12 MQ update

June 2000

In this issue

MQ Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: +44 1635 550955
e-mail: harryl@xephon.com

North American office

Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: +1 303 410 9344
Fax: +1 303 438 0290

Contributions

Articles published in *MQ Update* are paid for at the rate of £170 ($250) per 1000 words and £90 ($140) per 100 lines of code. For more information about contributing an article, please check Xephon's Web site, where you can download *Notes for Contributors*.

MQ Update on-line

Code from *MQ Update* is available from Xephon's Web site at www.xephon.com/mqupdate.html (you'll need the user-id shown on your address label to access it). If you've a problem with your user-id or password call Xephon's subscription department on +44 1635 33886.

Editor

Harry Lewis

Disclaimer

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs £255.00 in the UK; $380.00 in the USA and Canada; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the July 1999 issue, are available separately to subscribers for £22.50 ($33.50) each including postage.

MQSC commands, throughput, and Perl scripting

INTRODUCTION

MQSC commands are generally used to manage queue manager objects, such as channels, queues, and process definitions. With these commands it is possible to define, delete, and alter the attributes of these objects. Each OS supplies its own interface to these commands. On OS/390 they can be issued either as operator commands or by means of panels. On Unix, VMS, and Windows NT, the commands can be executed using the **runmqsc** utility.

In this article, we focus on issuing MQSC commands using the **runmqsc** executable. More specifically, we focus on using MQSC commands to extract active channel status information in order to generate extended reports on throughput.

MQSeries has several commands that are used to query the current status of queue manager objects. The *DIS CHSTATUS* MQSC command is specific to channels and can be used to query the status of active channels in the system. It can return, for example, the *SHORTRTS* status attribute, which records how many 'short retry' attempts are left on the channel, or the *CONNAME* attribute, which (in this case) shows the address of the machine at the other end of the channel – if the transport protocol used by the channel is TCP/IP, it returns the IP address of the machine. Channels that serve client connections (*SVRCONN*) may have multiple instances active; in such circumstances, the *CONNAME* attribute can be used to differentiate between machines when querying their status.

Nevertheless, all these attributes provide only basic measurements about channel status, leaving a lot of information. One of the most fundamental measurements that we, at our installation, felt was missing from reports is the throughput of active channels. We needed the answer to questions such as: how many bytes are flowing in each direction and at what rate? How many messages pass per second? What is the average size of messages passing through a channel?

There are some hefty third-party software packages around that can be used to command and control queue managers and dig out statistics such as those mentioned above. However, these packages are expensive, they require training, and they consume a fair chunk of a machine's resources. In this article, we present a lightweight and simple script that provides the statistics we need and others may also find of great value.

The script requires no complex and expensive command and control software. It's written in Perl (Version 5) and doesn't even use the MQI (or any other API, for that matter) – it uses only the MQSeries **runmqsc** utility for querying MQSeries. This script was tested on HP-UX versions 10.20 and 11.0 and Windows NT 4.0. I will also demonstrate how the script can easily be run indirectly on any other platform, such as IBM OS/390.

In general, what the script does is capture status information on the specified channel at fixed intervals. Status information is captured using replies from the *DIS CHSTATUS* MQSC command. The difference (or so-called 'delta') between each two measurements is printed. Take, for example, the field *BYTSSENT*. Suppose that, at one invocation of the script, it reads '150' and, at the next invocation, five seconds later, its value is '180'. From this, it is concluded that the average send rate was six bytes per second during that specific interval, as:

```
(180 - 150) / 5 = 6
```

The same method is used on several other fields.

The script is listed here in its simplest form. Many additions and enhancements can be made to it, and some are discussed after the listing.

MQTP.PL

```
#!/usr/local/bin/perl -w
#------------------------------------------------------------------------
#
# DESCRIPTION:
#
#     This perl script computes MQSeries channel throughput statistics
```

```
#     at fixed time intervals. It prints averages for the send rate,
#     receive rate, message rate, and message size during the interval.
#
#     For simplicity, all interaction with MQSeries is made using MQSC
#     commands and not the MQI or any other API set.
#
#     This script can be run on any platform that supports Perl and
#     the runmqsc executable.
#
# USAGE:
#
#     mqTP.pl <chl name> <sampling interval>
#
#     * "chl name" is the name of the channel to sample. This can
#       be a channel of any type.
#
#     * "sampling interval" is the number of seconds between
#       each measurement.
#
#     * The runmqsc executable must be in the path environment
#       variable.
#
#     Example:
#
#       perl mqTP.pl MQ.CHANNEL1 5
#
# INTERNALS:
#
#     * Channel status hash variables: the status of a channel is
#       stored as a hash variable containing the following fields:
#
#       active  Either 1 (channel is active) or 0 (otherwise).
#       now     The time when a sample is taken.
#       msgs    The number of messages sent and received. This
#               corresponds to the MSGS channel status attribute.
#       sent    The total number of bytes sent through the channel.
#       rcvd    The total number of bytes received through the
#               channel.
#
# AUTHOR:
#
#     jb, feb 00.  jb@lando.net
#
#-----------------------------------------------------------------------

if ($#ARGV+1 < 2)
{
    &print_usage ();
    die("\n");
}
```

```
$chl_name = $ARGV[0];
$sleep_time = $ARGV[1];

%old_chs = &get_chs ($chl_name);
&check_chs (%old_chs);

while (1)
{
    sleep ($sleep_time);

    %new_chs = &get_chs ($chl_name);
    &check_chs (%new_chs);

    print_stats (old_chs, new_chs);

    %old_chs = %new_chs unless $new_chs{"msgs"}-$old_chs{"msgs"} <= 0;
}

# ------------------------------------------------------------------
# Execute given runmqsc command and return response lines.
#
# TAKES: one-line string containing the mqsc command.
# GIVES: one-line string containing the runmqsc response.
# ------------------------------------------------------------------
sub runmqsc
{
    my ($line, $lines);

    # This pipes the given mqsc command through runmqsc and
    # grabs the output.
    open (CMDOUT, "echo \"$_[0]\" | runmqsc |") ||
        die("unable to runmqsc given command $_[0]");

    $lines = "";
    while($line=<CMDOUT>)
    {
        $lines .= $line;
    }

    close (CMDOUT);
    return ($lines);
}

# ------------------------------------------------------------------
# Retrieve channel status measurements.
#
# TAKES: one string containing the name of the channel.
# GIVES: a hash variable containing the channel status measurements.
# ------------------------------------------------------------------
sub get_chstatus
```

```
{
    my (%chstatus, $response);

    %chstatus = ("active" => 0,
                 "msgs" => 0,
                 "rcvd" => 0,
                 "sent" => 0,
                 "now"=> "");

    $chstatus{"now"} = time ();

    $response = &runmqsc("dis chstatus($_[0]) all");

    # Look up phrase "status(running)" in mqsc response.
    # If found, the channel is active.

    if ($response =~ /status\(running\)/i)
    {
        $chstatus {"active"} = 1;
    }
    else
    {
        return (%chstatus);
    }

    # Look up phrases "  msgs(*)", "bytssent(*)", and "bytsrcvd(*)".
    # Note the blank before the phrase "msgs" to differentiate it
    # from the similar expression "curmsgs(*)".

    $response =~ / msgs\((\d*)\)/i;
    $chstatus{"msgs"} = $1;

    $response =~ /bytssent\((\d*)\)/i;
    $chstatus{"sent"} = $1;

    $response =~ /bytsrcvd\((\d*)\)/i;
    $chstatus{"rcvd"} = $1;

    return (%chstatus);
}

# ------------------------------------------------------------------------
# Calculate statistics and print them out.
#
# TAKES: previous and current channel status measurements.
# GIVES: nothing.
# ------------------------------------------------------------------------
sub print_stats
{
    my ($deltat, $deltas,
```

```
    $deltar, $deltam, $BSPS, $BRPS, $BPM, $MPS);

    ##%old_chstatus = $_[0];  %new_chstatus = $_[1];
    $deltat = $new_chstatus{"now"}  - $old_chstatus{"now"};
    $deltas = $new_chstatus{"sent"} - $old_chstatus{"sent"};
    $deltar = $new_chstatus{"rcvd"} - $old_chstatus{"rcvd"};
    $deltam = $new_chstatus{"msgs"} - $old_chstatus{"msgs"};

    if ($deltat <= 0)
    {
        die("deltat must be > 0 ! aborting..\n");
    };

    if ($deltam == 0)
    {
        $deltam = 1;
    }

    $BSPS = $deltas / $deltat;             # Bytes sent per second
    $BRPS = $deltar / $deltat;             # Bytes received per second
    $MPS  = $deltam / $deltat;             # Messages per second (S/R)
    $BPM = ($deltas + $deltar)/ $deltam;   # Bytes per message.

    printf ("snd %7.7s KB/sec | ", $BSPS/1000);
    printf ("rcv %7.7s KB/sec | ", $BRPS/1000);
    printf ("%7.7s msg/sec | ",$MPS);
    printf ("%7.7s KB/msg", $BPM/1000);
    printf ("\n");
}

# ------------------------------------------------------------------------
# Print usage summary.
#
# TAKES and GIVES nothing.
# ------------------------------------------------------------------------
sub print_usage
{
    print "USAGE: perl mqTP <chl name> <sampling interval>\n";
}

# ------------------------------------------------------------------------
# Check the status of a channel.
#
# TAKES: the channel status hash variable.
# GIVES: true (channel is active) or false (inactive).
# ------------------------------------------------------------------------
sub check_chstatus
{
    my (%chstatus) = @_;
    if ($chstatus{"active"} == 0)
```

```
    {
        die("channel inactive or does not exist, so stopped.\n");
    }
}
```

