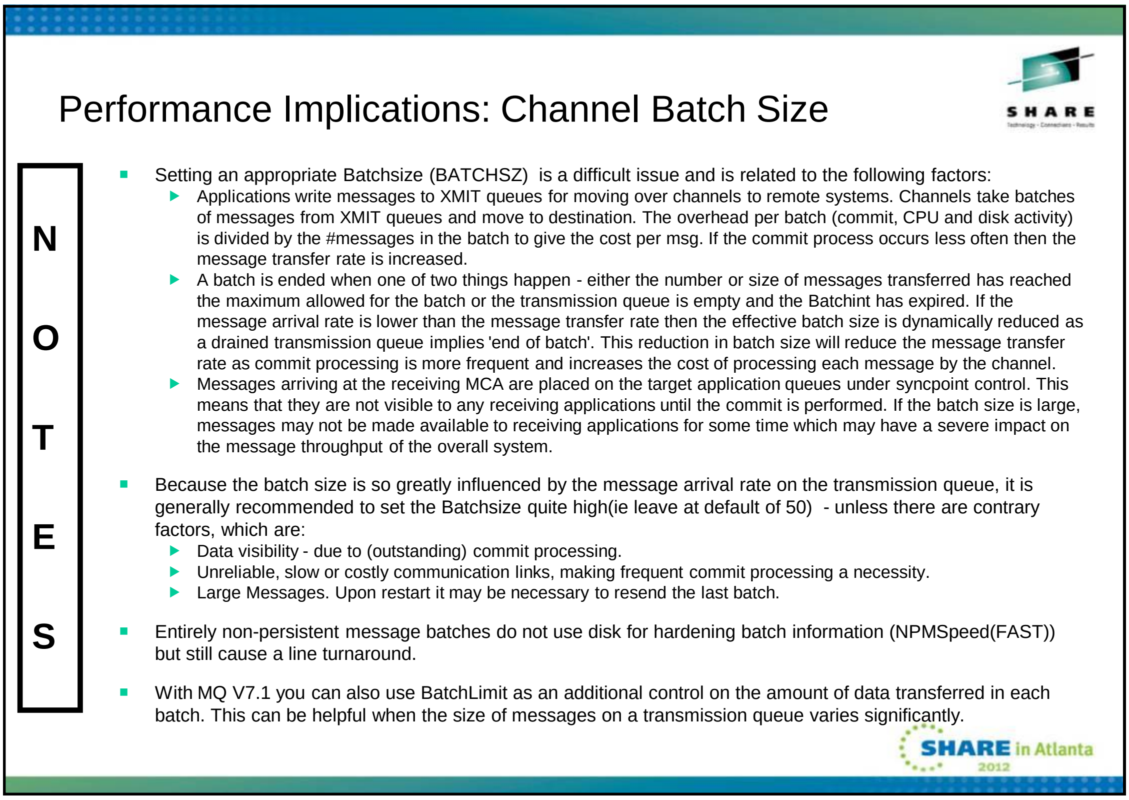
```
    Do until (batchsize/batchlimit reached) or (no more messages and batchint expired)
        Local MCA gets a message from the transmission queue
        A header is put on the data and sent using APPC, TCP etc.
      End
      Harden the message ids/indoubt flag
      Send "End of batch flag"
      Remote end commits
      Remote end sends "OK" flag back
      Local end updoubt synchronisation record to non-indoubt state and commits
```

- If there is any failure in the communications link or the MCA processes, then the protocol allows for re-synchronisation to take place and messages to be appropriately recovered.
- Probably the most misunderstood part of the message exchange protocol is Batchsize. Batchsize controls the frequency of commit flows used by the sending MCA. This, in turn, controls how often the communications line is turned around and - perhaps more importantly - how quickly messages at the receiving side are committed on the target application queues. The value for Batchsize that is negotiated at channel start-up is the maximum Batchsize only - if the transmission queue becomes empty then a batch of messages is automatically committed. Each batch containing Persistent messages uses the Scratchpad. The larger the effective batch size, the smaller is the resource cost per message on the channel. Batchint can increase the effective batch size and can reduce cost per message in the server.
- Pipelinelength=2 enable overlap of putting messages onto TCP while waiting for acknowledgment of previous batch. This enables overlap of sending messages while waiting for Batch synchronization at remote system.



Performance Implications: Channel Batch Size

- Batchsize parameter determines maximum batch size
  - Actual batchsize depends on
    - Arrival rate
    - Processing speed
- Low Batchsize may cause XmitQ to build up
  - Scratchpad housekeeping uses equiv of 3 Persistent messages
  - High Batchsize will make little difference - self throttling
- Use high (~50) Batchsize unless there are issues with:
  - Data visibility - throughput
  - Communication link reliability
  - Message Size: Can use BatchLimit with V7.1
- Batchint can increase effective batch size towards Batchsize
  - Delays message availability to application
  - Reduces CPU cost per message

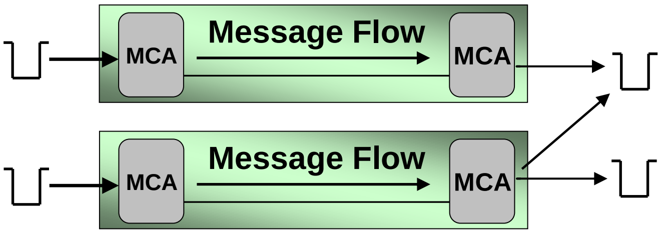


Performance Implications: Channel Batch Size

NOTES

- Setting an appropriate Batchsize (BATCHSZ) is a difficult issue and is related to the following factors:
  - Applications write messages to XMIT queues for moving over channels to remote systems. Channels take batches of messages from XMIT queues and move to destination. The overhead per batch (commit, CPU and disk activity) is divided by the #messages in the batch to give the cost per msg. If the commit process occurs less often then the message transfer rate is increased.
  - A batch is ended when one of two things happen - either the number or size of messages transferred has reached the maximum allowed for the batch or the transmission queue is empty and the Batchint has expired. If the message arrival rate is lower than the message transfer rate then the effective batch size is dynamically reduced as a drained transmission queue implies 'end of batch'. This reduction in batch size will reduce the message transfer rate as commit processing is more frequent and increases the cost of processing each message by the channel.
  - Messages arriving at the receiving MCA are placed on the target application queues under syncpoint control. This means that they are not visible to any receiving applications until the commit is performed. If the batch size is large, messages may not be made available to receiving applications for some time which may have a severe impact on the message throughput of the overall system.
- Because the batch size is so greatly influenced by the message arrival rate on the transmission queue, it is generally recommended to set the Batchsize quite high(ie leave at default of 50) - unless there are contrary factors, which are:
  - Data visibility - due to (outstanding) commit processing.
  - Unreliable, slow or costly communication links, making frequent commit processing a necessity.
  - Large Messages. Upon restart it may be necessary to resend the last batch.
- Entirely non-persistent message batches do not use disk for hardening batch information (NPMSpeed(FAST)) but still cause a line turnaround.
- With MQ V7.1 you can also use BatchLimit as an additional control on the amount of data transferred in each batch. This can be helpful when the size of messages on a transmission queue varies significantly.

