bmcsoftware

The MQ Insider

Focusing on MQSeries Technical Matters

bmcsoftware

VOLUME 1 ISSUE 1 – SPRING 1999

Welcome to the first issue of The MQ Insider - a quarterly technical newsletter from BMC Software that is committed to bringing you current and in-depth technical information regarding MQSeries. Whether you are interested in better administration of your MQ networks or looking to improve your MQSeries programming, our newsletter will provide a little inside help.

Written by industry experts, The MQ Insider will examine a whole range of MQSeries-related topics, and will not be restricted to the performance and high availability issues for which BMC Software is renowned. For this first issue we are focusing on MQSeries Clustering, one of the key capabilities of Version 5.1, and Programming MQSeries with ActiveX.

BMC Software is producing this newsletter for several reasons.

- We believe that MQSeries is an important technology for our customers and BMC is committed to supporting it.
- Enterprise Application Integration (EAI) is a key industry trend and many customers are using the MQSeries family to achieve this goal. BMC Software offers Application Service Assurance (ASA) for integrated infrastructures, improving the availability, performance and recoverability of business critical applications.
- We hope that by providing this newsletter customers will learn more about BMC's expertise in MQSeries.

The online version will be available to all subscribers as a PDF download from MessageQ.Com. Printed copies of the newsletter are also available. To subscribe to The MQ Insider please use the enclosed faxback form or register online at MessageQ.Com (www.messageq.com/mqinsider).

I hope you enjoy the first issue.

Phil Griston
European Solutions Manager, BMC Software

In focus this issue:

BMC News Headline

BMC Acquires New Dimension Software.

The acquisition complements BMC Software's pending merger with Boole & Babbage. As a result of these strategic acquisitions, BMC aims to accelerate the delivery of solutions that significantly reduce the complexity of managing large enterprises.

To subscribe to The MQ Insider visit:

Exploring MQSeries Clusters and Dynamic Workload Balancing

Much has been written about MQSeries, its capabilities and its shortfalls. With the recent announcement of MQSeries Version 5.1, IBM has answered the critics and done it in a big way. Probably the two most significant changes made to MQSeries in this release are the addition of the MQSeries Explorer (Windows NT Only) and MQSeries Clusters with Dynamic Workload Balancing on version 5 platforms[1] and on OS/390 with MQSeries Version 2.1.

MQSeries Explorer uses the Microsoft Management Console, MMC, to allow administrators and programmers the ability to configure, monitor and view their MQSeries environments with a tool much like Windows Explorer. In the past, users had to rely on the "runmqsc" utility, self written shell/command scripts or third party tools in order to work with their queue managers and the objects they control. The monitoring and management of MQSeries networks will be covered in depth in a future issue, the remainder of this article will deal primarily with the second significant change made for V5.1, MQSeries Clusters.

Clusters are not a new concept to IT. We have had High Availability Cluster Multi-Processing (HACMP) clusters for some time now, and most IT shops try to design most, if not all, of their systems/applications to provide "High Availability" to their users so that no server or service is unavailable at any given time. The Distributed Computing Environment (DCE) accomplishes this same availability using the notion of "Cells" and a naming service. A DCE service can be located on multiple host systems within a DCE cell. Client applications can access or bind to these services in one of three ways: automatic, implicit or explicit. Automatic allows the Remote Procedure Call (RPC) library to find the host and service to handle your request. Implicit will allow the client application some control over where the service is selected. Explicit allows the client application full control over the host and service for each RPC call made.

You're probably wondering what this has to do with MQSeries clusters. In MQSeries, a cluster is a set of MQSeries queue managers associated in a way that allows them to share resources and also be aware of each other's existence. They could represent a geographical region, different branches or departments of a company or simply a logical grouping of services. Clusters, like queue managers, should have unique names within an enterprise. This will allow for a queue manager to belong to more than one cluster. Within each cluster you will have a repository queue manager. This queue manager will hold a full set of information pertaining to every other queue manager in the cluster. This information contains the queue managers, the cluster queues that they host and the channels used to pass data to and from these queue managers and repositories. This information is stored in the SYSTEM.CLUSTER.REPOSITORY. QUEUE.

