WebSphere MQ Queue Sharing Group in a Parallel Sysplex environment

Introduction

This Redpaper provides guidance for application programmers who want to create applications that run in a WebSphere MQ for z/OS queue-sharing group (QSG) in a z/OS Parallel Sysplex environment. It addresses new terminology and discusses application considerations when using queue managers that are members of a QSG.
A short introduction to WebSphere MQ

WebSphere MQ products enable applications to participate in message driven processing. With message-driven processing, applications can communicate across the same or different platforms by using the appropriate WebSphere MQ products. WebSphere MQ products implement a common application programming interface (API) that applications can use to achieve the transfer of messages via queues.

The common API consists of the following verbs:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQBACK</td>
<td>Backs out a unit of work.</td>
</tr>
<tr>
<td>MQCLOSE</td>
<td>Closes a WebSphere MQ object.</td>
</tr>
<tr>
<td>MQCMIT</td>
<td>Commits a unit of work.</td>
</tr>
<tr>
<td>MQCONN, or</td>
<td>Connects to a WebSphere MQ queue manager</td>
</tr>
<tr>
<td>MQCONNX</td>
<td></td>
</tr>
<tr>
<td>MQDISC</td>
<td>Disconnects from a WebSphere MQ queue manager</td>
</tr>
<tr>
<td>MQGET</td>
<td>Retrieves a message from a WebSphere MQ queue.</td>
</tr>
<tr>
<td>MQINQ</td>
<td>Inquires on the attributes of a WebSphere MQ object.</td>
</tr>
<tr>
<td>MQOPEN</td>
<td>Opens a WebSphere MQ object for subsequent access.</td>
</tr>
<tr>
<td>MQPUT</td>
<td>Places a message on a WebSphere MQ queue.</td>
</tr>
<tr>
<td>MQPUT1</td>
<td>Opens a WebSphere MQ queue, places a message on the queue and closes the</td>
</tr>
<tr>
<td></td>
<td>queue.</td>
</tr>
<tr>
<td>MQSET</td>
<td>Sets selected attributes of a WebSphere MQ queue object.</td>
</tr>
</tbody>
</table>

A message is a string of bytes that is meaningful to the applications that use it. Messages are used for transferring information from one application program to another (or to different parts of the same application).

WebSphere MQ messages consist of a message descriptor that defines control information such as the type and priority of the message, and the application data.

A queue is a WebSphere MQ object that is essentially a data structure used to store messages. Queues are managed by a queue manager.

Other types of WebSphere MQ objects include processes, namelists, and the queue manager attributes object.

A queue manager is responsible for maintaining the queues it owns, for managing requests from applications as well as from sub components of the queue manager, for the storage and retrieval of messages from these queues, and for managing the set of resources that are used by WebSphere MQ. One such resource is the channel initiator.

The channel initiator provides and manages resources that allows the transfer of messages across a network between queue managers, or between client applications and queue managers. This process is known as distributed queuing or intercommunication. The resources involved include message channels, MQI channels, and message channel agents (MCAs).
Message channels define the communications links and behaviour used to transfer messages from one queue manager to another. Message channels can be of various types: sender, receiver, server, requester, cluster sender, and cluster receiver. A channel is established by defining one of these message channel types at one queue manager, and another compatible type at the other queue manager. Compatible combinations are sender-receiver, sender-requester, server-requester, server-receiver, and cluster sender-cluster receiver.

MQI channels are used to transfer messages between WebSphere MQ client applications and server queue managers. MQI channels can be of type client connection (to represent the client end) and server connection (to represent the server queue manager end). The only compatible combination is client connection-server connection. Essentially, the client issues WebSphere MQ API calls that are serviced by the server.

TCP/IP or LU6.2 is typically used to establish the physical communications link between the message channels.

MCAs are the components of the channel initiator that control the sending and receiving of messages. There is one MCA at each end of a message channel. The MCA at the sending end is known as the sender (or outbound) MCA. The sender MCA retrieves messages from a special type of queue known as a transmission queue and places them on a communications link. The MCA that receives these messages and places them on to the target queue(s) at the remote end is known as the receiver (or inbound) MCA.

A WebSphere MQ client is a part of the WebSphere MQ product that can be installed on a separate machine from the server machine where the base product is installed and where the queue manager(s) runs. A WebSphere MQ application can be run on a WebSphere MQ client and it can connect to one or more queue managers on one or more WebSphere MQ servers. An MQI channel is established at the time of connection and it remains active until the time of disconnection.

Queue managers can be grouped to form a cluster. Queue managers in a cluster can make the queues that they host available to every other queue manager in the cluster. Any queue manager can send a message to any other queue manager in the same cluster without the need for many of the object definitions required for standard distributed queuing. Each queue manager in the cluster has a single transmission queue from which cluster sender MCAs can transmit messages to any other queue manager in the cluster.

WebSphere MQ for z/OS V5.2 (and above) exploits the Coupling Facility (CF) technology and a shared DB2 database to allow queue managers to be configured to form a cooperating queue sharing group (QSG). Within a QSG, queues may be defined as shared queues which are accessible to all queue managers in the QSG. Queues which remain accessible by individual queue managers only are non-shared or private queues. A new class of channels known as shared channels are also provided. These channels offer enhanced behaviour and availability over non-shared or private channels.

A stand-alone queue manager

On z/OS, WebSphere MQ runs as a z/OS subsystem. Within the subsystem, the queue manager is activated by a startup procedure. The subsystem and the queue manager have the same name of up to four characters. Applications connect to the queue manager using this name. The queue manager owns and manages the set of resources that are used by WebSphere MQ. These include:

- Page sets that hold the object definitions and message data
- Logs that are used to recover messages and objects in the event of queue manager failure
- Processor Storage
- Connections through which different application environments (CICS, IMS, and Batch) can access the common API (via adapters that are part of the queue manager)
- Channel Initiator

Figure 1 illustrates a queue manager, a channel initiator, and connections to the queue manager from different application environments.

Queue managers in a queue-sharing group

A group of queue managers that can access the same WebSphere MQ object definitions and that can share message data is called a Queue Sharing Group (QSG). A QSG has a name of up to 4 characters. QSG names within a network must be unique, and different from any queue manager names. QSGs on WebSphere MQ for z/OS consist of the following components:

- a number of z/OS queue managers (running within a single z/OS sysplex) along with their associated channel initiators,
- a Coupling Facility containing one or more Coupling Facility structures (CF structures) that are used to hold messages on shared queues,
- a shared DB2 database (also known as the DB2 shared repository) that is used to hold shared object definitions.
Shared (or group) object definitions for any type of WebSphere MQ object are held in the DB2 shared repository. Each queue manager in the QSG connects to a DB2 subsystem that is a member of the same DB2 data sharing group. This allows each queue manager to access the definitions and create instances of them.

Figure 2 illustrates a QSG that contains two queue managers. Each queue manager in Queue Sharing Group QSG1 has an associated channel initiator and its own local page sets and log data sets, as well as access to the Coupling Facility and the DB2 shared repository.

Figure 2   A queue-sharing group

A shared local queue can hold messages that can be accessed concurrently by any queue manager in the QSG. For example, an application connected to a queue manager in the QSG can put a message to a shared queue while another application, connected to a different queue manager in the same QSG, can get this message. This provides a rapid mechanism for communication within a queue-sharing group without the need to define and start WebSphere MQ channels between queue managers.

**Coupling Facility list structures**

The messages on shared queues are held in Coupling Facility list structures. Two types of list structures are used. Administration list structures, for holding information required for unit of work recovery and for coordinating WebSphere internal activity across the QSG, and application list structures, for holding messages on shared queues.

Figure 3 on page 6 illustrates the two types of list structures
Each Coupling Facility list structure is dedicated to a specific QSG, but a Coupling Facility can hold structures for more than one QSG. Each queue manager can only belong to a single QSG, and queue managers in different QSGs cannot share data. Up to 32 queue managers in a QSG can connect simultaneously to a Coupling Facility list structure.

A single Coupling Facility list structure can contain message data for up to 512 shared queues. The size of a list structure limits the space available for message data. The size of a list structure itself is limited by the fact that it must lie within a single Coupling Facility, and that the available Coupling Facility storage may be shared between other structures for WebSphere MQ and other products.

Figure 4 illustrates a Coupling Facility being shared between two QSGs. It is possible that the two QSGs represent production QSGs (set up for two suites of application programs that for reasons of sensitivity must be restricted from interacting with each other), or a production and a test QSG (due to a limitation of there being only one Coupling Facility available, for example). In the latter case, the same named shared queues can be defined to the list structures for both QSGs to assist in migration from test to production.
Channel Initiators in a queue-sharing group

Multiple channel-initiators are employed within a queue-sharing group to obtain capacity and availability improvements over stand-alone channel-initiators. The architecture and behaviour of shared channels makes this possible, whilst requiring few, if any application changes.

There are two repositories that hold data relevant to running a channel. Firstly, the synchronization queue holds information about the last batch of messages transmitted. This queue is a regular MQ queue reserved for system use that holds the batch data in MQ messages. This data is used to enforce assured, once-only message delivery. Secondly, the channel status table stores information about the state of all active channel instances. It is used to enable correct channel behaviour. For example, an already running channel must ignore a subsequent start request, whereas a stopped channel requires a start-command to be issued against it before it may be used to transmit messages.

Two classes of channels are available. Private channels retain all the behaviour of pre-QSG versions of MQ, and are managed and run unchanged. Shared channels benefit from QSG related improvements. The class of a channel instance is determined by the type of the transmission queue and the address specified in the channel definition.

To enable high-availability, any channel initiator in the QSG must be able to run a shared channel, and the channel must be recoverable by any channel initiator following a failure. This requires that the state repositories for a shared channel must be accessible by any of the QSG channel initiators. In order to achieve this, the synchronization queue used by shared channels is a shared queue (private channels continue to use private synchronization...
WebSphere MQ Shared Queues, and a subset of a shared channel's status resides in the DB2 shared repository (private channels continue to use a status-table held in the channel initiator's address space). The full status record for a shared channel is also stored in-memory by the channel initiator currently running the channel.

The inbound-architecture is shown in Figure 5. An MQ channel initiator in a QSG may listen for inbound connections on two different kinds of port. In fact these ports may actually be two different ports on the same TCP/IP address, two entirely separate TCP/IP addresses, or different LU6.2 logical units, but the term port will be used here as a shorthand. A connection made through the private port causes the receiving end of the channel to act as a private channel. If a connection is made through the group port the receiving end of the channel behaves as a shared channel. In addition, a generic port is assigned to the QSG as a whole. This port may be managed by any front-end routing hardware or software, e.g. Sysplex Distributor, WLM DNS, 2216 router, Cisco routers, VTAM Generic Resources (GR). Any connections to this port should be rerouted by the front-end hardware or software to the group port of one of the QSG channel initiators (the choice is determined by the front-end mechanism, not WebSphere MQ).

The outbound-architecture is shown in Figure 6. A private channel transmits messages residing on a private transmission queue, whereas a shared channel transmits messages residing on a shared transmission queue.

When a start request is generated for an outbound shared channel, the channel initiator processing the request chooses the channel initiator that will run the channel via a workload-balancing algorithm. Start requests may result from trigger-starting the channel, or from a command. The algorithm calculates a load-number for each channel initiator in the QSG based upon the number of channels a channel initiator is currently running and the maximum number of channels it has been configured to run. These numbers are stored in the
CF for every QSG channel initiator, so that they are accessible by all the QSG members. The least loaded channel initiator is requested to run the channel. If the lowest load number occurs more than once a weighted-random choice is used to choose a particular channel initiator, with the weighting proportional to the headroom available (the current number of channels being run subtracted from the maximum number of channels it has been configured to run). The load balancing algorithm aims to target start channel requests to the most appropriate channel initiator to keep the loads on each channel initiator approximately balanced across the QSG.

It should also be noted that:

- the type of port (private or group) that an inbound channel is connected to determines whether the receiving end of a channel is private or shared
- the type of transmission queue (private or shared) that the sending end of an outbound channel services determines whether it is private or shared.
- a shared channel derives its benefits over private channels from the channel initiators in the QSG cooperating to provide enhanced behaviour; this is unlike a shared queue which is defined as a shared object, and is usable concurrently by all queue managers in the QSG.
- while a shared channel is running, it is managed by a single channel initiator in the QSG.

**Intra-group queuing**

WebSphere MQ for z/OS V5.2 introduced a mechanism called Intra-group queuing (IGQ). With IGQ messages may be transferred between queue managers in a QSG without the need to define private channels between these queue managers.
On each queue manager in the QSG an agent called the IGQ agent retrieves messages from a shared transmission queue called the SYSTEM.QSG.TRANSMIT.QUEUE. Messages that are put to this queue are tagged with a destination queue manager name. The agent running on the destination queue manager retrieves these messages and places them on the required target queue(s).

Figure 7 illustrates intra-group queuing in use between two queue managers in a QSG.

With IGQ, administration costs are reduced since there is no need to define channels between the queue managers.

The flow is as follows:

a. The requesting application opens, and puts a query on to, the remote request queue (the remote request queue is defined with RNAME set to the name of the local request queue, RQMNAME set to QM2 and XMITQ set to blanks; assuming that IGQ has been enabled, name resolution resolves this to the SYSTEM.QSG.TRANSMIT.QUEUE).

b. During the put request, queue manager QM1 tags the message with the name of the target queue manager QM2.

c. The IGQ agent on QM2 retrieves messages that are tagged with queue manager QM2, and delivers them to the target queue. In this case, the local request queue.

d. The server application processes requests on the local request queue and generates a suitable response (though, the flow for the response message has not been shown in Figure 7).
Benefits of configuring queue managers in a QSG

**High availability of queue managers**
As applications can connect to any queue manager in a QSG, and as all queue managers in a QSG can access shared queues, applications do not need to be dependent on the availability of specific queue managers. In the unlikely event of a queue manager failure, applications can continue to service shared queues by connecting to any remaining active queue manager in the QSG.

**High availability and capacity of server applications**
Cloned instances of server applications can be run on each z/OS image in the sysplex to provide higher server application availability and capacity. Each server instance can service request messages that are put to a shared request queue.

**Scalability**
New instances of server applications, or even new z/OS queue managers, can be easily introduced into the QSG to service shared queues and to run additional channels.

**Pull workload balancing**
Based on the available processing capacity of each queue manager in the QSG, ‘pull’ workload balancing is naturally achieved for server applications.

**Increased throughput of persistent messages**
Persistent message throughput is limited by the rate at which data can be written to the log. Running multiple queue managers, each with its own log, can result in a significant increase in the throughput. By using shared queues, the increase in throughput can be made available through a single queue or through a set of queues.

**Low cost messaging within the sysplex**
It is not necessary to define Websphere MQ channels to transfer messages between applications connected to different z/OS queue managers in the QSG. Messages can simply be put to and retrieved from shared local queues.

**Queue Manager Peer Recovery**
If a queue manager in the QSG should fail and abnormally disconnect from the Coupling Facility, the remaining queue managers in the QSG are notified of this, and one of these remaining queue managers will complete any pending shared queue units of work for the failed queue manager. This maximizes the availability of messages and avoids the need to wait until restart of the failed queue manager.

**Reduced administration**
As shared objects are stored in a DB2 shared database, once they are defined on one queue manager they become available to the entire group (hence, shared objects are also known as group objects). Similarly, any updates to shared objects need only be made on one queue manager in the QSG.

**Increased channel availability**
A shared channel is run by one of the channel initiators in the QSG. In the event of a failure affecting the communications subsystem, the channel initiator address space or the queue manager address space, the channel is restarted on one of the other QSG channel-initiators. It then re-synchronizes with its remote partner and continues message transmission.
Channel load-balancing

Outbound shared channels are run by a channel initiator chosen based on the number of channels each channel initiator is running, and the number of channels the channel initiator is configured to run. Inbound shared channels may be routed to one of the QSG channel initiators as determined by front-end routing software or hardware.

Issues to consider before proceeding to exploit QSGs and shared queues

Before you proceed to exploit QSGs and shared queues, you should take into account the following considerations (additional issues are addressed in the WebSphere MQ manuals):

1. The number of QSGs you need to create for your application environment.
   
   You may wish to setup different QSGs for production and test use, or define multiple production QSGs to achieve queue manager, and hence application, separation. By defining the appropriate level of access permissions at the QSG or queue manager level, you can control access to a QSG or to a given queue manager within a QSG, on a per application basis.

2. As with queue manager names, you/your system administrator will need to derive a suitable naming convention for your QSG names.
   
   The WebSphere MQ for z/OS Concepts and Planning Guide offers some suggestions for this.

3. The number of queue managers in each QSG.
   
   You will need to decide on the number of queue managers you need to run in each QSG to handle your application throughput.
   
   As persistent message throughput is limited by the rate at which data can be written to the log, running multiple queue managers (each with its own log) in a QSG can significantly increase the message throughput.
   
   The WebSphere MQ for z/OS V5.3 Performance Report (SupportPac MP1D) shows that a throughput rate of more than 11000 persistent messages (of size 1000 bytes) per second was achieved when running with four queue managers using a shared request queue and a shared reply to queue. This contrasts with a throughput rate of under 3000 persistent messages when running with a single queue manager in a QSG. Refer to the performance report for full details of the system configuration and design of the test application that was run to achieve these results.

4. Whether your application environment requires you to run cloned, or different class of service, queue managers within a QSG.
   
   a. With cloned queue managers, you should define all objects as group objects so that the same definitions are made available to all queue managers within the QSG. This will enable your application instances to run against any queue manager within the QSG.
   
   b. With different class of service queue managers, you may choose to vary the attributes of an object from queue manager to queue manager. This can be achieved by defining an object with the same name, but with different attributes, on each queue manager.
      
      Another example is to define one of your queue managers within a QSG to be a gateway queue manager, for communication to remote queue managers outside the QSG, by only running a channel initiator on this queue manager.

5. The number of Coupling Facility list structures that your applications intend to use.
A QSG requires a minimum of two list structures - the administration list structure and an application list structure. However, for data isolation reasons, for example, your application environment may demand that you define multiple application list structures. Chapter 16 of the WebSphere MQ for z/OS Concepts and Planning Guide lists other examples of why you may need more than one application list structure.

For performance reasons, it is recommended that you use as few CF application structures as possible.

If your applications issue MQGETs (say from a request queue) and MQPUTs (say to a reply to queue) within syncpoint, it is recommended that you use a single application structure for holding the messages associated with the two queues.

During tests, using a single queue manager with the request queue in a different application structure to the reply to queue, the unit CPU cost per reply/request was seen to increase by 21%. However, with all MQGETs and MQPUTs out of syncpoint, the cost of using one or multiple application structures was seen to be the same.

6. The administration list structure has a fixed name of qsg-nameCSQ_ADMIN (where qsg-name is the name of your QSG) but you will need to derive a suitable naming convention for your application list structures.

For example, you could use a name associated with the suite of applications that will use the shared queues whose messages are to be held in the list structure.

7. The size of your list structures.

It is recommended that the size of the administration structure is at least 10MB.

The size of your application structures will depend on the size of your messages and on the maximum number of messages that will be concurrently held on the shared queues that map to your application structures. Of course, you will also be limited by the amount of available storage in the Coupling Facility. The WebSphere MQ for z/OS Concepts and Planning Guide discuses this in more detail. You should also refer to the WebSphere MQ for z/OS V5.3 Performance Report (supportPac MP1D) which provides a simple formula for estimating the size of an application list structure.

8. Once you have decided on the number and size of your application list structures, your systems administrator will need to define them to your Coupling Facility Resource Manager (CFRM) policy.

9. WebSphere MQ for z/OS V5.3 introduces CFSTRUCT objects that can be used to define the backup and recovery parameters for Coupling Facility application list structures. You can define/alter your CF structures to be recoverable, and use the BACKUP CFSTRUCT command to backup persistent messages held in CF structures to logs. In the unlikely event of a CF failure, you can use the RECOVER CFSTRUCT command to recover persistent messages that had previously been backed up.

You can only backup those structures that are marked as recoverable. WebSphere MQ for z/OS V5.3 allows you to define/alter your CF structures to be recoverable. You should consider putting persistent messages to recoverable structures and non-persistent messages to non-recoverable structures.

Refer to the WebSphere MQ for z/OS System Administration Guide for further details.

10. The number of shared queues required by your applications, and the application list structure that each queue will map to.

Using a large number of queues in a single application list structure does not have any significant effect on the performance.

11. You will need to consider which of the following types of queues you want to define as shared queues:

   a. request,
b. reply to,
c. initiation,
d. transmission,
e. dynamic queues (possibly also being used as reply to queues).

When making this decision, consider the items listed in “Limitations/Restrictions” on page 22.

12. You will need to consider the unit of work (UOW) scope and peer recovery. UOW scopes include

a. Local only UOWs - exist when messages are placed or retrieved, within syncpoint scope, from private local queues only.
b. Shared only UOWs - exist when messages are placed or retrieved, within syncpoint scope, from shared local queues only.
c. Mixed UOWs - exist when messages are placed or retrieved, within syncpoint scope, from both private local and shared local queues.

In the unlikely event of a queue manager failure, for:

a. local only UOWs, peer recovery is not possible. The failing queue manager must be restarted to recover the local only UOWs.
b. shared only UOWs, peer recovery is possible.
c. mixed only UOWs, peer recovery is possible for shared messages only. To recover local messages, the failing queue manager must be restarted.

13. You need to consider if your application has any message sequencing issues. For example:

a. Does the design of your applications allow you to run multiple instances of them against a shared queue?
b. Do your applications open queues for input exclusive use? Can they be changed to open queues for input shared so that multiple instances of them can be run against shared queues?
c. Do your applications issue keyed MQGET calls? Will they continue to work with shared queues given the limitations with shared indexed queues. See “Limitations/Restrictions” on page 22.
d. You may need to consider using serialized applications to preserve message sequencing. See “Serialized applications” on page 17.

14. As far as non-application queues are concerned:

a. You should not define your dead-letter queue (DLQ) as a shared queue because the maximum message size limitation for shared queues will prevent messages of length greater than 63KB from being written to the DLQ.
b. You can define SYSTEM.ADMIN.*.EVENT queues as shared queues. As the correlation identifier of each event message gives an indication of the originating queue manager, this may be useful in making some deductions about load balancing across the QSG.
c. You could define your SYSTEM.DEFAULT.LOCAL.QUEUE to be a shared queue so that subsequent queues based on the default are defined to be shared.
d. If you plan to use shared channels or intra-group queuing, you must define SYSTEM.QSG.* queues as shared queues.

Note: The * above is used as a wild card. Refer to the WebSphere MQ manuals for details of the possible names for these queues.
How applications can exploit WebSphere MQ shared queues

Existing MQSeries for OS/390 and WebSphere MQ for z/OS applications will run on a queue manager that is a member of a QSG without change. However, to gain from the benefits of running queue managers in a QSG, you need to consider some of the issues highlighted by the following scenarios.

In the following scenarios, it is assumed that the applications follow a requester-server model. Requesting applications flow request messages into the queue manager from the network, back end server applications perform database updates or inquiries and return response messages to the requesting applications via reply to queues. Such a scheme works equally well whether the requesters use MQI channels to connect as clients directly into the queue manager, or whether downstream queue managers concentrate the requests and pass them on using message channels. In either case, the server applications are running on z/OS or OS/390 LPARs and connect directly to the QSG (see **group attach** below).

Multiple cloned servers

In the simplest case, consider server applications that run as batch applications, perhaps coordinating updates to a database using RRS. The server application flow could be something like:

1. MQGET a request message from a request queue, in syncpoint,
2. Perform a database update, in syncpoint,
3. Construct and MQPUT a reply message to a reply to queue, in syncpoint,
4. MQCMIT the unit of recovery.

With one instance of a server application running in an LPAR, request messages would be serviced one at a time. With multiple instances of the server application running in the same LPAR more than one request can be serviced simultaneously but, with no increase in capacity this will not result in a significant improvement in the overall time to service requests. However, by making the following changes, this type of server application can very easily exploit the additional capacity and availability provided by QSGs and shared queues:

1. Configure z/OS queue managers to be members of a QSG,
2. Convert the request queue to be a shared queue,
3. Modify the server application so that it issues an MQCONN call to the QSG name, instead of to a specific queue manager name (this could be achieved through modification of the default queue manager). This is known as **group attach**.
4. Run multiple cloned instances of the server application program on different LPARs.

If a server, or queue manager instance should fail, messages on the shared request queue can still be serviced by the server application instances connected to the remaining queue managers in the QSG.

Remote queue operations are usually more expensive than operations against shared queues because of the additional costs involved in moving messages over channels. If the requester and server queue managers are in the same QSG, converting the private local reply to queue on the requester queue manager to a shared queue will avoid the channel overhead.
Replicated local queues with shared inbound channels

In the above scenario, the request queue was converted to be a shared queue. If you do not wish to convert a request queue to a shared queue, but you have your queue managers configured in a QSG, then you could maximize on the capacity of your queue managers by running server applications on each queue manager. To achieve this you need to:

▶ Define a private local request queue of the same name on each queue manager in your QSG.
▶ Define cloned server applications on each queue manager.
▶ Configure your shared inbound channels to start through the generic port so that they are load balanced started across the available queue managers in the QSG, therefore allowing you to take advantage of the benefits of running shared channels.

Assuming that the inbound channels have been distributed across the available queue managers in the QSG, incoming requests will also be distributed across the available request queues.

IMS Bridge

A particularly common type of server application makes use of the IMS Bridge. Requesting applications construct messages in a format suitable for running legacy IMS transactions, maybe using the MQI1H header to provide additional control of the transaction environment. These messages are placed on “IMS Bridge” queues. IMS Bridge queues are linked by a WebSphere MQ storage class definition to a particular IMS control region. When messages arrive on the IMS Bridge queues, they are pulled off and transmitted via Cross System Coupling Facility (XCF) to the appropriate IMS regions by special IMS Bridge tasks inside the queue manager. The transactions run inside IMS and the responses that they generate are returned to the queue manager which places them on the appropriate reply to queues.

In a queue sharing group, the IMS Bridge can be configured for either capacity or availability, on a per queue basis.

Configuring a IMS Bridge queue for capacity

▶ Define the IMS Bridge queue to be a shared queue to allow it to be accessed by internal IMS Bridge tasks on each queue manager in the QSG.
▶ Define the queue with DEFSOPT(SHARED) and SHARE to allow multiple tasks to concurrently have it open for input.
▶ Ensure that the storage class definition on each queue manager targets a different IMS control region (ideally one which resides on the same LPAR as the queue manager).

Now, each queue manager can run its own internal IMS Bridge task against the shared queue and pass messages from the shared queue to a back-end IMS in parallel with its peers. Within IMS, shared IMS message queues may be used, as required, to increase availability or solve issues relating to affinity of message processing regions to LPARs.

Configuring a IMS Bridge Queue for availability

For some applications it is not appropriate to run multiple servers against queues. This typically is the case where messages must be processed in strict sequence. The IMS Bridge can be configured for these types of applications as follows:

▶ Define the IMS Bridge queue as a shared queue, but with options DEFSOPT(NOSHARED) and NOSHARE. This restricts access to the queue, so that only a single task can open the queue for input at a time.
As above, ensure that the storage class definition on each queue manager targets a different IMS control region.

Now, only a single IMS Bridge task can run against the shared queue at a given time. But, if the queue manager where the task is running should abnormally terminate, the other queue managers in the queue sharing group will be notified and a race will occur to start another instance of the IMS Bridge task to serve the shared queue.

Queue Partitions

The issue of processing messages in strict sequence does not fit well in a shared environment where the greatest gains are made by having cloned servers working in parallel on a single queue. It is not possible to simply open the queue for MQOO_INPUT_EXCLUSIVE as that limits the servers on the queue to one, regardless of the number of LPARs available.

A number of techniques are available to resolve this dichotomy, but each relies on the requesting application logically grouping the messages either by using a unique CorrelId for each group, or by using WebSphere MQ message grouping. Each server then works on a unique 'partition' of the queue.

Partition by CorrelId

Both WebSphere MQ clustering and IGQ use the technique of partitioning by CorrelId.

In the case of clustering, the SYSTEM.CLUSTER.TRANSMIT.QUEUE is used to hold all messages outbound from the queue manager. Multiple sender channels start against the queue, and each only attempts to retrieve messages with a CorrelId that matches the channel name.

In the case of IGQ, the SYSTEM.QSG.TRANSMIT.QUEUE is used to hold messages that are destined for target queue managers within the same QSG. Multiple IGQ agents (one per queue manager) serve the queue. The agent on each queue manager retrieves messages with a CorrelId that matches the queue manager name.

This is a simple technique and works best when there are a few, well defined, logical partitions which can be individually handled by different parallel servers.

Partition by Logical Group

The requesting application constructs logical groups of messages. Each group is assigned a unique GroupId and messages are sequenced within the group. The server application retrieves messages in logical order. Firstly, it retrieves the first message in any group (applications can optionally choose to work on complete groups only, that is, ones where all the messages for the group are already available on the request queue). Next, it retrieves successive messages in the group until the last message in the group has been retrieved, when it returns to finding any 'first in group' message.

This technique is more complicated to implement from an application programming standpoint, but works well where individual requests are composed of many messages, maybe from different sources, or which have travelled by different routes.

Serialized applications

Any application working in strict sequence on a queue, and using shared queues for availability may be subject to the following race condition:

1. Application instance 1 runs on queue manager A.
2. Either queue manager A, application instance 1, or both abnormally terminate, causing a rollback of the current unit of work.

3. An external monitor (e.g. ARM) notices the abnormal termination and reschedules instance 2 of the application on queue manager B in the same queue sharing group.

4. Application instance 2 restarts and performs MQGET from the queue BEFORE the rollback of the unit of work that was created by application instance 1 has completed. Instance 2 of the application may now have retrieved a message out of sequence.

In the simple case, where there is a single server application, opening the queue for input exclusive may be sufficient. In the case of an application failure, an open queue will not be closed until the rollback has been completed. In the case of a queue manager failure, the rollback of the inflight unit of work on the shared queue is performed by peer level recovery in a different queue manager - it too only removes the open interest in the shared queue after the rollback is complete. However, if the application is using two phase commit and the unit of work is indoubt at the time of failure, both mechanisms release the queue to new openers even though there is an uncommitted unit of work, which may rollback at some future time.

A more general issue is where multiple servers are working on different queue partitions and it is not possible to lock the entire queue to an application by opening the queue for input exclusive. The solution is to use a **serialized application**.

The concept is as follows:

- The application issues an MQCONNX call to connect to the queue manager. The application passes a serialization token of its choice on the connect call.

- Any recoverable changes performed by the application against a queue (that is, MQPUT or MQGET in syncpoint) are tagged with the serialization token.

- The tag is only removed when the unit of work has either been committed or backed out through normal unit of work completion, abnormal application termination, peer level recovery, or indoubt unit of work resolution.

A second connection attempt using the same serialization token will be refused while queues are still tagged with the token.

Therefore, serialized applications can be used to protect against the race condition described above, and to preserve the sequence in which messages are retrieved from a queue. By ensuring that both instances 1 and 2 of an application use the same serialization token, instance 2 can be prevented from starting until instance 1 has either successfully completed, or has been completely recovered.

**Shared queues and clustering**

You can define shared queues to be cluster queues. Such a queue is advertised as being hosted by each queue manager in the QSG which is also a member of the cluster.

**Inbound channels versus target queues**

The table below gives the front-end and back-end availability for various configurations of inbound channels.

In order for an inbound message to reach its intended target-queue, either any one of the QSG queue managers is sufficient, or a particular queue manager is required. This requirement varies depending on the type of the target queue, and the type of the port the inbound channel uses. The term front-end availability is used to describe this requirement (i.e. what resource is required to put the message to the target queue).
Similarly, to retrieve a message from a target queue either any one of the QSG queue managers is sufficient, or a particular queue manager must be used. This requirement depends on the type of the target queue. The term back-end availability is used to describe this requirement (i.e. what resource is required to get the message from the target queue).

<table>
<thead>
<tr>
<th>No</th>
<th>Target Queue</th>
<th>Port</th>
<th>Type of chl started</th>
<th>Front end availability</th>
<th>Back end availability</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PrivLQ</td>
<td>prvP</td>
<td>Private</td>
<td>QMgr</td>
<td>QMgr</td>
<td>Same behaviour as pre WebSphere MQ for z/OS V5.2 channels</td>
</tr>
<tr>
<td>2</td>
<td>ShrdLQ</td>
<td>genP</td>
<td>Shared</td>
<td>QSG</td>
<td>QSG</td>
<td>Natural shared local queue</td>
</tr>
<tr>
<td>3</td>
<td>PrivLQ</td>
<td>genP</td>
<td>Shared</td>
<td>QSG</td>
<td>QMgr</td>
<td>Replicated private local queues</td>
</tr>
<tr>
<td>4</td>
<td>ShrdLQ</td>
<td>prvP</td>
<td>Private</td>
<td>QMgr</td>
<td>QSG</td>
<td>Front end sufficient</td>
</tr>
<tr>
<td>5</td>
<td>PrivLQ</td>
<td>grpP</td>
<td>Shared</td>
<td>QMgr</td>
<td>QMgr</td>
<td>Shared synchronization</td>
</tr>
<tr>
<td>6</td>
<td>ShrdLQ</td>
<td>grpP</td>
<td>Shared</td>
<td>QMgr</td>
<td>QSG</td>
<td>Front end sufficient, shared synchronization</td>
</tr>
</tbody>
</table>

1. A message being targeted to a private local queue, through the private port, gives the same behaviour as channels in pre QSG versions of WebSphere MQ for z/OS. The front and back end availability is dependent on the targeted queue manager being available.

2. A message being targeted to a shared local queue, through the generic port, is using shared queuing and channels in the optimum way. The front end availability is at a QSG level since any queue manager in the group can be targeted with the conversation. The back end availability is also at a QSG level since any queue manager in the group can access the message on the shared queue.

3. A message being targeted to a private local queue, through the generic port, is a valid configuration if the replicated local queue model is in use. The front end availability is at a QSG level, but the back end availability is dependent on the selected queue manager remaining available.

4. A message being targeted to a shared local queue, through the local port, would be used when the front end queue manager is sufficiently available, but the back end availability is required at a QSG level.

5. A message being targeted to a private local queue, through the shared port, is choosing to use the shared synchronization mechanisms, but availability is as for model 1 (above).

6. A message being targeted to a shared local queue, through the shared port, is choosing to use the shared synchronization mechanisms for model 4 (above).

**Shared request and reply to queues**

The example migration scenario (see “Migration Scenarios” on page 26) addresses some of the issues that you might encounter if you attempt to convert private local request and reply to queues to shared local request and reply to queues.
Shared queue triggering

For trigger types *FIRST* and *DEPTH*, shared queue triggering makes use of Coupling Facility list monitoring services to monitor for transitions in headers, known as *list headers*, in Coupling Facility list structures. Transitions occur when messages are written to shared queues that map to the Coupling Facility list structures. For trigger types *FIRST* and *DEPTH*, the current depth of a shared local application queue is monitored for a transition from 0 to 1 and from N to the value of trigger depth, respectively.

For trigger type *EVERY*, when a trigger monitor opens an initiation queue, each queue manager maintains a list of application queues that name the initiation queue.

**Triggering using private local initiation queues.**

Figure 8 illustrates the use of private local initiation queues to trigger start instances of a server application to serve a shared application queue.

![Diagram](image.png)

**Figure 8** Using private local initiation queues to trigger start server application instances

**Note:**

- For clarity, the initiation queue, trigger monitor and server application have not been shown for queue manager QM3. The process objects on all queue managers have also not been shown.
- The initiation queue and the process object can be defined as group objects.
- A lot of conditions need to be true for a trigger event to occur. The intention is to explain the general flow for trigger starting applications to serve shared queues, and NOT to explain all the conditions. A full list of the conditions can be found in the WebSphere MQ manuals.
- There are also some special cases to consider when using triggering. Again, these are described in the WebSphere MQ manuals.
The flow for trigger type *FIRST* or *DEPTH* is as follows:

a. The trigger monitor for each queue manager opens the private initiation queue for input. As part of the open request, each queue manager turns on list header monitoring for those shared local application queues that name this initiation queue.

b. An application puts a message on to a shared local application queue.

c. Assuming that the put of this message results in a transition, each queue manager generates a trigger message and places it on the associated private initiation queue. So, with trigger type *FIRST* or *DEPTH*, n trigger messages are generated, depending on the number of trigger monitors started.

d. Each trigger monitor retrieves the respective trigger message and starts an instance of the server application. Depending on the design of the server application, e.g. cloned or not, one or more instances of the server application will be started.

The flow for trigger type *EVERY* is as follows:

a. The trigger monitor for each queue manager opens the private initiation queue. As part of the open request, the queue manager builds a list of those application queues that name this initiation queue.

b. An application puts a message on to a shared local application queue.

c. The queue manager on which this put is issued determines where trigger monitors are running and nominates one of the queue managers to generate the trigger message.

d. A trigger message is put to the private initiation queue of that queue manager. So, with trigger type *EVERY*, one trigger message is generated.

e. The trigger monitor retrieves the trigger message and starts an instance of the server application.

If you do not want to trigger start an instance of the server application on one of the queue managers in the QSG then you can simply stop the trigger monitor on that queue manager.

**Note:**

For a shared queue defined with trigger type *EVERY*, when an application puts the first message on to the application queue, the queue manager that the application is connected to nominates one of the queue managers in the QSG to generate a trigger message. Only one trigger message is generated and it is put to the private initiation queue defined on the nominated queue manager. If the trigger monitor serving the initiation queue, or the nominated queue manager, should fail before processing the trigger message, then the trigger message will remain unprocessed, or be lost (since trigger messages are non-persistent). An instance of the server application will not be started and a message will be left on the application queue.

To ensure that the trigger message is processed, and an instance of the server application started, you should consider using a shared initiation queue. Provided that a trigger monitor instance has been started on more than one queue manager in the QSG, in the event of failure, one of the trigger monitors should be able to process the trigger messages on the shared initiation queue.

**Triggering shared application queues using shared local initiation queues**

The rules for trigger starting applications to serve shared application queues using a shared initiation queue are as described above when using private initiation queues. A point to note though is that for trigger type *FIRST* or *DEPTH*, if one trigger monitor instance is busy, this leaves the potential for less busy trigger monitors to process more than one trigger message from the shared initiation queue. Hence, multiple instances of the server application may be
started against a given queue manager. Note that these multiple instances are started as a result of processing multiple trigger messages. Ordinarily, for trigger type FIRST or DEPTH, if an application instance is already serving an application queue, another trigger message will not be generated by the queue manager that the application is connected to.

Limitations/Restrictions

- With WebSphere MQ for z/OS V5.2, you can only write non-persistent messages to shared queues. With WebSphere MQ for z/OS V5.3, you can write both persistent and non-persistent messages to shared queues.
- As with z/OS queue manager names, QSG names are limited to up to 4 characters.
- You can define up to 32 queue managers to be members of a single QSG.
- Coupling Facility structure names can be up to 12 characters long and must begin with an alphabetic character.
- You can define a maximum of 512 Coupling Facility list structures per sysplex.
- Each queue manager in a QSG can connect to up to 64 Coupling Facility list structures. One of these is the administration list structure so this leaves the possibility for up to 63 application list structures.
- A Coupling Facility list structure can hold data for up to 512 shared queues.
- The maximum message size for shared queues is 63KB.
- You should not define a private local queue and a shared local queue with the same name. If you do manage to do this and you attempt to open either the private or shared version of the queue, then the MQOPEN (or MQPUT1) request will fail with reason code MQRC_OBJECT_NOT_UNIQUE. You will need to delete one of the definitions before you can successfully access the remaining definition.
- An MQINQ call for the IPPROCS and OPPROCS count of a shared local queue returns the count of applications that have the queue open for input or output, respectively, on the queue manager that issued the call. To obtain the count of applications that have the shared queue open for input or output throughout the QSG, you need to issue the MQINQ call on each queue manager in the QSG and essentially ‘total up by hand’.
- The models for shared dynamic queues have a DEFTYPE of SHAREDYN. Shared dynamic queues are created and destroyed in the same way as permanent dynamic (PERMDYN) queues. That is to delete them, you have to MQCLOSE them with the MQCO_DELETE* options.
- Queue indexing is limited for shared queues. If your application needs to retrieve messages by MsgId or CorrelId, then you must specify the corresponding index type for the shared queue. If you do not define a queue index, it is still permissible to select a message by message identifier but an attempt to select a message by correlation identifier will result in a failure with return code MQRC_CORREL_ID_ERROR. Additionally, if your queue is indexed by GroupId, then you can only retrieve specific messages by specifying a GroupId. If you specify a MsgId and a GroupId, for example, then you will receive the next available message with a matching GroupId (the MsgId may well be different to that which was requested).
- You should not start a shared channel instance between two z/OS queue managers within the same QSG. If you attempt this then you may end up corrupting the shared channel status for this channel in DB2. Also, for load balanced starts, depending on which queue manager processes the start request, you may end up starting the sending end or the receiving end. This defeats the purpose of having shared channels.
The maximum batch size (though this is not policed) that can be specified for a shared channel is 2650. This is due to the maximum size of message that can be written to the \texttt{SYSTEM.QSG.CHANNEL.SYNCQ} (which is defined as a shared queue). The typical batch size in use is 50 so, as it happens, having a limit of 2650 for shared channels should not be a cause for concern.

If you set the batch size for a shared channel to a value greater than 2650, then your channel may terminate abnormally with an error message that indicates that the MCA was unable to put a message to the \texttt{SYSTEM.QSG.CHANNEL.SYNCQ} for reason 2030 (\texttt{MQRC_MSG_TOO_BIG_FOR_Q}).

Group attach is only supported for batch applications. So, your CICS and IMS transaction management systems should continue to connect to a specific, named queue manager that is on the same LPAR as the transaction management system.

There is only one IGQ agent per queue manager.

Non-persistent messages delivered by IGQ are always delivered out of syncpoint scope. IGQ does \textit{NOT} support the equivalent of the function provided by the \texttt{NPMSPEED(NORMAL)} channel attribute.

**Things to watch out for**

- Since the coupling facility is resilient to software failures, and can be configured to be resilient to hardware failures and power-outages, messages stored in the Coupling Facility are highly available.

  Non-persistent messages on private local queues are deleted following a restart of a queue manager. Non-persistent messages on shared local queues are not deleted from the CF list structures following a restart of a queue manager. While most applications will probably view this as an advantage, if your applications have a dependency on non-persistent messages being deleted following a restart of a queue manager, you will need to write a clean up job(s) to deal with these unprocessed non-persistent messages.

  Also, it may not simply be a matter of clearing the queue. If the queue also has persistent messages on it, then you may not want these to be deleted.

  One way of ensuring that you remove the unwanted non-persistent messages safely is to disable the queue for puts (this will prevent the arrival of any new messages on the queue), get all messages off the queue with the \texttt{MQGMO_SYNCPOINT_IF_PERSISTENT} option (this will ensure that all persistent messages are retrieved within syncpoint scope), and issue a backout (this will restore all persistent messages to the queue). You will of course need to ensure that the number of messages in your unit of work does not exceed the value of the \texttt{MAXSMSGS} attribute for the queue manager that your clean-up application is connected to.

  Another way is to disable the queue for puts, browse a message on the queue with the \texttt{MQGMO_LOCK} option, and destructively get the message if it is non-persistent. By repeating this process for all messages, you will be able to remove all unwanted non-persistent messages from the queue. Of course, if the clean-up application is the only application running (and there is only one instance of it) against the queue, then you do not necessarily need to use the \texttt{MQGMO_LOCK} option.

- Unlike triggering for private local queues, triggering for shared local queues works on committed messages only.

- When starting shared channels, provided that your queue managers (and their associated channel initiators) in the QSG are running, an attempt will be made to load-balance start the channel on any one of the channel initiators. It is therefore important that you define your channels as group objects so that they are available on all queue managers. An
omission of a channel definition on one queue manager does not exclude the channel
initiator for that queue manager from being the target for start channel.

There is a way of targeting a start channel request to a specified queue manager and
‘fixing’ a channel to that queue manager. However, in the unlikely event of queue manager
or channel initiator failure, since the channel is ‘fixed’ to the queue manager, no attempt is
made to restart the channel on another available queue manager. The failing queue
manager must be restarted before the channel itself can be started. Nevertheless,
fix-started channels can be useful if you have a need to only ever start an instance of a
channel on one of your QSG channel initiators, for example.

The userids that are checked when IGQ is in use are controlled by the IGQAUT and IGQUSER
queue manager attributes.

It is important to note that while you can define more than one channel, there is only one
IGQ agent per queue manager. As each channel can have an MCAUSER userid associated
with it, access to WebSphere MQ resources can be granted for each MCAUSER userid. With
IGQ, there is only one IGQUSER userid per queue manager. Hence, access to WebSphere
MQ resources can only be granted for this one userid. However, depending on your
RESLEVEL switch settings, WebSphere MQ will check one or two userids. If you set IGQAUT
to CTX (context)/ALTIGQ (alternate userid), then if RESLEVEL is set to check two
userids, authorization checks will also be performed against the userid that is in each
message that the IGQ agent processes.
Migration Considerations

Migrating your existing applications to use shared queues

In order to migrate your existing applications to use shared queues, you will need to:

1. Migrate your existing private local queues to WebSphere MQ for z/OS V5.2/V5.3 shared queues, or migrate your existing WebSphere MQ for z/OS V5.2 shared queues to WebSphere MQ for z/OS V5.3 shared queues.

   If you have messages on your existing queues, then you will also need to move these. Though, given the maximum message size and queue index restrictions that apply to shared queues (see “Limitations/Restrictions” on page 22), you may not be able to move the messages on some of your private local queues.

   The procedure for moving messages from private local queues to shared local queues is described in the WebSphere MQ for z/OS System Administration Guide.

2. Modify your existing applications (if required) to check for and handle any new reason codes that can be issued on WebSphere MQ API calls. Some new reason codes that may be issued on WebSphere MQ for z/OS are:
   
   - MQRC_STORAGE_MEDIUM_FULL (but this has the same value as MQRC_PAGESET_FULL),
   - MQRC_DB2_NOT_AVAILABLE,
   - MQRC_OBJECT_NOT_UNIQUE,
   - MQRC_CF_NOT_AVAILABLE
   - MQRC_CF_STRUCTURE_NOT_AVAILABLE
   - MQRC_WRONG_CF_LEVEL,
   - MQRC_CF_STRUC_FAILED.

   You should refer to the description of each WebSphere MQ API call, in the WebSphere MQ Application Programming Reference manual, to determine the full list of new reason codes.

3. Identify affinities (if any) that your applications may have to
   
   a. A given queue manager.
      
      For example, you may have batch applications that are coded to connect to a given queue manager or that connect to the default queue manager.

      If you are planning to run multiple cloned instances of the batch application, then you should change your applications to connect to the QSG name, or you should change the name of your default queue manager to be the name of a default QSG.

   b. Resources owned by a given queue manager.
      
      For example, your applications may be using private WebSphere MQ object definitions defined to a given queue manager.

      If you want to ensure that your applications can run on any queue manager in the QSG, then you should ensure that the WebSphere MQ object definitions required by your applications are available on all the queue managers in the QSG. One way of achieving this is to use group objects.

   c. The way resources owned by a given queue manager are accessed
      
      For example, you may have server applications that open private local request queues with the MQOO_INPUT_EXCLUSIVE option.

      If you are planning to run multiple, cloned instances of your server applications simultaneously, then you should ensure that your server applications open shared local request queues with the MQOO_INPUT_SHARED option.
Of course, if you want to ensure that only one server instance can access a shared local request queue at a time, then your server application could open the request queue with the `MQOO_INPUT_EXCLUSIVE` option. Alternatively, you need to ensure that you only ever start one instance of the server application at a time (refer to “Serialized applications” on page 17).

d. A transaction manager, like CICS or IMS, that connects to a given queue manager.

If you are planning to create a QSG with cloned queue managers, then you should ideally have ‘cloned’ transaction managers that connect to each of these queue managers. So, for example, it should be possible to run instances of your transactions on each transaction manager. Additionally, if you are using triggering, then you should have trigger monitors started on each transaction manager.

A shared queue implementation may require application changes to address any such affinities and take advantage of parallel processing.

4. Identify affinities that your channels may have to:
   - Private transmission queues
     If you wish to benefit from running shared sender channels, then you will need to ensure that your sender channel definition refers to a shared transmission queue. If you have converted your private transmission queue to be a shared transmission queue, then this will not be an issue.
   - Specific port numbers.
     If you have sender, server, or fully qualified requester channels that today specify a local port in the connection name field of the channel definition, then you will need to change this to refer to the group port, or to the generic port, if you want to take advantage of running shared inbound channels.

5. Ensure that your applications do not generate messages that are greater than the maximum length of messages for shared queues. Otherwise, puts to shared queues will clearly fail.

6. Make note of all the shared queues limitations, see “Limitations/Restrictions” on page 22.

Migration Scenarios

The following scenario illustrates how a simple query application can be converted to use queue managers in a QSG and shared local queues, in place of distributed queuing and private local queues.

Figure 9 on page 28 illustrates an simple query/response scenario. The scenario consists of a client application connected to a front-end z/OS queue manager QM1, and a server application connected to a back-end z/OS queue manager QM2. QM1 and QM2 are connected by WebSphere MQ channels. The client submits a query to the server. The server processes the query and returns a response to the client.

**Note:**
- For simplicity, the client connects directly to QM1. The client may well be connected to a remote queue manager, QM3, running on a distributed server (e.g. AIX), with QM3 being connected to QM1 via WebSphere MQ channels.
- This example assumes prior knowledge of WebSphere MQ. Refer to the WebSphere MQ manuals for specific details.

The queues defined on QM1 are:
– QUERY.REQUEST, an alias queue definition, with TARGQ(QUERY.REMOTE).
– QUERY.REMOTE, a remote queue definition, with RNAME(QUERY.REQUEST), RQMNAME(QM2), XMITQ( ).
– QM2, a private local queue definition, with USAGE(XMITQ) - the transmission queue that is used to transfer queries to queue manager QM2. Transmission queue QM2 is served by a WebSphere MQ sender channel that has a partner receiver channel on QM2.
– QUERY.RESPONSE, an alias queue definition, with TARGQ(QUERYRESP).
– QUERYRESP, a private local queue definition.

The queues defined on QM2 are:
– QUERY.REQUEST, an alias queue definition, with TARGQ(QUERYREQ).
– QUERYREQ, a private local queue definition.
– QM1, a private local queue definition with USAGE(XMITQ) - the transmission queue that is used to transfer responses to queue manager QM1. Transmission queue QM1 is served by a WebSphere MQ sender channel that has a partner receiver channel on QM1.

The query/response flow is as follows:
1. The client application puts a query on to QUERY.REQUEST (name resolution resolves this to transmission queue QM2).
   On the put request the client application sets the ReplyToQ and ReplyToQMgr fields of the MQMD to QUERY.RESPONSE and blanks, respectively. Hence, the queue manager sets the ReplyToQMgr field to QM1.
2. The sender MCA for channel C1 transmits the query to the partner receiver MCA which puts it on to QUERY.REQUEST (name resolution resolves this to QUERYREQ).
3. The back-end server application retrieves the query, processes it, generates a response, and places the response on the associated reply to queue (name resolution resolves this to transmission queue QM1).
4. The sender MCA for channel C2 transmits the response to the partner receiver MCA which puts it on to QUERY.RESPONSE (name resolution resolves this to QUERYRESP).
5. The client application retrieves the response from QUERY.RESPONSE (name resolution resolves this to QUERYRESP).
Figure 10 on page 29 illustrates the same scenario but with QM1 and QM2 in a QSG and with shared local queues. Notice that there is no longer a need to define channels between the queue managers, and that there is no longer a need to use remote queues and transmission queues. Also note that both the request and response queues have been converted to shared queues. If the client has been using a private dynamic queue for a reply to queue, then this could have been converted to a shared dynamic queue, or alternatively just left as a private dynamic queue.

The queues defined on QM1 and QM2 are:

- QUERY.REQUEST, an alias queue definition, with TARGQ(QUERYREQ). Defined as a group object.
- QUERYREQ, a shared local queue definition.
- QUERY.RESPONSE, an alias queue definition, with TARGQ(QUERYRESP). Defined as a group object.
- QUERYRESP, a shared local queue definition.
Figure 10  Simple query/response scenario using shared queues

The query/response flow is as follows:

1. The client application puts a query on to QUERY.REQUEST (name resolution resolves this to QUERYREQ).

   On the put request, the client application sets the ReplyToQ and ReplyToQMgr fields of the MQMD to QUERY.RESPONSE and blanks, respectively. Hence, the queue manager sets the ReplyToQMgr field to QM1.

2. The back-end server application retrieves the query, processes it, generates a response, and places the response on the associated reply to queue i.e. QUERY.RESPONSE at QM1. Name resolution would resolve this to transmission queue QM1 but, we no longer have transmission queue QM1. Ideally, the server application should put the response to QUERY.RESPONSE at QM2 since name resolution would resolve this to QUERYRESP. This in turn would avoid any unnecessary distributed queuing. There are a number of alternative ways of achieving this:

   a. Change the client application so that it specifies ReplyToQ QUERY.RESPONSE and ReplyToQMgr QSG1. The server would then place the response on QUERY.RESPONSE at QSG1 which would result in the response being put to shared queue QUERYRESP.

      However, this is not ideal, as many applications are probably designed to leave the ReplyToQMgr blank so as to allow the queue manager to fill in the field.

   b. Define a queue manager alias definition (which is really a remote queue definition) on QM2 that maps QM1 to QM2. I.e. Define QREMOTE(QM1) with RNAME( ), ROQNAME(QM2) and XMITQ( ). So that, when the server puts the response to QUERY.RESPONSE at QM1, this actually resolves to QUERY.RESPONSE at QM2.

      However, if there was a need to define private channels as well as to configure queue managers in a QSG, then this method would not allow you to define a transmission queue called QM1 on QM2.
c. Define a reply to queue alias definition (which is really a remote queue definition) on QM1 that maps the ReplyToQMgr to the QSG name. Define QREMOTE(QM1, QUERY.REQUEST.REPLYTOQ.ALIAS) with RNAME(QUERY.RESPONSE), RQNAME(QSG1) and XMITQ( ).

Change the client application so that it specifies ReplyToQ QUERY.REQUEST.REPLYTOQ.ALIAS and a ReplyToQMgr of blanks. So that, the reply to queue information in queries is resolved, at the time the client issues the put request, to QUERY.RESPONSE at QSG1. Now, when the server puts the response to QUERY.RESPONSE at QSG1, the response is actually put to QUERYRESP.

The client application would continue to process responses from QUERY.RESPONSE on QM1.

This method is better as it resolves the reply to queue information at source. However, the client application must specify different reply to queue information in the query, and in the request to open the reply to queue for subsequent processing. If client applications currently make use of ReplyToQ alias definitions, then this should not give much of a problem. Otherwise, it will be necessary to change the client application to avoid unnecessary distributed queuing (via private channels) between queue managers QM1 and QM2.

3. The client application retrieves the response from QUERY.RESPONSE (name resolution resolves this to QUERYRESP).

To gain full benefit of the increased capacity and availability, the server application should be enabled to run on queue manager QM1, so that instances of client applications can connect into either queue manager for service.

The above scenario demonstrated how shared queues could be used to avoid the need to define transmission queues and channels. If the front-end queue manager, QM1, shown in Figure 9 on page 28 had been running on a distributed server (e.g. an AIX server), and the back-end queue manager, QM2, on z/OS, then an increase in capacity at the back-end could be exploited as illustrated in Figure 11 on page 31.

Figure 11 on page 31 illustrates how the simple query/response scenario can exploit the back-end capacity and availability. The client application connects to QM1 on AIX and submits a query to the back-end QSG. The cloned server application on one of the QSG queue managers processes the query and returns a response.
The queues defined on QM1 are:

- QUERY.REQUEST, an alias queue definition, with TARGQ(QUERY.REMOTE).
- QUERY.REMOTE, a remote queue definition, with RNAME(QUERY.REQUEST), RQNMNAME(QSG1), XMITQ( ).
- QSG1, a private local queue definition with USAGE(XMITQ) - the transmission queue that is used to transfer queries to the QSG. This queue is served by a sender channel.
- QUERY.RESPONSE, an alias queue definition, with TARGQ(QUERYRESP).
- QUERYRESP, a private local queue definition.

Queue managers QM2, QM3 and QM4 are cloned queue managers with:

- QUERY.REQUEST, an alias queue definition, with TARGQ(QUERYREQ). This queue can be defined as a group object.
- QUERYREQ, a shared local queue definition.
- QM1, a shared transmission queue.

Channels C1 and C2:

- Sender channel C1 establishes a session with the partner receiver channel via the generic port. Hence, the inbound receiver is load-balanced started on one of the queue managers (e.g. on QM2) in the QSG.
- Sender channel C2 is a shared channel as it serves shared transmission queue QM1. This too is load-balanced started on one of the queue managers in the QSG (e.g. on QM4).
- Receiver channel C1 and sender channel C2 can be defined as group objects.
The query/response flow is as follows:

a. The client application puts a query on to QUERY.REQUEST (name resolution resolves this to transmission queue QSG1).

   The client application sets the Rep1yToQ and Rep1yToQMgr fields of the MQMD to QUERY.RESPONSE and blanks, respectively. Hence, the queue manager sets the Rep1yToQMgr field to QM1.

b. The sender MCA for channel C1 transmits the query to the partner receiver MCA which puts it on to QUERY.REQUEST (name resolution resolves this to QUERYREQ).

c. The back-end server application on one of the queue managers (e.g. QM4) retrieves the query, processes it, generates a response, and places the response on the associated reply to queue (name resolution resolves this to transmission queue QM1).

d. The sender MCA for channel C2 transmits the response to the partner receiver MCA which puts it on to QUERY.RESPONSE (name resolution resolves this to QUERYRESP).

e. The client application retrieves the response from QUERY.RESPONSE (name resolution resolves this to QUERYRESP).

Notice that in the QSG, the inbound receiver channel was started against queue manager QM2 while the outbound sender channel was started on QM4. Though they may well have both started against the same queue manager.

It is also worth noting the changes that were made on the front-end queue manager QM1:

- The RQMNAME attribute in remote queue QUERY.REMOTE was changed to name QSG1.
- The name of the transmission queue on QM1 was changed from QM2 to QSG1.
- Sender channel C1 was changed to serve transmission queue QSG1.

If you have many distributed queue managers connecting into the QSG, then you will clearly need to make such changes on all these queue managers. However, this kind of impact may not be acceptable, as some customers are already discovering. A technique that has been considered is to give the QSG the same name as the back-end queue manager that the QSG is intending to replace. So, in Figure 11., the QSG could be named QM2, and the queue managers within the QSG could be called QM2A, QM2B, and QM2C, for example. Therefore, not necessitating the changes listed above at the front-end queue manager. Additionally, if you have any batch applications that connect to QM2, they would now connect to the QSG (i.e. they benefit from using the group attach facility).

Some recommendations

- The processing of large messages in a WebSphere MQ network often necessitates special design considerations (e.g. separate channels) that should be reviewed before simply transferring an application architecture to a parallel implementation. For example, in an image archiving system, immediate message availability may not be of prime importance.

- WebSphere MQ applications designed to gather messages for later processing i.e. Batch oriented applications, can store large amounts of message data before the batch processing is started. Bearing in mind some of the limitations for shared queues, and given that this type of application is not response time critical, delivering these messages via private channels may be more appropriate.

- For real time applications, the amount of message data that may arrive during an application outage (say for update or due to a failure), and the fact that messages may be trapped, needs to be considered. A solution that uses shared queues and supports parallel servers may be more suitable in this case.
Solutions that are running successfully from response, availability, and reliability perspectives should be lower priority for migration.

If significant application changes are required to remove affinities, it may be appropriate to leave these unchanged.

Clustering (or private channels) can handle larger messages than shared queues. If the maximum size of a reply message from a legacy application cannot be guaranteed to be less than 63K, it may be better to use clustering (or private channels) than to redesign the application.

A shared queue solution assures that messages are not trapped. The availability of queue managers in a QSG allows server applications to continue serving shared queues, and removes the need to wait for a failing queue manager to be restarted.
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.
This document created or updated on November 21, 2002.

Send us your comments in one of the following ways:

- Use the online Contact us review redbook form found at:
  IBM.com/redbooks
- Send your comments in an Internet note to:
  redbook@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYJ Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

- IBM eServer™
- Redbooks(logo)™
- AIX®
- CICS®
- DB2®
- IBM®
- IMS™
- MQSeries®
- OS/390®
- Parallel Sysplex®
- Perform™
- Redbooks™
- S/390®
- SP™
- SupportPac™
- VTAM®
- WebSphere®
- z/OS™

The following terms are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both:

- Lotus®
- Word Pro®

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.