Unit 7. WebSphere MQ internals

What this unit is about

This unit covers some of the inner workings of WebSphere MQ, explaining how the various components interact, and where they are located physically within the file system. The UNIX configuration files are explained, as well as ways to manually stop and remove queue managers.

What you should be able to do

After completing this unit, you should be able to:

- Explain the WebSphere MQ architecture and framework at a high level
- Describe the components and processes that make up WebSphere MQ
- List the directory structure on Windows and UNIX platforms
- Identify configuration files and where to locate them on Windows and UNIX systems
- Describe how to remove a queue manager manually

How you will check your progress

Accountability:

- Checkpoint
- Machine exercises

References

WebSphere MQ System Administration Guide

WebSphere MQ Intercommunication.

Unit objectives

After completing this unit, you should be able to:

- Explain the WebSphere MQ architecture and framework at a high level
- Describe the components and processes that make up WebSphere MQ
- List the directory structure on Windows and UNIX platforms
- Identify configuration files and where to locate them on Windows and UNIX systems
- Describe how to remove a queue manager manually

Figure 7-1. Unit objectives WM203 / VM2032.0

Notes:

Topic outline

- Components and structure
- Directory structure
- Configuration files
- Stopping and removing a QMGR manually

Figure 7-2. Topic outline

Notes:

Components and structure topic objectives

After completing this topic, you should be able to:

- List the functional components that make up WebSphere *for* MQ
- Describe the queue manager processes and their functions
- Describe the processing of the main queue operations:
  - MQCONN, MQOPEN, MQPUT, MQGET, MQCMIT
- Discuss how MQ channels assure delivery of messages

Figure 7-3. Components and structure topic objectives WM203 / VM2032.0

Notes:

Distributed MQ

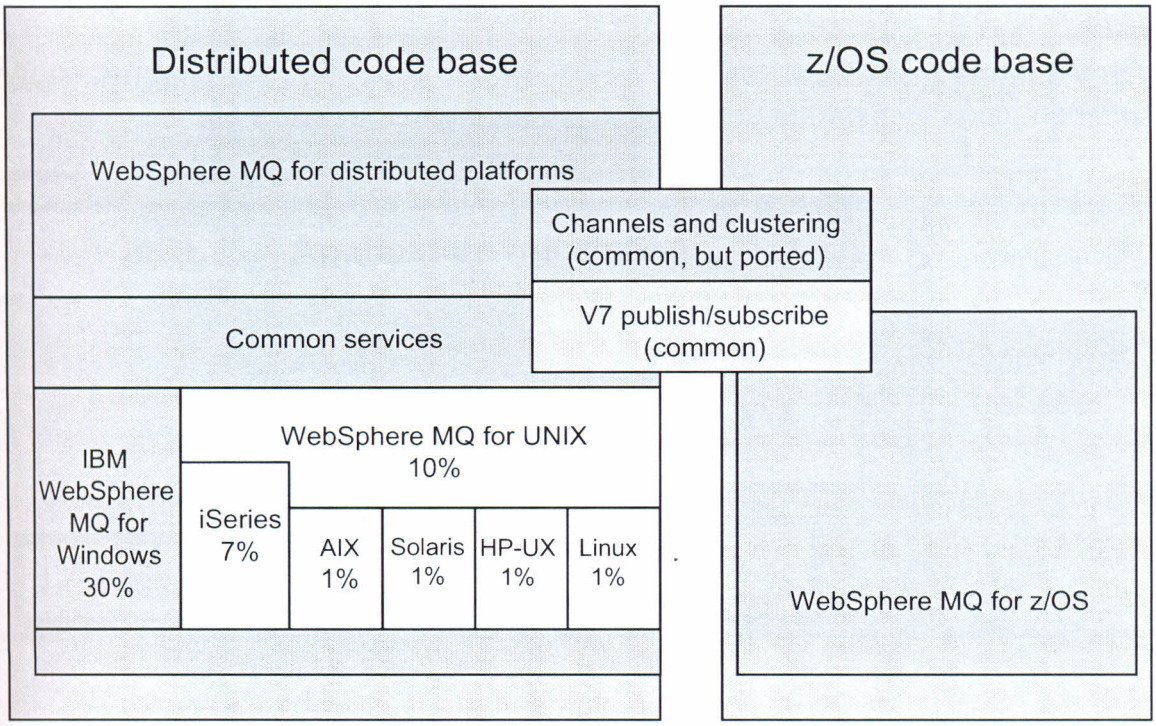


Figure 7-4. What is distributed MQ? WM203 / VM2032.0

Notes:

- The distributed version is the code base for range of workstation and midrange systems. Its design point is a full function, high performance queue manager with a focus on highly portable code.
  - There is a high percentage of common code between versions on different operating systems.
  - The code is written in C with special attention to portability. Even the environment to build the runtime code 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, such as Windows and iSeries.
- The z/OS version of WebSphere 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.
- iSeries 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 and journaling features.
- With V7, there is now truly common code between z/OS and Distributed: previously the "common" code was ported between environments.

Functional view

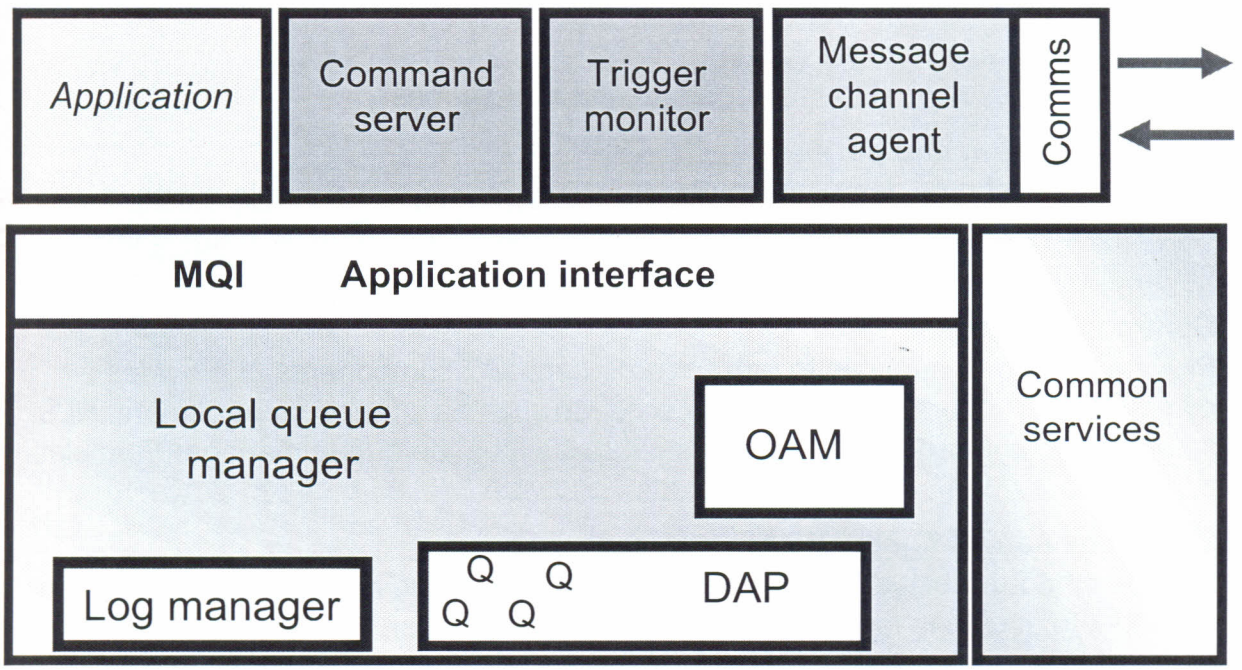


Figure 7-5. Functional view

Notes:

The visual depicts the functional architecture for queue managers. There are three main areas:

- A *local queue manager* (LQM) that manages the physical queues. Its main interface is the MQI.
- Some components that are applications that use the local queue manager.
- Some underlying services.

