Session 16205
MQ: Beyond the Basics

Neil Johnston
IBM UK
neilj@uk.ibm.com

#SHAREorg

SHARE is an independent volunteer-run information technology association that provides education, professional networking and industry influence.

SHARE in Pittsburgh 2014

Copyright (c) 2014 by SHARE Inc. Except where otherwise noted, this work is licensed under http://creativecommons.org/licenses/by-nc-sa/3.0/

Agenda

- The Async Consume API
- MQ Client/SVRCONN Channels
  - Async Put Response
  - Read-ahead of messages
- Async Consume in Action - The MQ Channel Initiator
- MQ Internals - The Queue Manager

The Async Consume API

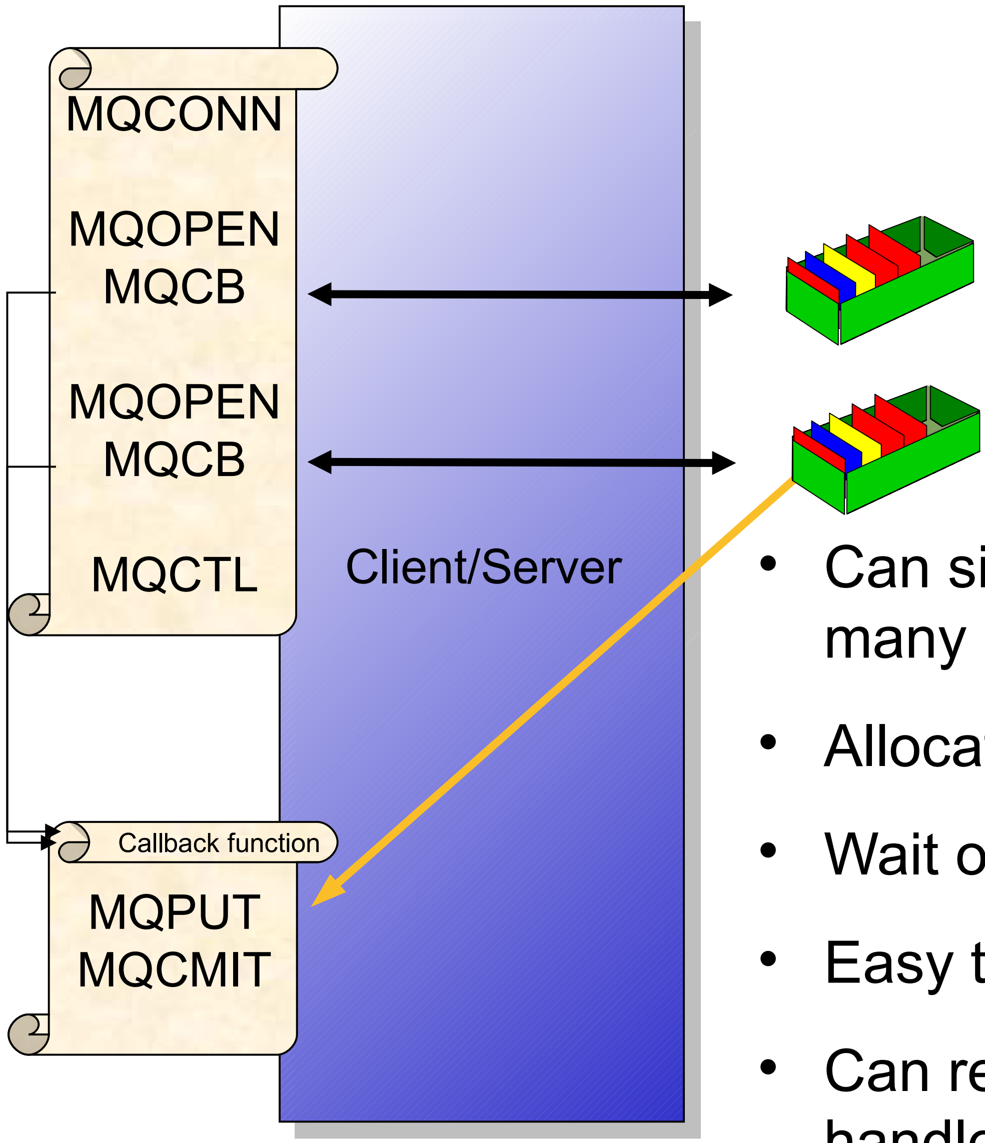
SHARE is an independent volunteer-run information technology association that provides education, professional networking and industry influence.

Copyright (c) 2014 by SHARE Inc. Except where otherwise noted, this work is licensed under http://creativecommons.org/licenses/by-nc-sa/3.0/

The Classic MQ API

- The 'classic' MQ API is pretty easy to grasp
  - You MQOPEN a queue
  - You MQPUT a message to that queue
  - You MQGET a message from the queue
- Everything happens synchronously and (from the application programmer's perspective) on a single thread

Asynchronous Consumption of Messages



- Can simplify programming in many cases
- Allocates message buffers
- Wait on multiple queues
- Easy to cancel
- Can register an Event handler

Asynchronous Consumption of Messages - Notes

N O T E S

- Asynchronous consumer allows the application to register an interest in messages of a certain type and identify a callback routine which should be invoked when a message arrives. This has the following advantages to the traditional MQGET application.

- **Simplifies programming**
  The application can continue to do whatever it was doing without needing to tie up a thread sitting in an MQGET call.

- **Allocates message buffers**
  The application does not need to 'guess' the size of the next message and provide a buffer just large enough. The system will pass the application a message already in a buffer.

- **Wait on multiple queues**
  The application can register an interest in any number of queues. This is very much simpler than using MQGET where one generally ended up polling round the queues.

- **Easy to cancel**
  The application can use either MQCTL or MQCB to stop consuming from a queue at any time. This is awkward to achieve when an application is using MQGET

- **Can register an Event handler**
  The application is notified of events such as Queue Manager quiescing or Communications failure.

Define your call-back functions

- MQOPEN a queue

  or MQSUB using MQSO_MANAGED
- MQCB connects returned hObj to call-back function
- Operations (MQOP_*)
- CallbackType:
  - Message Consumer
  - Event Handler

```
MQOPEN( hConn,
        &ObjDesc,
        OpenOpts,
        &hObj,
        &CompCode,
        &Reason);

MQCB  ( hConn,
        MQOP_REGISTER,
        &cbd,
        hObj,
        &md,
        &gmo,
        &CompCode,
        &Reason);
```

```
MQCBD   CBDesc = {MQCBD_DEFAULT};

cbd.CallbackFunction = MessageConsumer;
cbd.CallbackType     = MQCBT_MESSAGE_CONSUMER;
cbd.MaxMsgLength     = MQCBD_FULL_MSG_LENGTH;
cbd.Options          = MQCBDO_FAIL_IF_QUIESCING;
```

Define your call-back functions - Notes

N O T E S

The MQCB verb ties a function (described in the Call-Back Descriptor (MQCBD)) to an object handle. This object handle is any object handle that you might have used for an MQGET. That is, one that was returned from an MQOPEN call or an MQSUB call (using MQSO_MANAGED for example).

The MQCB verb has a number of Operations. We see an example of MQOP_REGISTER on this foil which tells the queue manager that this function (described in the MQCBD) should be called when messages arrive for the specified object handle. You can do the inverse of the operation with MQOP_DEREGISTER to remove a previously registered call-back function. Also we have MQOP_SUSPEND and MQOP_RESUME which we will cover a little later.

There are actually two types of call-back function you can define. A message consumer which is tied to an object handle, and receives messages or error notifications about the specific queue such as MQRC_GET_INHIBITED; and an event handler which is tied to the connection handle and receives error notifications about the connection such as MQRC_Q_MGR_QUIESCING.

One of the benefits of using asynchronous consume is that the queue manager manages the buffer your message is in. This means that your application doesn’t have to worry about truncated messages and acquiring bigger buffers in the case of MQRC_TRUNCATED_MSG_FAILED. The default is to use MQCBD_FULL_MSG_LENGTH, but if you wish to restrict the size of messages presented to your call-back function, you can put a length in the MaxMsgLength field of the MQCBD.

MQGMO differences

	MQGET	Asynchronous Consume
Combining MQGMO_BROWSE_FIRST + MQGMO_BROWSE_NEXT	MQRC_OPTIONS_ERROR	Delivers first message then automatically switches to BROWSE_NEXT
MQGMO_WAIT with MQGMO.WaitInterval = 0	MQGET will return immediately with MQRC_NO_MSGS_AVAILABLE if there are no messages	Only called with MQRC_NO_MSGS_AVAILABLE if just started or has had a message since last 2033
MQGMO_NO_WAIT		The message consumer will never be called with MQRC_NO_MSGS_AVAILABLE
MQGMO_WAIT with MQGMO.WaitInterval = MQWI_UNLIMITED	MQGET will never return with MQRC_NO_MSGS_AVAILABLE	
MQGMO_SET_SIGNAL	Allowed	Not allowed

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval

MQGMO differences - Notes

N O T E S

The MQCB call provides an MQGMO structure which you will be familiar with from using MQGET. The MQGMO is used for Asynchronous Consume as well as for MQGET. It is after all the way to describe how to consume your message whether synchronously or asynchronously. Some of the attributes/options in the MQGMO operate slightly differently when used for Asynchronous Consume and this foil details those differences.

MQGMO_WAIT with MQGMO.WaitInterval = 0 operates just like MQGMO_NO_WAIT when one uses on an MQGET, but in the case of asynchronous consumers we wish to avoid the consumer from polling in a busy loop in this case, so it operates more like a backstop marker to show when the end of a batch of messages has been reached.

Note that MQGMO_NO_WAIT, and MQGMO_WAIT with a WaitInterval of MQWI_UNLIMITED are quite different when passed to MQGET but with the MQCB call their behaviour is the same. The consumer will only be passed messages and events, it will never be passed the reason code indicating no messages. Effectively MQGMO_NO_WAIT will be treated as an indefinite wait. This is to prevent the consumer from endlessly being called with the no messages reason code.

Control your message consumption

MQCTL controls whether message consumption is currently active

Operations

- MQOP_START
- MQOP_START_WAIT
- MQOP_STOP
- MQOP_SUSPEND (MQCB too)
- MQOP_RESUME (MQCB too)

```
MQCTLO ctlo = {MQCTLO_DEFAULT};
ctlo.Options = MQCTLO_FAIL_IF_QUIESCIN
MQCTL( hConn,
       MQOP_START,
       &ctlo,
       &CompCode,
       &Reason);

...

MQCTL( hConn,
       MQOP_STOP,
       &ctlo,
       &CompCode,
       &Reason);
```

Give up control of the hConn for call-backs to use

Change current call-backs operating

- Either MQOP_SUSPEND the connection
- Or from within a currently called call-back

Control your message consumption - Notes

N O T E S

Once you have defined all your message consumers using MQCB calls – you may have several – then use the MQCTL call to tell the queue manager you are ready to start consuming messages. Once you have called MQCTL for a specific hConn you give up control of that hConn and it is passed to the call-backs to use. If you try to use it for any other MQ call you will receive MQRC_HCONN_ASYNC_ACTIVE. The exception to this is another call to MQCTL to either MQOP_STOP or MQOP_SUSPEND message consumption.

