This article describes some best practices when using WebSphere MQ shared queues to provide continuous availability of messages. Topics include Coupling Facility list structure and queue size, queue depth, and data serialization.

Introduction

Much has been written about the advantages of IBM® WebSphere® MQ shared queues, in particular their ability to provide continuous availability of messages. But a number of questions have arisen during planning and implementation about the best use of shared queues and their impact on applications. While some of the issues are well known, such as the application code needed to ensure data serialization, others have not received as much attention. Recommendations in this article are based on implementations in production environments, and often include best practices for all WebSphere MQ applications. Efforts to make the infrastructure robust and highly available do little good unless the applications are also robust and highly available, and therefore application best practices are also vital to the success of shared queue implementations.

Some shared queue topics are not covered in this article. A new feature of WebSphere MQ V6 allows messages larger than 63K in shared queues, but the effect of these messages on applications and resource utilization is not included because WebSphere MQ V6 was not implemented in any production environments when the article was written. In addition, shared channels and intra-group queuing are not covered.

Example application

The example application used in this article is called PTR. Its anticipated message volume is 1,000,000 messages per day (peak rates are discussed later), with an average message size of 3.5K including the message descriptor (MQMD). The messages are spread across two queues with the volume distributed equally between them. The messages will be processed by a CICS transaction called PTRC.

Issue 1: Coupling Facility list structure and queue size

Unlike private queues, where queue size is limited by pageset size (a maximum of 4G or 64G depending on the version of WebSphere MQ), shared queues are limited by the application list...
structure size. This structure is an allocated area of storage in the coupling facility (CF). The CF is a common resource, and is used by the sysplex itself: DB2, CICS, IMS, and other WebSphere MQ structures. The structure size is set by the z/OS system administrators via IXCMIAPIU. Each WebSphere MQ queue sharing group (QSG) requires at least two structures, one for administration and one or more application structures.

In practice, most queue sharing groups are defined with the administration structure and at least two application structures, one for persistent messages and one for non-persistent messages. WebSphere MQ administrators need to know if persistent messages may be put to a shared queue. Queues that can hold a mixture of persistent and non-persistent messages must be defined in the persistent list structure.

The size limitations imposed by the CF have made some customers question whether their high volume applications are suitable for shared queues. In some cases the CF capacity does prohibit the use of shared queues, such as in a batch job where millions of messages remain on a queue for extended periods of time. But in most cases this issue is overrated, and how the queue is actually used, for example, how long messages typically remain on the queue, is more important in determining whether the use of shared queues is appropriate.

The sample application structure delivered with WebSphere MQ is set to an initial size of 266 MB and a maximum size of 512 MB. This section shows you how to calculate how many of the PTR messages can fit into the structure as defined.

Discussions with the application group should be held to determine the maximum depth expected during peak periods, the size of the messages, and the anticipated duration on the queue. Making these estimates may be difficult until message processing rates are known and real customers are using the system. But as with any capacity planning endeavor, you need a baseline.

**Estimating how much of the application structure is needed**

Assume that the application developers request enough storage to hold 20% of one day's worth of the projected number of messages on the queue, to allow for peak periods when messages may flood into the system or the pulling transactions are not able to keep up with the number of requests. This behavior is the exception, and would usually occur only during a network or application outage. The CF structure size required to hold only the PTR messages would be:

- 20% of 1M messages = 200,000
- Average message size = 3.5K = 3584
- Number of messages * size = 200,000 * 3584 = 716,800,000
- Add 30% for overhead = 931,840,000 (see note below)
- Number of 1K pages needed = 910,000 = 889 MB = Almost 1GB

**Note:** The 30% allowance for overhead is based on observations at Coupling Facility Control Code (CFCC) Level 12. If you are at a different CFCC level, this value can vary. For more information, see WebSphere MQ SupportPac MP16: Capacity Planning and Tuning for WebSphere MQ for z/OS.
The 889 MB requirement may be unrealistic; in fact it is larger than the default application structure size defined with WebSphere MQ V5.3. It may also get laughter as the response when discussed with the z/OS systems programmers. If that value is too high, what about 5% of the expected daily message traffic? That would be slightly more than one hour’s worth of messages if the 1,000,000 MPD rate is fairly even. The calculations follow:

- 5% of 1M messages = 50,000
- Average message size = 3.5K = 3584
- Number of messages * size = 50,000 * 3584 = 179,200,000
- Add 30% for overhead = 232,960,000
- Number of 1K pages needed = 227,500 = 223 MB

This value may also be unrealistic as it would take almost the entire initial structure size. If the PTR queues fill, then other applications with queues defined in the structure may be impacted.