Here is additional information about the specific components:

- **Application interface (AI)**
  Provides the entry points for the MQI and internal calls used by other WebSphere MQ components. It also provides the language bindings for calls from C and COBOL. Its main purpose is to isolate queue manager resources from the applications and provides the environment and mechanism for the execution of MQI calls.
- **Queue manager kernel**
  Implements the functions of the MQI and the control commands. For example:

  - MQPUT, MQGET, MQOPEN
  - Trigger events
  - Management of object handles
  - Authorization using the OAM
- **Object Authority Manager (OAM)**
  Provides access control to the queue manager resources.
- **Data abstraction and persistence (DAP)**
  This component provides the following functions: storage and manipulation of objects, message operations, message persistence, transactional support, logging changes to queue manager resources
- **Common services**
  Provides low-level services to other WebSphere MQ components. It provides a common set of operating system-like services such as storage management, NLS, serialization, and process management that isolates the queue manager from platform differences.
- **Message channel agent (MCA)**
  Provides the function to move messages from one queue manager to another over a network. To avoid the loss of any persistent messages, an MCA communicates with its partner MCA using a defined set of WebSphere MQ formats and protocols, using the SPI.
- **Communications**
  An MCA uses a communications protocol such as SNA LU6.2 or TCP/IP in order to send data over the network to its partner MCA and receive data from its partner.
- **Command Server** is a special application. It is concerned with processing messages containing commands to manage 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 computer crash and for remembering which operations were in which transaction.