Use MQOP_STOP when your application is finished consuming messages. Use MQOP_SUSPEND (and then subsequently MQOP_RESUME) when you wish to briefly pause message consumption while you, for example, MQOP_REGISTER another MQCB call or MQOP_DEREGISTER an existing one. While the whole hConn is suspended none of the call-backs will be delivered messages. You may wish to only suspend a particular object handle, in which case you can use MQOP_SUSPEND on an MQCB call.

Calls to change the call-backs currently operating can also be made inside another call-back removing the need to suspend the connection in order to make changes such as this.

The Call-Back Function

Fixed prototype

Call-back context (MQCBC)

CallType – why fn was called

CompCode + Reason detail any error

State – Consumer state

- Saves coding all possible Reasons

```
struct tagMQCBC
{
  MQCHAR4   StrucId;
  MQLONG    Version;
  MQLONG    CallType;
  MQHOBJ    Hobj;
  MQPTR     CallbackArea;
  MQPTR     ConnectionArea;
  MQLONG    CompCode;
  MQLONG    Reason;
  MQLONG    State;
  MQLONG    DataLength;
  MQLONG    BufferLength;
  MQLONG    Flags;
};
```

```
MQLONG MessageConsumer( MQHCONN   hConn,
                        MQMD     * pMsgDesc,
                        MQGMO    * pGetMsgOpts,
                        MQBYTE   * Buffer,
                        MQCBC    * pContext)
```

The Call-Back function - Notes

N O T E S

Your call-back function can have any name you want, but it must conform to the prototype shown. When called with a message, you are passed the Message Descriptor (MQMD), the message buffer and the Get-Message Options structure (MQGMO) which contains a number of output fields about the message you have been given. You will know you have been given a message because the CallType field in the Call Back Context (MQCBC) will be set to either MQCBCT_MSG_REMOVED or MQCBCT_MSG_NOT_REMOVED (which one depends on the get options you used, i.e. browse or a few specific errors).

Your message consumer can also be called with CallType set to MQCBCT_EVENT_CALL (this is also the only way an Event handler can be called). The message consumer will be given events that are pertinent to the queue it is consuming from, for example, MQRC_GET_INHIBITED whereas the event handler gets connection wide events. If there is an error to report, in the case of an MQCBCT_EVENT_CALL or in some cases for MQCBCT_MSG_NOT_REMOVED, it will be reported in the CompCode and Reason fields of the MQCBC. When a Reason code is delivered to a call-back, the State field of the MQCBC details what has happened to the consumer as a result of the specific Reason. It can be used to simplify application programming by informing the application what has happened to the consumer function rather than the application having to know for each reason code what the behaviour will be. States such as MQCS_SUSPENDED_USER_ACTION which detail that some user intervention will be needed before message consumption can continue.

The MQ Client: Async Put Response

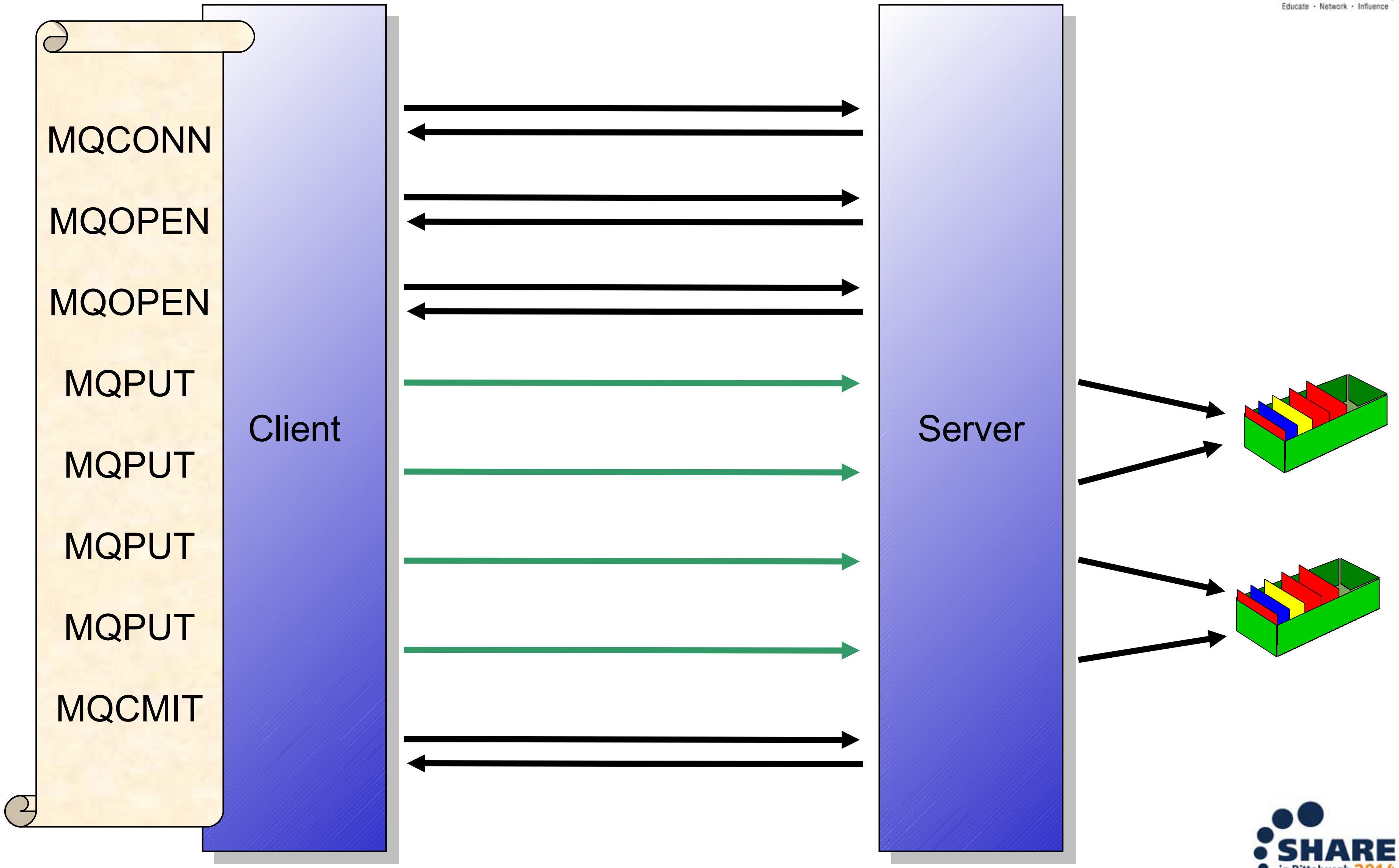
#SHAREorg

SHARE is an independent volunteer-run information technology association that provides education, professional networking and industry influence.

Copyright (c) 2014 by SHARE Inc. Except where otherwise noted, this work is licensed under http://creativecommons.org/licenses/by-nc-sa/3.0/

SHARE in Pittsburgh 2014

Asynchronous Put Response



Asynchronous Put Response - Notes

N
O
T
E
S

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.

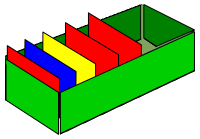
Once the application has competed it's put sequence it will issue MQCMIT or MQDISC etc which will flush out any MQPUT calls which have not yet completed.

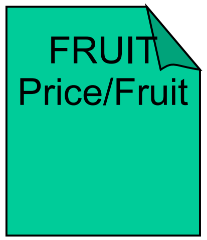
Because this mechanism is designed to remove the network delay it currently only has a benefit on client applications. However, it is recommended that applications that could benefit from it, use it for local bindings as well since in the future there is the possibility that the server could perform some optimisations when this option is used.

Put Response Options

MQPMO_ASYNC_RESPONSE
MQPMO_SYNC_RESPONSE

MQPMO_RESPONSE_AS_Q_DEF
MQPMO_RESPONSE_AS_TOPIC_DEF





DEFPRESP
- SYNC
- ASYNC

Returned (output) Message Descriptor (MQMD)

ASYNC

- `ApplIdentityData`
- `PutApplType`
- `PutApplName`
- `ApplOriginData`
- `MsgId`
- `CorrelId`

SYNC

- Full MQMD is completed

Put Response Options - Notes

N O T E S

You can make use of asynchronous responses on MQPUT by means of an application change or an administration change. Without any change your application will be effectively using MQPMO_RESPONSE_AS_Q_DEF which will be resolved to whatever value is defined on the queue definition. You can choose to deliberately use asynchronous responses by using MQPMO_ASYNC_RESPONSE, and you can choose to always have synchronous responses by using MQPMO_SYNC_RESPONSE.

The queue and topic objects have an attribute DEFPRESP which is where the MQPMO_RESPONSE_AS_Q_DEF/TOPIC_DEF are resolved from. This has values ASYNC and SYNC.

Apart from not being informed of any failures to put the message on the queue, the other change when using ASYNC is that only some fields in the Message Descriptor (MQMD) are actually filled in when it is returned as an output structure to the putting application. The remaining fields are undefined when using ASYNC responses.

Last Error Retrieval

Application will not find out about a failure to put to the queue

- Ignore the situation
- Issue an MQCMIT
- Issue the new verb MQSTAT
  - Flush in-flight messages
  - Return success/failure count

```
struct tagMQSTS
{
  MQCHAR4    StrucId;
  MQLONG     Version;
  MQLONG     CompCode;
  MQLONG     Reason;
  MQLONG     PutSuccessCount;
  MQLONG     PutWarningCount;
  MQLONG     PutFailureCount;
  MQLONG     ObjectType;
  MQCHAR48   ObjectName;
  MQCHAR48   ObjectQMgrName;
  MQCHAR48   ResolvedObjectName;
  MQCHAR48   ResolvedQMgrName;
};
```

```
MQSTS sts = {MQSTS_DEFAULT};
MQSTAT(hConn,
       MQSTAT_TYPE_ASYNC_ERROR,
       &sts,
       &CompCode,
       &Reason);
```

Last Error Retrieval - Notes

N
O
T
E
S

Because the client does not wait for a response from the MQPUT call it will not be told at MQPUT time whether there was a problem putting the message. For example, the queue could be full. There are three things the application can do :

Ignore the situation
In many cases of say a non-persistent message the application does not care too much whether the message makes it or not. If no response it received then another request can be issued within a few seconds or whatever.

Issue an MQCMIT
If the messages put are persistent messages in syncpoint then if any of them fail they will cause a subsequent MQCMIT call to also fail.