Before you begin creating your clusters, it is a good idea to layout your server/queue manager topology and where you want your services/applications to reside. When laying out your architecture, if 100% availability is a necessity, try to design your services/applications so that they reside on different servers. These servers should also reside on different network segments that use different network routers. This will help eliminate outages that could be caused by network failures, and allow you to maximize the benefits of your MQSeries clusters.

If you already have your queue managers defined and your topology in place, you can create a cluster using MQSeries Explorer and the Cluster Wizard. The wizard will take you through several screens that will ask you questions about the cluster you are creating and the queue managers to be included. You will be prompted for channel names needed to connect this queue manager to the cluster.

Now that we have created a cluster, we can begin to create queues to be used by the cluster queue managers. It is important to remember that cluster queues are only beneficial to applications sending data "to" these queues. The applications that will be retrieving the messages from these queues will still need to connect to the queue manager that actually hosts the queue in order to remove messages from the queue.

[1] Version 5 platforms include, AIX, HP-UX, OS/2, Sun Solaris and Windows NT.

Since we now have repositories that hold information about queues and queue managers that belong to a cluster, remote queues and channel definitions, other than the CLUSSDR and CLUSRCVR channels, are no longer needed. However, remote queue usage is still needed to access other MQSeries platforms that do not support clusters. Now when you create a queue and indicate its participation in a cluster, the local queue manager will send data to a repository queue manager so this queue is advertised throughout the cluster. Any application connected to any queue manager belonging to the cluster can send (issue MQPUT's or MQPUT1's) to that queue without any further intervention from the MQSeries administrator.

What we have just created is a cluster and a queue that is shared within that cluster. See Figure 1.

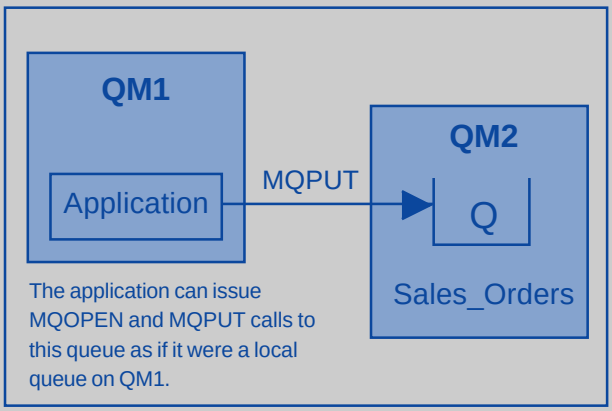

Figure 1. Sharing a queue within a cluster.

Let's look at some of the new commands that administrators can use to view information pertaining to their MQSeries clusters. Although we have the new features provided by MQSeries Explorer, you still have the option to issue commands directly to the queue manager using the runmqsc interface.

DIS CLUSQMGR – This command, when issued from a repository queue manager, shows information pertaining to every queue manager belonging to that cluster. When issued from a queue manager not having a full repository, the output will only show the queue managers that have full repositories and every queue manager you have sent messages. When issuing this command, a specific queue manager can be requested, or an asterisk * can be used which will list all queue managers that are known to this platform and the cluster. Like all other MQSeries commands issued with the ALL suffix, this will show all attributes of all objects of that type.

DIS CLUSQMGR(*) ALL

SUSPEND QMGR – This will temporarily suspend the queue manager's participation in the cluster. A suspension notification is sent to all other queue managers in the cluster and, in this way, the workload balancing function will not attempt to send a message to the suspended queue manager. This feature is useful for applying maintenance to your servers or for installing new releases of application or system software.

SUSPEND QMGR CLUSTER(*clustername*)

RESUME QMGR – This will notify the cluster repositories that the queue manager is now available for work. The repository queue managers will, in turn, notify all of the queue managers that have shown an interest in this queue manager or one of its resources.