Below is some sample output of the script run on a sender channel (type *SDR*) that connects two queue managers communicating over a WAN link. Note that the average send rate is 130 KBytes per second (with a rather large standard deviation). The receive byte rate is very low compared to the send rate, as would be expected on an *SDR* channel.

```
> perl mqTP.pl CHANNEL1 5
snd  45.388 KB/sec | rcv       0 KB/sec |     0.2 msg/sec |  226.94 KB/msg
snd   48.63 KB/sec | rcv       0 KB/sec |     0.1 msg/sec |   486.3 KB/msg
snd   48.63 KB/sec | rcv       0 KB/sec |  0.0625 msg/sec |  778.08 KB/msg
snd 61.4911 KB/sec | rcv       0 KB/sec | 0.14285 msg/sec | 430.438 KB/msg
snd 47.8628 KB/sec | rcv       0 KB/sec |     0.4 msg/sec | 119.657 KB/msg
snd 116.104 KB/sec | rcv       0 KB/sec |       2 msg/sec |  58.052 KB/msg
snd 128.124 KB/sec | rcv       0 KB/sec |     1.2 msg/sec |  106.77 KB/msg
snd 126.558 KB/sec | rcv       0 KB/sec |       2 msg/sec | 63.2790 KB/msg
snd 171.397 KB/sec | rcv       0 KB/sec |     0.8 msg/sec | 214.246 KB/msg
snd 175.175 KB/sec | rcv  0.0056 KB/sec |       3 msg/sec | 58.3936 KB/msg
snd 139.182 KB/sec | rcv       0 KB/sec |     0.4 msg/sec | 347.956 KB/msg
snd 140.790 KB/sec | rcv       0 KB/sec |       2 msg/sec | 70.3950 KB/msg
snd 196.703 KB/sec | rcv       0 KB/sec |     0.4 msg/sec | 491.758 KB/msg
snd 142.387 KB/sec | rcv       0 KB/sec |     2.2 msg/sec | 64.7214 KB/msg
snd 162.711 KB/sec | rcv       0 KB/sec |     0.5 msg/sec | 325.422 KB/msg
snd 202.157 KB/sec | rcv       0 KB/sec |     0.2 msg/sec | 1010.78 KB/msg
snd 202.157 KB/sec | rcv       0 KB/sec |     0.1 msg/sec | 2021.57 KB/msg
snd 188.671 KB/sec | rcv       0 KB/sec |     0.2 msg/sec | 943.359 KB/msg
snd 81.8728 KB/sec | rcv       0 KB/sec |     0.2 msg/sec | 409.364 KB/msg
snd 142.515 KB/sec | rcv 0.00466 KB/sec | 2.66666 msg/sec |  53.445 KB/msg
```

POSSIBLE ENHANCEMENTS TO THE SCRIPT

1 The script currently calculates average measurements over fixed intervals. A useful improvement would be to catch the '^C' break signal and print overall averages for the entire duration of the run. A possible way of implementing this would be to modify the script to generate an output log of the results, and then use the logged values off-line to calculate the overall averages and plot them.

2 The script currently quits if it finds the specified channel is

inactive. It may be extended to 'endure' this in a manner similar to that of the TCP/IP **ping** utility.

3 The script may also be modified to display the status of the corresponding transmission queue. Each channel has a specific *XMITQ* that it feeds from. This attribute can be queried and the current status of that queue (in other words, current queue depth, *IPPROCS*, *OPPROCS*, etc.) can be displayed.

4 The script is meant to run on systems that use the **runmqsc** executable, as stated above. It is possible, though, to alter the script so that it can be run 'remotely' on another machine. This is explained in more detail in the section below.

NOTES

This is a concise 'walk through' that describes how to use the **runmqsc** executable to manage remote MQSeries objects. Full details on this can be found in the *Administering remote MQSeries objects* chapter in the *MQSeries System Administration Guide*.

runmqsc can be used to administer objects that belong to both local and remote queue managers. However, before **runmqsc** is run, several MQSeries objects should be defined and configured on both the local and the target remote queue manager.

Configuration on the local queue

1 A local queue manager must be used and, furthermore, it must be the default queue manager. The reason for this is that the **runmqsc** utility is invoked with only two parameters: the timeout (see below) and the name of the target queue manager. There is no way of specifying the name of a local queue manager, so the default must be used. The local queue manager should have the following items defined on it:

   – A sender channel that will convey the MQSC command messages from the source queue manager to the remote queue manager.

– A receiver channel that will receive the responses coming from the target queue manager.

– A transmission queue with the name of the target queue manager.

2 The target queue manager is the queue manager whose objects we plan to control remotely. It should have definitions for the following items:

– A receiver channel for receiving MQSC commands.

– A sender channel for sending responses back to the source queue manager.