Issue the new verb MQSTAT
This new verb allows the application at any time to flush all messages to the server and respond with how many of the messages succeeded or failed. The application can issue this verb as often as required

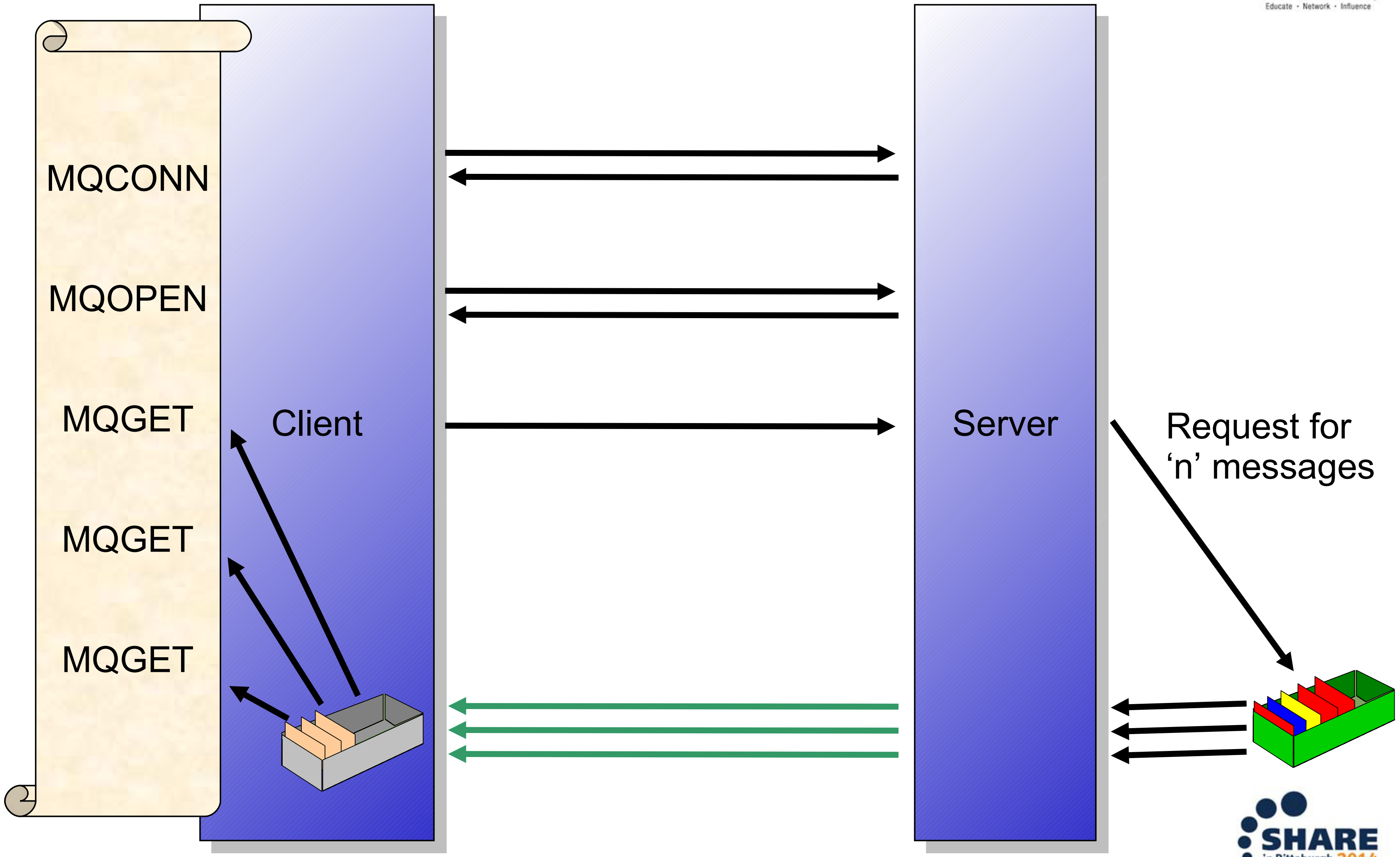
MQ Client/SVRCONN Read-Ahead of Messages

#SHAREorg

SHARE is an independent volunteer-run information technology association that provides education, professional networking and industry influence.

Copyright (c) 2014 by SHARE Inc. Except where otherwise noted, this work is licensed under http://creativecommons.org/licenses/by-nc-sa/3.0/

Read-Ahead of Messages



Read-Ahead of Messages - Notes

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.

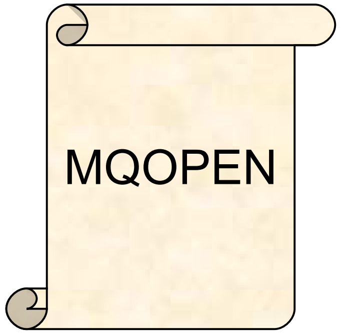
If an application requests read ahead but the messages are not suitable, for example, they are all persistent then only one message will be sent to the client at any one time. Read ahead is effectively turned off until a sequence of non-persistent messages are on the queue again.

The message buffer is purely an 'in memory' queue of messages. If the application ends or the machine crashes these messages will be lost.

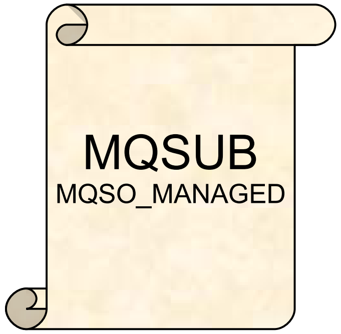
Because this mechanism is designed to remove the network delay it currently only has a benefit on client applications. However, it is recommended that applications that might benefit from it, use it for local bindings as well since in the future there is the possibility that the server could perform some optimisations when this option is used.

Read-Ahead Options

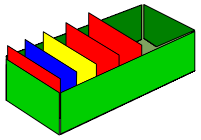
MQOO_READ_AHEAD_AS_Q_DEF
MQOO_NO_READ_AHEAD
MQOO_READ_AHEAD
MQSO_READ_AHEAD_AS_Q_DEF



When using managed queues
- MQSO_NO_READ_AHEAD
- MQSO_READ_AHEAD



DEFREADA
- NO
- YES
- DISABLED



Read-ahead Options - Notes

N O T E S

You can make use of read-ahead on MQGET by means of an application change or an administration change. Without any change your application will be effectively using MQOO_READ_AHEAD_AS_Q_DEF on MQOPEN which will be resolved to whatever value is defined on the queue definition. You can choose to deliberately use read-ahead by using MQOO_READ_AHEAD on your MQOPEN, and you can choose to turn off read-ahead by using MQOO_NO_READ_AHEAD.

If you are using a managed destination on MQSUB, by default your application will be effectively using MQSO_READ_AHEAD_AS_Q_DEF and taking its value from the model queue that is used to base managed destinations on. Non-durable subscriptions using the default provided model, SYSTEM.NDURABLE.MODEL.QUEUE, will find that read-ahead is turned on. You can choose to deliberately use read-ahead by using MQSO_READ_AHEAD on your MQSUB, and you can choose to turn off read-ahead by using MQSO_NO_READ_AHEAD on your MQSUB.

Queue objects have an attribute DEFREADA which is where the MQOO/SO_READ_AHEAD_AS_Q_DEF are resolved from. This has values YES and NO for this purpose and additionally a value DISABLED, which over-rides anything specified by the application and turns off any request for read-ahead on this queue.

Application Suitability for Read-Ahead

Suitable for:

- Non-persistent, non-transactional consumption of messages intended for this client only
- Non-durable subscriber
- Response messages to a query

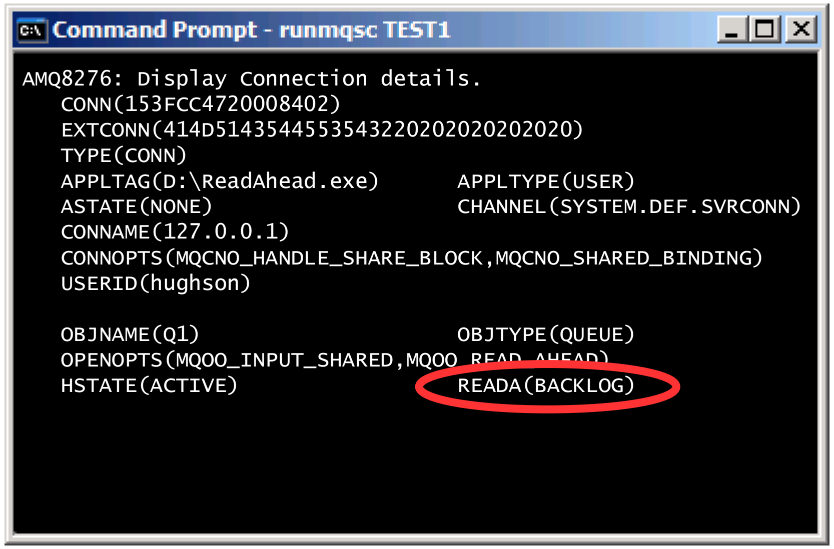
Not suitable for:

- Persistent, transactional messages
- Queues with many clients getting messages
- Applications that continually change message selection criteria

Application Suitability for Read-Ahead

Changing message selection criteria can leave unconsumed messages in the read-ahead buffer

- Highlighted by DISPLAY CONN TYPE(HANDLE) with READA(BACKLOG) if the number of these gets so high as to affect the efficiency of read-ahead



```
AMQ8276: Display Connection details.
   CONN(153FCC4720008402)
   EXTCONN(414D514354455354312020202020202020)
   TYPE(CONN)
   APPLTAG(D:\ReadAhead.exe)          APPLTYPE(USER)
   ASTATE(NONE)                       CHANNEL(SYSTEM.DEF.SVRCONN)
   CONNAME(127.0.0.1)
   CONNOPTS(MQCNO_HANDLE_SHARE_BLOCK,MQCNO_SHARED_BINDING)
   USERID(hughson)

   OBJNAME(Q1)                        OBJTYPE(QUEUE)
   OPENOPTS(MQOO_INPUT_SHARED,MQOO_READ_AHEAD)
   HSTATE(ACTIVE)                     READA(BACKLOG)
```

Application Suitability for Read-Ahead

Use of some options implicitly turns off read-ahead:

- Persistent messages – read-ahead turned off for that message
- Certain MQGMO options – read-ahead turned off for whole use of that object handle (see next page)

MQGMO options with Read-ahead

	MQGET MQMD values	MQGMO fields	MQGET MQGMO options

Permitted when read-ahead is enabled and can be altered between MQGET calls	`MsgId`
	`CorrelId`
			`MQGMO_WAIT`
			`MQGMO_NO_WAIT`
			`MQGMO_FAIL_IF_QUIESCING`
			`MQGMO_BROWSE_FIRST`
			`MQGMO_BROWSE_NEXT`
			`MQGMO_BROWSE_MESSAGE_UNDER_CURSOR`

Permitted when read ahead is enabled but cannot be altered between MQGET calls	`Encoding`
	`CodedCharSet ID`
	`Version`
		`MsgHandle`	`MQGMO_SYNCPOINT_IF_PERSISTENT`
			`MQGMO_NO_SYNCPOINT`
			`MQGMO_ACCEPT_TRUNCATED_MSG`
			`MQGMO_CONVERT`
			`MQGMO_LOGICAL_ORDER`
			`MQGMO_COMPLETE_MSG`
			`MQGMO_ALL_MSGS_AVAI`[illegible]
			`MQGMO_ALL_SEGMENT`[illegible]`LABLE`
			`MQGMO_MARK_BROW`[illegible]
			`MQGMO_UNMARK_`[illegible]`E_*`
			`MQGMO_UNMARKED_BROWSE_MSG`
			`MQGMO_PROPERTIES_*`
			`MQGMO_NO_PROPERTIES`
			MQRC_OPTIONS_CHANGED

MQGET Options that are not permitted when read ahead is enabled			`MQGMO_SET_SIGNAL`
			`MQGMO_SYNCPOINT`
			`MQGMO_MARK_SKIP_BACK`[illegible]
			`MQGMO_MSG_UNDER_CU`[illegible]
			`MQGMO_LOCK`
			`MQGMO_UNLOCK`
			MQRC_OPTIONS_ERROR

MQGMO options with Read-ahead - Notes

N O T E S

As noted on the previous page, some values you can specify on MQGET will cause read-ahead to be turned off. The last row of the table indicate which these are. If they are specified on the first MQGET with read-ahead on, read-ahead will be turned off. If they are specified for the first time on a subsequent MQGET then that MQGET call will fail with MQRC_OPTIONS_ERROR.