RESUME QMGR CLUSTER(*clustername*)

REFRESH CLUSTER – Not needed during normal running, this instructs the queue manager to drop all known information it has about the specified cluster and to re-advertise its presence and resources to the cluster. The queue manager will also request updates about resources within the cluster in which it has interests.

REFRESH CLUSTER(*clustername*)

RESET CLUSTER – This will forcibly remove a queue manager from a cluster, and is the only way to remove auto-defined cluster sender channels. It must be issued on a repository queue manager.

```
RESET CLUSTER(clustername)
QMNAME(qmgrname)
ACTION(FORCEREMOVE)
```

We now know that we can share queues between queue managers in a cluster. Let's talk about the same queue existing on multiple queue managers within a cluster.

Say you have a queue manager that has a local queue called SALES_ORDERS, and it is responsible for receiving sales orders from your customer interface applications, processing the orders and then forwarding them onto a data server for storage. This is a classic three-tier architecture. This queue and service have been processing and posting your sales orders for a few months now, but recently your business has increased and you now have more orders coming in every day. You find yourself in need of additional resources to help keep up with the

traffic. You have other servers in your enterprise that are not as heavily used as the one hosting the SALES_ORDERS queue, so you decide to define another queue called SALES_ORDERS on another queue manager in your cluster. Applications now have a choice, unknown to them, as to which server can process their sales orders.

When applications open a queue that exists on more than one queue manager within a cluster, they are directed to a queue by a dynamic workload algorithm automatically invoked by the queue manager. This algorithm provides for smart workload distribution. This means that if, for some reason, one of the possible target queue managers hosting the queue is unavailable, the message will not be sent to that destination. This algorithm uses a round robin approach to sending messages to their target destinations. Be aware that some applications in your enterprise might send messages that need to be correlated in some fashion. These messages are said to have affinities to one another. In these circumstances, application programmers can specify that they would like to "bind" to one instance of the queue during the course of their MQPUT processing. This binding request is accomplished at the time the queue is opened using a new open option called MQOO_BIND_ON_OPEN. Figure 2 is an example of setting the option for the MQOPEN call.

```
MQLONG Options;

Options = MQOO_OUTPUT +
MQOO_FAIL_IF_QUIESCING +
MQOO_BIND_ON_OPEN;

MQOPEN (Hconn,
          &ObjDesc,
          Options,
          &Hobj,
         &CompCode,
           &Reason);
```

Figure 2. Setting the option for MQOPEN.

If it is determined that the target queue is local to the queue manager, and if it is PUT(ENABLED) and has the ability to accept the message, this queue will always be selected.

Cluster queue managers are constantly keeping each other informed about changes in their environments. Say for instance that the same queue exists on two servers within a cluster. If an administrator sets the queue to PUT(DISABLED), this information is propagated to the repository queue managers and then to all other queue managers that have shown an interest in this queue. In this way, the workload balancing mechanism will always have up-to-date information about the environment in which it is distributing messages.

The dynamic workload algorithm will more than likely satisfy the majority of scenarios for most users. However, in not wanting to constrain users with very specific routing needs, or users requiring more control over the distribution of messages within their enterprises, MQSeries V5.1 provides for a work load exit. The exit is associated with a queue manager and can be designed to further enhance your messaging environment.

Workload exits are written much in the same way as channel exits are written today. These exits need to have a standard entry point, and will return the destination chosen based on the needs of your applications or the outcome of your routing rules or message interrogation.

As you can see from the snippet of code in figure 3, the MQWXP structure is made available for the exit code to use. Within this structure are pointers to several other structures as well as a pointer to the message descriptor (MQMD) of the message being routed. Below is a list of the structure names and a brief description of the information they contain.