– A transmission queue to be used by the target queue manager for temporarily holding responses until the sender channel ships them. Again, the name of the *XMITQ* should be the same as that of the source queue manager.

The timeout flag **-w** must be specified when the **runmqsc** executable is invoked. This serves two purposes: to indicate that the utility is to run in 'indirect mode' on a remote queue manager and to specify the timeout used for commands issued. The second parameter is the name of the target remote queue manager.

Now that I've outlined the set-up, I'll mention a few points that are pertinent to the mechanism used.

- Remember that the transmission queues should have the same name as the participating queue managers.
- I recommend that you use channel trigger definitions in the transmission queue. This ensures that the sender channel is launched automatically. If trigger definitions don't exist, you have to start both sender and receiver channels manually, otherwise the MQSC command messages and the replies are left unsent in the queue manager where they originate.
- Remember that the *SYSTEM.ADMIN.COMMAND.QUEUE* command queue must be defined on the target queue manager. If it isn't, the commands are sent to the dead-letter queue instead of the command queue.

- Remember that the command server must be running on the remote queue manager. If isn't, the MQSC command messages are not picked up from the command queue, in the target queue manager.

JB (jb@lando.net)
MQSeries Specialist (Israel)

MQSeries and high availability

INTRODUCTION

This article describes the concept of high availability (HA), describing the fundamental approaches to minimizing the amount of time during which critical services or applications are not accessible by clients or business processes.

The article describes how high availability relates to MQSeries, examining the relationship between MQSeries Queue Manager Clusters and HA clusters. However, in order to tackle this topic, it is necessary first to do some groundwork on the subject of high availability. Those already familiar with HA may consider moving directly to the section entitled *MQSeries and HA*.

WHAT IS HIGH AVAILABILITY?

Modern businesses are increasingly operating on a round-the-clock basis. The rapid growth of the Internet and the need to conduct business on-line have led to the need to provide access to business systems on a 24-hour basis, as clients from around the world initiate e-business transactions around the clock.

It is, therefore, becoming increasingly important that critical applications are kept running day and night. This includes Web

servers, application servers, enterprise databases, and all the infrastructure that binds these systems together.

There are times when a computer system has to be shut down in order to perform planned maintenance. There are also times when hardware or software failures cause unplanned system outages. These periods are referred to as 'downtime'. Of key interest to enterprises is how to minimize periods of downtime, whether planned or accidental.

The availability of a system is commonly expressed as a 'number of nines'. People talk about 'three nines' or 'five nines' availability. These are references to the percentage of time for which a system is available. Three nines means that a system is available 99.9% of the time. Five nines means that it's available 99.999% of the time. While these may sound similar – both indicate that the system is available virtually all the time – they become much more revealing if instead of looking at the time for which a system is available one looks at the figures in terms of downtime over a fixed period. For example, a system with 99.9% availability can be expected to suffer over 8.5 hours of downtime a year, while a system with 99.999% availability will be down for only a little over 30 seconds a year. Such availability does not come cheap, and there is a cost involved in minimizing downtime. To be able to replace a faulty component or move a failed system's workload in just 30 seconds is no mean feat. This would either require hardware that has a mean-time-between-failures of far more than a year, so that an average of 30 seconds per year could be sustained, or that can switch either components or workload very rapidly. Alternatively, you would need to employ the fastest hardware engineers in the west, and they would have to take up residence in your machine room.

Apart from all the 'nines', there are other less specific terms used to describe systems that maximize availability. These include 'continuous availability', which strictly speaking means that there should be no discernible downtime, even in the event of a component failure. A continuously available system should detect an error that would result in a loss of availability immediately and be in a position to provide an alternative component that's already 'revved up' and ready to go. A continuously available system should also support the scheduling of

planned maintenance by allowing workload to be seamlessly transferred away from the components or subsystems that are the subject of maintenance activity. Another common term is 'fault tolerance', which means almost the same as continuous availability, but is used to describe systems that can switch to a replacement component in a matter of a few milliseconds in the event of a failure, this being fast enough to be described as 'near instant'. Further down the scale is high availability, which generally means that a system can detect a single failure and react to it automatically in a matter of at most a few minutes. There are two significant aspects to this definition of high availability: the system should survive a single failure, though a second failure may result in a loss of service, and the detection of faults and the triggering of action to accommodate faults should be automatic, requiring no manual intervention.

It should be apparent from the above that high availability is much cheaper than either continuous availability or fault tolerance. High availability is generally achieved by grouping relatively inexpensive hardware systems and connecting them together using common interconnections and storage mechanisms. This replication of whole computers is considerably cheaper than alternatives that use proprietary hardware architectures and is sufficient for many organizations. Systems built this way are often called 'HA clusters' or 'shared disk clusters', because they frequently rely on the presence of shared disks, as described below.

HOW DOES AN HA CLUSTER WORK?

The physical architecture of an HA cluster is that there are two or more complete computers that are connected together. Each computer runs a separate operating system image and the computers are referred to as 'nodes'. Most vendors of HA cluster technology offer scalability of between four and 32 nodes participating in a single cluster, although there are some that are limited to two nodes. If one node fails, then the (highly available) workload that was running on the node is made to 'failover' to another node in the cluster.

In order to achieve a failover, all nodes must be able to access the data, and this is a pre-requisite to the service. This doesn't mean that more

than one node needs to be able to access the data at the same time, it just means that critical data should not be stored in the disks or memory of just one node. Otherwise, if the node were to fail, the data would be inaccessible until the node is recovered. There are two principal means of achieving common access to critical data. One is to replicate the data by mirroring it, and there are a number of commercially-available systems that provide this sort of support. The other is to attach the nodes physically to a shared data bus and to attach the disks containing the critical data to the bus. The shared bus is normally based on SCSI technology, though there is increasing support for higher bandwidth interconnects, such as Fibre Channel or Memory Channel.

Both these approaches have their advantages and disadvantages. A mirrored system can provide greater separation between nodes, which is useful if the cause of a node failure is environmental, such as an earthquake, landslide, or power outage that affects a whole community. The drawback with mirrored systems is that they are either slower (sometimes as a result of the greater distance between nodes) or asynchronous (which may mean that, at the time of a failure, the only up-to-date copy of the data is at the node that has just been rendered inaccessible). The advantages of a shared disk system are that it's cheap and synchronous, its disadvantages are that it offers less protection from environmental disasters, as only one copy of the data exists. Another disadvantage of shared disks is that the SCSI standard restricts the maximum separation between nodes to 25 metres typically, though a separation of as much as one kilometre is possible if repeaters are used.

To address all these issues is outside the scope of this article, which will focus on shared disk HA clusters, but it is important for those that require high availability to consider all possible risks and to develop plans to counter or mitigate each type. A robust back-up procedure and a disaster recovery plan can both complement a shared disk solution.

As well as requiring access to critical data on shared disk, each node in the cluster also needs to be monitored by some other node, so that, if a node unexpectedly fails, the monitoring node can report the failure and the remaining nodes in the cluster can initiate corrective action.