Performance Implications: One or Multiple Channels



- Use one channel if it can handle the throughput
  - Monitor depth of XMIT queue
  - One high-use channel is more efficient than two low-use channels
    - The actual batch size will be larger
  - Multiple channels can yield some performance benefit
    - Depends on network and arrival rate
- Multiple channels for separate classes of work
  - Large messages only delay large message
  - Encryption cost on taken on worthwhile messages
  - Small interactive messages do not get delayed

Performance Implications: One or Multiple Channels

N O T E S

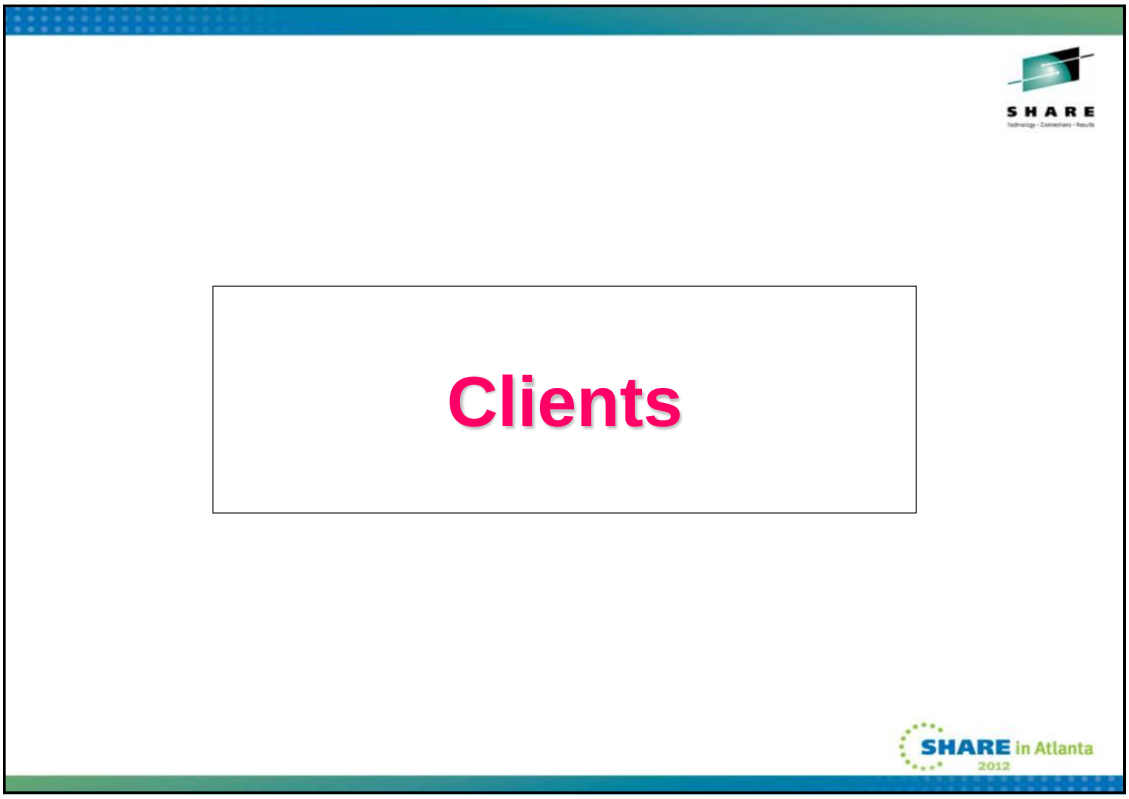
- For the reasons previously outlined, it is most appropriate to have as high a channel utilization as possible. This means that it is appropriate to have as few channels as can handle the load between any two queue managers.

- However, where there are different classes of message being moved around the MQ network, it may be appropriate to have multiple channels to handle the different classes.

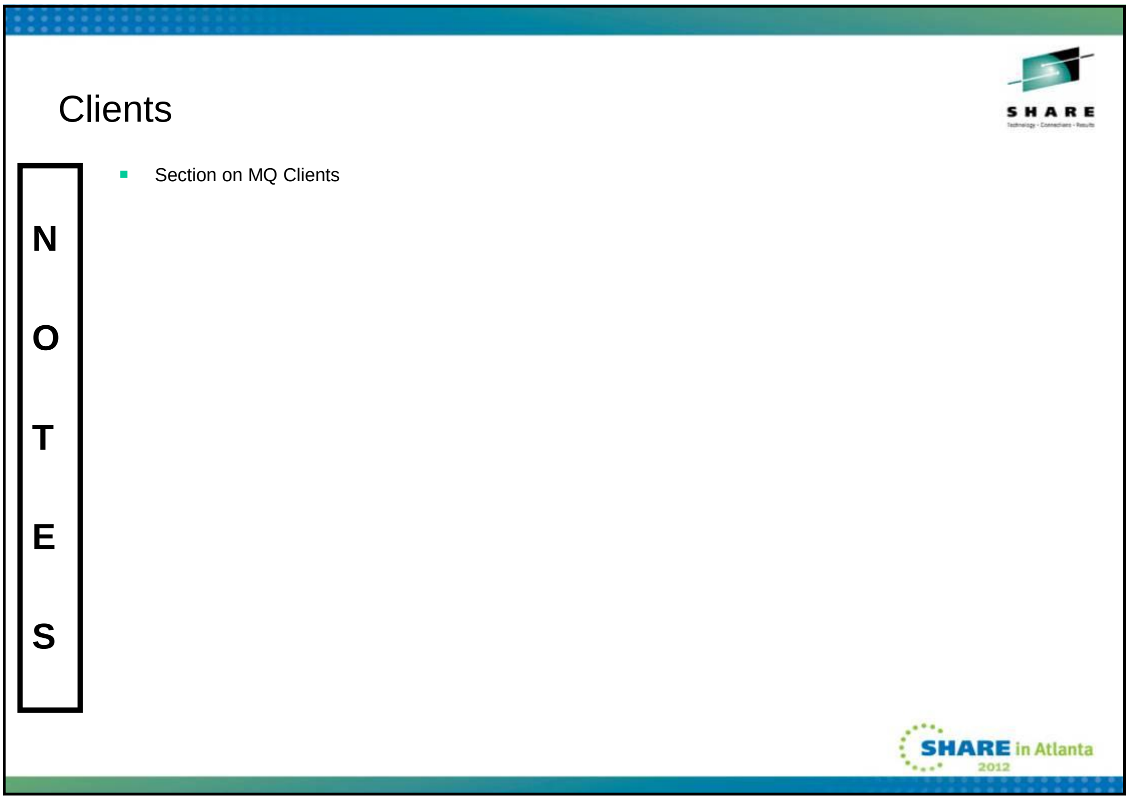
- Messages deemed to be 'more important' may be processed by a separate channel. While this may not be the most efficient method of transfer, it may be the most appropriate. Note that a similar effect may be achieved by making the transmission queue a priority order queue and placing these message at the head of the transmission queue.