- MQWXP – Exit parameter structure. This is the main structure passed to the exit. It contains specific information about the exit, why it was invoked, and pointers to all other data and structures sent to the exit.
- MQWDR – Destination records. A pointer to an array of these structures is sent. There will be an entry for every possible destination of this message. This structure contains information about the destination queue manager: the type of queue manager, its availability to receive messages, the state of the channel connecting it to the local queue manager and information about the cluster itself.
- MQWQR – Queue record structure. This contains information about the individual queues to which the message could be sent, such as queue type, the default binding attribute, the persistence indicator, default message priority, and whether or not put operations are allowed.

```
#include <stdio.h>
#include <stdlib.h>
#include <cmqc.h>
#include <cmqxc.h>
#include <cmqcfc.h>

void MQENTRY MQdwle(MQWXP
  *WLEparms)
{
your code
…
return;
}
```

Figure 3. Making the MQWXP structure available for the exit code.

- MQWCR – Cluster record structure. This contains information about the cluster that hosts the destination queue. In the event that this queue belongs to more than one cluster, there will be one of these records for each cluster. Also included in this structure is information about the type of queue manager that hosts this instance of the queue and how the cluster channels are defined, manually or automatically.
- MQCD – Channel definition structure. This contains very specific information about the channel this message will use to reach its destination. Channel name, transport type, retry counts and intervals, batch size and connection name are just a few of data fields made available to the workload exit.

Along with these structures, a pointer to the complete message descriptor is made available for the exit, along with some or all of the actual message data. The amount of message data sent to the exit is determined by the queue manager attribute *ClusterWorkloadLength*. The length of message data passed to the exit as well as the actual full length of the message itself can also be used in determining its destination.

The exit can be invoked for several reasons. These reasons are passed to the exit so the appropriate activity can take place at the appropriate time. For instance, at queue manager startup when the exit is loaded, the reason MQXR_INIT (initialize) is sent. In this way, if any outside routing information or rules are stored in files or databases, it can be accessed and cached for later use when making the destination choice. In order to keep this information dynamic, it would be a good idea to periodically re-cache this information throughout the day in order to pick up any new information. The exit is also aware when an MQOPEN call is made, as well as the MQPUT and MQPUT1 calls.

The goal of the exit is to populate the *DestinationChosen* field of the MQWXP exit parameter structure, based on message characteristics, data content, message size or the current state of the target environments. The actual reasoning behind your decision will be based on the needs of your business requirements. Once your destination choice has been determined, whether based on routing rules specific to the message or simply the availability of the target queue, the exit will populate the *DestinationChosen* field of the MQWXP structure with the destination number.

As you can see, MQSeries clusters and the dynamic workload features which are now possible will give architects the opportunity to design their systems for 100% availability, and still allow them to easily accommodate system maintenance and unplanned outages. Version 5.1 of MQSeries truly provides the ability for fault tolerant, asynchronous messaging systems and their components.

END

DID YOU KNOW?

BMC Software is the 7th largest software-only vendor in the world.

Programming with MQSeries in ActiveX

This article assumes a basic understanding of MQSeries programming, ActiveX, Visual Basic and JavaScript.

Recently, IBM introduced a set of ActiveX Automation Classes (MQAX) for MQSeries. The classes give programs that are written for the Component Object Model (COM), Microsoft's object-based programming model, full access to the features and functionality of the MQSeries API as well as interconnectivity to other platforms and environments. COM allows MQSeries services to be accessed from:

- Internet Explorer
- Active Server Pages (ASP) executing within Internet Information Server (IIS)
- Components executing under the control of Microsoft Transaction Server (MTS)
- Applications written in Visual Basic, Visual C++ or J++.

COM exists on Unix and MVS, but the ActiveX Automation Classes are only available for Windows NT, 95 and 98.

Properties, *methods*, *events*, and *exceptions* define ActiveX class behaviors. Properties enable read or write access to underlying data members. Methods perform synchronous actions that return results. Events are asynchronous actions, such as a mouse click, that an object can respond to. Exceptions are unexpected events that disrupt normal processing.