In a two-node cluster, both monitoring and initiation are performed by the surviving node, but more scalable cluster architectures can adopt various topologies, including ‘rings’ and ‘trees’. For a node to be monitored by another node, some kind of connection must exist between the nodes. In general this is implemented as an additional ‘private’ network between the nodes. There are alternatives, such as using a public network or a SCSI channel, though a private network generally benefits from being immune to other traffic and more secure. A private network can be implemented cheaply by adding a second Ethernet adapter to each node. In the case of a two-node cluster, the nodes’ Ethernet adapters can be directly connected by a crossover cable.

There are a number of other features that may be found on HA clusters, including multiple private and public networks (to avoid the network being a single point of failure) and RAID storage for data. The more robust cluster offerings enforce these features and will prevent you from configuring or starting a cluster that has fewer than two private networks or has data that is stored on a single drive. Such clusters can also employ advanced fault detection algorithms that are able to analyse the response from a network to determine whether a fault lies in the local network adapter, the cable (or network infrastructure), the remote network adapter, or the remote node.

It is also necessary to consider the question of client connectivity. If network clients connect to a service or application running on a node that subsequently fails, it is necessary somehow to move those network connections to the node to which the service itself has been moved. Moving network connections is not easy – in general, it is necessary to break the connections, which happens to be a natural consequence of the failure of a node in TCP/IP, and to remake the connections with the node to which the service or application has been moved. Clients, therefore, need to be written in such a way that they tolerate a broken connection and will try, perhaps for a limited time or number of attempts, to reconnect. For these reconnection attempts to succeed, the address (or name) to which they are trying to connect must be transferred to the new node. Some clusters support the failover of SNA (LU 6.2) LUNAMEs, while others support the failover of IP addresses. Some clusters support both. The failover of

an IP address is known as 'IP Address Takeover' or IPAT. During IPAT, network routing information is updated, so that future packets sent to the IP address being failed over are routed to the MAC address of the new node.

From an administrative point of view, it is simplest if a cluster is treated as a single computing resource. If the nodes are identical in all respects other than network address and name, and all have identical versions of software installed on them, then it is far easier to configure applications and services to run on the cluster in such a way that they provide the same functionality and performance regardless of which node they are running on. As part of this symmetry, it is also preferable if the cluster has a common namespace and security model.

From a capacity planning point of view, it is important to provide enough spare capacity on each node to enable it to cope with the additional demands that could be placed on it should other nodes in the cluster fail. In this regard, it is important to decide whether you want to run the cluster in 'Active/Passive mode', which means that some nodes merely act as standbys for other nodes, or in 'fully Active mode', which means that all nodes have their own workload and also act as standbys for one or more other nodes. A popular way of distributing workload and standby responsibility in a cluster is to run the production workload (the work that has to be performed to keep the business going) on some nodes, while others handle workload that is less critical (for example, development work) and also act as standby nodes for those that run the production workload.

MQSERIES AND HA

One of the key concepts in an HA cluster is the 'unit of failover'. This is the term used to describe the smallest indivisible collection of resources, data, and processes that have to be present on the same node for the desired service to be provided. The unit of failover differs from one type of application or service to another. For a simple name service, it may comprise one disk file that contains a directory or database and a daemon that reads the file and responds to queries arriving over an IP network. In this case it would be inappropriate to move just the disk file, the daemon, or the IP address to another node

– failover works only if they are all moved. In the case of MQSeries, the smallest unit of failover is a Queue Manager. This is because each Queue Manager has a single recovery log, which is replayed on restart to recover incomplete units of work. It would not make sense to try to move just a queue to a new node without moving the whole Queue Manager to which the queue belongs.

When making an MQSeries Queue Manager highly available, as many as three different types of data may need to be accessible by the standby node. Two of them are both obvious and essential – they are the recovery log and the queue files. These should be stored on shared disk drives. The third type of data that may be needed, depending on the platform, is configuration settings, which may, for example, be stored in a machine's registry.

To make an MQSeries Queue Manager operate within an HA cluster, and be able to move from one node to another, the Queue Manager needs to be configured so that its log and queue files are on shared disks. The Queue Manager could be created this way or reconfigured either manually or (preferably) programmatically. An example of the latter is in the MQSeries SupportPac MC74 for Windows NT, which provides Microsoft Cluster Server support and includes a utility that will modify an existing Queue Manager whose files are on a local drive and move the files to a shared drive (chosen by the user). The utility also updates the relevant registry entries accordingly. The same utility is able to move files back to the local drive, should the need arise.

The steps needed to integrate MQSeries with an HA cluster depend on the platform that is used and the choice of clustering software. The most common approach requires the creation of a number of scripts that know how to start, stop, and monitor the state of a Queue Manager. The interface to and implementation of these scripts depends on the specific clustering support that is being used, which in turn generally varies from one platform to another, even if the basic principles are fairly consistent.

Some platforms use a style of HA clustering that doesn't rely on scripts, instead offering a choice of either very simple stop/start

actions, which are not customizable to a particular application or service, or programming of a custom library of methods that must conform to an interface specified by the cluster software. Most Unix platforms support script-style clustering, while Windows NT supports both the generic and custom approaches. As it's usually better to stop most types of service and application under controlled conditions, I recommend the writing of scripts or a custom resource library to handle this task. This has already been done for MQSeries on some platforms – at the time of writing, the MQSeries SupportPac site contains solutions for AIX, HP-UX, and Windows NT.

On most clustering systems, it's possible to configure the cluster so that the cluster software performs an IP Address Takeover and migrates (in other words, takes control of) shared disks. Consequently, by the time MQSeries-specific scripts and methods are called, the only things that need to be handled are starting or stopping the Queue Manager and starting or stopping any fault monitoring processes that are used to keep track of the Queue Manager. Before MQSeries Version 5.1, it was also necessary to restart channels, though channel states are now bound to the recovery log. When a Queue Manager is restarted on a new node, channels maintain their previous states. An exception to this is 'Requester' channels, though, in fairness, it's unlikely that channels of this type will be used on a server cluster.

RELATIONSHIP TO MQSERIES QUEUE MANAGER CLUSTERS

MQSeries has built-in support for Queue Manager Clusters. These are different from HA clusters and the two are complementary. A Queue Manager Cluster can be used to provide much simpler administration and workload distribution and higher availability than was previously available with normal distributed queueing.

The simpler administration is a result of the introduction of 'Cluster Repositories'. These allow you to define one cluster-receiver channel and one cluster-sender channel per Queue Manager, and MQSeries takes care of the rest. This simplifies the addition of Queue Managers as they 'learn' from the repositories and additional channels are automatically defined on your behalf. Furthermore, it is possible to connect to any Cluster Queue Manager and put a message on a Cluster

Queue. Cluster Queue Managers can locate Cluster Queues on your behalf, again using the repositories.

The ability to distribute workload is derived from the ability to define multiple instances of the same Cluster Queue on different Cluster Queue Managers. A message that is put on a Cluster Queue of which there are multiple instances can be automatically routed to any of the instances. Furthermore, you can define how the workload distribution is handled. The improvement in the availability of Queue Manager Clusters results from the fact that, if a node fails and a Cluster Queue Manager becomes unavailable, other Cluster Queue Managers still provide access to the remaining instances of the Cluster Queue. New messages being put on the Cluster Queue just by-pass the failed instance. A message that was previously routed to the failed Queue Manager and is waiting on a transmit queue to make the final 'hop' is automatically re-routed to one of the surviving instances of the Cluster Queue.