Some values cannot be changed if you are using read-ahead. These are indicated in the middle row of this table and if changed in a subsequent MQGET then that MQGET call will fail with MQRC_OPTIONS_CHANGED.

The client applications needs to be aware that if the MsgId and CorrelId values are altered between MQGET calls, messages with the previous values may have already been sent to the client and will remain in the client read ahead buffer until consumed (or automatically purged).

Browse and destructive get cannot be combined with read-ahead. You can use either, but not both. You can MQOPEN a queue for both browse and get, but the options you use on the first MQGET call will determine which is being used with read-ahead and any subsequent change will cause MQRC_OPTIONS_CHANGED. You cannot therefore use MQGMO_MSG_UNDER_CURSOR which is using the combination of both browse and get.

Async Consume in Action
The MQ Channel Initiator

#SHAREorg

SHARE is an independent volunteer-run information technology association that provides education, professional networking and industry influence.

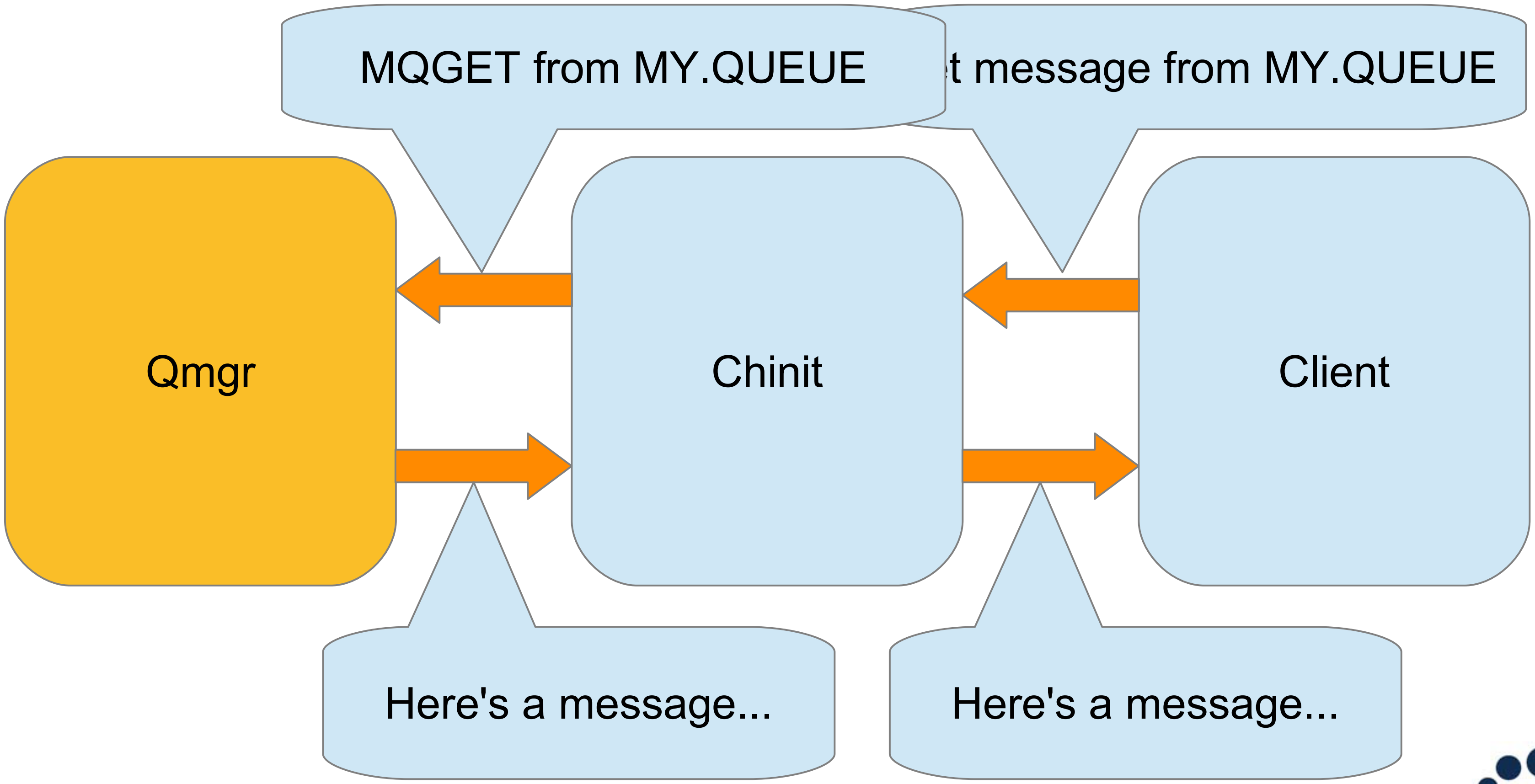
Copyright (c) 2014 by SHARE Inc. Except where otherwise noted, this work is licensed under http://creativecommons.org/licenses/by-nc-sa/3.0/

SHARE in Pittsburgh 2014

Async Consume In Action

- Here's a real-world example of the power of async consume
- The channel initiator is a (very sophisticated) MQ application
- In 'classic' SHARECNV(0) operation, an MQ SVRCONN channel will just use the traditional MQ API verbs to manipulate messages:
  - MQGET
  - MQPUT
- In principle the SVRCONN MCA 'mirrors' the MQ API requests of the client

SVRCONN Channels with Traditional MQGET Verbs – Getting a Message



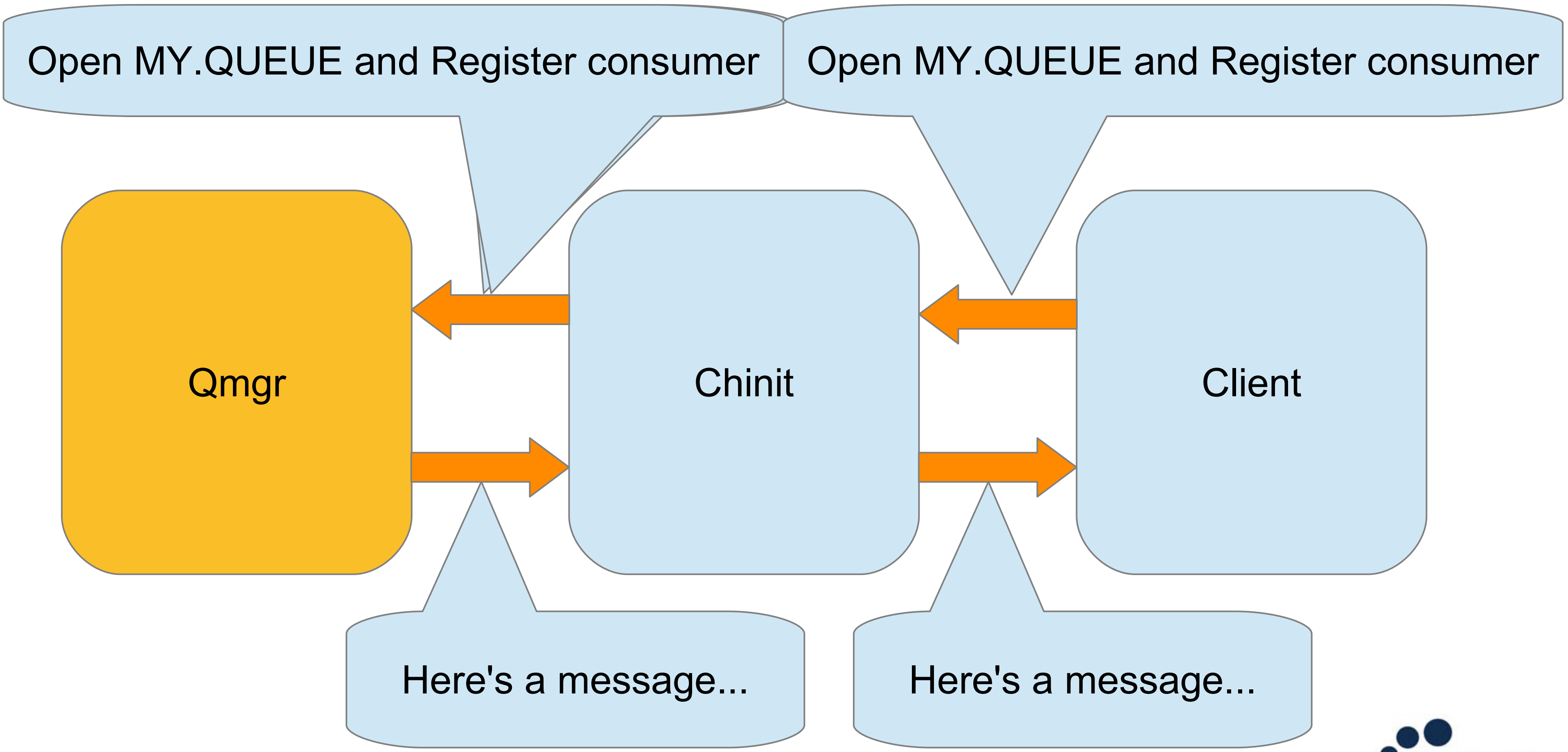
Async Consume In Action

- Let's modify this model to use async consume for getting and putting messages
- Rather than using MQGET, we're going to get messages as quickly as possible by consuming them asynchronously from a queue
- ⚠ Requires significant design and implementation changes
  - Not just replacing verbs

Async Consume – Getting Messages With Read-Ahead

- Simple example consuming from a single queue
- We still MQOPEN the queue as usual
- Rather than getting from the queue we use:
- MQCB(*register*) to establish our MQGET options (MQMD and MQGMO)
- MQCTL(*start*) will begin sending messages

SVRCONN Channels with Async Consume – Getting Messages With Read-Ahead


Open MY.QUEUE and Register consumer
Open MY.QUEUE and Register consumer
Qmgr
Chinit
Client
Here's a message...
Here's a message...

Other Advantages of the Async Consume Model for SVRCONNs

- Duplexed channels for additional control
  - Administrative STOP-QUIESCE
  - Heartbeats
  - Performance

Considerations – Read Ahead

- Do we consume messages “forever” or can the client throttle us?
- How do we stop consuming messages?
- What happens to messages we've sent but may not have been received/processed?

