IBM WebSphere MQ
Best Practices

What does this presentation cover?

- IBM® WebSphere® MQ is IBM's message-oriented middleware product. It enables independent and potentially non-concurrent applications on a heterogeneous system to communicate with one other. It is supported on more than 80 different platforms and environments.
- One of the strengths of MQ is its ability to be highly configurable and customizable for diverse customer environments and data transmission needs. However, this strength can open the door for poorly configured systems that don't support future expansion, changing development standards and protocols, and robust security.
- This presentation describes the most common best practices in designing, building, running, and maintaining MQ solutions in order to achieve the full benefits of MQ. Keep in mind that not all recommendations are appropriate for all situations, and that they are offered as guidelines, not hard-and-fast rules.

BEST PRACTICES

DESIGN

Best Practices --- Design
Use short names for queue managers and MQ objects

- Generally, names of MQ objects can have up to 48 characters, except for channel names, which can be no more than 20 characters. You can use upper and lower case A-Z, numeric 0-9, underscore (_), period (.), and two special characters: forward slash (/) and percent (%)
- Use full UPPERCASE for all objects, including the queue manager, to prevent portability issues in heterogeneous environments
- Queue manager names should be unique in the MQ network and reflect the location, function, and environment of the queue manager (dev, test, etc.). Channel names should reflect their function and origin/destination queue manager connection flow; use FROMQUEUEMANAGERNAME.TOQUEUEMANAGERNAME for all sender and receiver channels
- Keep queue manager names short, no more than 8 characters
- Do not use spaces in any names of MQ objects, including system host names
- Although the forward slash (/) and percent (%) special characters are allowed, avoid them because they can cause cross-platform difficulties

Best Practices --- Design
Always assign a dead letter queue (DLQ) to the queue manager

- The DLQ is a local queue that is also referred as the undelivered-message queue. It's a good practice to create and assign one for each queue manager and use it to catch messages that are not sent due to network or destination issues
- If you do not define a DLQ, errors in the application programs may cause channels to shut down. If this happens, not only will the queue stop receiving messages, but it may effect the normal operations of MQ
- In addition, the DLQ should be monitored as messages arriving in that queue are likely an error. DLQ's should be defined, available, and sized for the largest messages the system is expected to handle
- In more robust environments, set up a dead-letter handler as a trigger so that the dead messages are automatically re-tried without intervention
- If you have not associated a DLQ to an existing queue manager, use the MQSC ALTER command to add the DLQ to the queue-manager object

Best Practices --- Design
Use standards like JMS whenever possible

- Today's dynamic business needs are driving IT departments to implement standards-based computing. JMS is a J2EE based standard for messaging, and provides a standard API for applications to use when performing enterprise messaging with application portability
- Using standards such as JMS in your solution will facilitate application portability and reduce dependency on vendor-specific API's. This will in turn ease integration challenges, and reduce vendor tie-in
- SOAP/JMS messaging is increasingly becoming a Web services supported platform, which will help customers in their SOA-based implementations
- MQ is one of the more popular JMS providers to J2EE applications in the market today. MQ JMS implementations can interoperate with other MQ programs without the need for bridges
- The MQ JMS implementation supports both point-to-point and publish/subscribe messaging models
- With standardized JMS APIs, you can upgrade to other JMS brokers, such as WebSphere Message Broker, without redevelopment of the applications or interfaces

Best Practices --- Design
Build with performance in mind

- Use persistent messages for critical or essential data only. Persistent messages are logged to disk and can reduce the performance of your application
- Retrieving messages from a queue by message or correlation identifiers will reduce application performance. It causes the queue manager to search all messages in the queue until it finds the desired message. If applications have high-performance requirements, applications should be designed to process messages sequentially
- Ensure that messaging applications are designed to work in parallel with each other and with multiple instances of applications. The queue manager executes one service request within a queue at a given time to maintain integrity. Avoid programs that use numerous MQPUT calls in a sync point without committing them
- Keep connections and queues open if you are going to reuse them instead of repeatedly opening and closing, connecting and disconnecting
- Configure channels with a disconnect interval so that they can go inactive when there is no activity on the channel after a period of time. This will reduce overhead and help improve overall performance
- MQ performance is commonly bound by disk I/O writes. Ensure that the storage team is involved with disk layouts to ensure the fastest reliable disk writes possible

Best Practices --- Design
Avoid location assumptions and fixed queue names in programs

- Avoiding the use of hard-coded MQ objects and location-names in your applications will provide for significant architectural flexibility, portability, and deployment flexibility
- Deployment flexibility will be enhanced by leveraging alias queues and/or model queues along with WebSphere environment variables
- Adhering to these rules enables applications to be deployed on different environments (unreliable network, mixed operating system, communication protocol, and mixed language environments) without changing the application

Best Practices --- Design
When feasible, cluster your MQ servers

If your infrastructure can support it and your budget allows it, hardware and software clustering provide a great way to increase the resiliency, scalability, and performance of your MQ solutions. Here are some recommendations when considering MQ clustering:

- MQ queue managers do not restart automatically after crashing, and in general, neither MQ clustering nor hardware-based clustering like HACMP provide this behavior by default
- Applications that have message affinities may lead to cluster workload management routines that are complicated and less than optimal
- With appropriate permissions, you can PUT messages with a destination of any queue in the cluster; however you can only GET messages from a local instance of a cluster queue
- You must choose at least one and preferably two queue managers in the cluster to serve as the full repository. Adding more than two full repositories often degrades overall performance, because the cluster will need to send additional traffic and spend more time maintaining all of the repositories
- When deciding which servers will host the full repositories, select servers that are highly reliable, well-managed, and have a static IP
- Use hub and spoke or bus models for maximum flexibility and performance, especially in large environments

Best Practices --- Design
Design infrastructures with fewer queue managers with more queues

- For architectural as well as performance reasons, it is usually better to create one queue manager with 100 queues as opposed to 100 queue managers with one queue apiece
- Where it makes sense, try to limit the number of queue managers in an MQ environment. A single queue manager per server can usually fulfill the needs of all of the queues and applications on that server. While the queue manager can fulfill multiple roles, the segregation of responsibility should occur at the queue level, preferably with a function identifier in the queue name

BEST PRACTICES

BUILD

Best Practices --- Build
Capture completion and reason codes in all application MQI calls

- Design your applications to take advantage of MQ completion and reason codes returned from MQI calls, which enables an application to determine if the message arrived safely and was processed correctly, or if there was a problem with the delivery of the message or processing
- MQ methods will throw an exception whenever the completion code or reason code resulting from an MQ call is not zero
- When using API's other than MQI, be sure to capture the linked exception. In the case of the MQ implementation of JMS, MQ raises a java MQException that contains the completion code, reason code, and details of the exception. Do not return just the JMSexception, as it will not contain the necessary completion and reason codes

Best Practices --- Build
Code applications to continually process messages

- When writing an MQ application, consider pulling messages off the queue as soon as possible and deciding whether to process them immediately or to send them to a failure queue
- Do not design the application code that requires messages to be cleared (such as messages that are not in the anticipated format) or needs administrator intervention prior to the application being able to process additional messages off of the queue

Best Practices --- Build
Close and disconnect connections properly

- Code applications to properly close and disconnect their connections prior to disconnecting or shutting down
- Failure to do so, especially when client connections are in use, will result in hung connections, which will increase resource consumption and may continue until the maximum connections are reached, which will block new connections

Best Practices --- Build
Be aware of the different features of various MQ clients

- Understand the connection options, such as client binding, server binding, managed client binding modes, and the limitations of the client for C, JMS, Java, or .NET applications. For example, the .Net MQ client does not have support SSL channel encryption, XA transactions, and channel compression. Another example is that the Java MQ client supports only the TCP/IP transport and does not read any of the MQ environment variables at startup
- Network and firewall problems may cause MQ connections to fail, and may be misdiagnosed as MQ problems
- Assign connecting applications their own unique channel definitions, so that administrators can easily determine which application is connecting, as well as implement additional security segmentation options
- In production systems, the SYSTEM.DEF.SVRCONN channel should have a non-privileged user added to the MCAUSER field, and potentially be placed in a stopped status. Applications should not be configured to use this channel, and should instead be assigned their own distinct channel

BEST PRACTICES

RUNTIME

Best Practices --- Runtime
Do not use the sample get/put utilities for production purposes

- MQ comes with sample programs that illustrate how to get and put messages. These samples come in both compiled and un-compiled forms.
- Do not use these programs verbatim as production-grade applications to get or put messages, since they have buffer limits and may not have the robustness or functionality needed by production-grade applications

Best Practices --- Runtime
Use runmqlsr in lieu of inetd listener

- In MQ V5.3 or later, the runmqlsr network listener program allows for multi-threaded connections. The advantage of using a multi-threaded process as opposed to initiating a new process per connection via amqcrsta is decreased system resource usage, as well as eliminating potential administrative outages.
- If you have systems with large numbers of connections, such as an integration hub or ESB, and existing listeners running via the UNIX-based inetd listening service (in combination with the amqcrsta program), consider migrating those listeners to runmqlsr.
- Place the listener port in the qm.ini file in the TCP stanza (or in the Windows Registry) so that ports are well associated with queue managers. Then, when the listeners are started, no command-line port number needs to be referenced

Best Practices --- Runtime
Use SupportPacs to extend MQ functionality

- MS03: savequeue manager -- Provides code that exports all MQ object settings from runmqsc in a format that can later be reused to import (and recreate) queue-manager configurations. Depending on your environment, it can be vital for backup and recovery.
- MS0E: runmqadm --- Provides an administrative wrapper that offers runmqsc and administrative-level access to users who are not members of the mqm group, enabling you to grant privileges in a much-more-granular fashion.
- MA01: q utility -- Lets you browse messages and move then between queues.
- MO03: qload utility -- Lets you move messages to and from files, for transport to different systems or for later reuse.
- MC91: High availability -- Provides scripts and instructions for configuring MQ on highly-available UNIX systems such as HACMP and Sun Cluster

BEST PRACTICES

MAINTAIN

Best Practices --- Maintain
When not using third-party configuration tools, automate all configuration and changes via MQSC scripts

- Make a practice out of writing any changes to queue managers into runmqsc commands in the form of a script.
- Any commands that can be typed into a runmqsc editor can also be written to a file for later execution.
- The script can be validated prior to execution using the runmqsc -v < filename switch.
- At execution time, output from the scripts can easily be captured to a log file for later review.
- This practice will help avoid costly typos and speed up execution of changes at execution time

Best Practices --- Maintain
Use the appropriate logging mechanism

IBM WebSphere MQ logs enable recovery of persistent messages from various types of failure. When the system is running properly, the logging process is an overhead that reduces the peak messaging capacity of the system in return for increased reliability.

Circular logging

- Circular logs are used to contain the messages while they are inside of a transaction. During restart of the queue manager, the log files are reconciled against the queue files to determine the disposition of these transactional messages.
- Circular logs are allocated once and then reused as needed. Because the total log allocation is finite, there is no danger of the logs growing to exceed the allotted file space

Linear logging

- Linear logging provides a superset of circular logging. Queue manager restart operations using linear logs function the same as with circular logs: the log files are reconciled against the queue files to determine the disposition of transactional messages.
- In addition to the transactions under syncpoint, linear logs also contain a copy of all persistent messages. If one or more queue files are damaged, the queue can be recovered to the last known good state by replaying the linear logs. This is known as media recovery
- Unlike circular logs which are reused, the number of linear logs increases without limit as messages move through the queue manager. The amount of log data produced in a daily processing cycle is proportional to the amount of data processed as persistent messages during that same period

Best Practices --- Maintain
Use the appropriate logging mechanism

Category	Circular logs	Linear logs
Recovery	Circular logs are used to reconcile units of work that were outstanding at the time of failure. No provision to recover from damaged queue files.	Linear logs contain a copy of all persistent messages that are queued. In a normal restart, linear logs perform the same function as circular logs -- recovery of outstanding units of work. In addition, linear logs support recovery of data when queue files are damaged.
Performance	Circular logs are allocated once and then reused. Therefore no time is required to allocate and format new log extents or to delete or archive them.	New linear logs must be allocated periodically which degrades performance. In addition, the logs must be deleted or moved to prevent filling the underlying file system. Drive head contention during archive operations reduces performance.
Overhead	No administrative overhead is required during normal operations.	Administrators must provide for management of the log files. In addition, the file system must be monitored to prevent the log files from consuming all available space. Human processes touch the administration, operations and support teams.
Operational risk	The loss of a queue file results in loss of all messages on that queue. Loss of a disk partition under the queue files results in loss of all messages on that queue manager.	A normally running queue manager will eventually fill all available disk space if log files are not managed regularly. This will result in an outage of the queue manager if allowed to happen.

Best Practices --- Maintain
Configure automated maintenance of queue managers

- Consider creating scripts to automate the regular maintenance of queue managers.
- Maintenance should include a regular savequeue manager (MS03), security setting backup (amqoamd), linear-log cleanup (if applicable), and backup of other important settings such as .ini files.
- Also, consider reviewing the contents of the FDC error directory for any FDC files that are old and should be removed or are new and should be examined since they indicate critical errors in the MQ system.
- Many third-party products can help provide this functionality. Consideration these vendor products as well as the freely available SupportPacs before writing and maintaining your own scripts, which will require maintenance with each new release of MQ

Best Practices --- Maintain
Schedule and apply fix packs on a regular basis

- Many calls to IBM Support occur because of problems that have already been fixed in an IBM fix pack. Therefore we encourage administrators to plan, schedule, and apply MQ maintenance on a regular basis
- Regular upgrades not only prevent problems from occurring in your environment but also maintain it at a suitable level from where future upgrades can be managed much more effectively and in smaller cycles
- Finally, keeping your environment synced-up with the latest product version will ensure that you never end up in a situation where you need to carry out an urgent upgrade activity to avoid having an out-of-support version