The limitation to how much Queue Manager Clusters can improve availability centres on what happens to messages that were previously routed to the failed Cluster Queue Manager and have either been received (and acknowledged) by that Queue Manager or have not yet been acknowledged and are consequently 'In-Doubt'. Unless these messages were the subject of an MQGET before the failure, the delivered messages will be stored in the failed instance of the Cluster Queue. Such messages will not be retrievable until the failed Queue Manager is brought back on-line. In-Doubt messages cannot be re-routed without breaking the transactional model provided by MQSeries and so remain In-Doubt until the failed Queue Manager is brought back on-line. This is where an HA cluster can help – if each node of the Queue Manager Cluster has a standby node that, on failure, can take over and restart any Queue Managers running on the failed node, then problems associated with messages being held up, either because they are In-Doubt or irretrievable, can be resolved quickly and automatically.

Graham Wallis (wallisgd@hursley.ibm.com)
MQSeries Design
IBM Hursley (UK)

More MQSeries standards and guidelines

In my previous article (*MQSeries coding standards and guidelines* in the May 2000 issue of *MQSeries Update*), I discussed the critical need for reliable and secure middleware and showed the ability of IBM's MQSeries to meet this need relatively seamlessly. I also established that, to use MQSeries effectively, you must implement standards and guidelines for application construction before coding begins. This makes it easier to build applications as much of the MQSeries groundwork is already laid. In addition, this method of educating developers is relatively simple as key concepts are concisely documented. My previous article presented essential background information on messaging models, followed by eleven standards for application construction. This article provides ten guidelines that you can use to establish successful application construction. As with my previous article, it also explains the justification for each guideline.

TEN CONSTRUCTION GUIDELINES

The following ten guidelines are important as they help ensure that MQSeries applications function predictably and are easy to maintain. When drawing them up, consideration was given to MQSeries features and options that enable effective performance management, portability, scalability, and reliability. This list of guidelines is by no means exhaustive. Rather, it addresses some of the typical application construction issues that arise when installation begins. Bear in mind that a guideline is a recommendation that you should follow in most cases, but that exceptions are bound to arise. If you don't follow a guideline, the effect on your organization is unlikely to be disastrous. However, the fact remains that your organization would also benefit if you followed the guidelines in most cases.

1 Applications should use the queue attributes *BOTHRESH* and *BOQNAME* when checking the back-out count.

Overview

When a message is retrieved under syncpoint and is subsequently

backed out (either explicitly or implicitly), the *BackoutCount* field in the Message Descriptor is incremented. If *BackoutCount* is above a predefined threshold, alternative processing should be used to handle the message. Applications may hard-code a threshold to which *BackoutCount* is then compared, but a more dynamic method uses the queue attributes *BOTHRESH* and *BOQNAME*. These may be obtained using the *MQInq* API call. The back-out count is then compared to the *BOTHRESH* attribute's value. If it exceeds this value, the message is moved to the queue named in the *BOQNAME* attribute for subsequent processing.

Justification

Externalizing both the back-out count threshold and the pointer to the queue where the problematic message is moved makes the application more flexible and less likely to require changes if processing conditions change.

2 Applications should check the *FeedBack* code after MQGETs.

Overview

MQSeries enables applications to communicate through the *FeedBack* code in the Message Descriptor. While this code can be used to control applications, if the applications themselves don't check it, then this feature can't be relied upon and is, therefore, unavailable. After receiving a message successfully, applications should execute a block of code that checks the Message Descriptor's *FeedBack* field. Additional processing may then be carried out, based on the value of this field. An example of this is a product that uses *MQFB_QUIT* to send a signal to an application that it should end gracefully. Another example would be an application that defines codes that are used to enable or disable additional processing, such as logging.

Justification

Checking the *FeedBack* code will result in more flexible and dynamic applications.

3 Do not use queue-defined message persistence – applications should, instead, explicitly set a message's persistence.

Overview

Message persistence is an attribute that is message-specific. In most cases, only the originator of the message knows the impact of losing it. Therefore, the originator should explicitly set message persistence and avoid using the queue-defined value. Consider the following example: an application that generates update messages happens to use the queue-defined value of message persistence. This application is now vulnerable to the value of the queue attribute *DEFPSIST* – if this is unintentionally set to 'no', update messages may be lost.

Justification

Using the queue-defined value of message persistence will not ensure that a message's persistence is set to an appropriate value. Setting this value explicitly ensures that it is appropriate to the needs of the originating application.

4 Applications should set the persistence of inquiry messages to 'no'.

Overview

In most cases, there are no adverse consequences if an inquiry message is lost, as it's usually acceptable to expect the requesting application to re-send it. Therefore, there is no need to set the persistence attribute of an inquiry messages to 'yes' as the Queue Manager doesn't need to restore it. Making inquiry messages non-persistent prevents them from being logged and decreases the impact on the network.

Justification

This reduces MQSeries resource consumption and increases throughput.

5 When sending messages between applications on heterogeneous systems, use string format.

Overview

A benefit of MQSeries is that it has built-in support for data

conversion between different character sets and types of encoding. For MQSeries to convert a message successfully, it must know the *CCSID* and encoding of both the sending and receiving systems, as well as the message's format. In most cases, the *CCSID* and encoding are handled automatically and there is no need for the application to process them. However, the message format must be explicitly set for MQSeries to carry out a conversion request. A number of formats ship with product and will allow for conversion. These are indicated by the eight-character constants *MQFMT_**.

Justification

MQFMT_STRING is the only format that both allows the application data portion of an MQSeries message to be anything you want and permits conversion. The data, however, must be in string format and is further restricted to the subset of characters that is 'displayable'.

6 Inquiry and reply messages should have their expiry attribute set to avoid stale messages.

Overview

In most cases, requesting applications expect replies to arrive within a reasonable time period. If replies do not arrive in a timely manner, then the request is either aborted or re-sent by the requesting application. If request messages and their replies are not processed in a specified timeframe, they should be discarded as they are stale.

Justification

Setting messages to expire if they are not processed within a specified timeframe saves resources. This setting prevents messages from accumulating when the applications or systems that are responsible for replying to them are not available.

7 Set the message type to reflect the message content.

Overview

MQSeries does not enforce the use of the message type attribute.

Setting it in the Message Descriptor field – whether the message is a *datagram*, *request*, *reply*, or *report*, for example – can help determine how the message should be processed.

Justification

Setting the message type attribute improves message management. This attribute can also be used when the messages expected on a reply queue may vary depending on the options in place (for example, reports and replies going to the same queue).

8 By default, use the convert option when getting messages from a queue unless message conversion is never required. Note that the format *MQFMT_NONE* may be used for individual messages that should not be converted.

Overview

MQSeries provides built-in facilities to perform data conversion between heterogeneous environments. These facilities may be invoked either by coding the convert parameter on the sender-side channel to 'yes' or by using the convert option on the MQGET call (*MQGMO_CONVERT*). The receiving application should perform this conversion, not the channel. If no conversion is required, the Message Descriptor's format field should be set to 'none' (*MQFMT_NONE*). This option suppresses conversion.