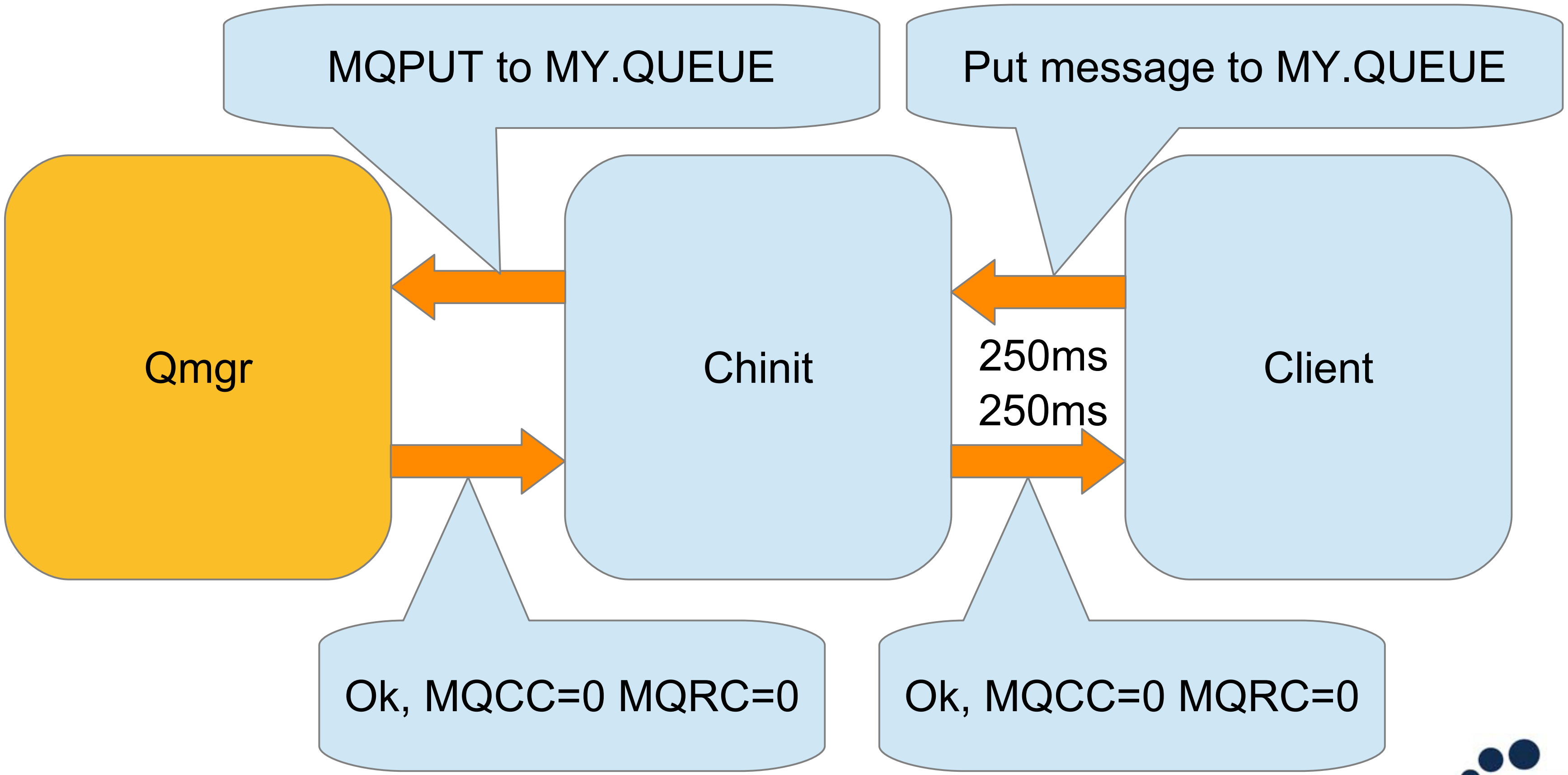
Chinit Internals – Read Ahead

- MQ Client implements a “proxy queue” which acts like a mini queue
  - Holds messages that have been streamed to the client via async consume but have not yet been delivered to the application
- Client provides feedback to the server on the size of this proxy queue, expressed as a size
  - “I can now accept 1.5MB more data”...
  - “I can now accept 500KB more data”...
- Chinit tracks how much is sent compared to most recent response and keeps sending until proxy queue is assumed full or new size information sent.

Async Put for the MQ Client

- Offers similar performance improvements to Async Consume but for messages sent **to** the queue manager
- Under the control of the application
- Application needs to know it's happening so it can check for and respond to the asynchronous failures

SVRCONN Channels with Traditional MQGET Verbs – Putting a Message



MQ Clients

MQ V8 Client Attachment Feature is now

FREE!*

*Also zero charge on WMQ 7.1 with PTF for APAR PI13429

MQ Internals
The Queue Manager

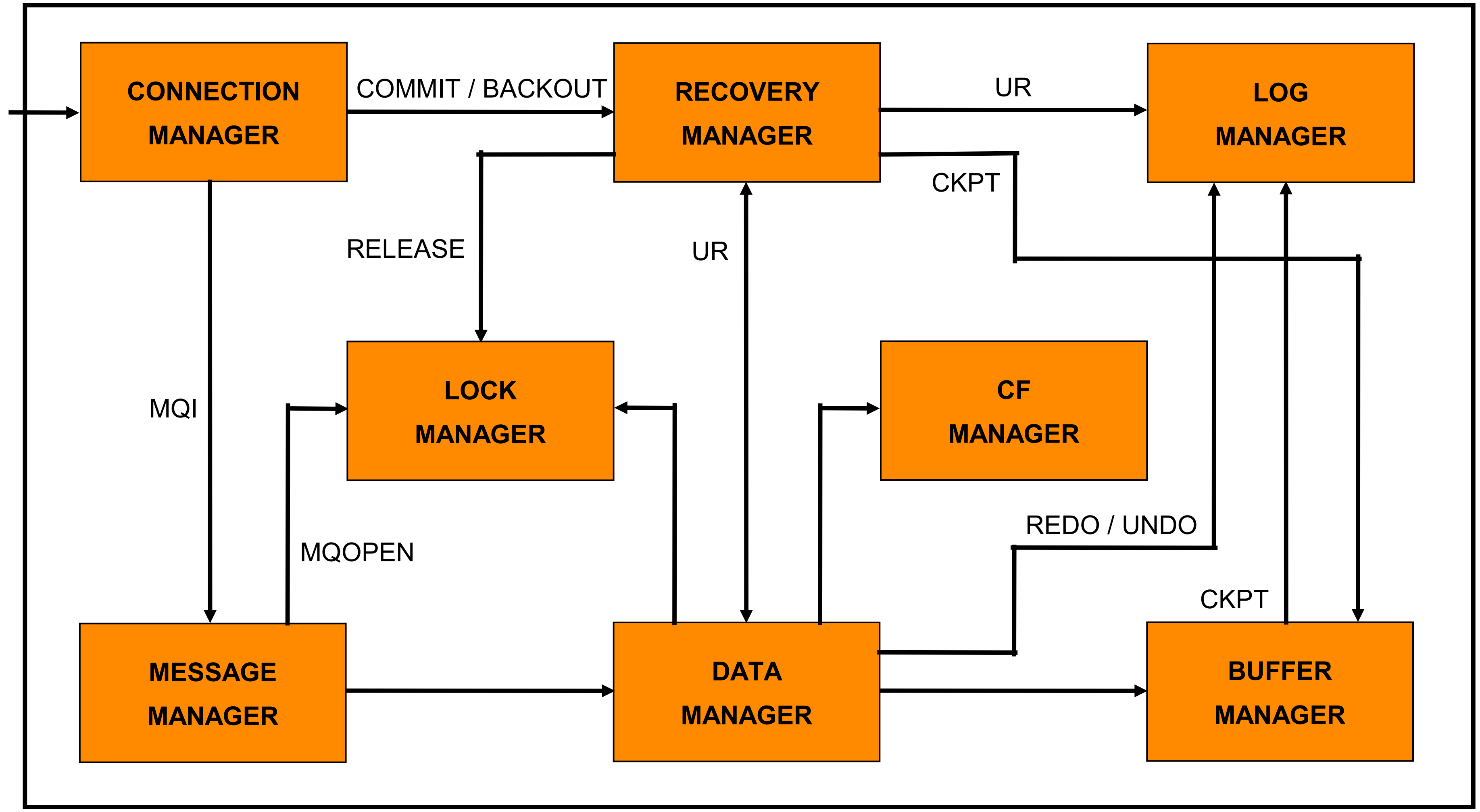
#SHAREorg

SHARE is an independent volunteer-run information technology association that provides education, professional networking and industry influence.

Copyright (c) 2014 by SHARE Inc. Except where otherwise noted, this work is licensed under http://creativecommons.org/licenses/by-nc-sa/3.0/

SHARE in Pittsburgh 2014

MQ Internals – QM Resource Managers



Building blocks – resource managers

NOTES

The queue manager is built from many resource managers (RMs)

A resource manager is a code package that encapsulates operations relating to a logically consistent set of operations on a given resource (e.g. the log). If a party wishes to modify or query a resource, it must access it through the appropriate resource manager.

Resource managers interact with each other to provide the MQI verb set.

The queue manager operation can be by understood by decomposing it into the interactions between these resource managers. The most important are:

Connection Manager

Processes the application thread as it enters the queue manager.

Message Manager

Handles operations relating to messages, objects and triggering.

Data Manager

Processes objects and the physical storage of messages on disk pagesets, in the coupling facility or in DB/2.

Begins transactions.

Buffer Manager

Handles the physical I/O to pagesets and ensures efficient buffering of data for data manager.

Handles checkpoints.

Building blocks – resource managers

N O T E S

Recovery Manager

Handles transactional operations (commit, backout).

Restart.

Log Manager

Provides interfaces to read from and write to the log efficiently, offloading of active logs, and the management of the archive logs.

Lock Manager

Provides locks to enable transactional isolation.

Coupling Facility Manager

Provides queue manager relevant interfaces to the coupling facility.

Equivalent function to a combination of the other RMs.

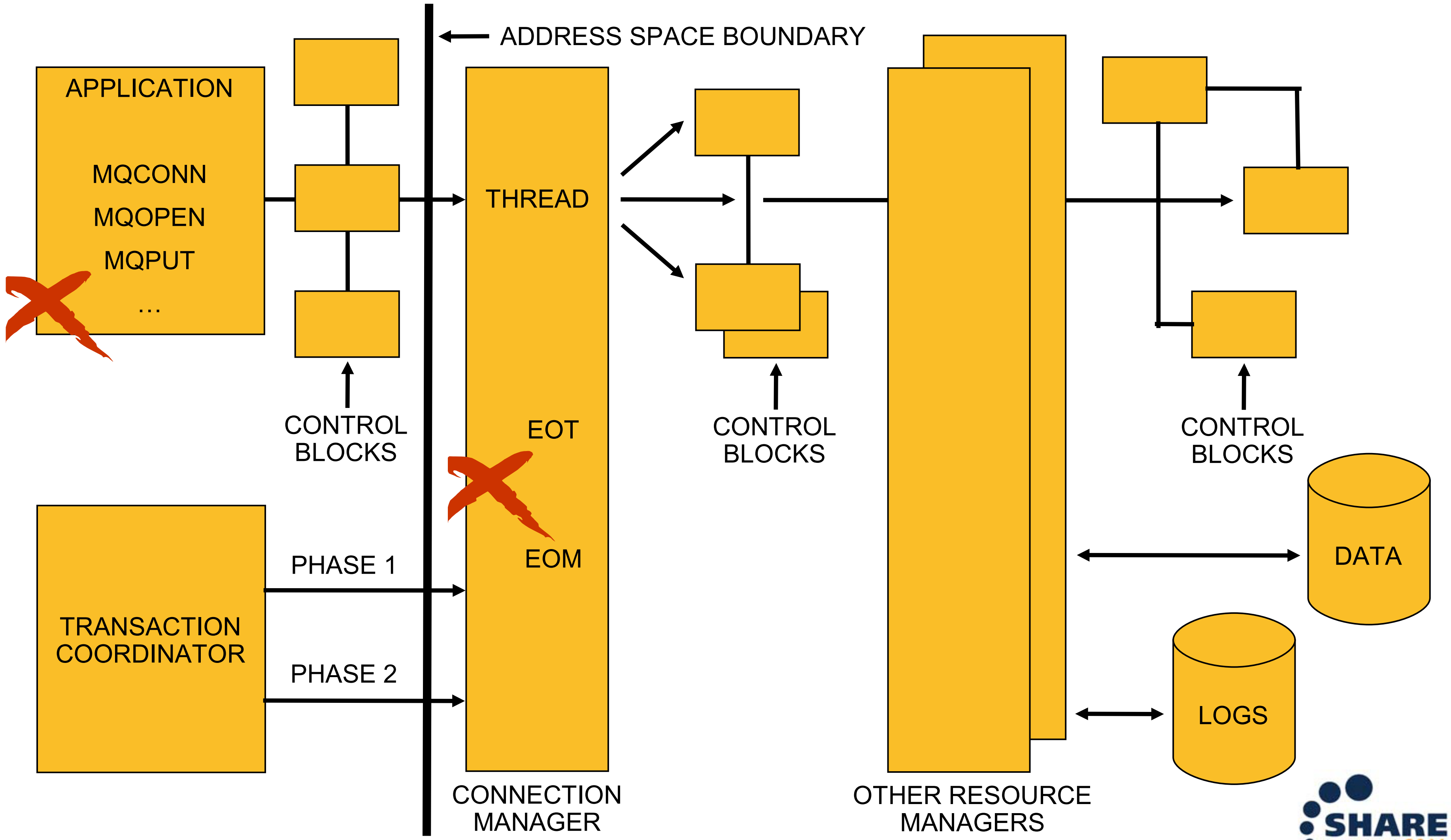
Now we will discuss these queue manager resource managers in detail

We will also discuss the most important data structures, namely how data objects and messages are stored on disk pagesets and Coupling Facility structures.