- Very large messages may hold up smaller messages with a corresponding deterioration in message throughput. In this instance, providing a priority transmission queue will not solve the problem as a large message must be completely transferred before a high priority message is handled. In this case, a separate channel for large messages will enable other messages to be transferred faster.

- If it is appropriate to route message traffic through different parts of the underlying network, multiple channels will enable those different network routes to be utilized.



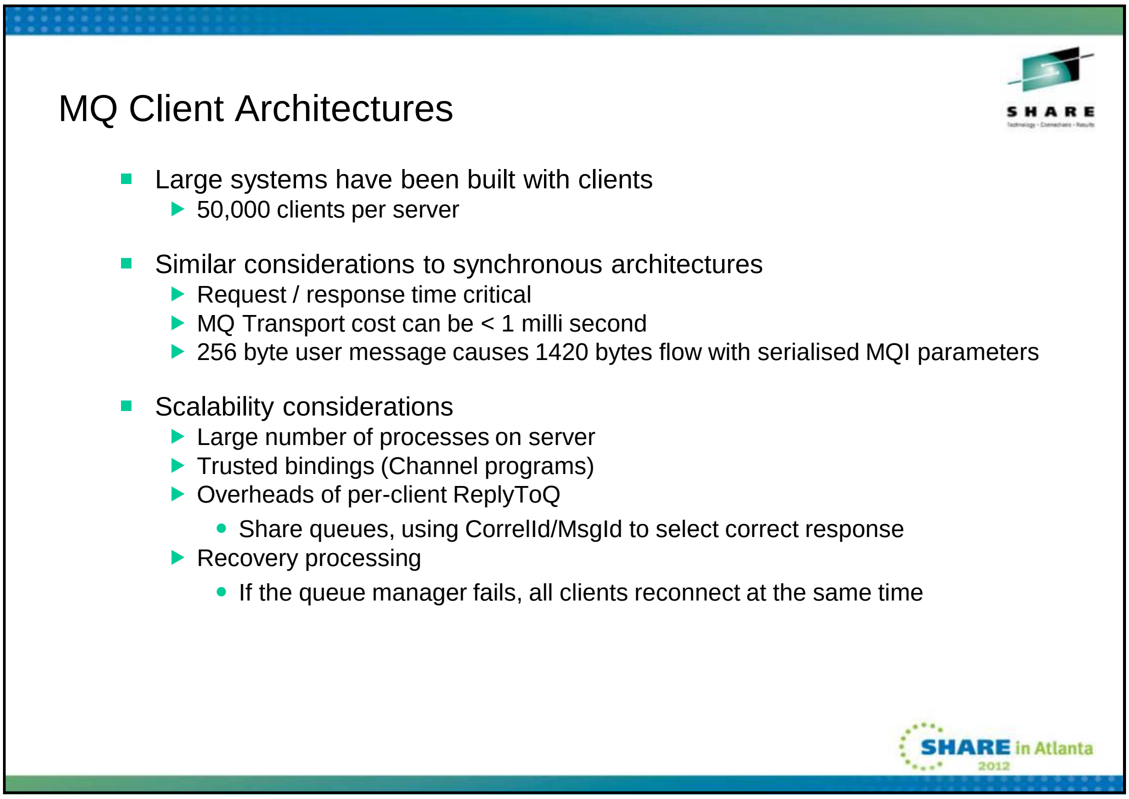
Clients



Clients

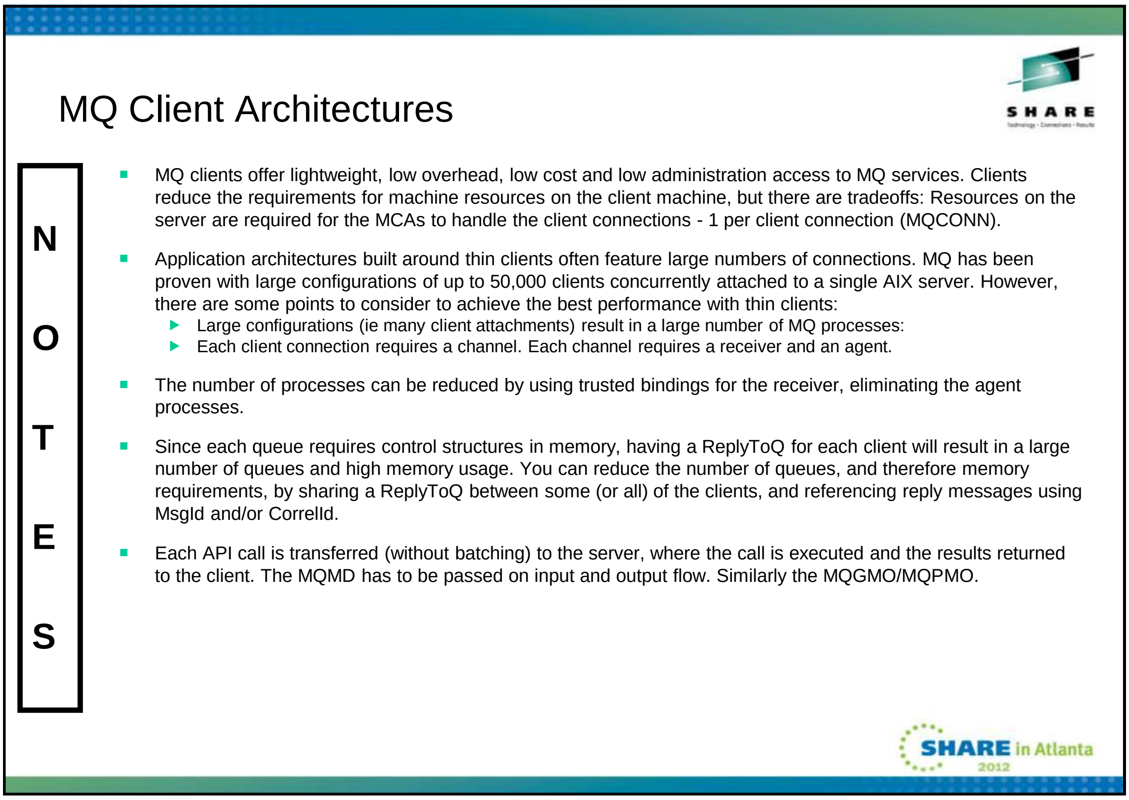
NOTES

- Section on MQ Clients



MQ Client Architectures

- Large systems have been built with clients
  - 50,000 clients per server
- Similar considerations to synchronous architectures
  - Request / response time critical
  - MQ Transport cost can be < 1 milli second
  - 256 byte user message causes 1420 bytes flow with serialised MQI parameters
- Scalability considerations
  - Large number of processes on server
  - Trusted bindings (Channel programs)
  - Overheads of per-client ReplyToQ
    - Share queues, using CorrelId/MsgId to select correct response
  - Recovery processing
    - If the queue manager fails, all clients reconnect at the same time

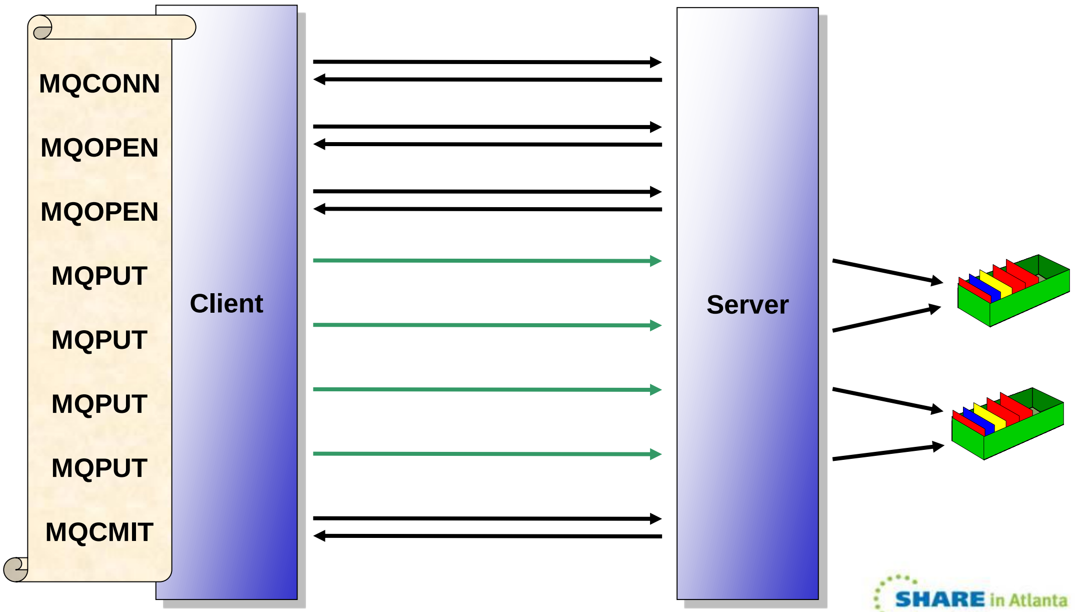


MQ Client Architectures

N O T E S

- MQ clients offer lightweight, low overhead, low cost and low administration access to MQ services. Clients reduce the requirements for machine resources on the client machine, but there are tradeoffs: Resources on the server are required for the MCAs to handle the client connections - 1 per client connection (MQCONN).
- Application architectures built around thin clients often feature large numbers of connections. MQ has been proven with large configurations of up to 50,000 clients concurrently attached to a single AIX server. However, there are some points to consider to achieve the best performance with thin clients:
  - Large configurations (ie many client attachments) result in a large number of MQ processes:
  - Each client connection requires a channel. Each channel requires a receiver and an agent.
- The number of processes can be reduced by using trusted bindings for the receiver, eliminating the agent processes.
- Since each queue requires control structures in memory, having a ReplyToQ for each client will result in a large number of queues and high memory usage. You can reduce the number of queues, and therefore memory requirements, by sharing a ReplyToQ between some (or all) of the clients, and referencing reply messages using MsgId and/or CorrelId.
- Each API call is transferred (without batching) to the server, where the call is executed and the results returned to the client. The MQMD has to be passed on input and output flow. Similarly the MQGMO/MQPMO.