Justification

By explicitly specifying data conversion on the MQGET call and avoiding the channel convert parameter, you prevent messages from being sent to the system's undeliverable message queue. Also, if the message channel agent is used to convert messages, channel performance decreases and effective throughput is reduced.

9 Applications should not over-allocate buffer space for MQGET and MQPUT requests.

Overview

The MQSeries Server allocates memory based on the buffer size specified on the MQGET or MQPUT call. If applications over-

allocate the buffer space on a get or put operation, then the queue manager will allocate memory to handle these requests, wasting memory.

Justification

This reduces memory requirements of the queue manager subsystem.

10 Applications that handle request queues should open the queues for inquiry.

Overview

An application handling request queues should check the back-out count after successfully getting a message. The application may also need to inquire about the back-out threshold and/or back-out queue attributes in order to determine whether the message has exceeded a pre-determined back-out value. To allow this, the application should open the input queue for inquiry as well as input.

Justification

This allows for both MQGET and MQINQ (inquiry) calls on a queue.

CONCLUSION

The guidelines set out in this article are not universal – some may make sense for your organization while others may not. It's also worth remembering that this list is by no means comprehensive. You should research the options that are available and carefully choose those that reflect the goals of your organization.

Marc Verhiel Marc_(Verhiel@candle.com)
Candle Corporation (USA)

Modifying and setting up a batch trigger monitor

As a result of the increased use of MQSeries in our company – especially in the handling of EDI messages – the requirement arose for a method of executing a batch job when one or more messages arrive in a queue. While we did not have a mechanism in place at the time to implement this functionality, we were aware of the existence of something called a 'batch trigger monitor', and that sounded like it would do the job.

So we investigated the subject and found that a batch trigger monitor is an MQSeries trigger monitor that will indeed start batch jobs or schedules when messages arrive in a triggered queue.

The process is like that of a CICS or IMS trigger monitor, except that, instead of starting a CICS or IMS transaction when a message is placed in an initiation queue, a batch trigger monitor will start a batch job (or interface with some other function to start a batch job).

THE PROCESS

When a message is *MQPUT* in a queue that has both triggering set to 'Y' and a batch trigger monitor's *InitQ* defined, MQSeries places a trigger message in the defined *InitQ*. The batch trigger monitor waits for a message to be placed on its *InitQ* (with an *MQGET Wait Interval*), retrieving the trigger message when it arrives. The batch trigger monitor then starts the required job.

In our case, there is really no need for an additional process (an MQSeries definition). While a process definition could be used to provide a value for that particular queue (something that could also be achieved using trigger data, for that matter), our implementation does not require this.

OUR REQUIREMENTS

We had certain requirements that needed to be met to implement this process successfully at our company. They were:

- To make the process as simple as possible – in other words, stick to the 'KISS' principle (Keep It Simple, Stupid).
- To use our existing job scheduling product to handle all job starting and scheduling. This ensures a centrally controlled and (better still) automatically tracked and monitored process. This requirement means we cannot submit any jobs or start any started tasks and must instead find a way to interface with our job scheduler.
- To find source code in a language with which we were comfortable, as we had to modify the code based on our other requirements.

OUR IMPLEMENTATION

The IBM-provided Batch Trigger Monitor in Support Pac MA12 is a COBOL implementation that writes JCL from a *DD* statement to the *INTRDR* to kick off a batch job. While this is fine, we wanted a process that would interface with our MVS scheduler instead, as this would then be responsible for kicking off the required schedule. We used a schedule instead of just one batch job in case we need more than one job to process the incoming message or messages.

We found the source for an Assembler Batch Trigger Monitor in one of the MQSeries Red Books that worked a similar way to the COBOL offering. One major difference was that this process issued an MVS *START* command to kick off the job. As we were more comfortable with Assembler, and the method used to start the batch job was closer to our preferred method, we contacted the author (Dave Shogren of IBM) and asked him if he could send us the source code for the program. He agreed and sent it along with an accompanying source module that was also required to get the process working (this additional program was a subtask that was attached to send a *WTOR* and process the responses).

We looked at the code and found that we would have to make three major changes to it to satisfy our requirements and fix a problem:

1 We changed the *Start* process (which directly started a job) to a *WTO* that would just indicate that a message was found in a batch-triggered queue. We then arranged for our MVS automation

package to catch this message and contact the MVS scheduler to kick off the schedule (see the sample below).

2 When attaching the subtask that issues the *WTOR*, we passed the queue manager name so that the *WTOR* message would contain that value in its body. This was required to allow us to distinguish between the various batch trigger monitors we have running (we use one per queue manager), since we automate the start-up of our batch trigger monitors.

3 We also had to change the *MQGET Wait Interval* process to a *SIGNAL* process, as the original *Wait* process wasn't working. So now, we issue an *MQGET* with the *SIGNAL* option, wait on the *ECB*, and re-issue the *MQGET* to retrieve the trigger message, when it's posted.

IMPLEMENTATION STEPS

The changes required to implement the batch trigger monitor are as follows:

- Code and test the changes to the assembler code and link the modules into a load library that is accessible by the batch trigger monitor *Proc*.
- Implement the changes needed by the automation software (rules, scripts, etc).
- Implement the changes needed by the scheduler software. This includes a table that the scheduler uses to kick off the required schedule based on the queue name in the message from the batch trigger monitor.
- Set up the required MQSeries definitions (*IniItQ* and *Process*) and add trigger parameters to the queues that require the 'kick off' process.
- Ensure MQSeries security rules are in place to allow the batch trigger monitor to access its *InitQ*.
- Create the *Proc* for the batch trigger monitor.

- Arrange for a user-id, ‘Down’ and ‘Up’ instructions, and performance parameters for the batch trigger monitor STC (Started Task).
- Ensure that the batch trigger monitor is started when required and stopped during an outage.

Note that a batch trigger monitor should be used only in certain situations – namely when messages are arriving in a queue that require immediate processing. If messages arrive in a queue and require rapid processing, but not to such an extent as to warrant the use of a trigger, then my advice is to leave them queued and schedule a batch job to run on a regular basis to process them.

Below are samples of the MQSeries set-up items.

Trigger monitor (STC) messages at start-up and kick off the scheduler messages:

```
+MQM04: PARAMETERS ACCEPTED. MONITOR WILL CONNECT TO : MCE1
921 MCE1: Batch Trigger Monitor Running. To Terminate reply "STOP"
+MQM50 MCE1: TRIGGER SCHEDULE FOR CISR.MKTT404I
```

The *MQM50* message is the one that’s trapped by our automation package; the other two messages are the batch trigger monitor start-up messages.