Distributed queuing is not part of the queue manager

Moving messages between queue managers is the responsibility of a separate address space called the channel initiator ("Mover").

MQ Internals – Connection Manager



Handling applications – Connection Manager

N O T E S

Connection Manager handles all MQI requests

Connection Manager can be considered the "front door" to the queue manager.

There is sufficient complexity in task management and recovery, and application environments, to justify a separate resource manager just for these functions.

Connection Manager understands the adapter tasking scheme with reference to thread management.

However, MQCONN is not necessarily passed to Connection Manager

In environments with relatively simple tasking schemes, an application issuing an MQCONN verb causes its adapter to identify with Connection Manager.

In more complex environments where the adapters are separate units of execution (TCBs), the adapters identify with Connection Manager at application environment initialisation. An application performing an MQCONN in these environments does not cause interaction with Connection Manager. This is why an application doesn't need to issue MQCONN in these environments, most notably CICS.

MQCMIT and MQBACK requests are passed to Recovery Manager

When a transaction is using the queue manager to request WebSphere MQ resource co-ordination, these requests are passed to Recovery Manager.

Such recovery requests are treated differently to the rest of the MQI since they do not relate explicitly to WebSphere MQ resources, but rather to the Unit of Work (UOW) state.

In two phase commit scenarios (e.g. co-ordinating message input/output with database INSERTs in CICS, IMS, RRS), Connection Manager is not called from the application, but from the syncpoint co-ordinator.

Handling applications – Connection Manager

N
O
T
E
S

All other MQI verbs are passed to Message Manager

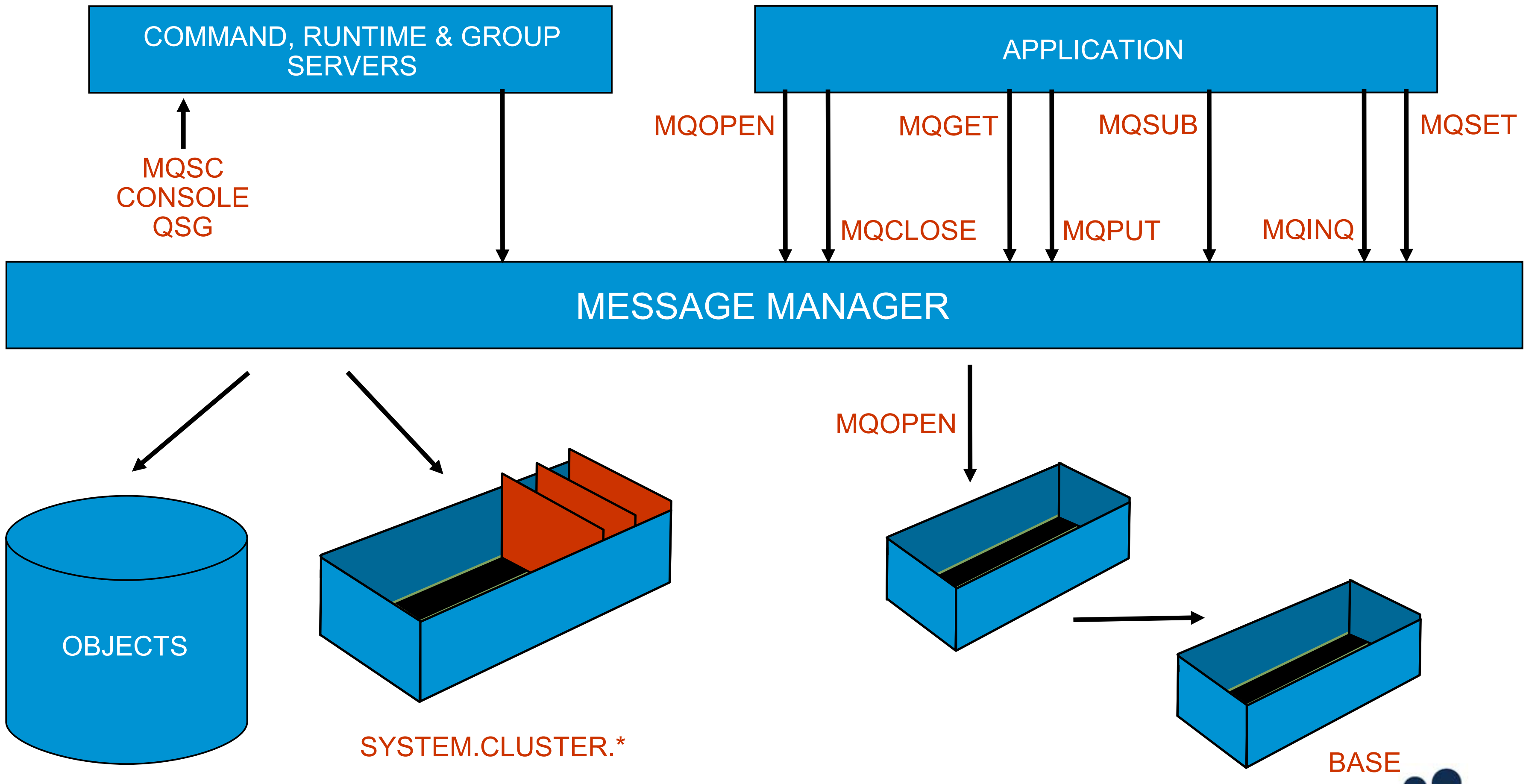
It is Message Manager that processes all requests relating to messages and queues.

Connection Manager handles task termination

Normally and abnormally ending applications have transactional implications. The former have their work committed; the converse is true for the latter.

As the queue manager uses the subsystem interface (SSI) it is notified upon task termination, and can thus implement the appropriate transactional semantics. Note that this can be extremely complex for environments in which the unit of execution moves (e.g. RRS with DB2 stored procedures) and is usually discussed under the subject "Context Management". This is beyond the scope of this presentation.

MQ Internals – Message Manager



VALIDATION, CONSISTENCY CHECKING, TRIGGERING, GET-WAIT PROCESSING

MQ Internals – Message Manager

N O T E S

Message Manager manages all the WebSphere MQ resource related MQI

It is the component that is most oriented towards applications. Other components deal with more abstract concepts (from the application's perspective), such as disk pagesets, buffers and logging.

It controls MQOPEN, MQCLOSE, MQGET, MQPUT, MQPUT1, MQINQ, MQSET, etc. and many other non-externalised application environment APIs.

Message manager manages all MQSC requests

It processes all MQSC requests related to queues, processes, namelists and storage classes. These originate from the Command, Runtime and Group servers.

If a command or resource modification relates to clustering, Message manager notifies the Repository Manager through using the SYSTEM.CLUSTER.COMMAND.QUEUE.

Validation of MQI requests and MQSC commands

It checks validity and consistency of MQI requests (e.g. MQOO_ value, can't open for shared and exclusive input).

Similar validity and consistency checking is performed for MQSC.

Name resolution for remote, alias and cluster queues

Open processing resolves the requested queue name to a base queue name.

At put time, any appropriate headers are added.

MQ Internals – Message Manager

NOTES

Manages security processing

Using user ID information extracted from the QRPL, MQMD, and application environment control blocks, calls Security Manager (not further discussed) to verify access to opened queue.

Implements shared/exclusive access to queues

At MQOPEN it creates locks that allows single or multi user input access to a queue. Uses Lock Manager services for these locks. These locks are released when the queue is closed, so called “allocation duration locks".

Performs processing to enable triggering and get-wait processing

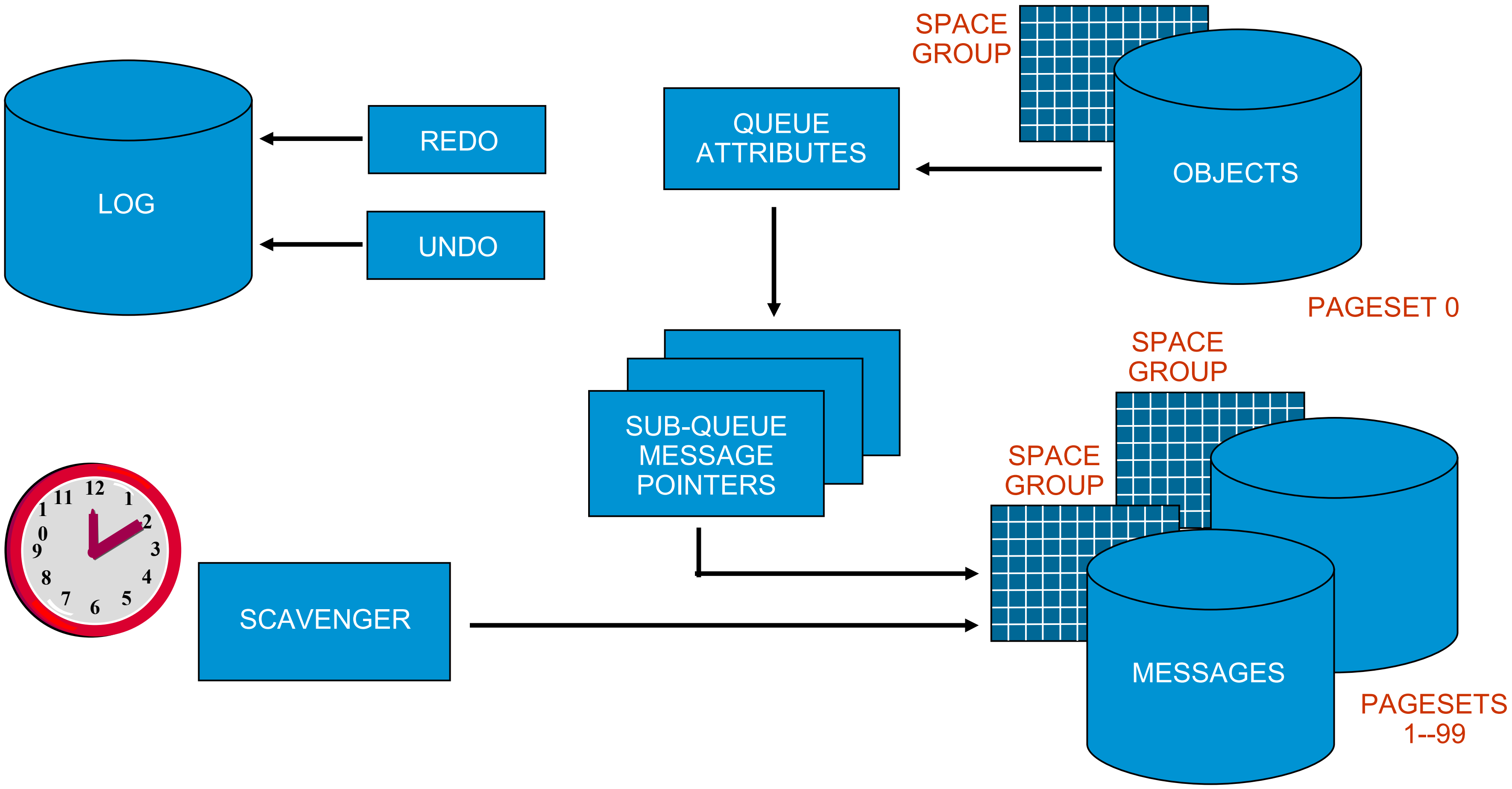
After a successful MQPUT to a queue, it

- Checks triggering rules to see if a trigger message is needed.
- Checks for any MQGET waiters that could be satisfied.