Asynchronous Put Response

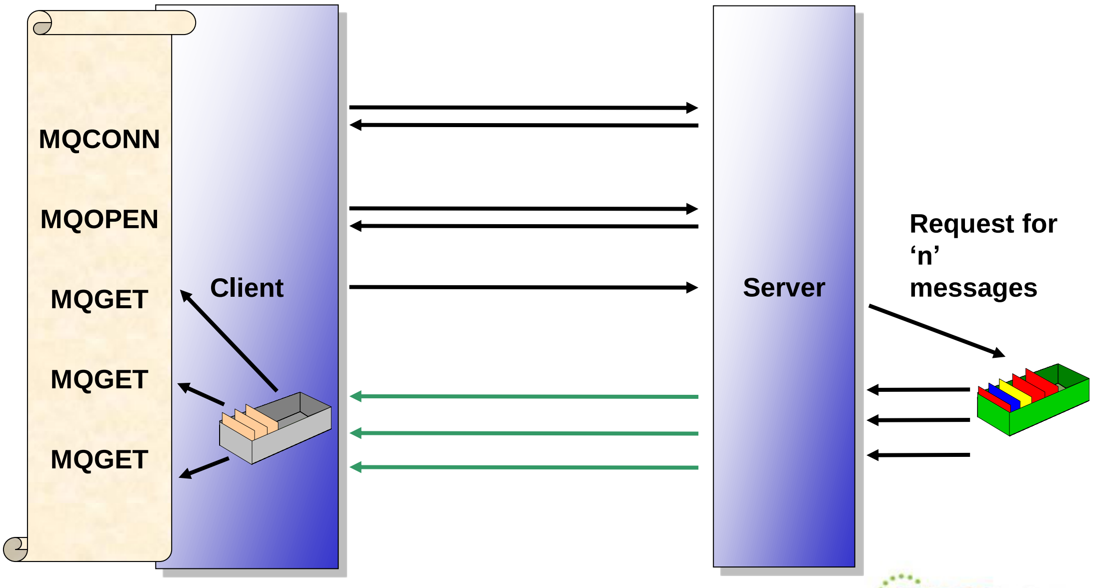


Asynchronous Put Response

NOTES

- Asynchronous Put (also known as 'Fire and Forget') is a recognition of the fact that a large proportion of the cost of an MQPUT from a client is the line turnaround of the network connection. When using Asynchronous Put the application sends the message to the server but does not wait for a response. Instead it returns immediately to the application. The application is then free to issue further MQI calls as required. The largest speed benefit will be seen where the application issues a number of MQPUT calls and where the network is slow.
- Once the application has competed its put sequence it will issue MQCMIT or MQDISC etc which will flush out any MQPUT calls which have not yet completed.
- Because this mechanism is designed to remove the network delay it currently only has a benefit on client applications.