BATCH TRIGGER MONITOR PROC

```
//         PROC
//*
//*********************************************************************
//* REVISION:TSSPEEG # 0001   04 JUNE  1998 (NEW PROC)                *
//*********************************************************************
//*               IBM MQSeries for MVS/ESA                            *
//* STARTED TASK PROCEDURE FOR THE BATCH TRIGGER MONITOR              *
//*********************************************************************
//*
//PROCSTEP EXEC PGM=MQMMON,
//         PARM='MCP1,BATCH.MONITOR.INITQ'
//STEPLIB  DD DISP=SHR,DSN=SYSMQS.SUPA.LOAD
//                  DD DISP=SHR,DSN=SYSMQS.MCP1.AUTH
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
```

```
- MQSeries definitions

DEFINE NOREPLACE
QLOCAL('BATCH.MONITOR.INITQ')
 STGCLASS('SYSTEM')
 DESCR('Batch Trigger Monitor Initiation Queue')
 PUT(ENABLED)
 DEFPRTY(5)
 DEFPSIST(NO)
 MAXDEPTH(100)
 PROCESS(' ')
 NOTRIGGER
 MAXMSGL(1000)
 BOTHRESH(0)
 BOQNAME(' ')
 INITQ(' ')
 USAGE(NORMAL)
 SHARE
 DEFSOPT(EXCL)
 MSGDLVSQ(FIFO)
 RETINTVL(999999999)
 TRIGTYPE(NONE)
 TRIGDPTH(1)
 TRIGMPRI(0)
 TRIGDATA(' ')
 NOHARDENBO
 GET(ENABLED)
 QDEPTHHI(80)
 QDEPTHLO(40)
 QDPMAXEV(ENABLED)
 QDPHIEV(DISABLED)
 QDPLOEV(DISABLED)
 QSVCINT(999999999)
 QSVCIEV(NONE)

DEFINE NOREPLACE
PROCESS('BATCH.MONITOR.PROC')
 DESCR('Process for starting Scheduled  Batch Production Jobs')
 APPLTYPE('MVS')
 APPLICID(' ')
 USERDATA(' ')
 ENVRDATA(' ')
```

Scott Morningstar
Computer Systems Officer
CN (Canada)

MQSeries Everyplace

INTRODUCTION

MQSeries Everyplace (MQSE – it was originally launched as 'MQSeries Everywhere') is the newest member of the MQSeries family. It's a lightweight derivative of MQSeries targeted at handheld computers, telephones, personal information managers, PDAs, and the like. Its main objective is to provide reliable messaging and queuing between the above types of device and MQSE servers, plus connectivity to MQSeries, MQSeries Integrator, and MQSeries Workflow.

This article, along with a follow-up article on MQSE configuration in next month's issue, examines the origins of MQSE and its place in the MQSeries family. This includes consideration of:

- What are the differences between MQSeries and MQSE
- How they'll work together
- The broad shape and availability of MQSE
- Possible applications.

THE UNDERLYING CONCEPT

The development of MQSeries Everyplace started around 1998 as part of IBM's 'Pervasive Computing' initiative. Today, the MQSE project is well advanced and Version 1.0 for Windows is now out.

As a member of the MQSeries family, MQSE provides entry-level messaging and message queuing. In practice this means that, with MQSE, IBM will offer reliable messaging software that is compact enough for the smallest devices – MQSE has a target base memory requirement of just 70 kilobytes (about two orders of magnitude smaller than standard MQSeries). That being said, if you add some of the additional functions, such as queue managers and security, the base memory requirement grows, so that a reasonably-featured implementation of MQSE will probably need between 100 and 250

kilobytes of memory, though this is still acceptable for both today's and tomorrow's handheld devices.

MQSE has not forgotten MQSeries. What MQSE has in common with MQSeries is that it:

- Is equally general purpose (perhaps more so)
- Offers the same reliable, once-only messaging model
- Works with MQSeries itself.

MQSE is, however, built on a brand new code base. Opting to deliver the product this way has enabled IBM to take advantage of many of the lessons learned from MQSeries over the past decade.

In addition, MQSE provides small-footprint messaging capabilities to other IBM offerings. For example, DB2 Everywhere (DB2E) makes use of MQSE's communications technology. Other IBM Pervasive Computing solutions are also likely to exploit MQSE, as part of IBM's efforts to capture what Lou Gerstner has described as the opportunity to connect "a billion people interacting with a million e-businesses with a trillion intelligent devices".

THE MARKET FOR MQSE

There are four basic reasons to explain why IBM's customers and partners need MQSE. The first is simply the ability to exploit reliable messaging and queuing using small devices.

This is important because it allows organizations to build their businesses and applications around such capabilities. While many organizations already use MQSeries, they are now looking to extend messaging and queuing out to the new ranges of device – mobile phones, PDAs etc – that will support future geographically dispersed business.

The second reason concerns the use of mobile computing devices in 'hostile communications environments', where people have to go into tunnels, down manholes, or to locations supported by poor communications infrastructures. The common factors here are that:

- Communications have to work in a hostile environment.
- Interruptions are frequent (both in terms of transmissions and work patterns).
- Data volumes are small.
- Locations are dispersed and diverse.

'Big' MQSeries was not designed to work in a world like this. It presumes that a 'fat' network is mostly present and available and much of the work involved with MQSeries focuses on how to recover from those few occasions during which the network is not working.

By contrast, the hostile communications environments for which MQSE is intended comprise networks that are assumed to be inherently unreliable and lacking in capacity. MQSE's design addresses these needs without losing key MQSeries capabilities.

The third reason for interest in MQSE is simply the need for physical mobility. This is required by those who need to connect to a network from different places at different times: today it may be London, tomorrow San Francisco, and the day after Tokyo. Supporting reliable roaming communication links in a flexible way is a significant development goal, and the result is that MQSE's solution is fundamentally different from MQSeries' approach.

Finally, the fourth reason for interest in MQSE is that many users and IT managers want to reduce the administrative overhead associated with a messaging infrastructure. These people don't want to know about channels, trigger monitors, and all the other paraphernalia that accompanies the standard MQSeries product. They want to minimize the number of things that may go wrong and need administrators to fix. Instead, they want a communications environment that's simple, reliable, and transparent.

APPLICATION SCENARIOS

There appear to be four broad categories of application for which MQSE may be relevant:

- Personal productivity applications

- Consumer applications
- Mobile workforce applications
- Control applications.

The first of these is reasonably familiar – for instance, replicating mail, updating calendars, updating databases (for example, on sales, prices and inventory), taking orders, etc. People who today lug a laptop around typically want something smaller. But that smaller device, be it a PDA or an enhanced phone or whatever comes next, will only aid the task in hand if it benefits from dependable and reliable communication.

The second and third categories embrace consumer applications and the mobile workforce. For example, supermarkets might want to give away MQSE-equipped devices to customers in return for customer loyalty (essentially the customer receives a small, cheap device with an attractive user interface that's proprietary to one store chain). Similarly, you can imagine the application of custom devices needing consistent communications in a mobile workforce environment. Examples of industries where such devices would provide benefits include travel, ticketing, restaurants (waiters ordering from the kitchen), package delivery, etc. In the case of package delivery, a recipient might sign for a delivery and that signature is then (within seconds) transmitted directly to the originator.

Finally, control applications are likely to be a source of major interest in MQSE. Oil, gas, water, electricity and similar utility-type organizations have a need to be able to send and receive sensor information from difficult locations.

USER REQUIREMENTS

As might be imagined from the range and diversity of the above, there is a long list of requirements that must be satisfied. For MQSE to be successful, it needs to deliver on as many of them as is practical (as well as on some of the requirements 'inherited' from 'big' MQSeries – functions that customers requested but have not yet been delivered, as the logical place for them is in a new product like MQSE).