In some shared queue environments, there is Coupling Facility assistance for trigger and get-wait processing.

Uses Data Manager access method to process messages and objects

MQ Internals – Data Manager



MQ Internals – Data Manager

N O T E S

Data manager relates WebSphere MQ messages and objects to buffers in pagesets

Data Manager maps every object (queue, namelist...) and message to a buffer on a page in a pageset.

Pagesets are defined using `DEFINE PSID(pageset id) BUFFPOOL(buffer pool id)`. Uses DD CSQP00xx.

There are several types of buffer corresponding to message or object size (see later).

All WebSphere MQ object definitions are held on pageset 0.

For messages held in shared message queues, the Coupling Facility is used rather than pagesets (see later).

Data Manager manages space in pagesets

The usage of pages in a page set is recorded in the "space map" within each pageset.

The space map is used to allocate new pages for new and updated objects or MQPUT operations.

MQGET operations result in the deallocation of pages.

Page deallocation can also be performed asynchronously by the scavenger process.

Data Manager begins the transaction and logs all transactional operations

A transaction is implicitly started when the first MQPUT or MQGET is performed within syncpoint.

It writes REDO and UNDO operations to the log that can be used to complete or rollback the transaction in the event of failure.

Pageset recovery can be performed at restart to bring pagesets to a consistent point. Data Manager uses "before" and "after" images (position, length and content) of a pageset change recorded on the log to do this.

MQ Internals – Data Manager

N O T E S

Data Manager acquires locks on pages, messages and queues

It acquires exclusive locks on messages, which are not released until commit time to ensure message isolation. This is necessary for both MQPUT and MQGET, since changes are not visible until commit (unless same UOW).

It acquires shared locks on pages to ensure that pages are not deallocated when it is using them.

It acquires shared locks on queues to prevent deletion for in-doubt transactions.

Data Manager uses Buffer Manager to optimally access pages

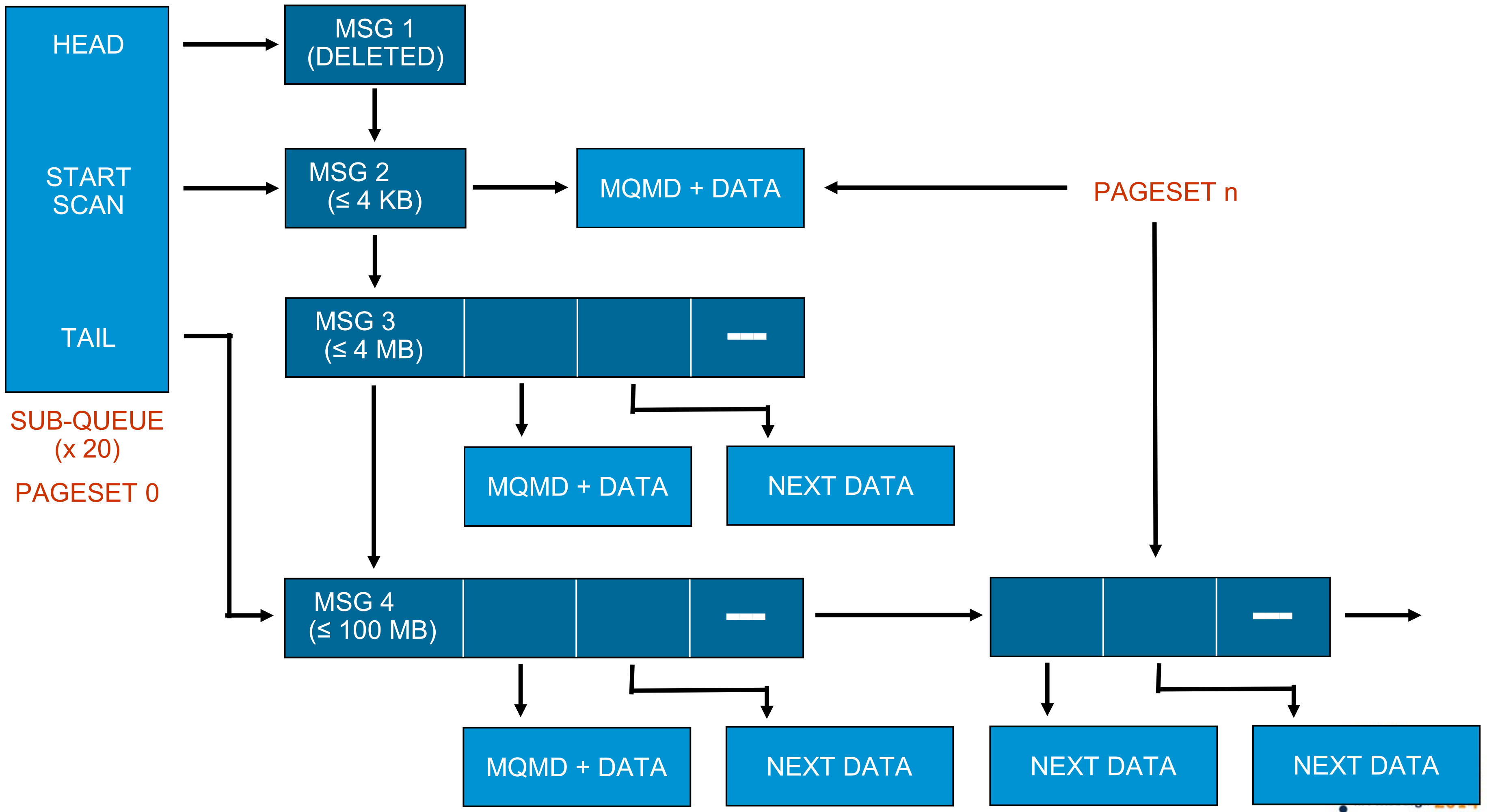
Performance can be greatly enhanced by buffering input and output to pagesets. Data Manager relies on Buffer Manager to read and write virtual storage representations of the pagesets.

A queue is implemented as 20 sub-queues

These sub-queues correspond to the persistence and priority of messages. This improves performance of messages of a given priority at put time, though typically all messages on a queue have the same priority.

Having separate persistent and non-persistent message pages improves restart time, since non-persistent message pages can be marked as deallocated in the space map at restart.

MQ Internals – Local Message Queue Storage



MQ Internals – Local Message Queue Storage

N O T E S

Queues store messages on pages

Each page is 4 KB in size.

Each of the 20 sub-queues has a different anchor point identifying the first and last message pages.

A start position is held to enable more efficient MQGET processing.

According to their size, messages are defined as being short, long or very long.

Different sized messages have different page representations.

Short messages are contained completely within a page

When a getter is traversing a queue and locates an eligible short message, it can directly copy all the data from the page. The message is then marked as deleted.

Long and very long messages have a hierarchical structure

A message greater than 4 KB has a reference in the traversal chain. These references point to message text, or lower level references, and so on. Reference structure means no architected maximum to very long message size.

Pages are deallocated as messages are removed

Messages that reside completely within a page, or the high level reference to long or very long messages do not cause synchronous page deallocation at MQGET time. This would cause too much contention.

Asynchronous deallocation is performed by the scavenger process, who acquires exclusive page locks for pages with all messages deleted.

Text pages do not suffer contention and are deallocated synchronously during MQGET processing.

MQ Internals – Local Message Queue Storage

N O T E S

Large messages *may* have performance impacts

Merely enabling large message support does not impact performance.

The structure of very large messages does not degrade the traversal characteristics of the queue. Notice how long and very long messages are no more difficult to traverse at the high level.

The getter of a 100 MB message has to traverse an increased number of pages.

Putting a very large message can cause a large amount of logging.

Objects are stored in a similar way to messages

All WebSphere MQ objects are held on pageset 0 and are anchored from page 0.

Page 0 contains an array of object types, and objects are chained in a similar way to messages.

Short and long objects (namelists are an example of the latter) have a similar structure to short and long messages.

MQ Internals – Log Manager

- Log read and write functions
- Log shunting
- Multiple active log data sets and archive
- Archive inventory management
- Duplexed for reliability
- “Bootstrap” file
  - End of log location
  - Archive inventory
- Various Utilities

MQ Internals – Log Manager

N O T E S

Log Manager provides log read and write functions

- The log is a VSAM linear dataset.

Log Manager keeps old, "active" log records near the end of the log (since V6)

- The "log shunt" task spots long running units of work
- A condensed form of their log records is periodically rewritten
- "Immunizes" the queue manager from delayed restart times due to long running units of work. The original log records are required in the unlikely event that media recovery is required.

Log Manager provides active log management

- Log Manager writes to the currently active log. Multiple active logs can be configured.
- Log Manager configures its active log characteristics using `CSQ6LOGP`. This includes software duplexed logging for reliability. `CSQJU003` is used to configure the bootstrap dataset (BSDS) that holds active log dataset information.
- Active logs are "offloaded" to archives on tape or disk as they become full.

Log Manager provides archive log management

- Log Manager configures its archiving using `CSQ6ARVP`, e.g. archive naming schemes, volume units etc.
- The queue manager adds archive log names to the bootstrap datasets as logs are archived.
- The BSDS is used at queue manager start-up to identify the active and archived logs necessary to perform recovery.

MQ Internals – Log Manager

N O T E S

Bootstrap contents can be examined

Use the print log map utility, `CSQJU004`, to examine the active and archive log names in the BSDS.

For each log, active and archived, the BSDS contains their RBA ranges. For information, the dates and time associates with these logs is also available.

The log contents can be examined

Use the log print utility, `CSQ1LOGP`, to view the transaction records in the log.

More detailed analysis and replay (`CSQ4LOGS` sample) may be performed.