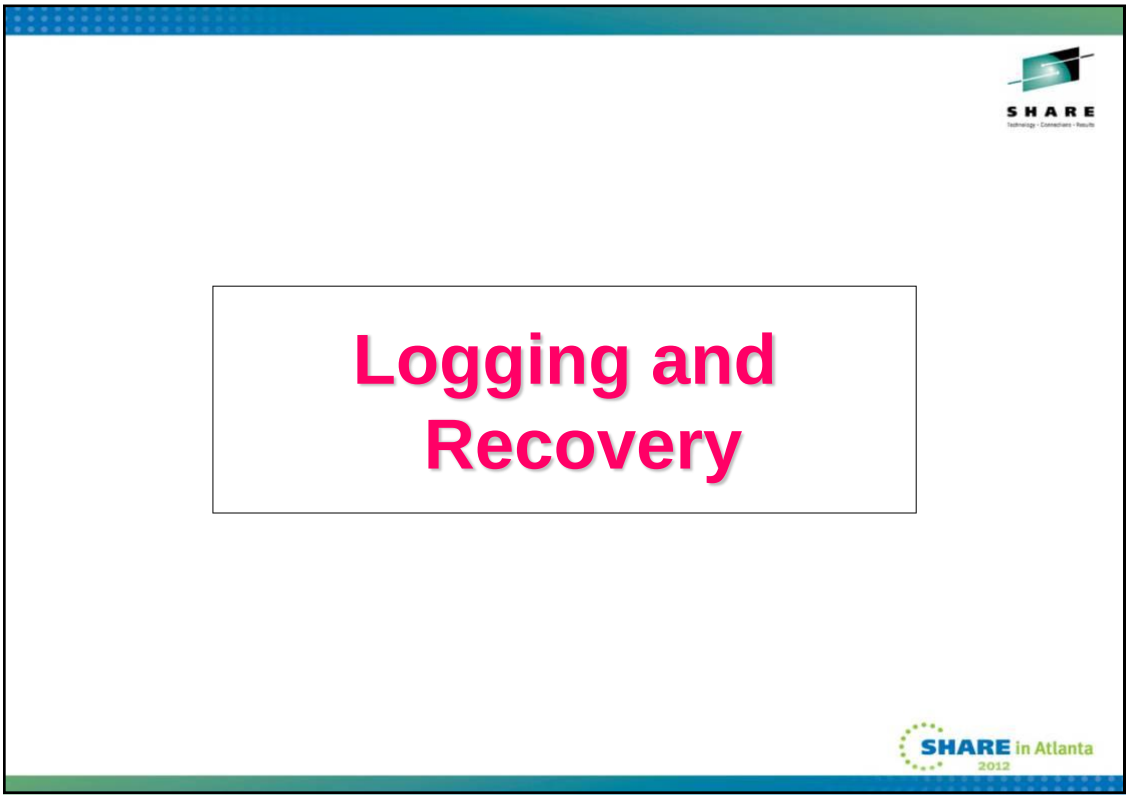
Read-ahead of messages



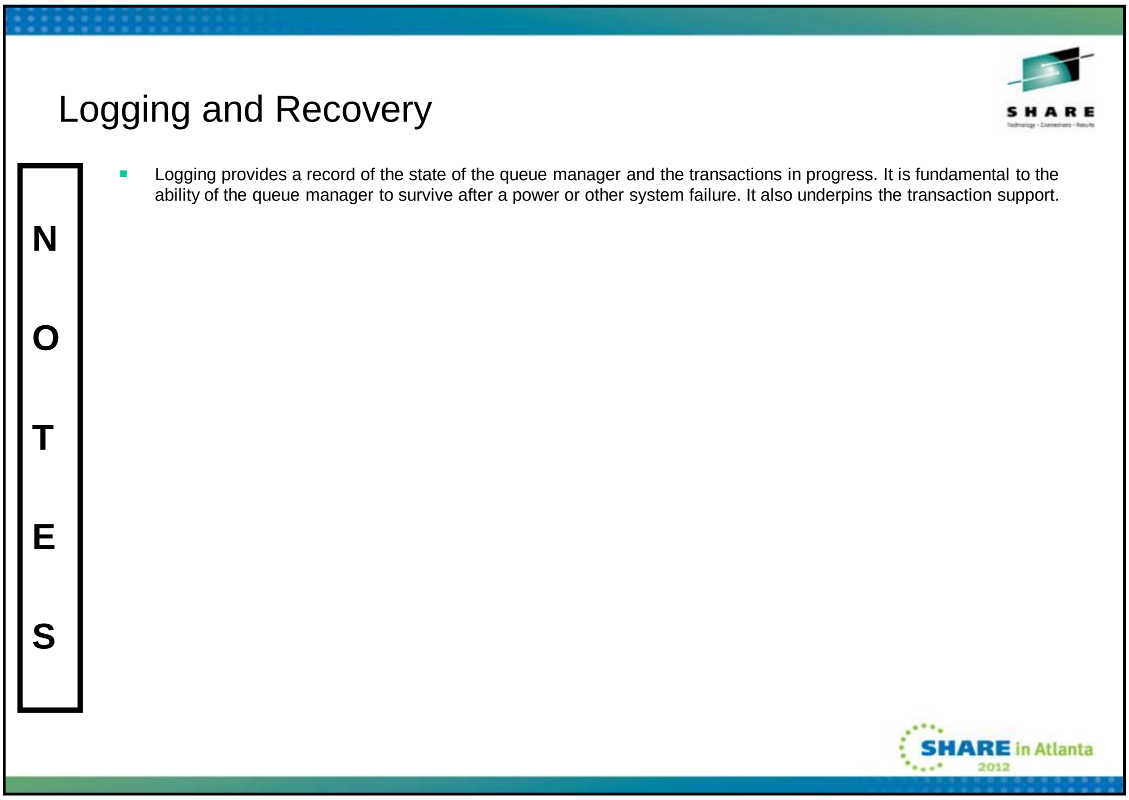
Read-ahead of messages

N O T E S

- Read Ahead (also known as 'Streaming') is a recognition of the fact that a large proportion of the cost of an MQGET from a client is the line turnaround of the network connection. When using Read Ahead the MQ client code makes a request for more than one message from the server. The server will send as many non-persistent messages matching the criteria (such as MsgId) as it can up to the limit set by the client. The largest speed benefit will be seen where there are a number of similar non-persistent messages to be delivered and where the network is slow.
- Read Ahead is useful for applications which want to get large numbers of non-persistent messages, outside of syncpoint where they are not changing the selection criteria on a regular basis. For example, getting responses from a command server or a query such as a list of airline flights.
- If an application requests read ahead but the messages are not suitable, for example, they are all persistent then only one message will be sent to the client at any one time. Read ahead is effectively turned off until a sequence of non-persistent messages are on the queue again.
- The message buffer is purely an 'in memory' queue of messages. If the application ends or the machine crashes these messages will be lost.
- Because this mechanism is designed to remove the network delay it currently only has a benefit on client applications.



Logging and Recovery



Logging and Recovery

NOTES

- Logging provides a record of the state of the queue manager and the transactions in progress. It is fundamental to the ability of the queue manager to survive after a power or other system failure. It also underpins the transaction support.

WebSphere MQ Objects

- Recoverable entities known by the LQM
  - Queue, Process, Queue Manager, Channel etc definitions
  - Scratch Pads
- Objects have security control information
  - Attempts to access them are mediated by the OAM
- Information is stored in Object files
  - May be part of other data in same file
  - Queue File contains messages and attributes
- Topics are objects, but Subscriptions are not
  - Subs stored as messages on queue
- Object Catalog points at object files
  - dspmqfls

SHARE in Atlanta 2012

WebSphere MQ Objects

N

O

T

E

S

- The term "MQ Object" has a special meaning inside the queue manager. It refers to entities which are recoverably updated. As well as queue (message) data, system configuration information is stored in these objects. Before V6, channels were not considered Objects in this sense.
- There is one object catalog file which lists all the objects (QMQMOBJCAT) and each object then has its own file containing its attributes. The QFiles contain the message data AND the queue definitions.
- "Scratchpad" objects are used by the channel programs for faster updates of synchronisation information. These scratchpads are not exposed by any API or command, so they cannot be used by user programs. These objects replace the SYSTEM.CHANNEL.SYNCQ in the majority of cases. The SYNCQ is still used to hold the status of channels (disabled, retry) across system restarts.
- The channel programs still have a separate file (AMQRSYNA) containing what is now an index into the scratchpad objects. This file is not an MQ object, although it is recognised by the rcdmqimg command. It is updated only when channels are created or removed; it does not take part in batch commit processing.

SHARE in Atlanta 2012

What's the point of logging?

- A log record is written for each persistent update
  - The log record describes the update
- Optimisations to minimise serialisation points
- Write-Ahead Logging
  - The log is always more up-to-date than the actual data
- Log is a sequential file
  - Sequential I/O is much quicker than random
  - Single point of writing rather than to individual object files
- Log and actual data are reconciled during strmqm
  - Progress information displayed from MQ V6
- Point of consistency – Checkpoint
  - Log control file: amqhlctl.lfh – in log directory
  - Checkpoint amqalchk.fil – qmgr directory
  - Backup queue managers with MQ V6

What's the point of logging?