Physical view

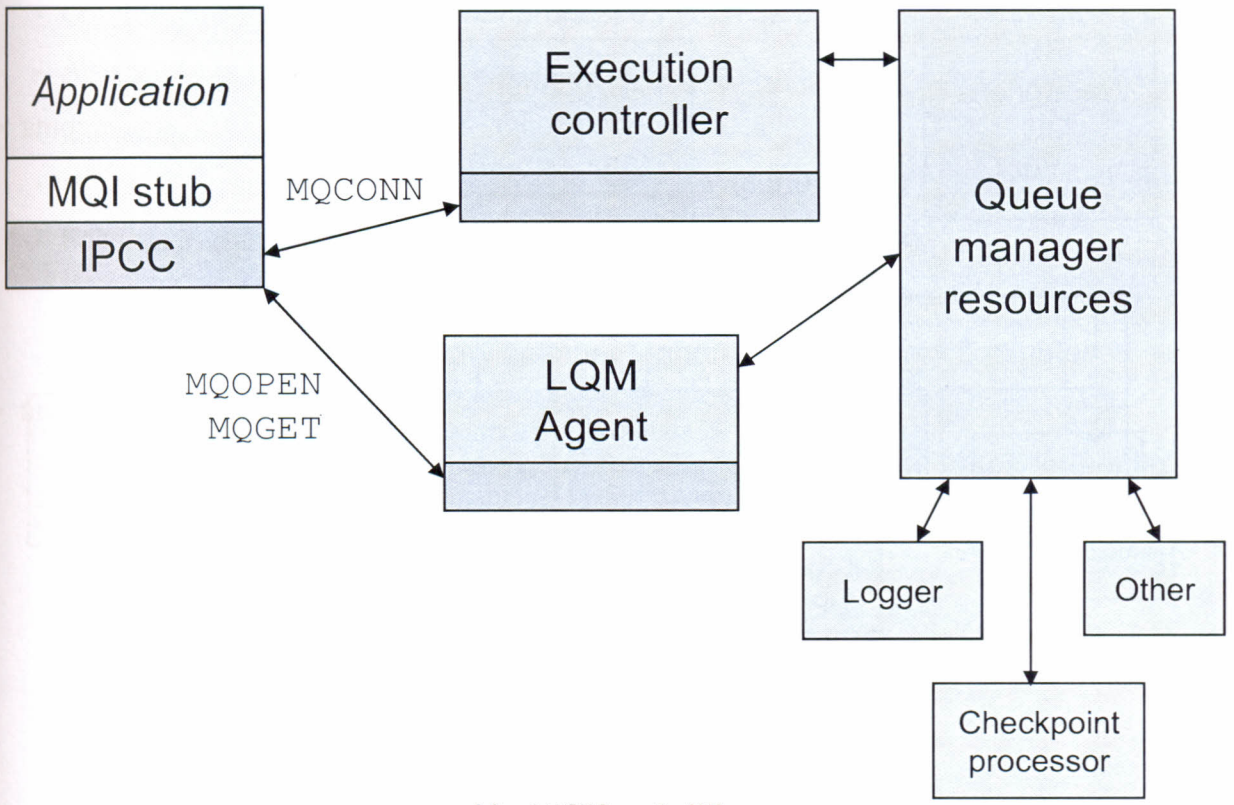


Figure 7-6. Physical view

WM203 / VM2032.0

Notes:

- The *execution controller* is the first process to start when the queue manager is started. It oversees all other processes, controls starting and shutting down, and manages connection requests. IPCC provides local interprocess communications between the queue manager components. It abstracts underlying operating system facilities and is based on shared memory and synchronization primitives.
- *An LQM agent* performs work on behalf of applications. It is where the queue manager kernel and the DAP run and is assigned by the execution controller during MQCONN.
- On WebSphere MQ for Windows, the local queue manager agent is thread-based and so a single agent process can support a number of connected applications.
- The *MQI stub* provides the language binding to the application. It first manages the connection of an application to a queue manager and, once that has been accomplished, it packages subsequent MQI calls and sends them to the local queue manager agent.

Queue manager process tree – V6

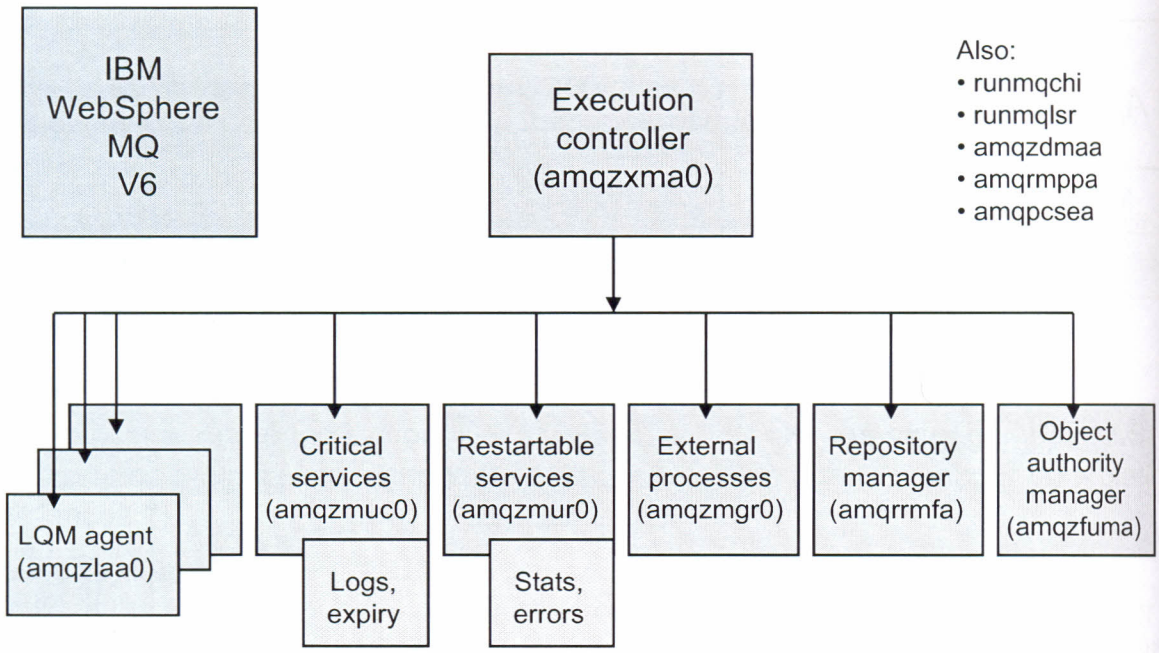


Figure 7-7. Queue manager process tree – V6 WM203 / VM2032.0

Notes:

The Execution Controller is program **amqzxma0**. This program 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 shared resources of the queue manager. It is concerned with managing and monitoring the other queue manager processes and the applications that connect.