MQ Internals – Log Print Example

```
00000000D569   URID(00000000D569)   RM(RECOVERY)   TYPE( START UR )
       ****   00640024 00200001 03000000 0000D569 00000000 D545
       0000   00240000 0000D000 00000000 00000700 00000000 00000000 00000000 0000D6C4
       0020   D6E6C4C1 4040B5B4 8FA08793 02864040 40404040 4040C2C1 E3C3C840 4040D6C4
       0040   D6E6C4C1 40400000 00000000 0000

00000000D5CD   URID(00000000D569)   RM(DATA)   LRID(00000000.00000E01)   TYPE( UNDO REDO )
   SUBTYPE( DECREMENT BY )
       ****   002A0064 0600000F C9000000 0000D569 00000000 D569
       0000   00000000 00000E01 00040326 00000001 00000001

00000000D5F7   URID(00000000D569)   RM(DATA)   LRID(00000001.00000201)   TYPE( UNDO REDO )
   SUBTYPE( DELETE )
       ****   0026002A 06000008 C9000000 0000D569 00000000 D5CD
       0000   00000001 00000201 00000000 00000E01

00000000D61D   URID(00000000D569)   RM(RECOVERY)   TYPE( START COMMIT1 )
       ****   007C0026 00200002 03000000 0000D569 00000000 D5F7
       0000   00240000 0000D000 00000000 00000700 00000000 00000000 00000000 00004040
       0020   40404040 40400000 00000000 00000000 00000000 00000000 00000000 00000000
       0040   00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
       0060   00000000 0000

00000000D699   URID(00000000D569)   RM(RECOVERY)   TYPE( PHASE 1 TO 2 )
       ****   0024007C 0020000C 03000000 0000D569 00000000 D61D
       0000   00240000 0000D000 00000000 0000

00000000D6BD   URID(00000000D569)   RM(RECOVERY)   TYPE( END COMMIT2 )
       ****   00240024 00200010 03000000 0000D569 00000000 D699
       0000   00240000 0000D000 00000000 0000
```

MQ Internals – Log Print Example

N O T E S

The log print utility, `CSQ1LOGP`, has been used to print a simple transaction. The transaction consists of a single MQGET within syncpoint, followed by a commit. The transaction completed successfully.

The transaction begins when the first recoverable action is performed, i.e. MQGET under syncpoint is issued.

The `TYPE(START UR)` action represents the start of the transaction (formally called a "Unit of Recovery"). This occurs at RBA position `D569` on the log.

The identifier for this UR is also `D569`. This links together all the log records for this transaction.

Notice that it was Recovery Manager who wrote this log record, `RM(RECOVERY)`.

Processing explicitly related MQGET

Firstly, the queue depth is decremented by 1. This is represented by the `SUBTYPE(DECREMENT BY)` action.

This action is performed by Data Manager, represented by the `RM(DATA)` tag.

The queue information that is changed resides at Logical Record IDentifier `LRID(00000000.00000E01)`. Read this as pageset 0, page E, record 01. Remember all objects live on pageset 0. Also note that this is the first record on this page.

This record has a `TYPE(UNDO REDO)`. This describes what happens when the transaction is rolled back or committed, respectively. If the transaction commits, then we can reapply (REDO) this change to the page at LRID. If the transaction is backed out then we need to perform the compensating action. Can you guess what this might be?

The second record relating to the MQGET operation has `SUBTYPE(DELETE)`. This corresponds to the removal of `LRID(00000001.00000201)`, i.e. pageset 1, page 2, record 1. The compensating action would have `SUBTYPE(UNDELETE)`. Can you say why?

MQ Internals – Log Print Example

N
O
T
E
S

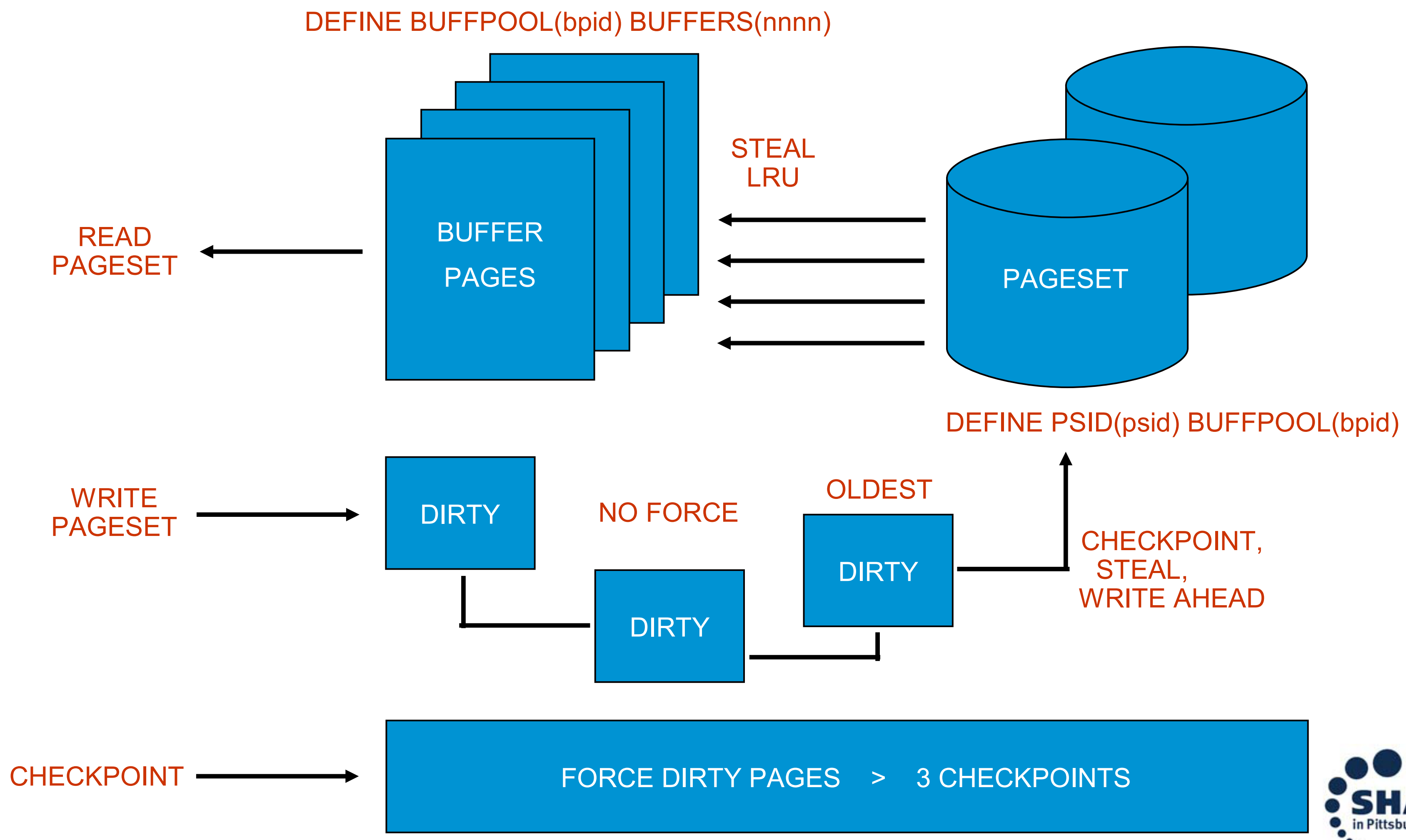
MQCMIT is issued and the transaction completes successfully

Three log records are written for the single phase MQCMIT.

The most important in this scenario is the second log record, which is forced. This is the first time the transaction must wait for disk I/O as this operation must reach the log before control returns to the application.

The transaction end is marked by the `TYPE(END COMMIT2)` record.

MQ Internals – Buffer Manager


DEFINE BUFFPOOL(bpid) BUFFERS(nnnn)
BUFFER PAGES
STEAL LRU
READ PAGESET
PAGESET
DEFINE PSID(psid) BUFFPOOL(bpid)
WRITE PAGESET
DIRTY
NO FORCE
OLDEST
DIRTY
DIRTY
CHECKPOINT, STEAL, WRITE AHEAD
CHECKPOINT
FORCE DIRTY PAGES > 3 CHECKPOINTS

MQ Internals – Buffer Manager

N O T E S

Buffer Manager performs I/O operations to pagesets

The access methods provided for manipulation of logical pagesets result in I/O operations being performed to pagesets.

Buffer Manager caches I/O in virtual storage pages to improve performance.

Buffers are defined for a pageset using the `DEFINE BUFFPOOL` MQSC command. Buffer pools are mapped to pagesets using `DEFINE PSID`.

Buffer Manager caches pageset read operations

It caches the most recently referenced pages, including some read-ahead processing for pages. Applications performing unspecified MQGETs (just get the next message) benefit most from this.

When buffer pools become full Buffer Manager may use a "steal" policy to write out least recently used (LRU) pages to disk. These stolen pages are now available for the new data.

Buffer Manager caches pageset write operations

As all recoverable MQI operations are written to the log, Buffer Manager can defer writing pageset changes. This “no-force” policy greatly improves throughput.

Deferred writes for committed transactions will require REDOing at restart after an abnormal termination, since the change never made it to the pageset.

MQ Internals – Buffer Manager

N O T E S

Buffer Manager enforces “log write-ahead rule”

- Buffer Manager ensures that any pages written out to pagesets have their corresponding log records forced out to the log. This means the log must always be ahead of the oldest deferred page.
- Ensures that any written buffers (including stolen) are consistently recovered with the log.

Recovery Manager broadcasts checkpoint requests to Buffer Manager

- Checkpoint processing is driven by Recovery Manager according to the number of log operations. The `LOGLOAD` parameter defaults to checkpointing every 10K log operations.
- Whenever a buffer page becomes more than three checkpoints old, it is forced out to disk.
- Long checkpoint intervals increase queue manager restart time after an abnormal termination, since more of the log needs to be analysed to bring the pagesets up to date.

MQ Buffers

- Keeping messages in MQ buffers is equivalent to keeping data in storage/RAM
- When your queue(s) are too large to be wholly contained within the available buffers, we have to start paging to DASD
  - This can hurt performance
  - With non-persistent messages especially, if you have not tested the effects of filling your buffers you may get an unpleasant performance surprise in production!
  - If messages continue to arrive at a high rate, recovering can be an uphill struggle

MQ Buffers

- With MQ V8 on z/OS we offer additional help for problems with buffers
  - Improved read-ahead and write algorithms
  - 64 Bit Buffer Pools allow us to move from ~1.6GB of buffers to (theoretically) 16 EB
    - Architecturally: 10 BILLION times larger!
    - Reality: No practical limit.
- More information:
  - “MQ for z/OS New Features Deep Dive”
  - Tuesday, 4:15pm here at SHARE Pittsburgh

Any questions?

This was session 16205 - The rest of the week ……

	Monday	Tuesday	Wednesday	Thursday	Friday
08:30			Application programming with MQ verbs	The Dark Side of Monitoring MQ - SMF 115 and 116 Record Reading and Interpretation	CICS and MQ - Workloads Unbalanced!
10:00					
11:15	Introduction to MQ	What's New in IBM Integration Bus & WebSphere Message Broker	MQ – Take Your Pick Lab	Using IBM WebSphere Application Server and IBM WebSphere MQ Together	
12:15					
01:30		All about the new MQ v8	MQ Security: New v8 features deep dive	New MQ Chinit monitoring via SMF	
03:00	**MQ Beyond the Basics**	MQ & DB2 – MQ Verbs in DB2 & InfoSphere Data Replication (Q Replication) Performance	What's wrong with MQ?	IIIB - Internals of IBM Integration Bus	
04:15	First Steps with IBM Integration Bus: Application Integration in the new world	MQ for z/OS v8 new features deep dive	MQ Clustering - The Basics, Advances and What's New in v8		

Please Fill in Your Evaluations

Complete your session evaluations online at www.SHARE.org/Pittsburgh-Eval