A minimum requirement of 1% of the volume may be the only practical option:

- 1% of 1M messages = 10,000
- Average message size = 3.5K = 3584
- Number of messages * size = 10,000 * 3584 = 35,840,000
- Add 30% for overhead = 46,592,000
- Number of 1K pages needed = 45,500 = 45 MB

**Defining the shared queues**

When defining the request queue, keep the maximum messages to the number of messages used when sizing the structure. The default queue depth (MAXDEPTH) on z/OS is 999,999,999 messages. While a structure/storage media full condition would be encountered before this limit is reached, it is important to remember that unless restrained by the queue depth, it is possible for messages building up on one queue to use the entire application structure. Filling the structure will impact every other application that has queues defined in that structure.

To define the request queue to hold 1% of the anticipated volume, the definitions would look like this:

```sql
DEFINE QLOCAL('PRTA.REQUEST') CMDSCOPE(' ' ) QSGDISP(SHARED)
   REPLACE
   DEFPSIST(YES)
   BOQNAME('PRT.REQUEST.BOQ' )
   BOTHRESH(3)
   CFSTRUCT(QSG1PERS)
   HARENB0
   MAXDEPTH(5600)
   QDEPTHI(80)
   QDEPTHLO(40)
   QDPHEIV(ENABLED)
   QDPLOEV(ENABLED)
   QDPMAEXV(DISABLED)

DEFINE QLOCAL('PRTB.REQUEST') CMDSCOPE(' ' ) QSGDISP(SHARED)
   REPLACE
   DEFPSIST(YES)
   BOQNAME('PRT.REQUEST.BOQ' )
   BOTHRESH(3)
   CFSTRUCT(QSG1PERS)
```
Other queue definition considerations

In the example above, the queues defined will take a maximum of 45 MB of the structure. If there are other application queues that need to be defined in the structure, do the same sizing exercise for them.

Another consideration when adding queues to an application structure is overlapping peak times. For example, if there are two queues that may need up to 45MB of storage and the structure is only 64MB, you may never run into a structure or queue full situation if their peak processing times do not overlap, or if the queue depth never approaches the maximum.

Issue 2: Queue depth

Even with high message volumes, shared queues can work well with most applications. The real limiting factor is not how many messages pass through a queue, but how deep a queue normally gets -- which should be a very low number. For high-volume applications, the server processes need to be highly available and able to pull messages from the queue rapidly.

When you evaluate queue depth requirements, message processing rates are just as important as peak message volumes. You can determine message volumes and processing rates by examining the System Management Facility (SMF) queue accounting records (SMF116). If the records are not available, and many customers do not turn on that level of reporting in production environments, you can estimate the volume and rates based on the statistics records (SMF115), audit logs from the application, average number of database actions driven by the messages, or other application indicators.

Another estimating tool is the RESET QSTATS command. If issued at regular intervals during a normal production or test cycle, you can capture and track the number of messages that have been processed during that interval and the queue high depth.

If this is a new application or if message rates cannot be calculated, you must estimate and derive message volume and processing rates from the SMF data produced during testing. When doing these estimates, do not assume an even distribution of messages throughout the day. Many types of requests are much higher during predictable peak periods, such as during normal business hours. A daily cycle can have multiple peaks, as in the financial industry, where there are often major peaks just after markets open and just before they close. Other peak periods may not depend on daily cycles, but may instead be related to month end, year end, pre-holiday periods, or less predictable volume spikes such as promotional activities or failover.
Discuss the anticipated distribution of message traffic with your application development team, and then test and evaluate peak processing rates during pre-production testing. Then be sure to monitor message traffic, especially on shared queues, after the application goes into production.

As an example of daily peak message rates, if we assume that 75% of the anticipated 1,000,000 messages comes in during an eight-hour period, the message traffic would average 26 messages arriving on the request queue per second during that period. If the peak period is shorter, or if a higher percentage of the messages comes in during the peak period, messages per second will of course increase. To keep the queue depth low, the server process would have to run at a higher rate.

**Effect of processing rates on queue depth**

In the example below, the message putting rate is twice as fast as the getting rate, and you can see the queue depth growing rapidly. The 2:1 rate difference is for simplicity -- every application will vary and the rates usually change over time. The pattern of the serving application being slower than the requesting application is normal because of the work the typical server application does – read tables, make calculations, update rows, and so on.