The LQM agents are program **amqzlaa0**. Agents perform the operations required to process MQI calls on behalf of applications. Nearly all of the code beneath the MQI is executed by the agents.

The separation of application programs from the critical resources of the queue manager 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.

- **amqrmppa** is the channel pooling process
  - 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

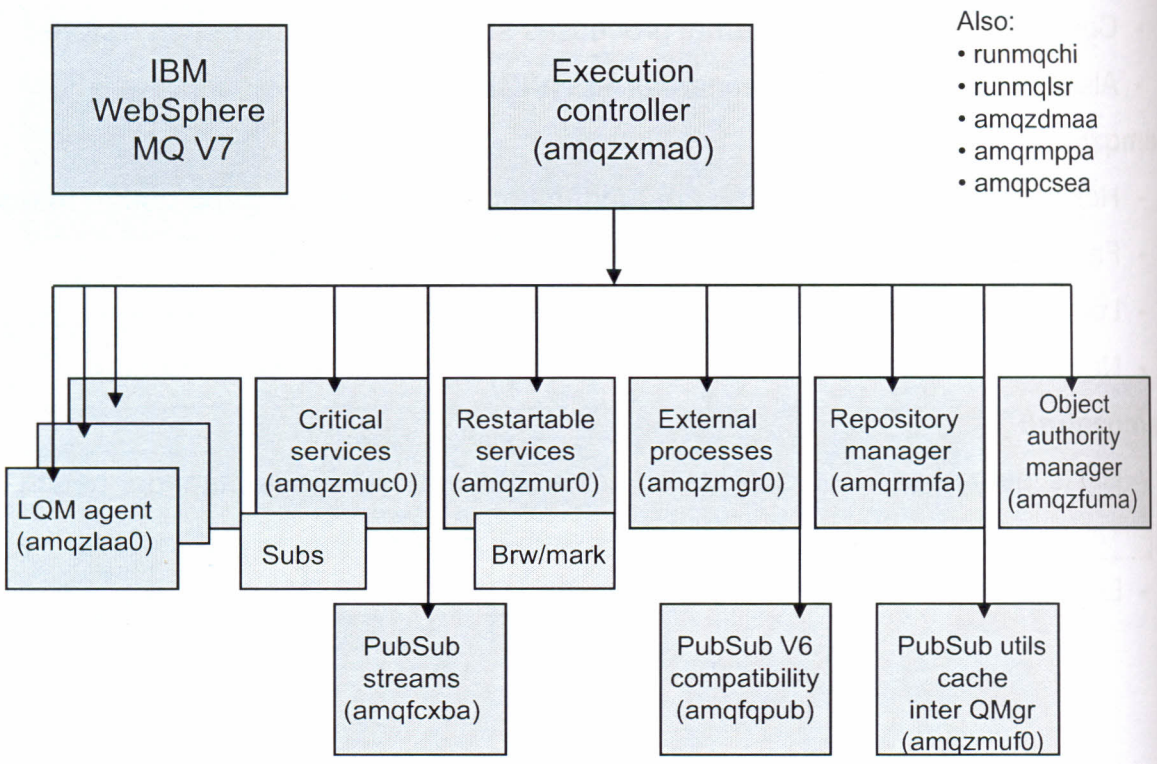


Figure 7-8. Queue manager process tree – V7 WM203 / VM2032.0

Notes:

WebSphere MQ V7 has new processes to handle publish/subscribe operation:

- **amqfcxba, amqfqpub**
  - Provides compatibility with V6 queued publish/subscribe processing for streams
- **amqzmuf0**
  - Publish/subscribe utility container
  - Cache management
  - Inter-queue manager publish/subscribe daemon

  Other tasks added to existing controllers
  - Browse/Mark scanner (restartable)
  - Durable Subscription manager (critical)