As ActiveX objects go out of scope, they are automatically destroyed and held resources are recovered, e.g. the connection to a queue manager, handles to queues and allocated memory. This simplifies the development of MQSeries applications.

The MQAX architecture will not surprise knowledgeable MQSeries users. The following list briefly describes all MQAX classes:

MQQueueManager – configures, creates and manages a connection to a queue manager.

MQQueue – manages access to a queue.

MQMessage – configures a message descriptor (MQMD) for sending or receiving a message and provides a buffer for application data.

MQPutMessageOptions – controls the options for sending messages.

MQGetMessageOptions – controls the options for receiving messages.

MQDistributionListItem – manipulates elements of an Object record (MQOR), Put Message record (MQPMR) and Response record (MQRR). These structures are used in distribution lists.

MQDistributionList – a collection of local, remote or alias queues, which are represented as MQDistributionListItem instances.

MQSession – a root class that contains error status.

Through MQSession class properties, users can derive CompletionCode, ReasonCode and ReasonName values. Another property tells MQAX classes when to throw exceptions – for CompletionCode values of MQCC_WARNING or MQCC_FAILED.

MQSession methods clear CompletionCode, ReasonCode and ReasonName values and return references to other MQAX classes.

The Visual Basic example in Figure 1 creates an MQSession object, sets the exception level to MQCC_FAILED and then connects to the default queue manager. A reference to a MQQueueManager object is returned and results are checked.

```
' "New" creates an object instance
Dim ssn As New MQSession
' Object reference created w/o New
Dim qm As MQQueueManager

ssn.ExceptionThreshold = MQCC_FAILED

Set qm = ssn.AccessQueueManager("")

MsgBox Str$(ssn.CompletionCode)  & " " & ssn.ReasonName
```

Figure 1. Using MQSession to check error status.

An MQSession object provides a simple way to check error conditions, but only one instance is created per pro-

cess. However, MQAX classes support a free threading model and will be loaded into a multi-threaded apartment[1]. MQSession objects return accurate results in single-threaded GUI applications, e.g. ones written in Visual Basic, but are dangerous to use in multi-threaded applications, including server applications hosted by Microsoft Internet Information Server or Transaction Server.

Other MQAX classes also include CompletionCode, ReasonCode or ReasonName properties. After a method call on an MQAX object, e.g. MQQueue, programmers should check the error properties associated with that object. Below, for example, we connect to a queue manager and then check error status using the MQQueueManager object:

```
qm.Connect()
MsgBox Str$(qm.CompletionCode) _
   & " " & qm.ReasonName
```

In Figure 2, we send a message from an Active Server Page written in JavaScript. An MQQueueManager object is created to connect to queue *pp.LQ1* associated with queue manager *ppDEFQM*. The queue is opened using a value of MQOO_OUTPUT. Then we create an MQMessage object, assign some message data, send the message and send a result string back to the client browser.

```
<%@ Language="javascript" %>
<HTML><HEAD><BODY>
<%
qMgr = Server.CreateObject("MQAX200.MQQueueManager");
putQ = qMgr.AccessQueue("pp.LQ1", MQOO_OUTPUT,
   "ppDEFQM");
pMsg = Server.CreateObject("MQAX200.MQMessage");
pMsg.MessageData = "MQAX/JavaScript/ASP Message";
putQ.Put(pMsg); %>
</BODY></HTML>
```

Figure 2. Sending a message from ASP.

Figure 2 uses the MessageData property to set a character string into the message. The MQMessage class also provides numerous methods for setting different types of data into and reading data out of a message buffer. Some methods also maintain a cursor into the message buffer making it fairly easy to write or read incremental amounts of data.