The first requirement is a small footprint. While IBM's initial objective was to fit the base product to 50 KB, the revised requirement of 70 KB doesn't seem that much of an increase given that some of the more interesting functions will take a typical implementation up to the 100 KB to 250 KB range mentioned already (there is even a possibility that the core product will shrink below 70 KB threshold when MQSE ships).

The second requirement is for dependable messaging in both reliable and unreliable communications environments, together with automatic recovery and support for TCP/IP, HTTP, and WAP (Wireless Application Protocol). Of these, perhaps the most interesting is the HTTP support. The rationale behind requiring this protocol is simple: many organizations already possess firewalls configured for HTTP. If MQSE can provide reliable communication via HTTP, a minimum of reconfiguration and testing is needed. This is, therefore, a major benefit that may even make up for some of HTTP's inadequacies.

A third requirement is for a tight, efficient protocol that results in as small a message as possible being sent. This was (and still is) not a strength of'big' MQSeries, which shows a clear preference for fat communication pipes, though this is not, in fairness, a problem in a data centre environment. By contrast, bandwidth may be narrow and/or expensive in hostile environments, and fat pipes may be uneconomic, impractical, or both.

This explains why IBM devised a new, efficient protocol for MQSE. Indeed, the justification is demonstrated in a control application example. If data from remote pumping stations has to be transmitted by satellite, reducing each message's length by just a few bytes can deliver major cost savings as well as reducing the bandwidth needed.

In MQSE, IBM appears to have achieved this without losing the reliable messaging and queuing for which the MQSeries family is known.

CODE BASES AND PLATFORMS

IBM initially thought that one code base would be sufficient for the entire MQSE family. By building the product to run on a Java Virtual

Machine (JVM), IBM hoped it would work on multiple operating systems. This, however, turned out to be less than straightforward.

The principal problem was size: any JVM takes up valuable memory in its own right. In addition, MQSE would sit on top of the JVM. To add to this, the application that is the raison d'être of the customer's investment would sit on top of the combined MQSE and JVM.

The risk was that, by requiring a JVM beneath MQSE, there would be too little space left on some key platforms for applications. Furthermore, several target devices (like the Palm Pilot or specific controllers) don't support multi-threading or have peculiar memory models that don't fit the JVM model.

In the end, IBM decided to accept the burden of delivering MQSE on several small operating systems, including the following:

- Windows (CE, 95/98, Windows NT, Windows 2000)
- Java
- Palm OS
- EPOC (Psion, etc.)
- Zaurus (Sharp).

The downside is that, in order to support such a wide variety of environments without an intermediate JVM, IBM has to maintain more than one code base. That being said, none of the code bases will be that large (remember that MQSE is only 70 KB to 250 KB in total), so the maintenance of the various code bases shouldn't present IBM with too onerous a task.

OPERATIONAL CONSIDERATIONS

MQSE provides assured delivery for both synchronous and asynchronous messaging and message queuing. In MQSE, synchronous messaging is additional to what the standard MQSeries product offers. Providing this functionality enables a sending application to know in real-time that a message has arrived (delivery occurs when the message is queued at the destination – not at the

sender, as would be the case in the asynchronous model). Banking applications, in particular, have for some time needed this capability.

To facilitate a broader range of operational possibilities, MQSE also includes:

- Roaming support via device-to-device connectivity, as well as device-to-server connectivity.
- Remote and local queue access, including support for mailbox-type delivery.
- Minimal administration needs alongside high availability.
- Policy-based operation/administration for customizing the user's environment.

In the case of the last of the above points, MQSE even addresses circumstances like fading batteries. You can set policies to tell MQSE what priorities to observe – and which messages to send – if, say, only five minutes of battery power remain.

An equally important operational requirement is compatibility with existing MQSeries installations, though this requirement brings with it a number of complications. For example, the nature of the environment in which MQSE has to work (MQSE must operate in diverse and exposed environments, including the Internet) demands a very strong security implementation. By contrast, standard MQSeries is generally protected by a corporate IS security structure.

The result is that IBM has built security directly into MQSE. Out of the box, it will include support for:

- Authentication
- Encryption
- Non-repudiation
- Compression.

ELABORATING ON SECURITY

To achieve its security objectives, MQSE delivers security at three distinct levels, providing security in ways that enable potential users and developers to select and shape how the levels are applied.

The three levels are:

- Determination of identity (there are a number of options available to achieve this, such as the use of certificates or authentication via operating system services).
- Queue-based security, where you have some target queue that is going to receive messages. To make everything secure, all you need to do is to set the attributes for encryption, compression, and/or authentication at the target queue (and everything else that's needed is handled automatically and transparently – even to the point of creating new channels to assure end-to-end security).
- Message-based security, which is independent of both where messages are going to be sent and the channels and links through which they may pass. This requires application-level encryption and authentication, and MQSE supplies the necessary services for this.

For message-based security MQSE uses public keys and asymmetric encryption. While this is slower than queue-based security, you end up with both proof of the sender's identity and protection for the message's contents.

Charles Brett
President
C3B Consulting (UK)

July 1999 – June 2000 index

Below is an index of all topics covered in *MQ Update* since Issue 1, July 1999. The numbers in **bold** are issue numbers and the ones in brackets are page numbers. Back issues of *MQ Update* are available from Xephon – see page 2 for details.

MQ news

IBM has made a number of substantial MQSeries announcements, including the first release of MQSE, a version of MQSeries V5.1 for Compaq True64 Unix, MQSeries for Linux V5.1 Technology Release, a preview version of MQSeries for OS/390 V2.2, and MQSeries Workflow V3.2.2 (more details on these products will be published in next month's issue).

IBM also announced Microsoft Cluster Server (MSCS) support for MQSeries for Windows NT V5.1, which is in addition to the clustering support already available for MQSeries for OS/390 V2.1 and AIX V5.1. Also new is support for the Microsoft Transaction Server (MTS), which allows COM objects registered under MTS to call MQSeries. All COM/MTS threading models are supported.

Finally, IBM announced support for Java Message Service (JMS), a set of Java classes that allow applications to send MQSeries messages to either existing MQSeries or new JMS applications.

For further information contact your local IBM representative.

* * *

Computer Network Technology's Enterprise/Access software now supports MQSeries, allowing it to speed up the deployment of MQSeries and, it's claimed, reduce the cost associated with providing MQSeries access to mainframe and midrange systems.

Enterprise/Access software provides enterprise information real-time on-line, enabling faster integration with core systems.

No other details were forthcoming.

For further information contact:
Computer Network Technology Corp, 605 N Highway 169, Suite 800, Minneapolis, MN 55441, USA
Tel: +1 612 797 6000
Fax: +1 612 797 6800
Web: http://www.cnt.com

* * *

Financial Fusion has announced Project Indigo, a joint initiative to build connectivity between its Web, wireless, and server applications and IBM's Application Framework for e-Business, including MQSeries, AIX, DB2 Universal Database, and WebSphere application server family as well as the S/390 enterprise server platform.

The two will target Global 1000 financial institutions with a single integrated platform. The suite will be promoted and marketed globally by the two firms, with integration services and support to be provided primarily by IBM Global Services.

For further information contact:
Financial Fusion Inc, 55 Greens Farms Rd, Westport, CT 06880, USA
Tel: +1 203 341 7400
Fax: +1 203 341 7442
Web: http://www.financialfusion.com