  Look in error log to see these processes starting

Queue manager process model

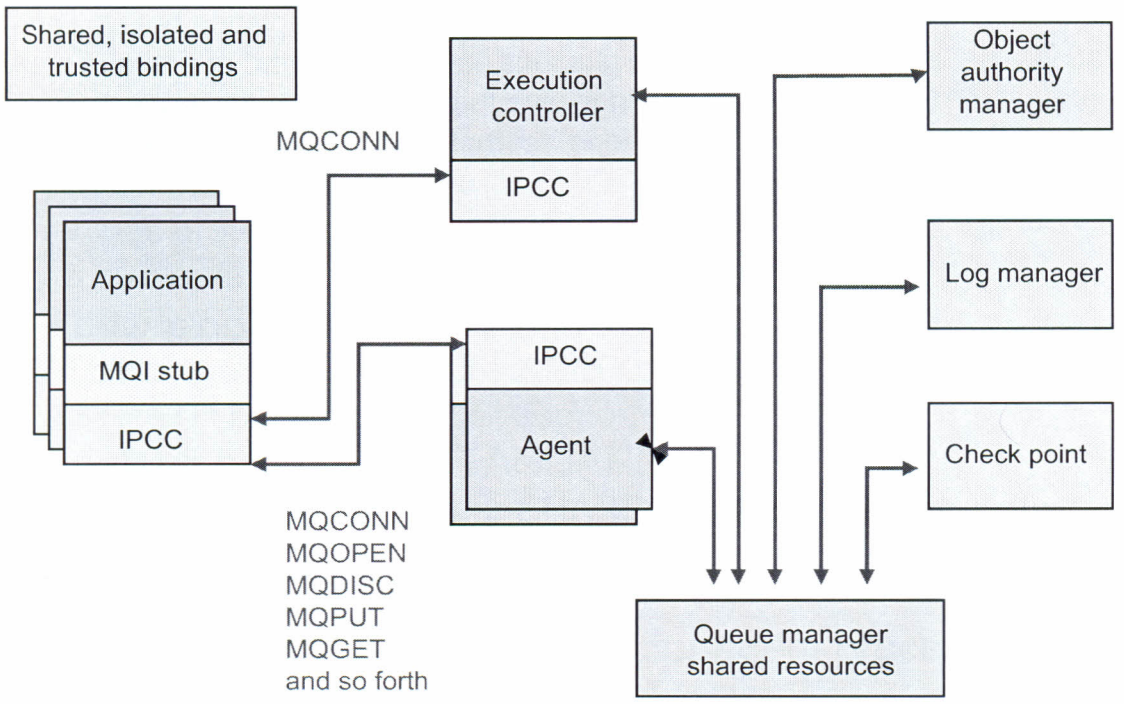


Figure 7-9. Queue manager process model WM203 / VM2032.0

Notes:

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 WebSphere 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 using the IPCC. The agent process performs the MQI calls on behalf of the application. 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.

Isolated bindings

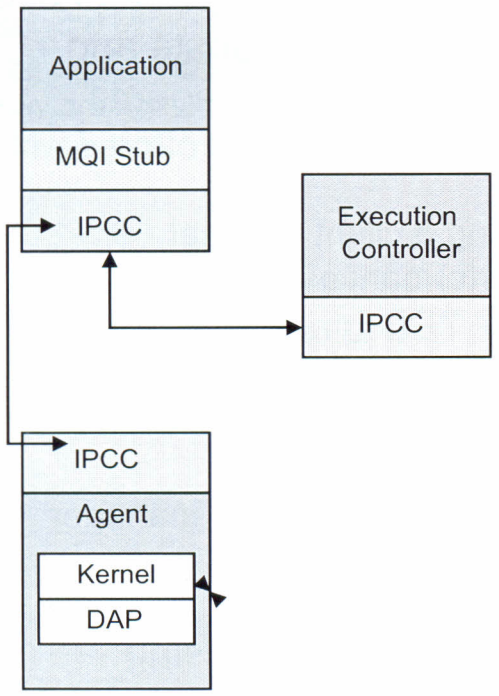
- The traditional WebSphere MQ application binding model uses shared memory and semaphores to transfer requests and replies between applications and queue manager agents.
  - has implications for restart and an alternative binding is offered which uses UNIX domain sockets
  - provides greater isolation without needing a client channel
  - At a measurable but not outrageous performance cost
- The new binding known as an ISOLATED binding
  - Old binding is known as a SHARED binding
  - FASTPATH bindings are still supported
- The default binding is "STANDARD"
  - Maps to either ISOLATED or SHARED based on queue manager configuration
  - Set in ini file
- Uses amqzlsa instead of amqzlaa for agent processing