Figure 2 did not use put message options, but it could have done. For example, we can tell MQSeries to fail if the queue manager is quiescing. To do this, create an MQPutMessageOptions object, set its Options property to MQPMO_FAIL_IF_ QUIESCING and include the MQPutMessage Options object as part of the Put operation. This is demonstrated in Figure 3.

```
<HTML><HEAD><BODY>
<%
// all code prior to putQ.Put(pMsg)
// in the previous example
mPMO = Server.CreateObject
   ("MQAX200.MQPutMessageOptions");
mPMO = MQPMO_FAIL_IF_QUIESCING;
putQ.Put (pMsg, mPMO);
%>
</BODY></HTML>
```

Figure 3. Using put message options to send a message.

In Figures 2 and 3, objects are created and destroyed after IIS finishes processing an Active Server Page. For high performance Web applications, programmers should consider storing references to MQAX instances in the *Contents* collection of a *Session* or *Application* object. Figure 4 demonstrates in VBScript how to store an MQQueueManager object reference, and reuse it later to examine the ReasonName property.

Figure 5 gives an example appropriate for back-end applications that need to receive a request message and process it. Using Visual Basic, we create queue manager, message and get message option objects. A connection to queue *pp.LQ1* is established and a reference to a queue is returned. We choose to wait 3 seconds for a message to arrive. When the Get operation returns, the program either receives a message or MQSeries returns errors through CompletionCode, ReasonCode or ReasonName properties.

```
Set Session("QM") = Server.CreateObject _
   ("MQAX200.MQQueueManager")
' Do some work and then check the ReasonName property
MsgBox Session("QM").ReasonName
```

Figure 4. Using put message options to send a message.

```
Dim qMgr As New MQQueueManager
Dim gMsg As New MQMessage
Dim gmo As New MQGetMessageOptions
Dim getQ As MQQueue
Set getQ = qMgr.AccessQueue( “pp.LQ1”, _
  MQOO_INPUT_AS_Q_DEF, “ppDEFQM”)
gmo.Options = MQGMO_WAIT
gmo.WaitInterval = THREE_SECONDS
getQ.Get gMsg, gmo
‘ Check for errors
’ Process the request
```

Figure 5. Getting a message using get message options.

Once a message has been received, we can examine elements of the MQMD structure through the many properties of the MQMessage object. For example, we might check the message ID, correlation ID or group ID.

Suppose we had wanted to receive the message within a unit of work. This would allow the program to coordinate the message receive operation with work performed in response to the request. If the work succeeds, it is committed and the message is removed from the queue. If an error occurs, the work is rolled-back and the message remains on the queue.

```
qMgr.Begin
getQ.Get gMsg, gmo
‘ Other work controlled by
‘ the unit of work (transaction)
qMgr.Commit
```

Figure 6. Getting a message within a unit of work.

To send or receive messages within a unit of work, the programmer brackets related operations between Begin and Commit calls to the queue manager. Figure 6 shows how this is done. **END**

[1]COM follows an apartment threading model. In any process, one multi-threaded apartment (MTA) and any number of single-threaded apartments (STA) is allowed. At any instant in time, more than one thread can execute in an MTA, but only one thread can execute in an STA. Classes are marked in the Registry as supporting one type of threading model, i.e., can execute using free-threading or single-threading. COM handles thread, apartment, and objects creation; marshals interface calls across apartment boundaries; and serealizes calls from an MTA into an STA.

Queued-Up

In each issue we feature useful MQSeries resources. Future issues will review hot sites with MQSeries content and the MQSeries shows and conferences you need to attend.

For your diary:

Transaction and Messaging Congress
October 11-15, 1999 - Prague, Czech Republic.
One of the major IBM shows that complements the CICS and MQSeries technical conference recently held in Dallas. Watch this space.

Some useful online resources:

MQSeries Home Page
The latest news and information about MQSeries, including partner support and downloads, case studies, education and events.

www.ibm.com/software/mqseries

MessageQ.Com
The independent site for Application Integration, providing daily news, features and in-depth interviews on all aspects of Enterprise Application Integration, including MQ topics.

www.messageq.com

BMC Software
Find out more about BMC’s Application Service Assurance solution for MQSeries.

www.bmc.com