26

MQ update

August 2001

In this issue

3 IP multicasting and enterprise messaging

9 Three ways to connect MQSeries clients

27 MQSeries and firewalls

34 Using MQ clusters for large communications buffers

41 Increasing availability on MQ for OS/390

44 MQ news

© Xephon plc 2001

MQ Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38126
From USA: 01144 1635 38126
Fax: 01635 38345
E-mail: info@xephon.com

North American office

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

Contributions

When Xephon is given copyright, articles published in *MQ Update* are paid for at the rate of £170 ($260) per 1000 words and £100 ($160) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £50 ($80) per 100 lines. In addition, there is a flat fee of £30 ($50) per article. For more information about contributing an article you can download a copy of our *Notes for Contributors* from www.xephon.com/contnote.html.

MQ Update on-line

Code from *MQ Update,* and complete issues in Acrobat PDF format, can be downloaded from our Web site at www.xephon.com/mqupdate.html (you will need to supply a word from the printed issue).

Commissioning Editor

Peter Toogood
E-mail: PeterT@xephon.net

Managing Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com

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.

© Xephon plc 2001. All rights reserved. None of the text in this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior permission of the copyright owner. Subscribers are free to copy any code reproduced in this publication for use in their own installations, but may not sell such code or incorporate it in any commercial product. No part of this publication may be used for any form of advertising, sales promotion, or publicity without the written permission of the publisher. Copying permits are available from Xephon in the form of pressure-sensitive labels, for application to individual copies. A pack of 240 labels costs $36 (£24), giving a cost per copy of 15 cents (10 pence). To order, contact Xephon at any of the addresses above.

Printed in England.

IP multicasting and enterprise messaging

WHAT IS MULTICASTING?

Most high-level network protocols, such as ISO transport protocols or TCP or UDP, only provide a unicast transmission service. That is, nodes of the network have only the ability to send to one other node at a time. All transmissions in a unicast service are inherently point-to-point. If a node wants to send the same packet of data to multiple nodes using a unicast transport service it must perform a replicated unicast and send *n* copies of the packet to each destination in turn.

A better way to transmit data from one point to many nodes is to use multicasting. With a multicast transport service a single node can send data to many destinations by making just a single call over the transport service.

Multicasting is useful because it allows the construction of truly distributed applications and provides important performance optimizations over a unicast service. Many underlying media, eg Ethernet, provide support for multicast and broadcast at the hardware or the media access level. Multicasting is currently being used in real-time audio and video-conferencing applications.

IP multicast is a protocol used for transmitting IP datagrams from one source to many destinations in a LAN or WAN of hosts that run on the TCP/IP protocol. The basic facility provided by the IP service is a unicast servicing mechanism. The current standard for IP only provides for the unreliable transmission of datagrams from a single source host to a single destination host. Multicast routing over IP is available but many vendors have to build the reliability mechanism on top of this protocol.

However, MTP (the newer breed of multicast service) provides application programs with a guarantee of atomicity and reliability in sending datagrams in a multicast fashion.

IP MULTICASTING

On systems that support multicast protocols provision must also be

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

made for sending and receiving multicast datagrams. There is a standard mechanism for using IP multicast facilities over sockets in 4.4 BSD Unix and other operating systems that support sockets. Every multicast group has a group address. These addresses are also known as Class D addresses. Class D addresses are not assigned to individual hosts, they are assigned to multicast groups. A class D network address is one defined as having the range of 224.0.0.0 through 239.255.255.255. These addresses are reserved for IP multicast. Hardware vendors such as CISCO, and software vendors including IBM, Microsoft, and Sun have incorporated IP multicasting into their software.

IP multicasting is making significant advances. Using IP multicasting would not only save a Web broadcaster's bandwidth, but its use would mean that the equipment sending out the broadcast would have to handle only one stream. The adoption of IP multicasting is not just about bandwidth congestion. The widespread adoption of this technology is an essential ingredient in the Web's development as a mass medium.

BASIC IP MULTICAST PROGRAMMING

The IP multicast routing protocol uses the TTL (Time-To-Live) property of IP datagrams to decide how far from a sending host a given multicast packet has to travel. The default value of the TTL field is 1, which means it can send the packet to other hosts in the local network.

A SetSockOpt(2) option may be used to change the TTL of the datagram – see below.

```
Unsigned char ttl;
SetSockOpt(sock,IPPROTO_IP,IP_MULTICAST_TTL,&ttl,sizeof(ttl));
```

As the values of the TTL field increase, routers will expand the number of hops they will forward a multicast packet.

Sending a multicast packet is easy. The program uses a sendTo.

```
System call with the host address to send the packet.
The "hello world" message could be used as an example here.
 Sender.c pgm.
#include
#include
#include
```

 © 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

```
  #include
  #define HELLO_PORT 5423
  #define HELLO_GROUP "226.0.0.38"
  main(int argc, char *argv[])
  {
     struct sockaddr_in addr;
     int fd, cnt;
     struct ip_mreq mreq;
     char *message="Hello, World!";
     /* create what looks like an ordinary UDP socket */
     if ((fd=socket(AF_INET,SOCK_DGRAM,0)) < 0) {
                      perror("socket");
                      exit(1);
     }
     /* set up destination address */
     memset(&addr,0,sizeof(addr));
     addr.sin_family=AF_INET;
     addr.sin_addr.s_addr=inet_addr(HELLO_GROUP);
     addr.sin_port=htons(HELLO_PORT);
     /* now just sendto() our destination! */
     while (1) {
                      if (sendto(fd,message,sizeof(message),0,(struct
sockaddr *)&addr,
                                            sizeof(addr)) < 0) {
                           perror("sendto");
                           exit(1);
                      }
                      sleep(1);
     }
}
```

LISTENER.C PGM

```
  #include
  #include
  #include
  #define HELLO_PORT 5423
  #define HELLO_GROUP "226.0.0.38"
  #define MSGBUFSIZE 256
  main(int argc, char *argv[])
  {
     struct sockaddr_in addr;
     int fd, nbytes,addrlen;
     struct ip_mreq mreq;
     char msgbuf[MSGBUFSIZE];
     /* create what looks like an ordinary UDP socket */
     if ((fd=socket(AF_INET,SOCK_DGRAM,0)) < 0) {
                      perror("socket");
                      exit(1);
     }
```

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

```
      /* set up destination address */
   memset(&addr,0,sizeof(addr));
   addr.sin_family=AF_INET;
   addr.sin_addr.s_addr=htonl(INADDR_ANY); /* N.B.: differs fromsender*/
   addr.sin_port=htons(HELLO_PORT);
   /* bind to receive address */
   if (bind(fd,(struct sockaddr *) &addr,sizeof(addr)) < 0)
      {perror("bind");
      exit(1);
      }
      /* use setsockopt() to request that the kernel join a multicast
      group */
    mreq.imr_multiaddr.s_addr=inet_addr(HELLO_GROUP);
    mreq.imr_interface.s_addr=htonl(INADDR_ANY);
 if (setsockopt(fd,IPPROTO_IP,IP_ADD_MEMBERSHIP,&mreq,sizeof(mreq)) < 0)
{
                        perror("setsockopt");
                        exit(1);
      }
      /* now just enter a read-print loop */
      while (1) {
                        addrlen=sizeof(addr);
                        if ((nbytes=recvfrom(fd,msgbuf,MSGBUFSIZE,0,
                             (struct sockaddr *) &addr,&addrlen)) < 0) {
                        perror("recvfrom");
                        exit(1);
                        }
                        puts(message);
      }
}
```

ENTERPRISE MESSAGING AND IP MULTICASTING

Many software vendors in the enterprise messaging area have incorporated IP multicasting in their JMS (Java Message Service) implementations. JMS is becoming a very important part of J2EE, as the inclusion of message-driven Beans in the EJB 2.0 specification illustrates. Application developers are finally getting to see the power that needs to be harnessed from asynchronous messaging models. It's no wonder that products from the application servers marketplace, such as BEA's Weblogic and Alliare's JRUN, now incorporate IP multicasting in their JMS servers. IBM's MQSeries also provides support for HTTP firewall tunnelling through both client and server-side firewalls.

SonicMQ (from Progress Software) also provides support for HTTP firewall tunnelling. The HTTP tunnelling supports client-side proxy

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

servers as well as server-side reverse-proxy servers. The next generation of SonicMQ will include a new deployment option based on multicast architecture. A very good example of a product written entirely in Java and supporting IP multicasting is FioranoMQ Multicast 4.5. It offers a distributed architecture based on IP multicast, but it does not provide any persistence mechanism for its JMS clients.

Typically, most of the enterprise messaging architectures are built or oriented on centralized architectures, where we have a central message server or a cluster of servers to provide the messaging backbone, onto which clients may push and/or pull messages. The implementations are usually hub-and-spoke models, but distributed or decentralized architecture implementations are also increasing.

Tasks such as message logging, which are currently carried out by the server in a centralized architecture, need to be incorporated at the client level if decentralized or IP multicasting is being used. These logging mechanisms will come at cost of disk I/O on the client machines, which could affect the performance of a distributed application.

IP multicasting is very much analogous to the publish-subscribe messaging domain rather than a point-to-point setup. With a pub/sub domain a producer puts a message onto a ‘Topic’ and the consumers that subscribe to the topic get the messages. Applications like this are ubiquitous in B2B exchange architectures.

Multicasting maps naturally onto the way messaging systems were intended to work. Most vendors have built some kind of reliability mechanism over UDP to do this. Multicast has its drawbacks as well – UDP traffic is usually not allowed through a firewall so you may have to negotiate with your network administrators. Furthermore, multicasting relies heavily on special routing software. Most modern routers support multicasting but there might be some old routers in the service path. As a configuration and maintenance consideration, multicast addresses must be coordinated across the network to avoid address collisions.

Multicasting may be the ideal choice if we were to use an enterprise messaging architecture within the enterprise network, but its powers are yet to be formed on the Internet. So, if for some reason a JMS client

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

needs to access a server over TCP/IP (Internet), multicasting may not be the ideal choice. In recognition of the problems of IP multicast, messaging vendors that use IP multicast provide software bridges to carry traffic across routers and firewalls. If most of your messages have to go through these bridges performance might not be very apparent.

So, the rule-of-thumb is to use regular TCP/IP-based messaging systems if messages travel across the firewall and use IP multicasting with reliability mechanisms inside a corporate LAN or VPN.

But under current conditions, if NBC, for example, has to push a news story or video to 10,000 users, it has to send out 10,000 individual video streams. Ideally, NBC would obviously prefer to broadcast the 10,000 streams simultaneously.

This example says it all, really. IP multicasting has immense power as far as content delivery mechanisms or high-speed messaging systems are concerned. With the support of leading software and hardware vendors the IP multicasting-based messaging mechanism will eventually become the standard for messaging architectures, not only in places where asynchronous messaging is required, but it might also evolve as a medium of choice for synchronous messaging as well.

Sudhanshu Sekhar Kar
Pricewaterhousecoopers (USA)
© Xephon

E-mail alerts

If you'd like to be notified when new issues of *MQ Update* have been placed on our Web site, you can sign up for our e-mail alert service, which notifies you when new issues (including new free issues) have been placed on our Web site. To sign up, go to http://www.xephon.com/alerts.

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

Three ways to connect MQSeries clients

"Two roads diverge in the yellow woods … and I have taken the one less travelled and that made all the difference …" *(Robert Frost)*.

PREFACE

Every time I meet a customer just getting into MQSeries I find them using MQSeries Client for everything possible – from the simplest client/server applications to the most complex transactions. I am not going to discuss which applications are suitable for MQSeries Client and which are not – this would require much more then a few pages – but, rather, if you choose to use MQSeries Client what is the best way to go about implementing it?

I will, however, mention the types of application that are using MQSeries Client – at least the ones I know of or helped set up.

BEFORE WE START…

I am assuming the following procedures have been completed. If this is not the case you should review the relevant *MQSeries Quick Beginnings* chapter, or read *Appendix 1* to this article.

- MQSeries is installed on both the client and the server.
- There is at least one queue manager defined on the server.
- Communication is up and running between both systems (I would advise that you don't go up against firewalls and such on the first attempt).
- MQSeries Communication is up and running on the server.

I would also suggest reading the following:

- Verification procedure: on AIX (as a sample Unix queue manager).

  http://www-4.ibm.com/software/ts/mqseries/library/manualsa/amqaac03/amqaac030s.htm#HDRMANVER

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

- Verifying a local installation: on Windows NT.

  http://www-4.ibm.com/software/ts/mqseries/library/manualsa/amqtac01/amqtac010w.htm#HDRAMQ72ZW

- Configuring communications for the MQSeries Client.

  http://www-4.ibm.com/software/ts/mqseries/library/manualsa/amqtac01/amqtac0115.htm#HDRAMQ72A8

To make things easier, I will be using TCP/IP as the transport medium.

MQSERVER – QUICK, SIMPLE, AND GETS THE JOB DONE

I remember the first time I tried setting up an MQ Client machine (back in V2 days); it was a long and painful task, mainly because the MQSeries Client book was somewhat obscure on how to get the whole thing up and running. It dealt at great length on the client connection channels and how to deploy them, but failed to explain how to create them. Today, when I meet customers, I see very little has changed: most of these people are still trying to conduct a simple test of their own environment.

So, the following procedure is for all those people – it will get the whole thing up in less then ten minutes.

Getting started

1 Logon as an MQSeries administrator user (any user belonging to group *mqm*) to the MQSeries server.

2 Enter a command line mode (DOS in the Windows environment or shell in Unix).

3 Type **runmqsc**

4 Type

```
"define channel (MY.SVRCONN)       +
     chltype  (SVRCONN)            +
     trptype  (TCP) "
```

5 Type **end**.

6 Logon as an MQSeries administrator user (any user belonging to group *mqm*) to the MQSeries Client.

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

7 Use the following command on Windows:

```
set MQSERVER=MY.SVRCONN/tcp/Server_ip_Address
```

or in Unix ksh:

```
export MQSERVER=MY.SVRCONN/tcp/Server_ip_Address
```

or:

```
MQSERVER=MY.SVRCONN/tcp/Server_ip_Address
export MQSERVER
```

8 Use the *amqsputc* and *amqsgetc* supplied with the client to make sure you can both put and get messages.

If all you are trying to do is to make sure you have a connection up and running, return codes such as 2035 (security issues) or 2085 (definitions) mean all is well – just check with your MQSeries administrator.

When to use it

I would suggest using this approach in the following cases:

- Testing the MQSeries Client environment is working with the server.
- A simple client/server environment where MQSeries is used as a transport layer and one does not want to go into complex definitions.
- An MQSeries development environment, where you don't want to have to install and maintain MQSeries on between five and ten developers' workstations.
- With the IBM-supplied *Support Pack MO71 – Remote Administrator*.
- When there is one and only one queue manager and no special features, ie channel exits are to be used.

Basically, from my experience, the best way to use MQSeries Client is in a small branch or a simple application development environment. As you can see from the above, it's easy to set up, and as long as you don't try to use it for things it is not meant for, it works well. Also, this

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

is the best way to find out whether or not the client connection works – you set things up, run the sample programs, and the answer will be immediately apparent!

Tips, and more

1 When running client code use a batch file to set up the environment. You can add other environment variables to these files, such as *MQCCSID*. Below are two sample batch files, one for DOS commands and one for K shell script.

RUNCLNT.CMD

```
rem Set the MQSERVER Var...
rem change the values of the IP address and run at will
set MQSERVER = SYSTEM.DEF.SVRCONN/tcp/127.Ø.Ø.1
set MQCCSID = 437
rem run your app from here...
```

RUNCLNT.SH

```
#!\bin\ksh
# Here is a sample shell script used to run client programs
# MQSERVER Varible  MQSERVER=Channel Name/Transport Type(TCP)/
     Connection Name
echo "Start script"
MQSERVER=SYSTEM.DEF.SVRCONN/tcp/127.Ø.Ø.1
export MQSERVER
echo $MQSERVER
# MQCCSID= Your code page
echo "End Script"
```

2 Try to use the same *CCSID* as the queue manager, but if need be don't forget client conversion is set by the local *MQCCSID* environment variable.

3 Remember that *MQSERVER* overrides the client connection tables so try not to set them in any global setting; it may cause problems to other users you may not be aware of.

4 If you encounter problems with privileges and security is not an issue (ie local development or a branch program using system user privileges), run the following command:

```
Alter channel (MY.SVRCONN) chltype (SVRCONN) mcauser('mqadmin_user)
```

where the user is an administrator user.

 © 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

CLIENT CONNECTION TO SERVER CONNECTION

Of all the ways to set up a client connection this is the most complex and cumbersome. But from the system point of view this is the best way to configure multiple queue managers on a client machine if you don't need to change the configuration very often, or you need to use special features that you don't want the programmers to be aware of (for example, encryption and security exits).

Getting started

The first four steps are the same as for the *MQSERVER* variable. But after that, it takes a turn for the worse! All the commands are detailed in the *def_clnt_svr.mqs* program, which is detailed after the following procedures.

1 Logon as an MQSeries administrator user (any user belonging to group *mqm*) to the MQSeries server.

2 Enter a command line mode (DOS in the Windows environment or shell in Unix).

3 Type **runmqsc**.

4 Type

```
define channel (MY.SVRCONN)      +
chltype  (SVRCONN)               +
trptype  (TCP)                   +
replace
define channel (MY.SVRCONN)      +
chltype  (CLNTCONN)              +
trptype  (tcp)                   +
conname  (Server_Ip_Address)     +
qmname    (DEV_SERVER)           +
replace
```

5 Type **end**.

6 Logon as an MQSeries administrator user (any user belonging to group *mqm*) to the MQSeries Client.

7 Copy the *amqclchl.tab* file to your client install root from:

– */var/mqm/qmgrs/qmname/@ipcc* on Unix machines

or

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

– *mq_root\qmgrs\qmname\@ipc*c on Windows NT or 2000 systems

where *qmname* is your queue manager name and *mq_root* is the high level qualifier for your data files install (usually *program file\MQSeries*).

8 Use the *amqsputc* and *amqsgetc* supplied with the client to make sure you can both put and get messages. Make sure you use the queue manager name or the code will not work. The reason for this is *SYSTEM.DEF.CLNTCONN* is one of the entries in the table, the *QMNAME* field there is blank, and so this is the default entry in the table.

9 If all you are trying to do is to make sure you have a connection up and running, return codes such as 2035 (security issues) or 2085 (definitions) mean all is well – just check with your MQSeries administrator.

DEF_CLNT_SVR.MQS

```
* This defines a set of both client and server connection channels.
* please make sure they have the same name. Also copy the
* amqclchl.tab file from the mq_home\qmgrs\queue_manager name\@ipcc
    directory.
* Server Connection
define channel (MY.SVRCONN)      +
chltype  (SVRCONN)               +
trptype  (TCP)                   +
replace
* Client Connection
define channel (MY.SVRCONN)      +
chltype   (CLNTCONN)             +
trptype   (TCP)                  +
conname   (192.168.1.34)         +
qmname    (DEV_SERVER)           +
replace
```

When to use it

I would suggest using this solution in the following cases:

- When there is a need for more then one queue manager to connect to and from the same run-time environment. (Remember, you can use a batch file to create your run-time environment.)

 © 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

- When there is a need for fields that do not appear in the *MQSERVER,* such as exits and tuning parameters.
- When there is a need for recovery or load balancing managed by queue manager name.

If you choose to use MQSeries Client as the base transport type for all client/server communications this is probably the best way to go about it. This will give you encryption if you need it, as well as load balancing and recovery through reconnect. (I know of a case where a customer chose to connect to their local queue manager with clients to achieve reconnect and encryption – I was against it.)

It is hard to maintain through the distribution of the *amqclchl.tab* files. In case of minor changes there is a need for extensive redistribution of the configuration files.

In case of widespread requirement for a client connection I would suggest writing a wrapper of some sort using LDAP and the *MQCONNX,* which is referred to in more detail later.

Tips, and more

1. If there is a need to use a default queue manager on a client machine I would suggest altering the *SYSTEM.DEF.CLNTCONN* and changing the *QMNAME* to a value which is not blank. Next, I would create a set of client and server connection channels with the name *DEFAULT* to be more explicit where the *QMNAME* is left as blank. It works really well!
2. Create all your client connection channels on the same machine if you can. Make sure it is not OS/390 or AS/400 to spare you the conversion and sort (as was detailed in the May issue of *MQ Update*). It's easiest to use Windows NT.
3. When using channel exits, remember clients can use security, send and receive exits (but not message exits) so take this into consideration when planning your implementation.

CONNECT FROM CODE USING MQCONNX (FOR THE VETERAN PROGRAMMER)

This method of operation is different from the two methods previously mentioned, as it moves control from the environment to the program itself, or in other words from the administrator to the programmer. This being so, I will simply add the program source for connecting to MQSeries using the *MQCONNX* call in the C programming language, and give an explanation as to where the code varies from regular MQSeries code.

I would not suggest trying to use this method unless you are proficient in writing MQSeries code. Furthermore, after coding this in both COBOL and C I can strongly recommend that you write in C!

This example demonstrates how to use the *MQCONNX* in a sample program.

CLNTTST

```
/* Program name: clnttst                                             */
/* Function:                                                         */
/*   clnttst is a C sample program which demonstrates how to use     */
/*   the following :                                                 */
/*     1) MQCONNX -  the connect call                                */
/*     2) MQCNO   -  The V2 structure that allows the client to      */
/*                   choose the qmgr without using MQSERVER or       */
/*                   the client connection channel table.            */
/*     3) MQCD    -  The Channel Defenition Structure, this          */
/*                   combined with the MQCD_CLIENT_CONN_DEFAULT      */
/*                   will be used to point to correct channel.       */
/*   Notes :                                                         */
/*   This program was linked with mqic32.lib.                        */
/*   I used the default protocol which is TCP/IP                     */
/*   clnttst has no input parms.                                     */
/*******************************************************************/
#include <cmqc.h>      /* MQ Header Files basic and                  */
#include <cmqxc.h>     /* Channel defs                               */
#include <stdio.h>     /* IO header                                  */
#include <string.h>    /* String functions                           */
int main()
{
    MQHCONN    hConn;        /* Connection handle                    */
    MQHOBJ     hObject;      /* Object handle                        */
    MQCHAR48   qmgrName;     /* Name of queue manager                */
    MQLONG     iCompCode;    /* Completion code                      */
    MQLONG     iReason;      /* Reason code                          */
```


© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

```
    MQLONG     iMessageLen;  /* Messsage Length                        */
    MQBYTE     buffer[1000];          /* Message data buffer        */
    MQLONG      iOpenOptions  ;
    /* These are the MQ Structs to be used in this program          */
    MQOD    mqod = {MQOD_DEFAULT};    /* Object description         */
    MQMD    mqmd = {MQMD_DEFAULT};    /* Message Descriptor         */
    MQPMO   mqpmo = {MQPMO_DEFAULT};  /* Put message options        */
    /* These are the special structures used         */
    MQCNO   mqcno = {MQCNO_DEFAULT} ; /* Connection options         */
    MQCD    mqcd  = {MQCD_CLIENT_CONN_DEFAULT};  /* Channel Defs    */
    /* Defined in cmqxc.h             */
 /* This part emulates data retrived from a database or LDAP Server. */
    /* I use the following:           */
    /*  1) qmgrName = ""              */
    /*  2) mqcd.ConnectionName = 192.168.1.34 = my dev server       */
    /*  3) mqcd.ChannelName = SYSTEM.DEF.SVRCONN = def server
           connection channel  */
    /*  can also retrive data such as queue name and such           */
    memset( qmgrName , 0 , MQ_Q_MGR_NAME_LENGTH) ;
    strncpy(mqcd.ConnectionName,
            "192.168.1.34",
            MQ_CONN_NAME_LENGTH);
    strncpy(mqcd.ChannelName,
            "SYSTEM.DEF.SVRCONN",
            MQ_CHANNEL_NAME_LENGTH);
 /* Point the MQCNO to the client connection definition and set its
          version to version 2  */
 /* or the ClientConnPtr will be ignored... */
   mqcno.ClientConnPtr = &mqcd;
   mqcno.Version = MQCNO_VERSION_2;
 /* MQCONNX - (Queue Manager Name, MQCNO, Connection handle, Completion
          code,
   Reason code ) */
    MQCONNX(qmgrName,
         &mqcno,
         &hConn,
         &iCompCode,
         &iReason);
 /* From here on it is a normal MQSeries program, but bear in mind, when
    receiving RC 2059 or 2009 check the AMQERR01.LOG for details.  */
    if (iCompCode == MQCC_FAILED)
    {
         printf("MQCONN failed with reason code %ld\n", iReason);
         return(iReason);
    }
    iOpenOptions   = MQOO_OUTPUT + MQOO_FAIL_IF_QUIESCING;
    strncpy(mqod.ObjectName, "SYSTEM.DEFAULT.LOCAL.QUEUE",
MQ_Q_NAME_LENGTH);
    /*  MQOPEN   */
    MQOPEN(hConn,
         &mqod,
```

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

```
        iOpenOptions,
        &hObject,
        &iCompCode,
        &iReason);
    if (iCompCode == MQCC_FAILED)
    {
        printf("MQOPEN failed with reason code %ld\n", iReason);
        return(iReason);
    }
    strcpy (buffer , "Hello from Client") ;
    memcpy (mqmd.Format , "MQSTR   " , 8 );
    iMessageLen = strlen(buffer)          ;
    MQPUT(hConn,
        hObject,
        &mqmd,
        &mqpmo,
        iMessageLen,
        buffer,
        &iCompCode,
        &iReason);
    if (iCompCode == MQCC_FAILED)
    {
        printf("MQPUT failed with reason code %ld\n", iReason);
        return(iReason);
    }
    /* MQCLOSE   */
    MQCLOSE(hConn, &hObject, MQCO_NONE, &iCompCode, &iReason);
    /* MQDISC - Do not forget to use DISC !!! This is Client Code   */
    MQDISC(&hConn, &iCompCode, &iReason);
    return(0);
}
#include <cmqc.h>
#include <cmqxc.h>
```

CINTTST.DSP

```
# Microsoft Developer Studio Project File - Name="clnttst" - Package
Owner=<4>
# Microsoft Developer Studio Generated Build File, Format Version 6.00
# ** DO NOT EDIT **
# TARGTYPE "Win32 (x86) Application" 0x0101
CFG=clnttst - Win32 Debug
!MESSAGE This is not a valid makefile.To build this project using NMAKE,
!MESSAGE use the Export Makefile command and run
!MESSAGE
!MESSAGE NMAKE /f "clnttst.mak".
!MESSAGE
!MESSAGE You can specify a configuration when running NMAKE
!MESSAGE by defining the macro CFG on the command line. For example:
!MESSAGE
```


© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

```
!MESSAGE NMAKE /f "clnttst.mak" CFG="clnttst - Win32 Debug"
!MESSAGE
!MESSAGE Possible choices for configuration are:
!MESSAGE
!MESSAGE "clnttst - Win32 Release" (based on "Win32 (x86) Application")
!MESSAGE "clnttst - Win32 Debug" (based on "Win32 (x86) Application")
!MESSAGE

# Begin Project
# PROP AllowPerConfigDependencies 0
# PROP Scc_ProjName ""
# PROP Scc_LocalPath ""
CPP=cl.exe
MTL=midl.exe
RSC=rc.exe
!IF  "$(CFG)" == "clnttst - Win32 Release"
# PROP BASE Use_MFC 2
# PROP BASE Use_Debug_Libraries 0
# PROP BASE Output_Dir "Release"
# PROP BASE Intermediate_Dir "Release"
# PROP BASE Target_Dir ""
# PROP Use_MFC 2
# PROP Use_Debug_Libraries 0
# PROP Output_Dir "Release"
# PROP Intermediate_Dir "Release"
# PROP Target_Dir ""
# ADD BASE CPP /nologo /MD /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D
"_WINDOWS" /D "_AFXDLL" /YX /FD /c
# ADD CPP /nologo /MD /W3 /GX /O2 /I "d:\MQSeries\tools\c\include" /D
"WIN32" /D "NDEBUG" /D "_AFXDLL" /D "_MBCS" /D "_CONSOLE" /YX /FD /c
# ADD BASE MTL /nologo /D "NDEBUG" /mktyplib203 /win32
# ADD MTL /nologo /D "NDEBUG" /mktyplib203 /win32
# ADD BASE RSC /l 0x40d /d "NDEBUG" /d "_AFXDLL"
# ADD RSC /l 0x40d /d "NDEBUG" /d "_AFXDLL"
BSC32=bscmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
LINK32=link.exe
# ADD BASE LINK32 /nologo /subsystem:windows /machine:I386
# ADD LINK32 mqm.lib /nologo /subsystem:console /machine:I386 /
libpath:"d:\MQSeries\tools\lib"
!ELSEIF  "$(CFG)" == "clnttst - Win32 Debug"
# PROP BASE Use_MFC 2
# PROP BASE Use_Debug_Libraries 1
# PROP BASE Output_Dir "Debug"
# PROP BASE Intermediate_Dir "Debug"
# PROP BASE Target_Dir ""
# PROP Use_MFC 2
# PROP Use_Debug_Libraries 1
# PROP Output_Dir "Debug"
# PROP Intermediate_Dir "Debug"
```

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

```
# PROP Ignore_Export_Lib 0
# PROP Target_Dir ""
# ADD BASE CPP /nologo /MDd /W3 /Gm /GX /ZI /Od /D “WIN32” /D "_DEBUG" /
D "_WINDOWS" /D "_AFXDLL" /YX /FD /GZ /c
# ADD CPP /nologo /MDd /W3 /Gm /GX /ZI /Od /I
"d:\MQSeries\tools\c\include" /D "WIN32" /D "_DEBUG" /D "_AFXDLL" /D
“_MBCS" /D "_CONSOLE" /YX /FD /GZ /c
# ADD BASE MTL /nologo /D "_DEBUG" /mktyplib203 /win32
# ADD MTL /nologo /D "_DEBUG" /mktyplib203 /win32
# ADD BASE RSC /l 0x40d /d "_DEBUG" /d "_AFXDLL"
# ADD RSC /l 0x40d /d "_DEBUG" /d "_AFXDLL"
BSC32=bscmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
LINK32=link.exe
# ADD BASE LINK32 /nologo /subsystem:windows /debug /machine:I386 /
pdbtype:sept
# ADD LINK32 mqic32.lib /nologo /subsystem:console /debug /machine:I386
/pdbtype:sept /libpath:"d:\MQSeries\tools\lib"
!ENDIF
# Begin Target
# Name "clnttst - Win32 Release"
# Name "clnttst - Win32 Debug"
# Begin Group "Source Files"
# PROP Default_Filter "cpp;c;cxx;rc;def;r;odl;idl;hpj;bat"
# Begin Source File
SOURCE=.\clientcode.c
# End Source File
# End Group
# Begin Group "Header Files"
# PROP Default_Filter "h;hpp;hxx;hm;inl"
# End Group
# Begin Group "Resource Files"
# PROP Default_Filter "ico;cur;bmp;dlg;rc2;rct;bin;rgs;gif;jpg;jpeg;jpe"
# End Group
# End Target
# End Project
```

We need to include two ‘include’ files:

1 *cmqc.h* – the regular ‘include’ to be used with MQSeries.

2 *Cmqxc.h* – the‘include file’ that contains the Channel definition structure.

```
MQHCONN      hConn;
MQCHAR48     qmgrName;
MQLONG         iCompCode;
MQLONG         iReason;
```


© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

These variables are needed for connecting to MQSeries and would be used in any MQSeries code. New in this code, however, are the following structures:

```
MQCNO        mqcno = {MQCNO_DEFAULT};
MQCD         mqcd  = {MQCD_CLIENT_CONN_DEFAULT};
```

MQCNO mqcno = {MQCNO_DEFAULT}; is used for *MQCONNX* for both client and server code. The difference is in a member called *ClientConnPt*r, which is the pointer to the MQCD structure.

MQCD mqcd = {MQCD_CLIENT_CONN_DEFAULT}, the channel definition structure, is where things really happen. It is used to build one's own client connection channel definitions. In this sample I use defaults for almost everything and use constants to populate the structure. If you choose to use this method of implementing MQSeries Client I would suggest using LDAP or a database; furthermore, I suggest you don't leave out fields not used in this sample, such as:

- Exit information.
- Tuning information.
- Transport type relevant information.

In this part I populate the minimal fields to achieve a connection.

I use the default queue manager name:

```
memset( qmgrName , Ø , MQ_Q_MGR_NAME_LENGTH  ; (
```

Because I use the default transport type (TCP/IP) I do not state it, but just place an IP address as the connection name:

```
strncpy(mqcd.ConnectionName,"192.168.1.34"
, MQ_CONN_NAME_LENGTH ) ;
```

I use a channel name I know exists on the server machine:

```
strncpy(mqcd.ChannelName,"SYSTEM.DEF.SVRCONN", MQ_CHANNEL_NAME_LENGTH);
```

The following code is important for the whole thing to work. I pass the pointer to the *MQCD* structure, and state the *MQCNO* is of Version 2 or else the pointer field will not be referenced.

```
mqcno.ClientConnPtr = &mqcd;
mqcno.Version = MQCNO_VERSION_2;
I now call the MQCONNX
```

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

```
MQCONNX (qmgrName,
         &mqcno,
         &hConn,
         &iCompCode,
         &iReason);
```

Make sure to check the return code after this call. If you receive reason code 2059 or 2009 please look at the *AMQERR01.LOG* for further details.

From here it is code as usual, but do not forget this; it is very important to code *MQDISC* in client code. I usually code the *MQCONNX* right along with the *MQDISC,* like *malloc* and *free,* just to be on the safe side.

```
    MQDISC(&hConn, &iCompCode, &iReason);
```

Prerequisites

Make sure of the following.

- You have MQSeries Client Development Tool Kit installed.
- You link your program with the client libraries.
- You know the following:
  - the connection name = IP address
  - the *SVRCONN* channel name; you can use the *SYSTED.DEF.SVRCONN*, but I would not make a habit of it
  - last but not least, the queue manager name.
- The user you choose is authorized to use MQSeries or you will get 2035 for every call you issue.

When to use it

I would suggest using this approach in the following cases:

- When you write embedded MQSeries code as part of an integration suite. This will ease the pain of integrating the product and will save the user a lot of configuration hassle. The best example of such code is the MQSeries Explorer using such a connection.
- If applications have multiple queue managers to connect to, and

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

managing the *amqclchl.tab* files is a problem (and it usually is), I would use this code for both load balancing and controlling the connections to the different queue managers. For this purpose I would suggest using either LDAP or a database to populate the *MQCD* in the program.

- If you build an MQSeries wrapper of one sort or another and want to make it available as both server and client code. Again, I would use LDAP or a database to hold the client definitions.
- When your programmers are proficient in using MQSeries and your code needs to have a short set-up time and must work on many types of systems where you cannot rely on the environment to supply the relevant data.

To summarize

This way is probably the best way to go about using clients for a multi-queue manager environment. But it requires previous planning, some central way to hold the definitions, and, last but not least, ample experience with MQSeries code to make sure you don’t get all the negative side-effects of moving data usually managed by the administrators to the control of the programmers (ie having code connecting and disconnecting all over the place and you don’t really now how and where to control it).

Tips, and more

- Define a channel for all your programs to use and have a *mcauser* defined, which limits the programs from doing things they are not supposed to do, such as creating their own definitions.
- Maintain your connection information on some central site, such as LDAP or a database.
- Make sure you leave ample room for definitions: basically, support all the fields in the client definition channel.
- Don’t forget the *MQCNO* version!!! Your code won’t work and you will not understand why (I have seen it in three separate cases where customers tried using this code).

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

APPENDIX 1: SETTING UP A SIMPLE MQSERIES ENVIRONMENT ON WINDOWS NT 4.0.

In this section I will show you how to create a basic queue manager on the Window NT/2000 system. I am assuming that MQSeries is already installed.

1 First, enter MQSeries Explorer (see Figure 1).

2 Right-click on 'queue managers' and select 'new->queue manager' (see Figure 2).

3 Enter the queue manager name and check 'make this the default queue manager' (see Figure 3).

4 Hit 'next'(see Figure 4).

5 Check both boxes (see Figure 5).

6 Select a port (1414 is best...) and click 'finish' (see Figure 6).

7 If you see the guy shown in Figure 7 you're the proud owner of a queue manager – use it wisely!

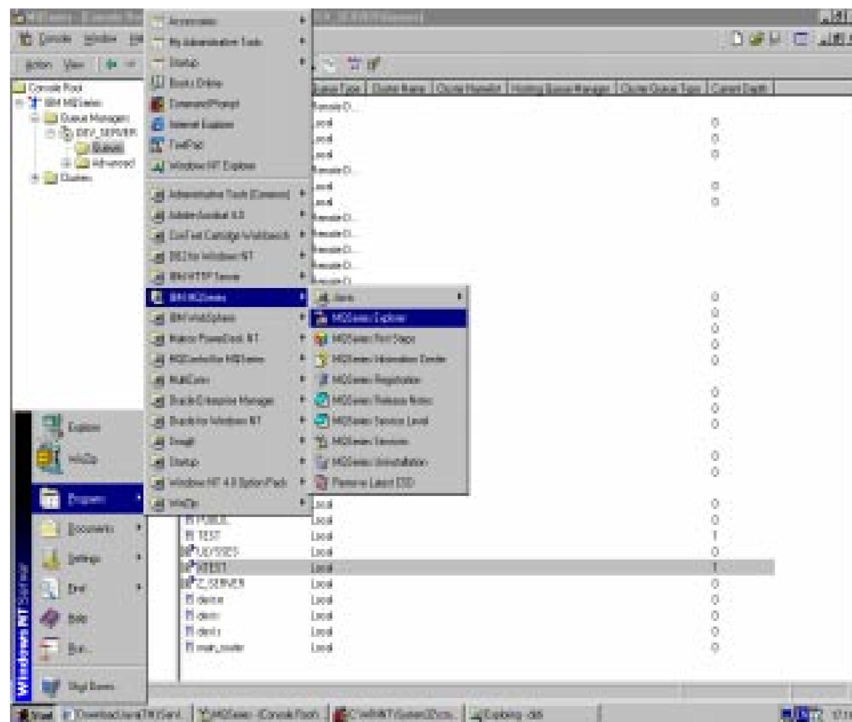

Figure 1: Enter MQSeries Explorer

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

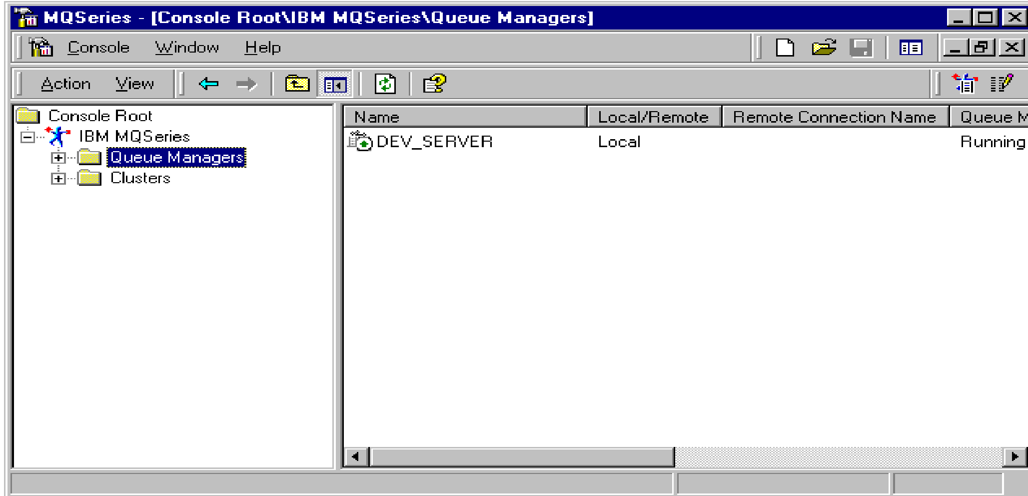

Figure 2: Right-click on 'queue managers' and select 'new->queue manager'

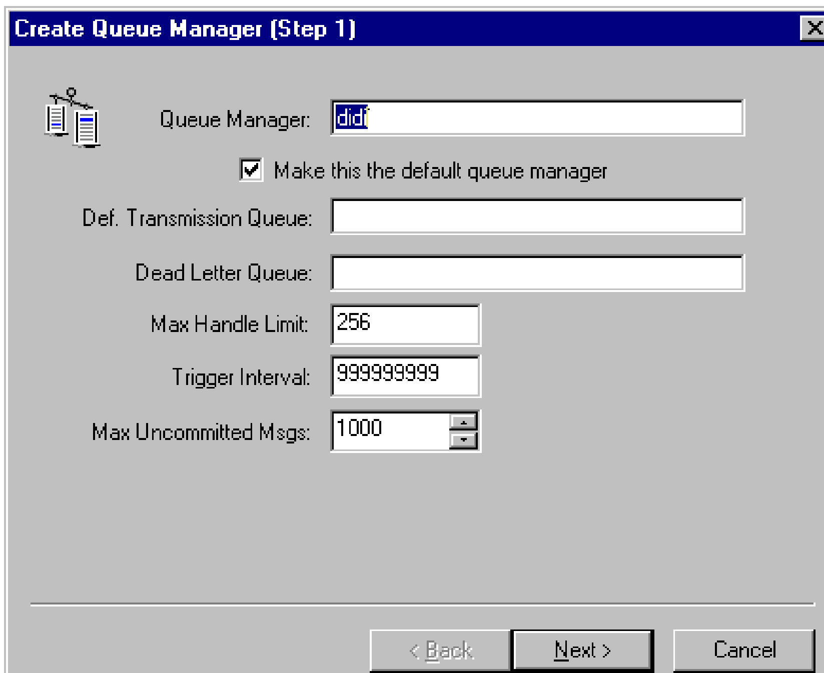

Figure 3: Enter the queue manager name and check 'make this the default queue manager'

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

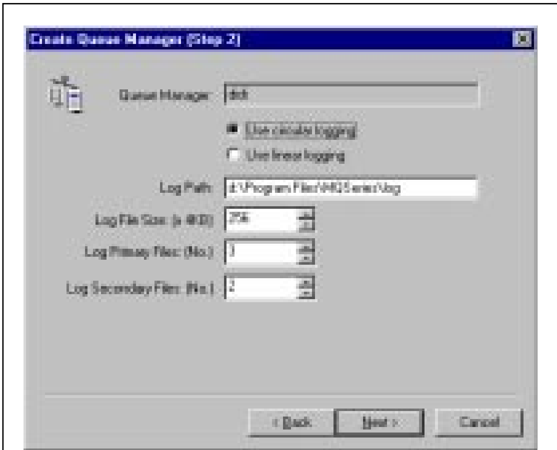

Figure 4: Hit 'next'

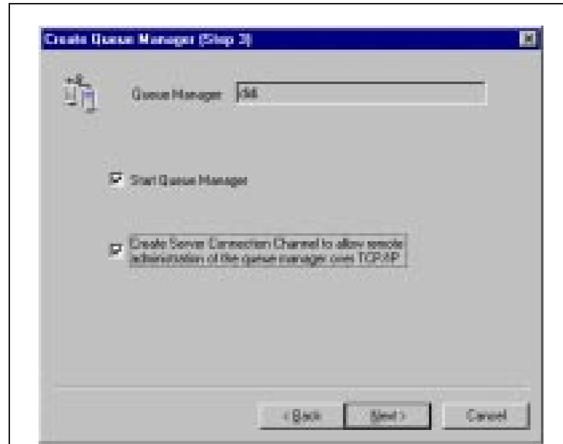

Figure 5: Check both boxes

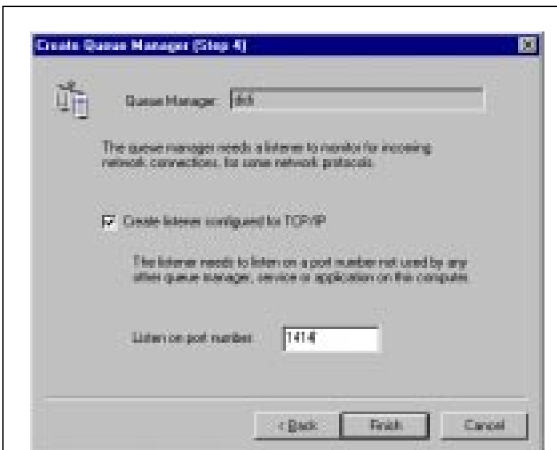

Figure 6: Select a port (1414 is best...) and click 'finish'

Figure 7: You're the proud owner of a queue manager

Didi Dotan
Technological Solutions Consultant
Multiconn (Israel)
© Xephon

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

MQSeries and firewalls

INTRODUCTION

As B2B commerce becomes more pervasive an increasing number of companies are finding they need to integrate their Web sites with back-office systems. MQSeries provides an ideal solution to this requirement.

In most scenarios, one or more firewalls will be employed to protect your own enterprise network from the Internet or another company's network. These firewalls may also provide Network Address Translation (NAT) to prevent situations where, for example, a TCP/IP address on your own network conflicts with an address on the other network. Consider the problems of providing a link between your company and a partner if you both use the IP address range of 192.168.xxx.yyy – both networks could have devices with the same IP address (eg 192.168.12.34).

IBM provides a SupportPac extension to the base product that can assist in connecting MQSeries over the Internet. The *MQIPT (MQSeries Internet Pass-Thru) SupportPac* allows channel communication to occur across firewalls by tunnelling the protocols through HTTP, or SSL (secure sockets), or by acting as a proxy. The SupportPac runs on AIX, Sun Solaris, HP-UX, Microsoft Windows NT, and Windows 2000, and provides an administrative graphical user interface for managing one or more MQIPT servers. For more information on the MQIPT see *SupportPac MS81* at: *http://www.ibm.com/software/ts/mqseries/txppacs/ms81.html*.

This article deals with the issues surrounding the use of MQSeries across firewalls without the *MQIPT SupportPac*. I will discuss my own experiences in setting up MQSeries running across firewalls, and also, briefly, some issues concerned with using MQSeries and MQSeries clusters that are affected by NAT firewalls.

MQSERIES AND TCP/IP

MQSeries intercommunication operates just like any other TCP/IP-

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

based application when carrying out the same task. A host using TCP/IP sockets selects a free local port at random and initiates a connection to a fixed TCP/IP port on a remote host. The remote host uses TCP/IP sockets to listen on the fixed port for these incoming connections.

TCP/IP sockets

Sockets are used by TCP/IP applications for conversations. Each socket is identified by four components: the source and destination IP addresses and port numbers. For example, a source on 192.168.12.34(1234) talking to a destination on 192.168.56.78(1414) would be one socket and a source on 192.168.9.10(3456) talking to the same destination on 192.168.56.78(1414) would be another.

Note that the destination IP address and port numbers are the same since the listener is only listening on one port number. Only one connection can exist with a specific 'source IP (port) address to destination IP (port) address' signature.

FIREWALLS AND TCP/IP

Firewall products typically apply rules to the four components of sockets to allow or restrict TCP/IP conversations. All four components are used to uniquely identify a conversation and different combinations of the components are used to allow or restrict specific IP address and/or port number combinations.

Typical firewall architectures look something like the diagram shown in Figure 1. Your specific implementation may be more complicated, depending on your network requirements. For example, some sites may choose to use two firewalls: one between the Internet and the Web server and a separate one between the Web server and your internal network.

Using two firewalls can increase security so that if somebody compromises the first firewall they still need to compromise the second to get onto your internal network. The principles of configuring MQSeries in a multi-firewall configuration do not change, however.

A firewall may only allow connections from the Internet to connect to port 80 on a Web server (ie HTTP). Applications on the Web server

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

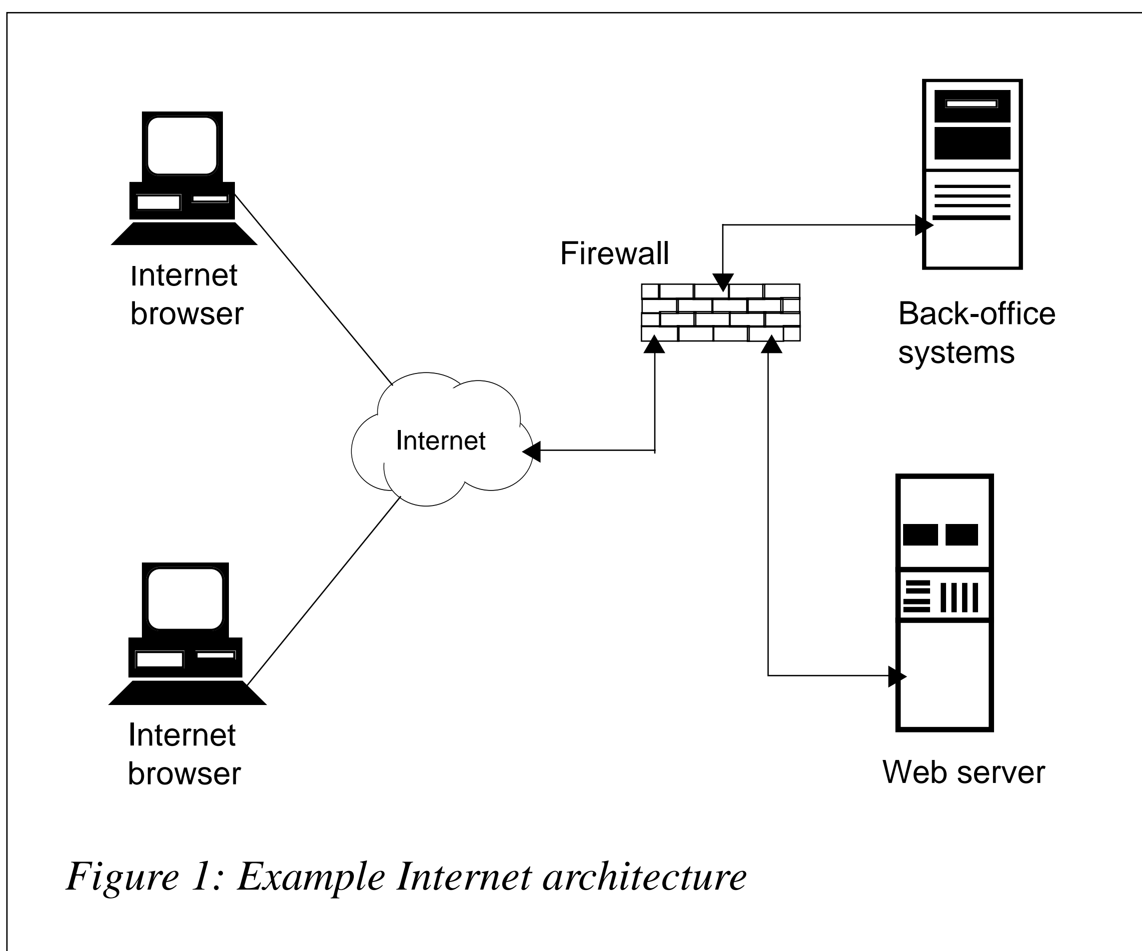

Figure 1: Example Internet architecture

may need to communicate with back-office servers in your internal network.

The firewall will generally restrict access to your internal network directly from the Internet. However, if somebody compromises the security of your Web server and manages to gain access, they can then use the Web server as a base to begin attacking your internal network, unless the firewall restricts access from the Web server to your internal network.

If your Web server is running MQSeries to allow intercommunication with your back-office systems the firewall will need to be configured to allow MQSeries channel agents to communicate over TCP/IP connections.

MQSERIES CHANNELS AND FIREWALLS

In order to configure firewalls to allow MQSeries channel agents to

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

communicate, it is important to understand the sequence of events during channel start-up. This is as follows:

1 The sending MCA selects a free port at random (greater than 1023 – these are reserved for services such as HTTP and FTP) and requests a connection to the MQSeries listener at the receiving end.

2 The MQSeries listener deals with the request by starting a receiver MCA for the channel.

3 The receiving MCA initiates a connection to the sending MCA using local port 1414 and the randomly-selected remote port on the sending end. In essence, it calls the sender back.

4 The original conversation is only used to request that the receiving MCA call the sending MCA back.

This sequence of events should apply to all channel types.

In the following examples I will discuss two hosts. The host outside the firewall I will call *outside_host* and it will run queue manager *OUTQM*. The host inside the firewall (on your internal network) I will call *inside_host* and it will run queue manager *INQM*. This example also assumes the default port 1414 is used by the MQSeries listener on both hosts. I will show the rules required to allow the channels *OUTQM.TO.INQM* and *INQM.TO.OUTQM* to be started. Figure 2 illustrates the required intercommunication.

In order for a channel from *outside_host* to *inside_host* (*OUTQM.TO.INQM*) to start, the following two firewall configurations are needed:

- Allow the *outside_host*, on any port greater than 1023, to initiate a connection to the *inside_host* on port 1414 only. This allows step one above in the channel start-up to occur.
- Allow the *inside_host*, on port 1414 only, to initiate a connection to the *outside_host* on any port greater than 1023. This allows step three above in the channel start-up to occur.

In order for a channel in the opposite direction (ie *INQM.TO.OUTQM*) to start, the following two additional firewall configurations are needed:

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

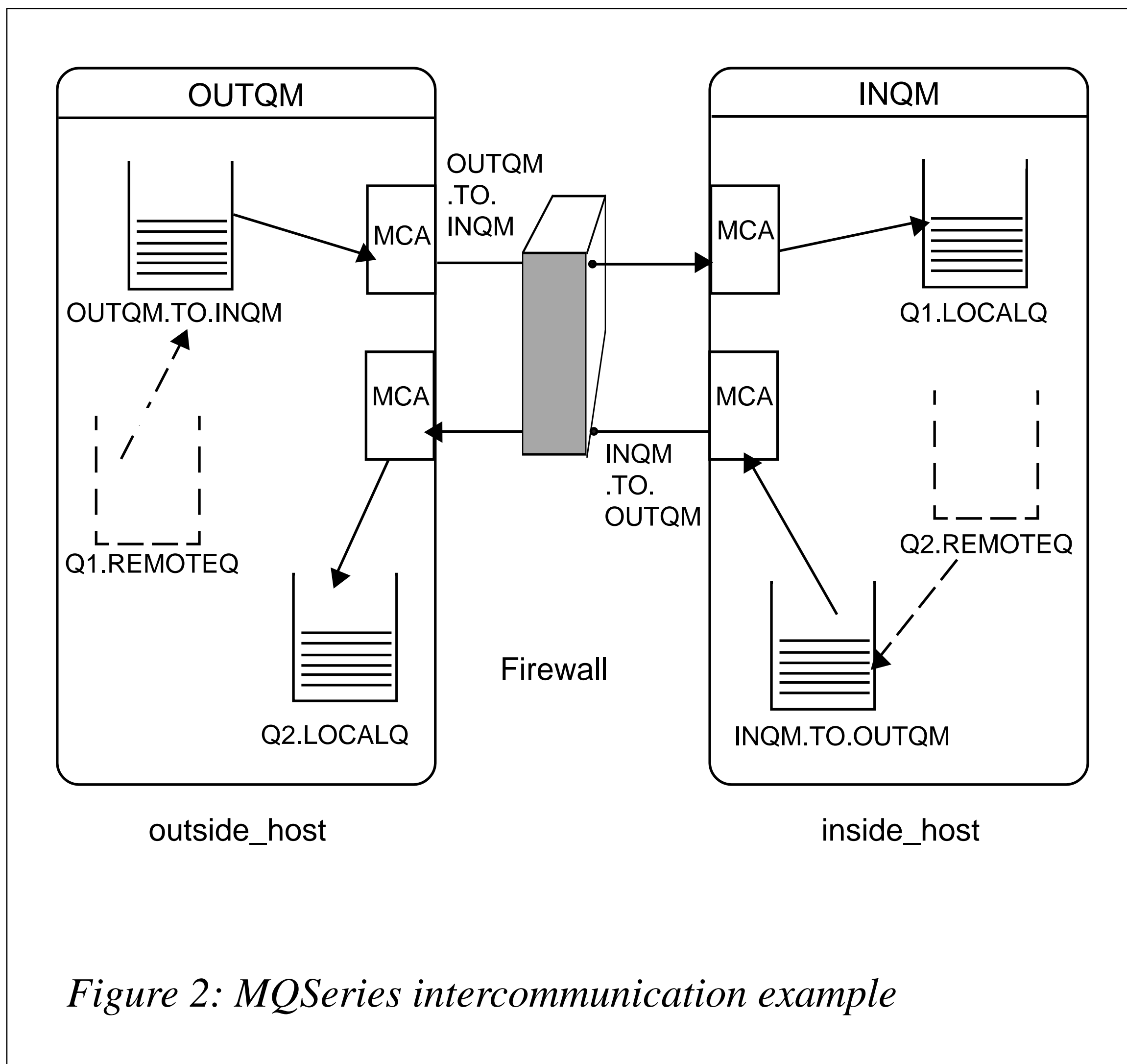

Figure 2: MQSeries intercommunication example

- Allow the *inside_host*, on any port greater than 1023, to initiate a connection to the *outside_host* on port 1414 only. This allows step one above in the channel start-up to occur.
- Allow the *outside_host*, on port 1414 only, to initiate a connection to the *inside_host* on any port greater than 1023. This allows step three above in the channel start-up to occur.

You should note that these two rules are the same as the first two, except that *inside_host* and *outside_host* have switched places. Therefore, to allow channels to start in both directions, the four rules shown in Table 1 are required. Depending on the functionality of your firewall, you may be able to reduce it to two rules (see Table 2).

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

Host	Port	Host	Port	Initiator
outside_host	>1023	inside_host	1414	outside_host
outside_host	>1023	inside_host	1414	inside_host
inside_host	>1023	outside_host	1414	inside_host
inside_host	>1023	outside_host	1414	outside_host

Table 1: Four rules to allow channels to start in both directions

Host	Port	Host	Port	Initiator
outside_host	>1023	inside_host	1414	Both
inside_host	>1023	outside_host	1414	Both

Table 2: Reducing the rules to two, depending upon firewall functionality

MQSERIES AND NETWORK ADDRESS TRANSLATION

Your MQSeries queue manager host inside your firewall corporate LAN may use a publicly-assigned private IP address (eg those beginning 192.168.x.y). The queue manager outside your firewall will be unable to communicate with it unless it is given a ‘real’ IP address.

One way to achieve this is by using Network Address Translation (NAT). This is where your firewall (or a network router) converts the internal IP address of your MQSeries host to a ‘real’ IP address for TCP/IP packets destined for the host outside your firewall. The ‘real’ IP address is also converted to your private IP address for TCP/IP packets destined for the host inside the firewall.

It is always a good idea to give your host names rather than IP addresses when defining channels, and if you use NAT it becomes particularly important.

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

In order to get MQSeries clusters working correctly across a NAT firewall, host names must be used. This is because some channels will be auto-defined by the queue manager based on actual definitions stored on other queue managers in the cluster. Since a remote host's IP address is different depending on which side of the firewall you are on, IP addresses cannot be used for cluster channel definitions.

For example, if you defined a cluster receiver channel (*TO.INQM*) with a connection name of 192.168.12.34, it would not work from outside the firewall because the IP address required is a 'real' IP address of (for example) 123.45.67.89.

Using 123.45.67.89 does not work from other queue qanagers in the cluster inside the firewall; they require the original definition using 192.168.12.34. Thus, using a connection name of *inside_host* will solve this problem. Just define *inside_host* in the *hosts* file outside the firewall as 123.45.67.89 and as 192.168.12.34 inside the firewall so that the channel definition is the same, regardless of the location of the queue manager.

CONCLUSION

MQSeries channels operate like other TCP/IP connections except that the receiving MCA 'calls the sender back', requiring that connections can be initiated in both directions.

Once you understand how channels start it is a relatively simple configuration task to allow those channels to run successfully over a firewall link.

With this knowledge, and careful consideration of the fact that host IP addresses may differ when working with Network Address Translating firewalls, it is even possible to get MQSeries clusters with auto-defined channels up and running.

John Scott
Senior Middleware Technical Specialist
Argos Ltd (UK)

© John Scott

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

Using MQ clusters for large communications buffers

INTRODUCTION

This article presents an idea for using MQSeries and clusters for deploying large communications buffers in a high-workload environment. It provides examples of specific strategies that may be used to overcome MQSeries limitations on queue space.

THE REQUIREMENT

MQSeries is generally a good solution for creating an asynchronous message link between two applications. However, as MQSeries pervades the IT industry its limitations are being stretched to capacity by some industry requirements – with specific reference to throughput and queue space. This article will not deal with the problematic aspects of using MQSeries in high-workload environments. Its intent is to suggest the use of MQSeries and clustering in order to achieve large asynchronous buffers that are needed in a high-end environment.

Consider a chain of store-and-forward applications: A→B→C→…→*N*. Each application in the chain is a link that receives messages from the previous link, does some unique processing, and forwards the processed message to the next link. The message rate between each pair of applications is very high (ie in the scale of 100MB per minute).

Buffer space is the area where messages reside on their way from the sender to the receiver application. Usually, the number of messages in the buffer space is small, because the system is tuned so that the receiver's message rate is the same as (or higher than) the sender's message rate. But if a malfunction occurs in the receiver application, inhibiting or delaying it from getting messages from the queue, the buffer space begins to fill up. If the problem persists, the buffer space eventually becomes full, inhibiting the sender from putting messages and possibly expanding the malfunction to the area of the sender application.

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

This is similar to a traffic jam caused by an obstruction of a highway exit: the traffic builds up behind the obstruction and the road must be cleared before the whole highway becomes blocked.

The time between the receiver's malfunction and the sender's malfunction (sometimes called 'breathing time') is a major factor in the design of such systems. This buffer space should allow enough breathing time for the correction and recovery of a problem in a link without causing the problem to expand up the chain.

SIMPLE DEPLOYMENT

Using MQSeries, buffer space translates directly to queue space. Except for mainframe environments, the maximum capacity of a single MQSeries queue in the current version is 2GB. A simple deployment consisting of a single queue manager and a single queue between the sender and the receiver applications (see Figure 1) will give 20 minutes of breathing time. This may be sufficient under certain circumstances but complex problems may take hours to resolve and recover. Furthermore, in time, as the data that flows in the chain becomes more complex (say XML replaces fixed-width binary fields), throughput tends to grow considerably and so does the required buffer space, so 2GB is hardly enough.

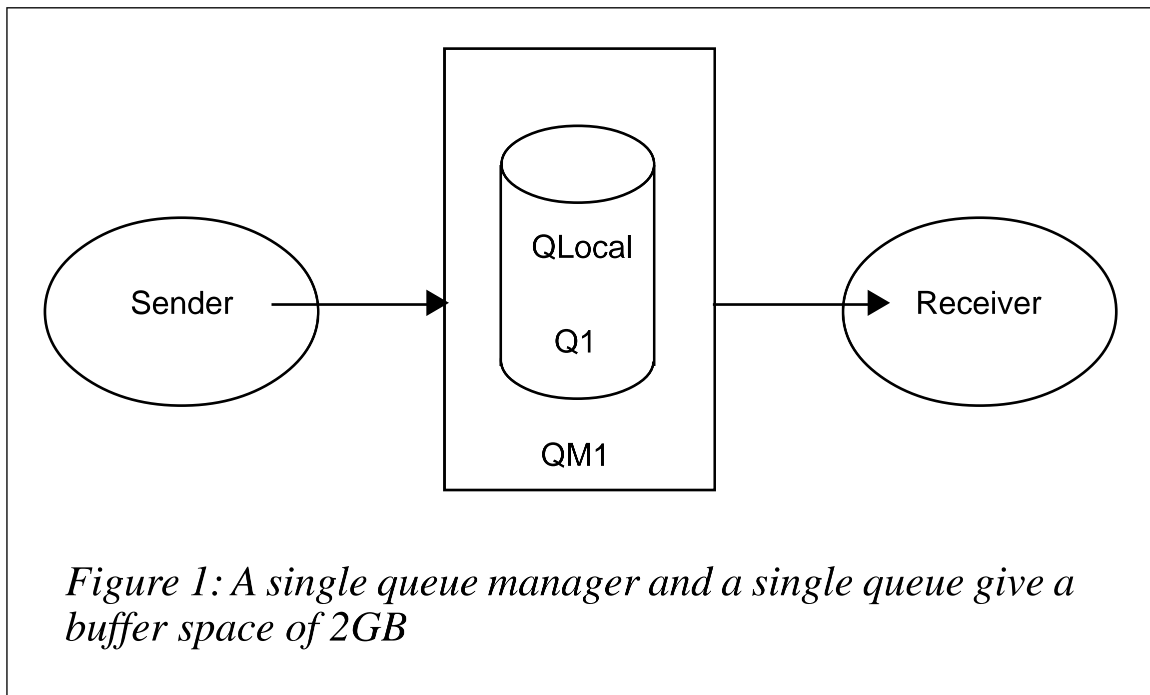

Figure 1: A single queue manager and a single queue give a buffer space of 2GB

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

The use of MQSeries clusters can lift the 2GB limit. Several physical queues on different queue managers can be pooled together to increase the buffer space. The sender application opens the queue specifying *MQOO_BIND_NOT_FIXED*, and each message is sent to a different 'instance' of the queue in the cluster (see Figure 2) using MQ's 'round-robin' algorithm.

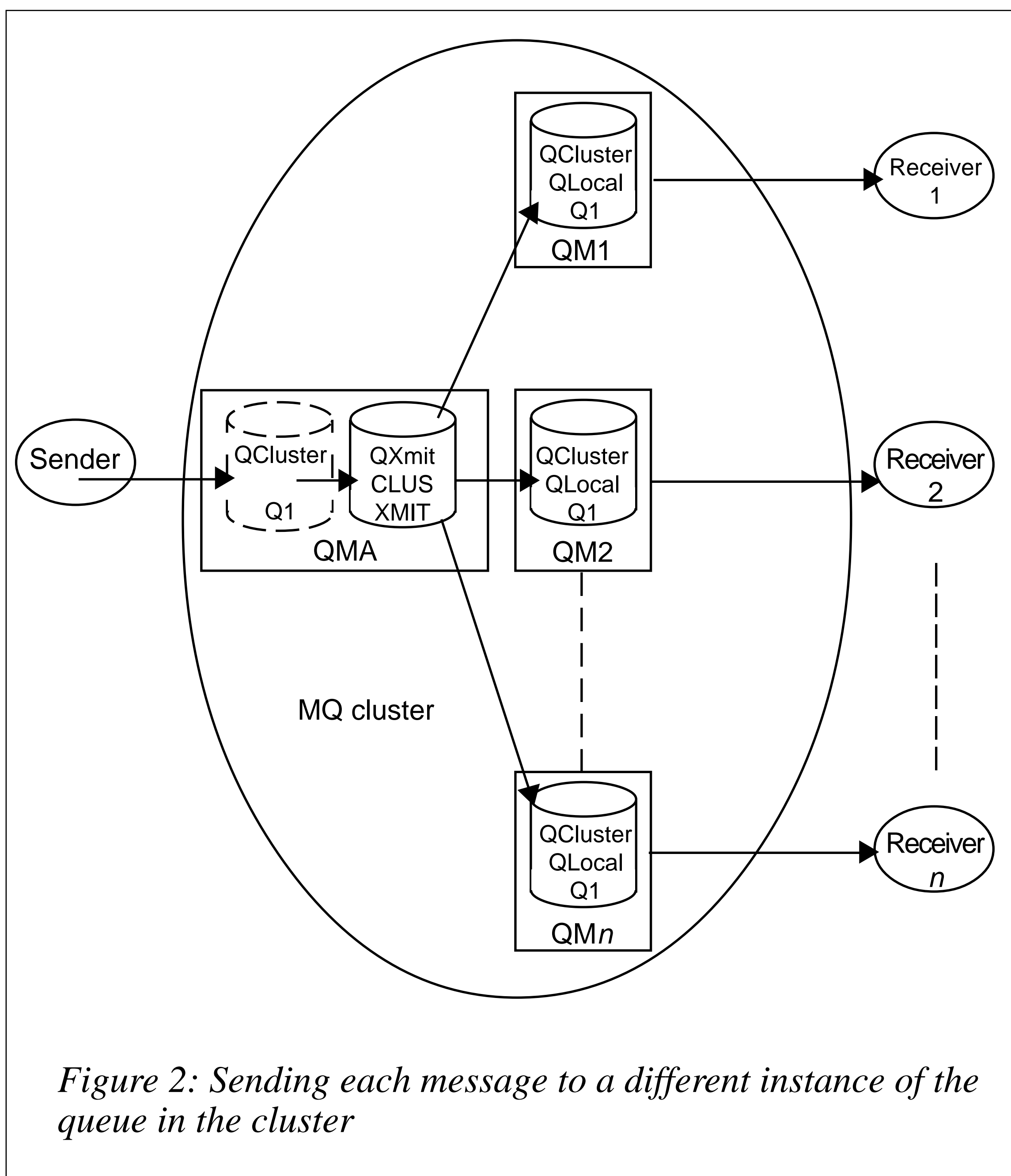

Figure 2: Sending each message to a different instance of the queue in the cluster

 © 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

In Figure 2, the sender connects to *QMA*, which only has a cluster queue definition of *Q1*. *Q1* is also defined as a local queue in *QM1..QMn*. Messages are stored in the cluster transmission queue (*SYSTEM.CLUSTER.TRANSMIT.QUEUE*) of QMA and in the local instances of *Q1* in *QM1..QMn*, giving a total buffer space of (n+1)*2GB.

n instances of the receiver application are required in order to read from all instances of *Q1*.

This lifts the 2GB limit at the cost of added complexity. You can have multiple queue managers in the cluster, each adding 2GB of buffer space and 20 more minutes of breathing time. For example, for three hours of breathing time you would have nine queue managers (eight with local queues plus one that the sender connects to).

The problem with this deployment lies in the fact that clustering only delivers 'push' load balancing – only the sender's puts are spread throughout the queue instances. The receiver must get messages from a specific instance of the queue on a specific queue manager. Since a single thread or process can only connect to a single queue manager at a time, the receiver application needs to launch as many receiver processes as there are instances of the queue. Having more than one process and reading from more than one queue is awkward at best, for the reasons outlined below.

- Internal synchronization of the separate receiver processes is needed. For example, if message priority is used, each separate queue has a separate prioritized queue, and a prioritization sync between the queues must be implemented.
- All receiver processes must be running. If a process halts, messages begin to accumulate in the queue that it serves.
- In case the receiver's design enforces a single process to receive messages (which happens frequently), the deployment may be impossible.

CLUSTERED DEPLOYMENT WITH FRONT ENDS

Another approach, which is illustrated in Figure 3, uses only one front-end queue manager per application and multiple back-end queue managers for the buffer space.

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

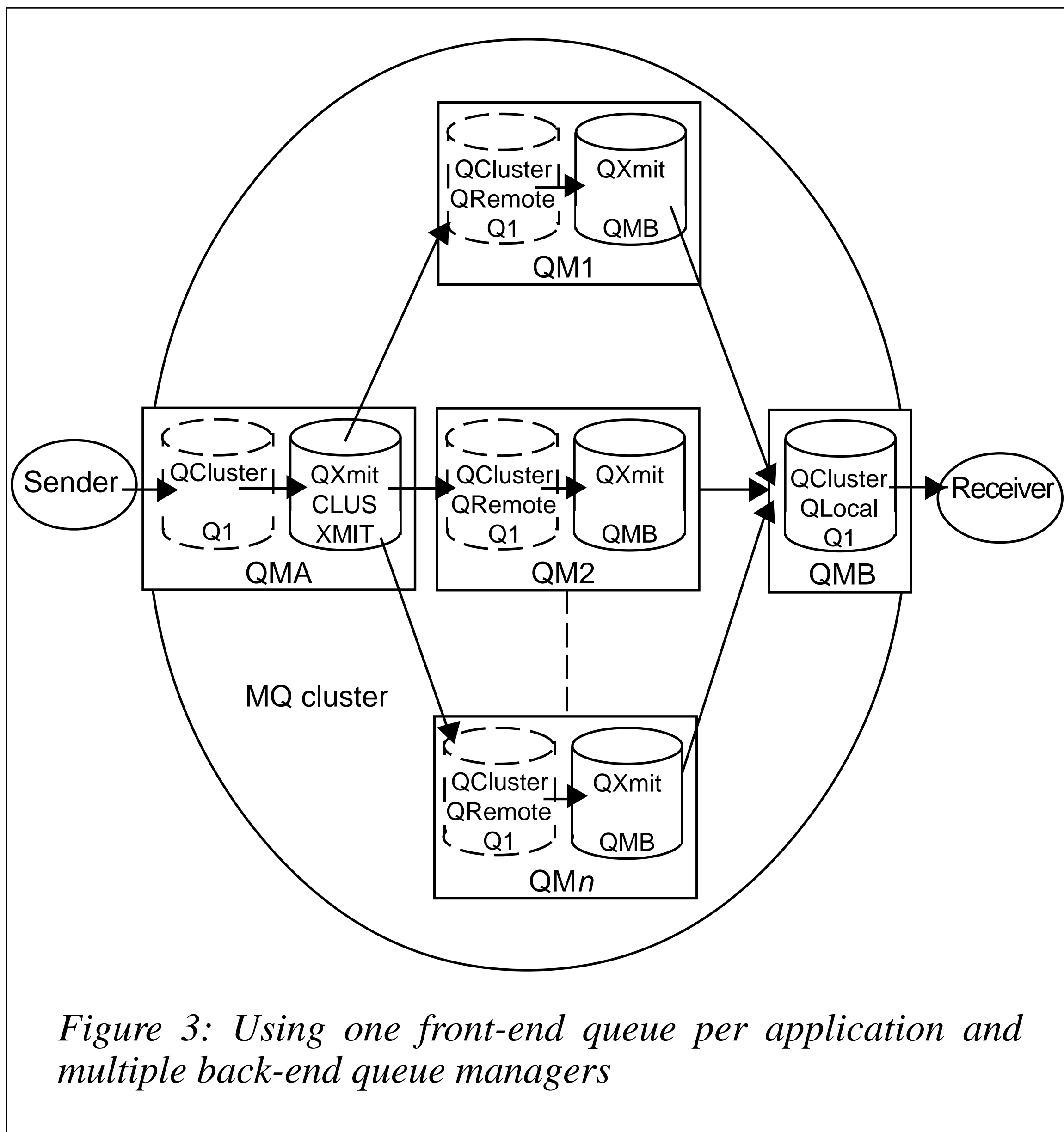

Figure 3: Using one front-end queue per application and multiple back-end queue managers

Figure 3 shows *QMA* and *QMB* as front-ends and *QM1..QMn* as back-ends. It is similar to Figure 2, except that *QM1..QMn* do not have a local queue *Q1* defined. Instead, a remote queue – *Q1* – is defined, which directs messages to *QMB* through a transmission queue. The receiver gets messages from the local queue *Q1* in *QMB*. This gives a buffer space of (n+2)*2GB and enables the receiver to have only one instance that reads from a single queue.

This eliminates the need for a multi-process receiver, since all messages get routed from the buffered back-end to the receiver's front-end.

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

The obvious drawback of this solution is its complexity. However, since more than one queue can be defined in each queue manager, a single deployment can be used for the entire chain of applications (see Figure 4). This can actually reduce the management overhead in certain environments.

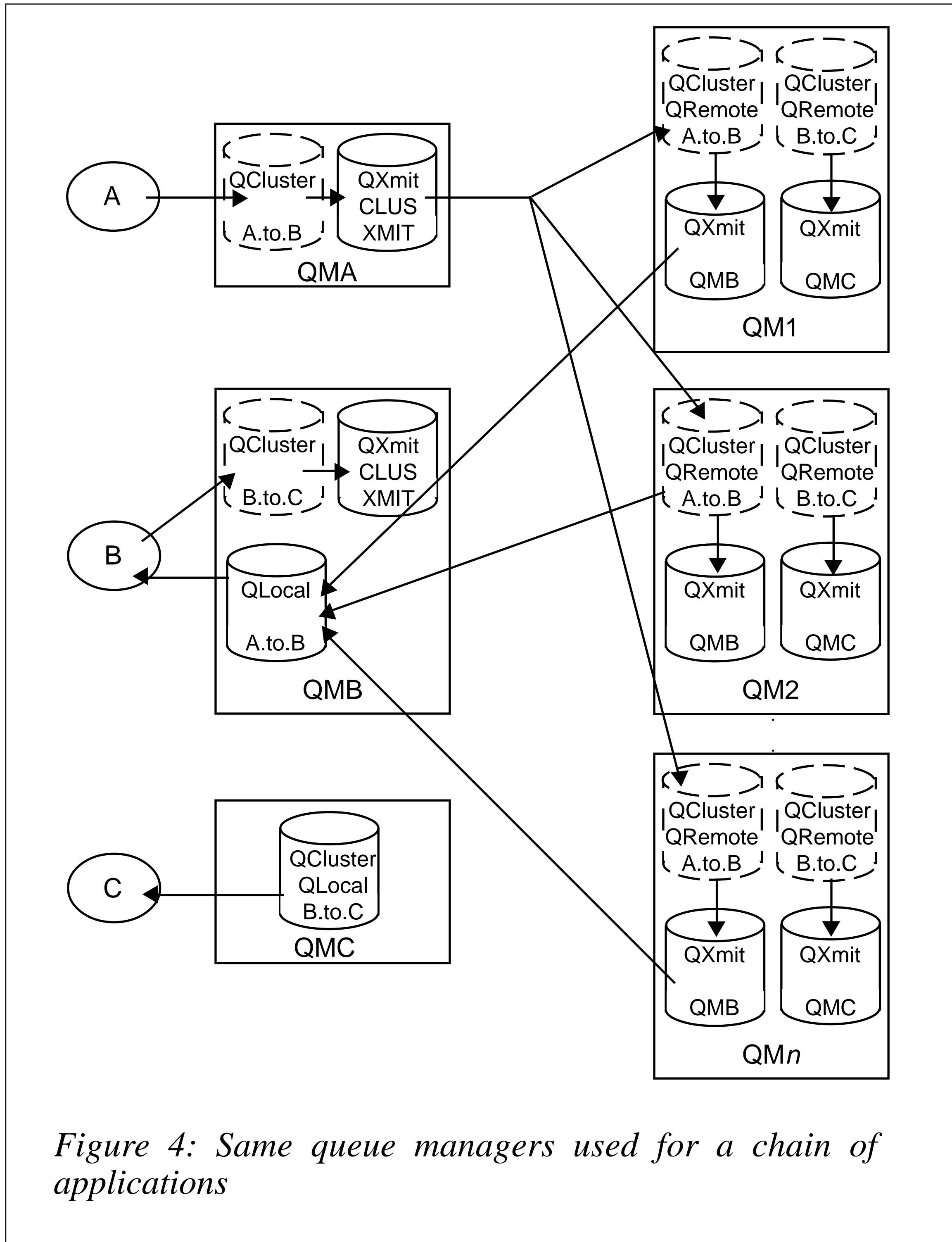

Figure 4: Same queue managers used for a chain of applications

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

Figure 4 shows that the same queue managers are used for a chain of applications: A→B→C. For each link in the chain a front-end queue manager is created (*QMA..QMC*). Also, in every back-end queue manager (*QM1..QMn*), definitions are made for remote queues, cluster queues, and transmission queues to all the front-end queue managers that need to receive messages. At the front-end queue manager a cluster queue is defined for every link that needs to be sent to, and a local queue is defined for every link that needs to be received from.

In Figure 4, A sends, C receives, and B does both. For clarity, arrows were only drawn for the link between A→B.

This deployment gives a buffer space of $(n+2m)$*2GB, where n is the number of back-end queue managers, and m is the number of links in the chain of applications.

IMPLEMENTATION

As previously stated, this article does not deal with specific implementation issues. It merely suggests an idea for an MQSeries deployment that lifts the queue space limit. Implementing such a deployment for production environments does not seem feasible at the moment, primarily because the clustering technology is not yet mature enough to be used on such a scale.

That being so, it does address two major problems:

- Installing, administering, and operating a large number of queue managers and computer boxes.
- Cluster technology does not scale well. Currently adding and removing queue managers from the cluster is a tedious task that often fails, creating zombie entries in the cluster.

A possible solution for the first problem resides in the uniformity of the many queue managers that comprise the back-end. When installed, all these queue managers are exactly the same in terms of definitions, except for their name. This suggests that the back-end queue managers might be a set of cloned hosts, for example cheap PC boxes that will only act as CPUs. A central high capacity storage facility (such as a box of RAID disks) can be connected to the CPUs to be used as protected storage for the queues. This way of creating a new queue

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

manager will consist of simply cloning a file system and connecting a CPU box to it. Since buffer storage resides on protected storage it makes the addition and replacement of queue managers in case of a hardware failure quite simple – disconnect the failed CPU box and replace it with a newly-cloned one.

The cloned installation can also be included in the remit of administration and operation tools (such as Candle Command Center or BMC's Patrol).

Another approach that can be combined is creating more than one queue manager per host to help reduce the amount of CPUs needed.

The second problem affects every cluster user but is yet to be resolved by IBM. Experience tells us that establishing a cluster for the first time is not as tricky as modifying it once it's established. A possible work-around, then, is to create as many queue managers as possible for the back-end. These queue managers will remain suspended from the cluster and the administrator will resume them on demand.

Roy Razon
MQSeries Consultant (Israel) © Roy Razon

Increasing availability on MQ for OS/390

MQSeries for OS/390 does not always benefit from the sysplex facilities of OS/390. MQSeries queues holding persistent messages cannot be shared between all LPARs in a sysplex. If this queue manager shuts down for any reason (ie maintenance or abend) all applications (in our case, CICS transactions) accessing the queues on this queue manager will fail. To overcome this situation we designed the following implementation.

We have two LPARs. We have created one queue manager named *PMQ1* on LPAR1, which holds all application queues. All CICS regions on that LPAR connect to *PMQ1* by default.

We have created another queue manager named *PMQ2* on LPAR2, which holds only one queue and is a transmission queue for *PMQ1*. All CICS regions on that LPAR connect to *PMQ2*.

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

We have defined a Namelist called *TARGET_QMGR* on both queue managers. Both namelists have only one entry (*PMQ1*) as shown below.

```
     Display a Namelist             Row 1 of 1
       Press Enter to refresh details.
       Namelist name              : TARGET_QMGR
       DESCRIPTION           :
       Last alteration time       :
            Name
            PMQ1
  **End of list**
```

All transactions in all CICS regions on any LPAR use a common program – let's call it *COMMQPUT* (common MQPUT). When a message wants to MQPUT to a queue the application program prepares a commarea containing the message and the target queue name and links to *COMMQPUT*. *COMMQPUT* gets the commarea, and makes an *MQINQ NAMELIST(TARGET_QMGR)*.

The result of the *MQINQ* is the target queue manager name where the target queue resides. Note that the target queue manager is always *PMQ1* holding all the application queues. This means that transactions running on LPAR1 are MQPUTing to the local queue manager *PMQ1* and transactions running on LPAR2 are MQPUTing on remote queue manager *PMQ1*, through local queue manager *PMQ2*, transmission queue *PMQ1*, and *PMQ2_TO_PMQ1* channel (see Figure 1).

If a planned interruption is to be carried out on one of the queue managers, let's say *PMQ1* on LPAR1, the only requirement is to modify the CICS definition of the program *COMMQPUT* to become a remote definition on another CICS region on LPAR2. So all transactions running on LPAR1 are using the *COMMQPUT* program remotely from LPAR2, and *COMMQPUT* MQPUTs messages on its local queue manager PMQ2. The messages will be held on the transmission queue *PMQ1* until the PMQ1 queue manager starts running (see Figure 2).

Where unplanned interruptions occur, the downtime is the time required to install the remote definition for the program *COMMQPUT* in CICS, which is a few seconds. This can be done manually or a program can automatically install the definition when the CICS MQ connection is down or Qmanager abends.

© 2001. Xephon UK telephone 01635 33848, fax 01635 38345. USA telephone (303) 410 9344, fax (303) 438 0290.

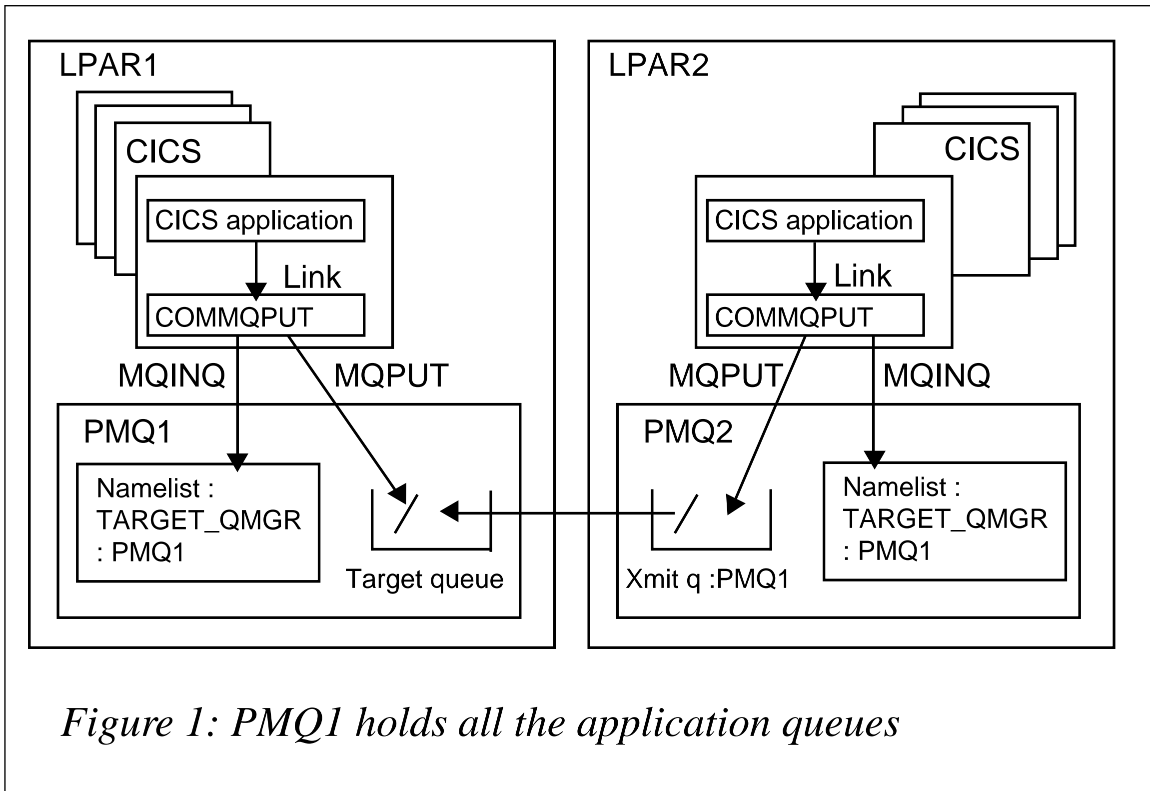

Figure 1: PMQ1 holds all the application queues

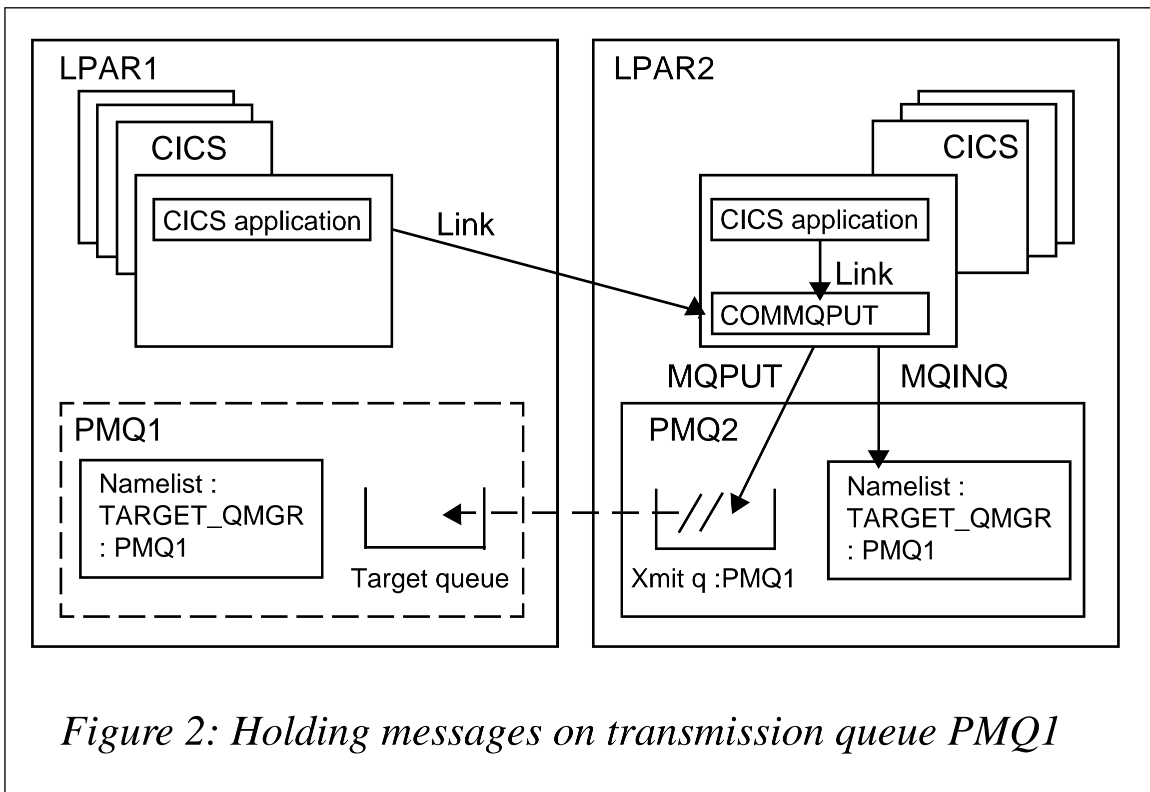

Figure 2: Holding messages on transmission queue PMQ1

Yucel Okan Senturk
Transaction and Messaging Systems Manager (Turkey) © Xephon

© 2001. Reproduction prohibited. Please inform Xephon of any infringement.

MQ news

Candle Corporation has recently announced Version 200 for MQSecure, which offers end-to-end security for MQSeries applications. MQSecure provides security for messages travelling over MQSeries networks and enables users to implement security services that currently are not provided by MQSeries software. In addition, says Candle, the product supplements the user authorization capabilities of various external security programs, such as RACF, ACF2, and Top Secret on OS/390, and operating system security tools on Unix and Windows systems.

MQSecure Version 200 incorporates a number of enhancements including: LDAP-based distribution of public keys, friendlier APIs, support for hardware encryption devices, support for MQSeries clustering, performance enhancements for node-to-node encryption, enhanced database support, Java API, and dual log support.

For further information contact:
Candle Corporation, 201 N Douglas St, El Segundo, CA 90245, USA
Tel: +1 310 535 3600
Fax: +1 310 727 4287
Web: Candle.com

Candle, 1 Archipelago, Lyon Way, Frimley, Camberley, Surrey, GU16 7ER, UK
Tel: 44 1276 414700
Fax: 44 1276 414777

* * *

CommerceQuest has announced the release of enableNet Business Process Integrator (enableNet BPI), an integration platform that, it's claimed, enables enterprise applications to be accessed, re-assembled, and re-used to improve business interoperability and speed business process automation.

Features include a graphical user interface and components to automate, change, and create new business processes. The framework interoperates with various data formats and applications. It runs on multiple platforms including MVS, Unix, AS/400, and Windows NT, and supports most common programming languages.

The company claims it is an automatic and simplified way to leverage IBM MQSeries and MQSeries Integrator to perform business process automation.

Within enableNet BPI, XML-based MQSeries or SOAP message formats are automatically created, thereby allowing components to be integrated in an open, loosely-coupled environment. enableNet BPI also provides native interfaces to essential communication protocols, such as HTTP, FTP, and SMTP.

For further information contact:
CommerceQuest, 2202 N Westshore Blvd, Tampa, FL, 33607, USA
Tel: +1 813 639 6300
Fax: +1 813 639 6900
Web: http://www.CommerceQuest.com

CommerceQuest (UK), Doncastle House, Doncastle Road, Bracknell, Berkshire, RG12 8PE, UK
Tel: +44 (0) 1344 861010
Fax: +44 (0) 1344 861011

* * *