Figure 7-10. Isolated bindings WM203 / VM2032.0

Notes:

When a queue manager is restarted it needs to ensure that no applications are still using its shared resources – that is why you sometimes see a message about processes still running when you run strmqm. Using isolated bindings stops that problem.

MQCONN processing



- Application (MQI Stub)
  - Verify parameters and handles
  - Construct a Connect message
  - Call API crossing exit
  - Send a message to the EC
- Application (MQI Stub)
  - Receive the reply
  - Construct an IPCC message
  - Send the message to the agent
- Application (MQI Stub)
  - Receive reply and call API crossing exit
  - Return HCONN
- Execution Controller
  - Choose an agent or start a new one
  - Construct a reply IPCC message
  - Return reply to application
- Agent
  - Check the permission of the application to connect
  - Allocate and assign agent resources
  - Send IPCC reply back to application

Figure 7-11. MQCONN processing WM203 / VM2032.0

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 been released by another application. It also creates 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 check 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 computer.

The parameters to MQCONN are bundled up into a connect message. The connect message 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 is 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.

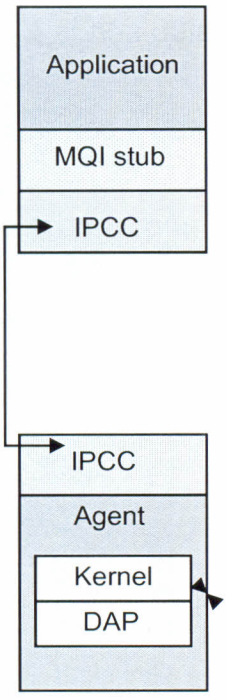
Otherwise, an existing agent thread is to be used and the application stub sends a connection message directly to the thread.

The Kernel checks that the application is authorized 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.

Fast path applications bypass most of the IPCC processing at the expense of integrity

MQOPEN of a queue



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

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

Application (MQI Stub)
- Receive reply
- Call API crossing exit
- Return HOBJ

Figure 7-12. MQOPEN of a queue WM203 / VM2032.0

Notes:

The MQI application stub first does basic parameter checking. This checking 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. The message 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 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 is a queue that is part of a cluster then some resolution of the target (to

the cluster transmit queue) is done here; final resolution to a specific queue manager may be done depending on the MQOO options.

The kernel sorts out this lot and opens the appropriate underlying queue. At the same time, the kernel also checks that the requester of the operation is authorized to perform it. It calls the OAM to perform these checks.

The DAP performs the operations needed to make the physical (local) queue available; referred to as *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. If the queue is already loaded, this work can be avoided.

Finally, the Kernel creates the 'handle' which the application uses 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.

MQPUT processing

- 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

Serialized

Figure 7-13. MQPUT processing

Notes:

The mechanism for reaching the kernel layer for MQPUT is the same as MQOPEN.

- The Kernel verifies the operation for validity. Many aspects 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 is allocated in one of the queue buffers, otherwise it is not allocated in the queue file.
- The operation normally results 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, a log record is not written.
- 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 is written to the queue

file using the file system buffer. If a log record is needed to record the update, it is written before the message data is written to the queue file. If the message is put under sync point, neither write is synchronous. A synchronous write to the log is 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 tracks 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 either completes or is removed completely. There is much 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.

MQGET processing

- 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
  - 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

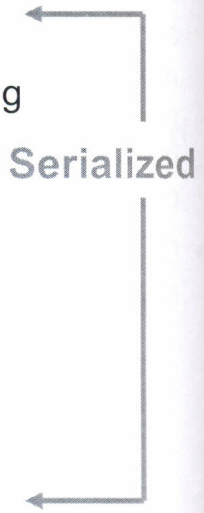


Figure 7-14. MQGET processing WM203 / VM2032.0

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 correlid, the space handles are used to optimize scanning for a suitable message. Expired messages can often be discarded at this stage. If a hashed identifier appears to match then the DAP looks 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 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 results 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 either completes or is removed completely.