During pre-production testing, you should determine both the put and get rates. While these rates may not match what you see in production, they should give you a starting point to determine how many copies of the serving application will have to run to keep up with the requesting applications.

**Figure 1**

![Diagram of queue depth](image)

<table>
<thead>
<tr>
<th>Requesting Application</th>
<th>Processing Messages on</th>
<th>Serving Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request application averages 20 request message puts per second.</td>
<td>Duration</td>
<td>Queue</td>
</tr>
<tr>
<td>10 Min</td>
<td>600</td>
<td></td>
</tr>
<tr>
<td>30 Min</td>
<td>18,000</td>
<td></td>
</tr>
<tr>
<td>60 Min</td>
<td>36,000</td>
<td></td>
</tr>
<tr>
<td>Serving application processes an average of 10 messages per second.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Obviously, if there is an 0.5 second response time service level agreement in place for this application, it would be quickly violated. And if the queue depth had been set to the 5000 calculated earlier, it would have been exceeded within the first half an hour. There are many techniques to remedy this situation, some of the more common are described below.
If there are no inhibitors to running multiple copies of the server application, the simplest solution to the server application processing messages at a slower rate than the requesting application is to run multiple copies of the server program. In this simple example, just two copies of the server application are required. In practice, multiple copies of an application usually do not run at the same rate, and application testing and evaluation is required to establish the real rates.

In our simple example, since the server process is a CICS transaction, simply having multiple copies of the server transaction running in multiple CICS regions would keep the queue depth very low. Similar techniques can be used for IMS transactions. Other processes (batch, application server, etc.) require more thought, but there are examples of using shared queues with nearly every process model in use.

Having enough processing transactions, enough programs doing the MQGETs, allows this high-volume queue to be shared without violating the SLA or exceeding the capacity.

**Issue 3: Data serialization**

Another common consideration is data serialization. If the messages must be processed in a strict sequence, then there can be only one putting and one getting application, which usually requires queues to be sized to hold more messages. For more information on application coding to guarantee serialization, see the IBM Redbook *Parallel Sysplex Application Considerations (SG24-6523)*.

Serialization requirements and their impact are not unique to WebSphere MQ processing. But it is often identified as an issue, especially when file-based batch processes are being replaced with real-time processes that can be initiated from multiple places. From a data availability standpoint, serialization requirements should be eliminated when possible. If elimination is not possible, or not probable in the near term, you can use the simple technique of targeted serialization to help manage serialization requirements.
**Targeted serialization**

An easy way to reduce the impact of a data serialization requirement is to see if it applies to a logical subset of the data. For example, in many cases, data serialization is required only within a specific account or item number. If this is the case, the requesting application can sort the messages into queues that hold defined ranges of the serialized data. Each request queue would have only one serving application to ensure serialization, but the total workload is not confined to one server process. As in the multiple servers example shown above, queue build-up may be avoided.

An example of targeted serialization is shown below:

**Figure 3**

This will require the application to open the multiple queues, apply the distribution logic, and so on. In addition, as message volume grows, the application may have to be altered to open more queues.

If the application is in production and changes are not possible, or if messages are flowing from a number of sources, using WebSphere Message Broker or a custom application can provide similar capability. The overhead of adding this processing is often far less than the impact of one high-volume serialized process. An example is shown below:
The advantage of this model is that as message volume grows, adding new queues does not require changes to the requesting applications. Also, if the distribution pattern changes over time, it is simpler to alter the criteria used to distribute the messages in a rules-driven engine than in distributed applications.

**Issue 4: Message affinities**

Some serialization requirements are driven not by a requirement that messages must be processed sequentially, but by affinities between individual messages. Often the messages are logically associated, as when multiple messages make up an application transaction, or ordered based on type, as when a change or cancellation can be processed only after an initial order has been accepted.

From an application coding standpoint, it is often easier to require serialization of all messages or records than to code for the affinities. This requirement may be a carryover from file-oriented batch processing, where it was simple enough to ensure that information came into the process in a predefined sequence.

Message affinities can be tight or loose. Tight affinities are when several messages must be processed together to complete a logical transaction, such as in a purchase order, which is typically made up of header information, one or more line items, and trailer information. Loose affinities are normally modification requests that can be processed only after the original transaction has been completed. For example, the modification of an order or the cancellation of a credit card charge can be processed only after the original transaction is complete.

**Using application logic to handle tight message affinities**

For tight message affinities, WebSphere MQ provides message grouping to logically tie multiple messages together as one logical transaction. From an application perspective, the putting application must:
1. Identify the messages via the MQMD GroupID field. You can do this either by setting the GroupID field from application information or letting WebSphere MQ generate the GroupID.
2. Keep the messages within a logical order (MQPMO_LOGICAL_ORDER).
3. Identify the logical position of each message within the group.

The getting application must:

1. Specify that messages are to be retrieved in logical order (MQGMO_LOGICAL_ORDER).
2. Not process messages until all messages in the group are available (MQGMO_ALL_MSGS_AVAILABLE).

**Using application logic to handle out-of-order conditions on loose message affinities**

In some applications, only small subsets of messages require serialization. The typical example is when an order cancellation must arrive after the initial order. If multiple servers are getting from a shared queue, a message may be retrieved out of sequence. Application designers may decide to include logic to handle those out-of-order situations, as in the following pseudocode example:

```plaintext
IF a message cannot be processed because it is out of sequence
   (e.g. an order change precedes an order create request):
   THEN DO
      DELAY for a small time (typically a few milliseconds)
      ROLLBACK the transaction (increments MQMD.BackoutCount)
   END
```

We assume MQGET processing already includes a check for Backout Count (see backout discussion below). The DELAY allows time for the message preceding the one being backed out to arrive. This technique works only if a very small subset (typically pairs) of messages require ordering.

**Issue 5: Running out of room and other errors**

Even private queues fill up from time to time, usually when there are extended network outages. Some customers have found that shared queues may be more prone to filling, because of the more restrictive size limits imposed by the CF. Putting applications should interpret and take appropriate action on the following return codes:

- MQRC_Q_FULL (2053, X'805') Queue already contains maximum number of messages
- MQRC_STORAGE_MEDIUM_FULL (2192, X'890') External storage medium is full
   (also known as MQRC_PAGESET_FULL)
- MQRC_CF_STRUC_FAILED (2373, X'945') Coupling-facility structure failed.
- MQRC_CF_STRUC_IN_USE (2346, X'92A') Coupling-facility structure in use.

The appropriate action depends on the application and environment. Some applications actions include:

**Stopping the putting application**

This is often the easiest to do programmatically, though it effectively makes the application unavailable.
Put inhibiting the shared queue
This action may be used by WebSphere MQ monitoring tools that detect the queue or media full condition prior to the application receiving the return code.

Throttling the putting applications
Essentially slowing them down by putting in a loop or wait interval. This technique can be effective for batch processing, though it requires application modifications and testing. If the putting application is a Message Channel Agent and you are on WebSphere MQ V5.3.1, throttling must be done via a receiver exit or in the remote programs. In WebSphere MQ V6, the retry parameter has been added to receiver channels on z/OS. This parameter controls the number of times a message will be retried before determining that a message cannot be delivered to its target queue.

When the shared queue fills, putting messages to a local queue
While this implies that the application "knows" it is using shared queues, it can be a good technique for any situation where a queue might fill. The application could even use a defined backout queue for this. This option requires some kind of recovery capability to process the messages or move them to the shared queue when it becomes available. A common technique is to apply the WebSphere MQ DLQ header to each message so routed, so that one of the generic DLQ handlers can be used to move the message to the shared queue without having to write a processor for it.

Issue 6: Message backouts
In the sample queue definition, backout information was included as highlighted below.

```
DEFINE QLOCAL('PRT.REQUEST') CMDSCOPE(' ' ) QSGDISP(SHARED)
REPLACE
DEFPSIST(YES)
BOQNAME('PRT.REQUEST.BOQ' )
BOTHRESH(3)
CFSTRUCT(MQM1DIVS)
HARDENBO
MAXDEPTH(10000)
QDEPHTHI(80)
QDEPHTLO(40)
QDPHIEV(ENABLED)
QDPLOEV(ENABLED)
QDPMAXEV(DISABLED)
```

While the MQ administrator sets the backout parameters, they are really used by well-behaved WebSphere MQ applications to remove potential "poisoned" messages from the queue, or messages that cannot be processed due to other issues. This is important for both shared queues and private queues, though the symptoms may differ. The parameters are:

**BOQNAME('PRT.REQUEST.BOQ')**
Name of queue to which applications should write messages that have been backed out.

**BOTHRESH (3)**
Number of processing attempts for each message.

**HARDENBO**
Harden the backout counter to disk when syncpoint is done.

If a message on a private queue cannot be processed and a rollback is issued, that message goes back to the top of the queue. The next MQGET for that queue will pick the problem message up...
and try to process it again. If that attempt fails and the transaction is rolled back, the message once again goes to the top of the queue. WebSphere MQ maintains a backout count that can notify an application when it is looping on the same message. If the application ignores the backout counter, the first indication of a poison message is that a queue depth begins rising and the getting processes continues to run.

On a shared queue, when there are multiple server processes running, queue depth may not increase dramatically, because when the poison message is being processed, other messages from the queue are being picked up and processed. There is just a rogue message, which keeps getting processed over and over again – eating up CPU cycles, but not stopping work.

In some cases, backing a message out and allowing a few processing attempts is reasonable. For example, if a database row or table is unavailable, it may become available the next time the message is processed.

A well-behaved application will check the backout count on every message read, and if it is non-zero, compare it to the backout threshold defined on the queue. If the count is greater than the threshold, the message should be written to the queue specified in the BOQNAME parameter and committed. Often a Dead Letter Header (MQDLH) is attached to the message to indicate why the message was written to the backout queue.

**Recommendations**

1. Use one backout queue per application, not per queue. One backout queue can be used for multiple request queues.
2. Do not use the queue-manager-defined dead letter queue as the backout queue, because the contents of the backout queue are usually driven by the application. The dead letter queue should be the backout queue of last resort.
3. Do not make the backout queue a shared queue, because backout queues are usually processed in a batch, and therefore any queue depth build-up on these queues may impact other applications using the same CF structure. Therefore, if you are defining a backout queue for a shared queue, you need an instance of the backout queue on every queue manager in the QSG that can host the server application.
4. If a DLH is applied to backout messages, use one of the many dead letter queue handlers to determine if a message should be restored to the original queue. Many customers use the standard handler, or tailor it to meet the needs of the application.

**Issue 7: Mixed shared and private queue options**

Some applications have very high availability requirements for certain classes of service. Premier customers or high economic value requests may need the continuous availability provided by shared queues, while other message traffic does not warrant that level of service. As in the “targeted serialization” example shown above, you can use the application programs or a brokering tool to route the selected messages to the shared queues and the others to private queues (probably defined as part of a cluster).
Issue 8: Other high availability options

Sometimes a shared queue is not the answer to the availability needs. For those applications, WebSphere MQ clustering can often provide a higher level of availability than a point-to-point implementation. In addition, you can use standard failover techniques for restarting a queue manager in place, or on a different LPAR.

Acknowledgements

As my great-grandmother always said, "Many hands make light work." Many people contributed experiences, helped refine ideas, checked my math, corrected typos, and repaired erratic grammar. Special thanks to Emir Garza, Bob Herbison, Mark Taylor, and Steve Zehner.
Related topics

- **Parallel Sysplex Application Considerations (SG24-6523)**
  This IBM Redbook introduces a top-down architectural mindset that extends considerations of IBM z/OS Parallel Sysplex to the application level, and provides a broad understanding of the application development and migration considerations for IMS, DB2, Transactional VSAM, CICS, and WebSphere MQ applications.

- **WebSphere MQ in a z/OS Parallel Sysplex Environment (SG24-6864)**
  This IBM Redbook looks at the latest enhancements to WebSphere MQ for z/OS and shows how you can use the z/OS Parallel Sysplex to improve throughput and availability of your message-driven applications. It helps you configure and customize your system to use shared queues in a high-availability environment and to migrate from earlier releases.

- **WebSphere MQ SupportPac MP16: Capacity Planning and Tuning for WebSphere MQ for z/OS**
  On the target page, go to Item 2, enter “MP16,” and press Enter.

- **WebSphere MQ SupportPac MP1D: WebSphere MQ for z/OS V5.3 and V5.3.1 Performance Report**
  On the target page, go to Item 2, enter “MP1D,” and press Enter.

- **WebSphere MQ product page**
  Product descriptions, product news, training information, support information, and more.

- **WebSphere MQ documentation library**
  WebSphere MQ manuals in PDF format.

- **WebSphere MQ V6 Information Center**
  A single Eclipse-based Web interface for all WebSphere MQ V6 documentation.

© Copyright IBM Corporation 2005

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)