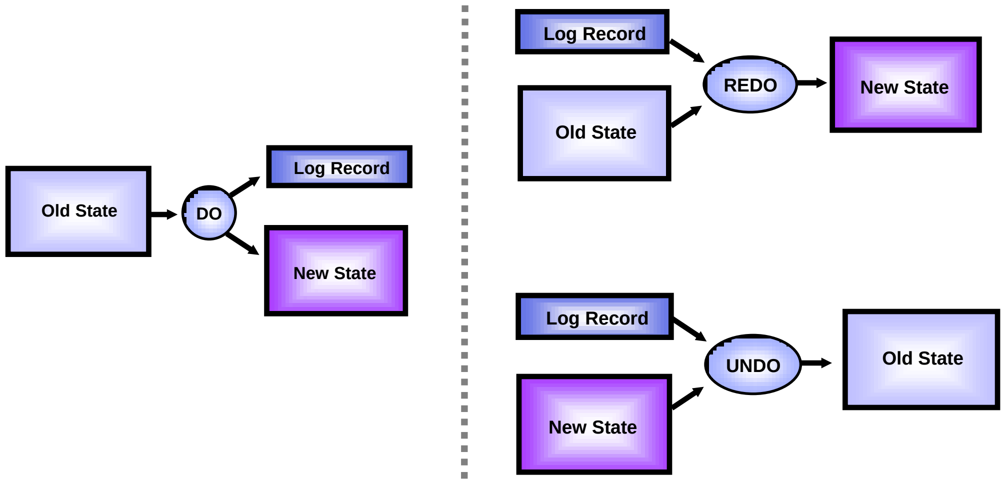
N O T E S

- Each update to persistent data is written to disk at least once. The first copy is a log record. The second copy may be the actual modified data on disk if the effect on the persistent data goes unchanged for a long period of time. This may sound like a lot of overhead but using the log does have advantages:
  - Log records are always written to the end of the log, whereas the updates to the data on disk are more or less random by comparison. The disk head is much more likely to be in the right place when writing to the log, especially if the log has a dedicated disk drive.
  - Special care is taken when writing log records to cope with power failures. This is fairly simple with a sequential file.
  - The log makes it easy to keep track of the operations which make up transactions.
- Writing the data twice does not mean that we wait for the disk twice. In fact, message operations under syncpoint do not result in synchronous I/O until commit or rollback.
- Non-persistent messages, even those that are spilled to disk, do not cause log records to be written.
- The log record describes the update in enough detail for the update to be recreated.
- The log records are written using a protocol called Write-Ahead Logging.
  - The log record describing an operation is guaranteed to arrive on disk before the data being updated.
  - The log is never less up-to-date than the actual data.
  - The contents of the log records can be used to perform the updates on the real data.
- Every now and again the log and data are brought into line. This point of consistency is called a checkpoint. At the end of a checkpoint, the queue files can be brought as up to date as the log at the start of the checkpoint if the queue manager was recovered.
  - During normal running, checkpoints are taken either every 30 minutes provided there are at least 100 log records, but also driven when 10000 log records have been written
- The log and data are reconciled during strmqm. This is called restart recovery. There are messages displayed as the queue manager goes through the phases of reconciliation.

DO, REDO and UNDO

Normal processing

Recovery processing



DO, REDO and UNDO

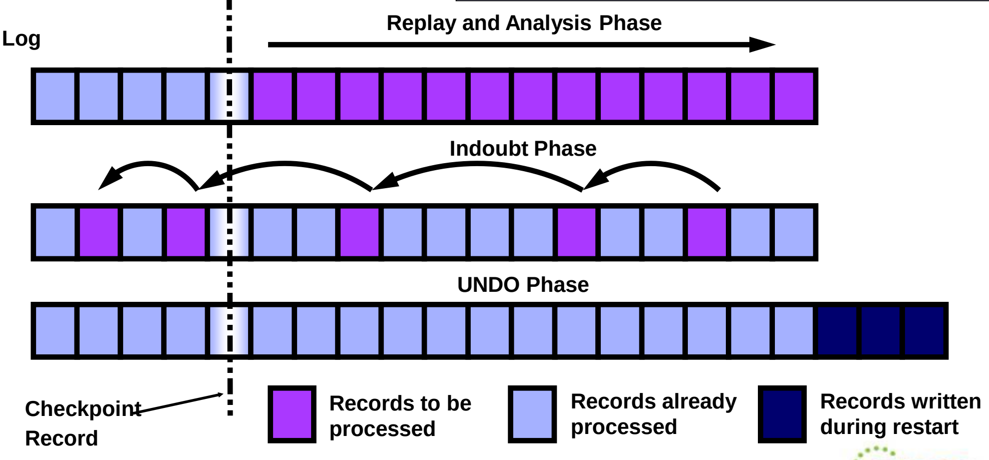
N O T E S

- MQ was designed to use a programming style known as DO-REDO-UNDO.
- All operations on recoverable data are split into three operations.
- DO
  - During normal processing, each operation on recoverable data is performed and an associated log record is generated. The log record contains an encapsulated version of the operation.
- REDO
  - During recovery operations, resource managers may need to reapply changes which were originally made. The contents of the log record and the old copy of the resources affected by the operation can be used to recreate the updated state of the resources from the last checkpoint.
- UNDO
  - After the REDO phase there may be certain operations which need to be undone such as partial transactions. The contents of the log record and the updated copy of the resources affected by the operation can be used to recreate the state of the resources as it was before the operations were performed.
- The DO-REDO-UNDO protocol is commonly used for resource and transaction managers. It relies on the correct information being logged. It also relies on the availability of programs which can perform the operations independently of the original application.
- The important point is that during restart, the log must contain all the information necessary to allow the resource to be recovered without the intervention of any code other than the resource manager. The applications do not have to be restarted.

Phases of Restart Recovery

- "Backup" Queue Managers
  - Only do REDO

```
$ strmqm QMC
WebSphere MQ queue manager 'QMC' starting.
9 log records accessed on queue manager 'QMC' during log replay phase.
Log replay for queue manager 'QMC' complete.
Transaction manager state recovered for queue manager 'QMC'.
WebSphere MQ queue manager 'QMC' started.
```


Log
Replay and Analysis Phase
Indoubt Phase
UNDO Phase
Checkpoint Record
Records to be processed
Records already processed
Records written during restart

Phases of Restart Recovery

N

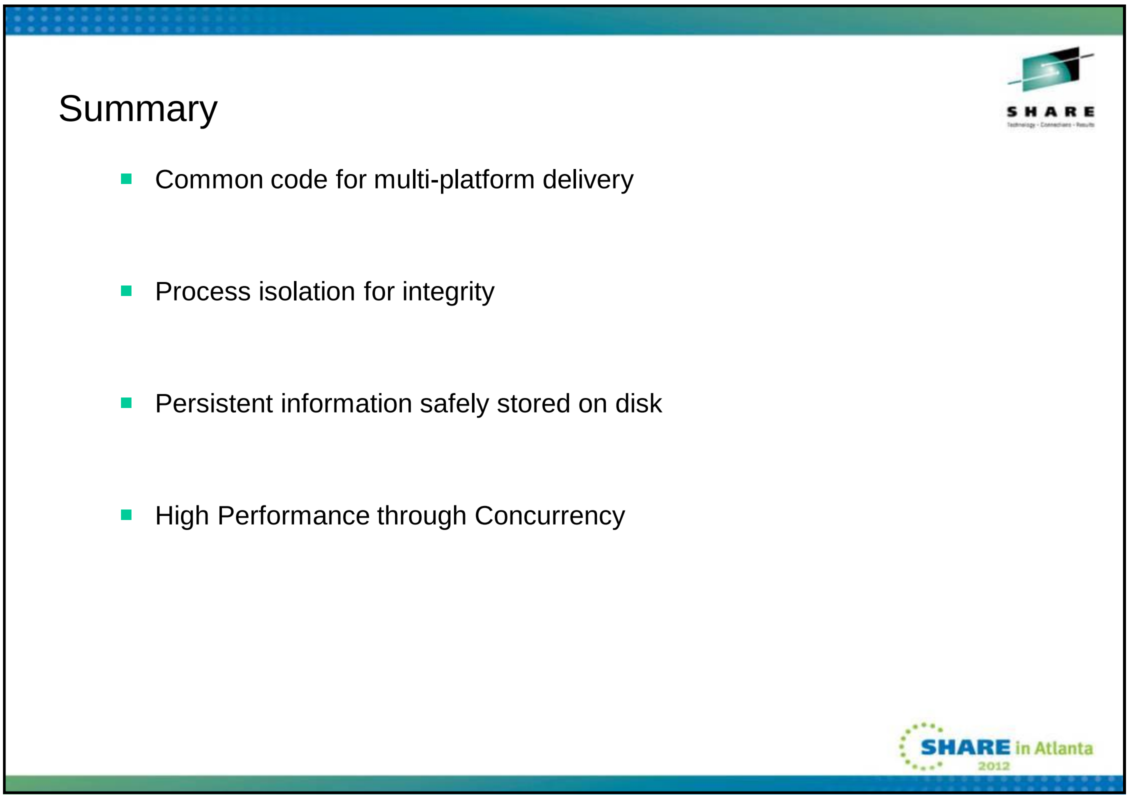
O

T

E

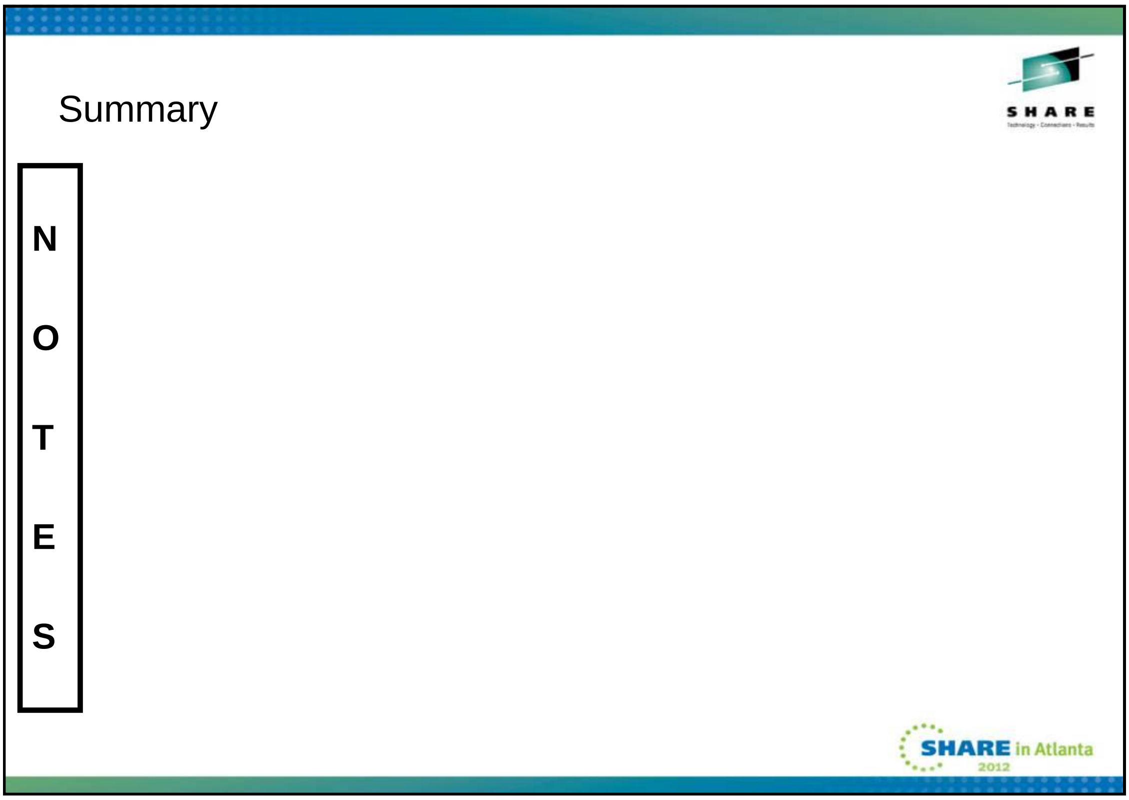
S

- Each time you restart, three phases of recovery are performed to restore the queue manager to a consistent state:
- Replay and Analysis Phase
  - All log records after the last checkpoint are REDONE. This is known as repeated history.
  - The checkpoint is the last known point of consistency between the log and the object files.
  - The transaction table is rebuilt during this phase. Any transactions mentioned in log records are added to a list as candidates for rollback in the final phase of restart. The state of all of the transactions found is also maintained during this phase. When the log records for the end of a transaction are found, the transaction can be removed from the list.
  - If the queue manager stopped cleanly (no in-flight transactions), the only processing required is to replay the most recent checkpoint. There will be no work to do in the next two phases.
- Indoubt Phase
  - We have a list of transactions in-flight at the time that the queue manager ended. For each transaction, we scan backwards through the log from the last record the transaction wrote following the links between records in the same transaction building up a picture of what the transaction was actually doing at the time the queue manager stopped.
- Undo Phase
  - Any transactions in the list which are not prepared are rolled back. This involves writing special undo log records called Compensation Log Records (CLRs). A CLR contains an after image only which corresponds to the before image of the log record it is undoing.
  - Once the CLR has been written, the operation which it describes will be ignored by the indoubt phase if we get into a situation where restart is interrupted and restarted at second time.
- At the end of restart, we may have some prepared transactions. The indoubt phase will have reconstructed the list of operations making up the transaction so we can subsequently commit or rollback the transactions when called by the transaction manager.



Summary

- Common code for multi-platform delivery
- Process isolation for integrity
- Persistent information safely stored on disk
- High Performance through Concurrency



Summary

NOTES