On return to the Kernel, the final responses for the application are generated as are any event or report messages required.

MQCMIT

- Kernel
  - Verify operation validity

Deal with 2PC protocol if an XA transaction

- 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

Figure 7-15. MQCMIT WM203 / VM2032.0

Notes:

The Kernel verifies the operation for validity.

The DAP locates the transaction to commit.

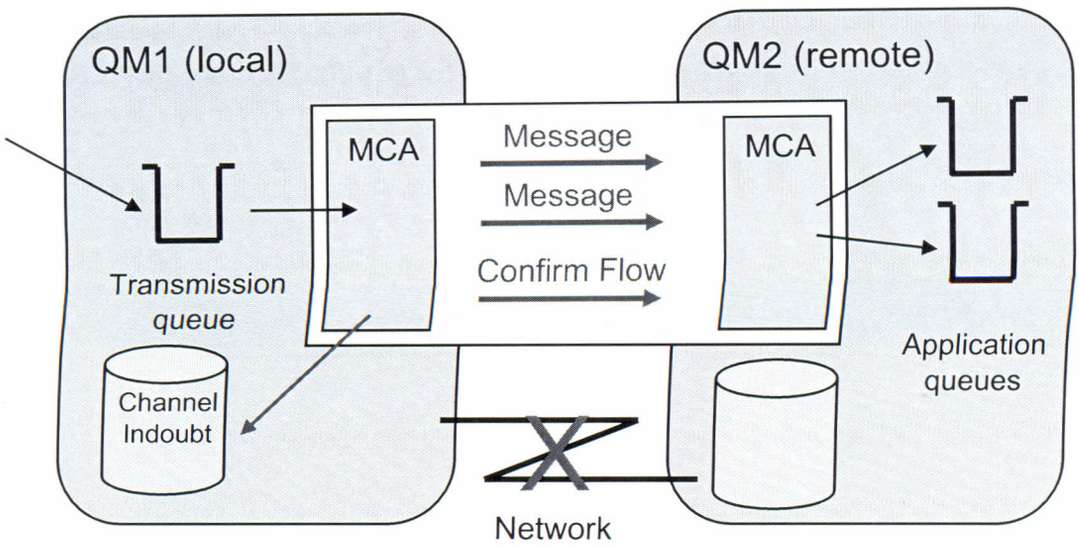
A log record is written to end the transaction and the log is forced to disk (written to the disk and not cached). Once the log is written, you know that all previous log records also have been forced out to the disk which does not include all log records from the puts and gets of this transaction.

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.

Channels assure delivery

- Channel synchronization uses scratchpads
  - A small area of data which can be part of two-phase commit processing
  - The SYNCQ is retained to hold channel status across restarts
  - Channel sync also uses file AMQRSYNA.DAT as an index into the scratchpads



Notes:

- Here you can see the two ends of the channel transferring a batch of messages. The MCAs write data to disk at the end of the batch 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 ends in doubt. When it reconnects to its partner, it notices from the data store that it was in doubt with its partner and asks 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 roll back the messages on the transmission queue.
- This synchronization data is viewable by issuing a DIS CHSTATUS(*) SAVED command. The values displayed should be the same at both ends of the channel.
- If the channel is restarted when it is indoubt, it automatically resolves the indoubt. However, it can only resolve 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 ends 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. In this case, use the output from DIS CHS(*) SAVED to ensure that the correct action COMMIT or BACKOUT is chosen.

Components and structure topic summary

Having completed this topic, you should be able to:
- List the functional components that make up WebSphere for MQ
- Describe the queue manager processes and their functions
- Describe the processing of the main queue operations:
  - MQCONN, MQOPEN, MQPUT, MQGET, MQCMIT
- Discuss how MQ channels assure delivery of messages

Figure 7-17. Components and structure topic summary WM203 / VM2032.0

Notes: