WebSphere MQ for Windows V5.3 - Performance tuning for large clusters

Version 1.0

14 February 2003

Lucas Partridge
Andrew Banks

WebSphere MQ
IBM UK Laboratories
Hursley Park
Winchester
Hampshire
SO21 2JN

Take Note!

Before using this report be sure to read the general information under “Notices”.

Notices

This report is intended to help the customer perform capacity planning. The information is not intended as the specification of any programming interfaces that are provided by WebSphere MQ.

References in this report to IBM products or programs do not imply that IBM intends to make these available in all countries in which it operates.

Information contained in this report has not been submitted to any formal IBM test and is distributed “as-is”. The use of this information and the implementation of any of the techniques is the responsibility of the customer. Much depends on the ability of the customer to evaluate the data and project the results to their operational environment.

The performance data contained in this report was measured in a controlled environment and the results obtained in other environments may vary significantly.

Trademarks and service marks

The following terms used in this publication are trademarks of the IBM Corporation in the United States or other countries or both:

IBM

MQSeries

WebSphere MQ

SupportPac

FFST

AIX

Microsoft, Windows, and Windows 2000 are trademarks of Microsoft Corporation in the United States, other countries, or both.

First Edition, February 2003
This edition applies to V1.0 of ‘WebSphere MQ for Windows V5.3 – Designing large clusters’ and to all subsequent releases and modifications until otherwise indicated in new editions.

Feedback on this SupportPac

We welcome constructive feedback on this report. Does it provide the sort of information you want? Do you feel something important is missing? Is there too much technical detail, or not enough? Could the material be presented in a manner more useful to you? Please direct any comments of this nature to: **WMQPG@uk.ibm.com**.

Specific queries about performance problems on your WebSphere MQ system should be directed to your local IBM MQ sales representative.

Acknowledgements

The authors are grateful to Paul Clarke and Alexander Russell for their help and advice.

CONTENTS

Context

The tests described here are based on a hub and spoke configuration of queue managers. This arrangement is typical of those found in retail businesses where it is convenient to connect a large number of branches at the spokes directly to a head office at the hub.

Objectives

We aim to answer two basic questions:

1. What do I need to consider when constructing a large cluster of queue managers?
2. How many branches can my cluster support?

Overview of the tests

The tests simulated sending a request message from a branch and receiving a response from the hub server. All of the messages were 4 KB in size and persistent. The request messages were sent at a rate of 100 per second, cycling through each branch. We have obtained results for between 1000 to 3000 branch queue managers, all contained within a single cluster.

To be successful these networks must be robust. One of the most demanding events that can occur is a total network failure near to the hub server which is long enough to cause all of the channels to fail and all of the message traffic to stop. We simulated this by disabling and later re-enabling the network adapter card on the server machine. Once the network card was re-enabled the time taken for both the original number of running channels and the original throughput of messages to be restored was measured.

To facilitate testing we simulated the entire set of branch queue managers by using a single queue manager located on a single driver system. We used a pair of cluster channels between the queue manager on the driver system and the queue manager on the server system to represent the two-way communication between an individual branch and the hub server.

All the tests were conducted using WebSphere MQ for Windows v5.3 running on Microsoft Windows 2000 Advanced Server. See the WebSphere MQ manual *Queue Manager Clusters* for an explanation of the concepts and terms used in clustering.

Summary of main results

- Message throughput before network failure was close to the expected value of 100 round trips/s for up to 3000 branches (channel pairs).
- Rapid and reliable recovery from failure of 1000 channel pairs could be achieved in under 3 minutes by suppressing specific channel error messages.
- Recovery from network failure at 2000 channel pairs was achieved (in under 4 minutes) by also running with trusted instead of non-trusted channels. Otherwise recovery failed because the agent processes on the server had run out of Windows address space.
- Recovery at 3000 channel pairs was achieved (in under 9 minutes) by additionally using strmqm and runmqlsr in preference to amqmdain, and by applying additional queue manager tuning parameters.
- We failed to achieve recovery from network failure at 4000 channel pairs due to inadequate processor power on our testing hardware.

Summary of tuning recommendations

Users are recommended to adopt the following configuration to promote rapid and reliable recovery of a clustered environment from network failure:

- Set the system environment variable MQ_CHANNEL_SUPPRESS_MSGS = 9001,9002,9202,9203,9206,9208,9209,9213,9514,9558,9999. Up to 20 AMQ channel error messages may be included in the suppression list. Only channel error messages may be suppressed.
- The attributes set on the CLUSRCVR channels control the behaviour of all channels in a cluster other than the first time a manually defined CLUSSDR starts.
    - Consider increasing the SHORTTMR attribute (default 60 seconds) if you can tolerate a recovery time greater than 60 seconds. Reducing SHORTTMR would speed up recovery but might overload the hub server.
    - Also consider increasing SHORTRTY attribute (default 10 attempts) if you want the channel to complete more short retry attempts before switching to its long retry attempts. Conversely, reduce SHORTRTY if you want to switch to long retry cycles sooner.
- Apply the following tuning parameters to the registry entry of the hub machine's queue manager (assumed to be called HUB.QM here):
    - amqmdain reg HUB.QM -c add -s TuningParameters -v AgentClassMap=(1,1,1)
    - amqmdain reg HUB.QM -c add -s TuningParameters -v AgentClassLimit1=(150,250,999)
    - amqmdain reg HUB.QM -c add -s Channels -v PPOptThreads=250
    - amqmdain reg HUB.QM -c add -s Channels -v PPMaxThreads=250
    - amqmdain reg HUB.QM -c add -s Channels -v PPOptProcesses=100
    - amqmdain reg HUB.QM -c add -s Channels -v PPMaxProcesses=-1

  These changes increase the number of threads in each agent process and channel pooling process and thereby help to conserve the memory in Windows available for desktop heap allocation.
- The previous recommendation is preferable to the next two because system stability is sacrificed to performance.
- Run the channels as trusted rather than non-trusted to reduce the number of processes and address ranges used. This also conserves virtual memory and CPU. Be advised that if a trusted channel fails, however, the entire queue manager might abend.
- To conserve address space further, use strmqm, endmqm and runmqlsr to start and end the queue manager and listener processes instead of amqmdain start, amqmdain end and amqmdain crtlsr respectively. Be advised that if you do this then you will not be able to log out of Windows without ending the queue manager.

See the section *Other considerations* for further options that may be applicable to your system.

Testing methodology

The scenario we were trying to simulate is one in which there are 1000 local branches or stores each containing a queue manager and one or more driving applications submitting and receiving messages. All the queue managers communicate with a single queue manager located at a regional head office. A multithreaded server application retrieves and processes messages arriving at the hub queue manager, and sends replies back to the branches. The head office queue manager and all the branch queue managers belong to the same cluster. This hub and spoke configuration, with a branch queue manager located on each of the spokes and the head office queue manager located at the hub, is illustrated in **Figure 1**:

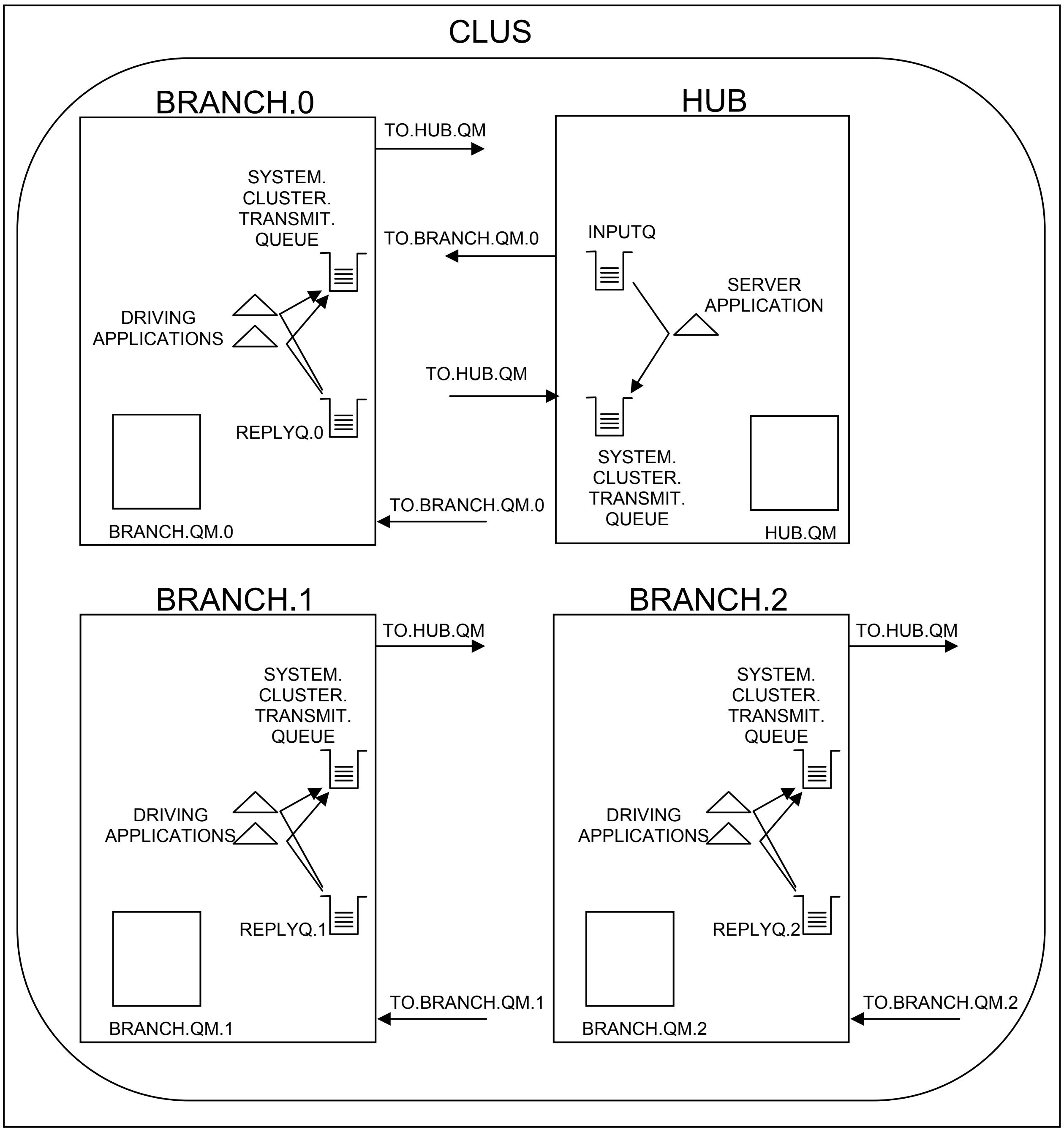

Figure 1 – Hub and spoke scenario being simulated

In Figure 1 there are three branch offices and one head office and all belong to the same cluster called CLUS. The queue managers HUB.QM and BRANCH.QM.0 both hold full repositories for the cluster. Arrows pointing away from a box represent CLUSSDR channel definitions and arrows pointing to, and touching, a box indicate CLUSRCVR channel definitions. Only channel definitions that need to be defined manually are shown.

Due to resource constraints it was not feasible to reproduce this scenario directly in the laboratory environment. The hub system is likely to come under the most strain because it has to reply to all of the request messages and host all the channels connecting to the branch machines. A branch machine on the other hand only has to host one direct connection to each server queue manager at the hub[1]. Since the hub system is the principle system under test the branch systems can be simplified by reducing them to as few machines as possible so long as they are not constrained and are able to drive a workload equivalent to the original number of branches. In fact we used just two machines altogether and configured them as shown in **Figure 2**:

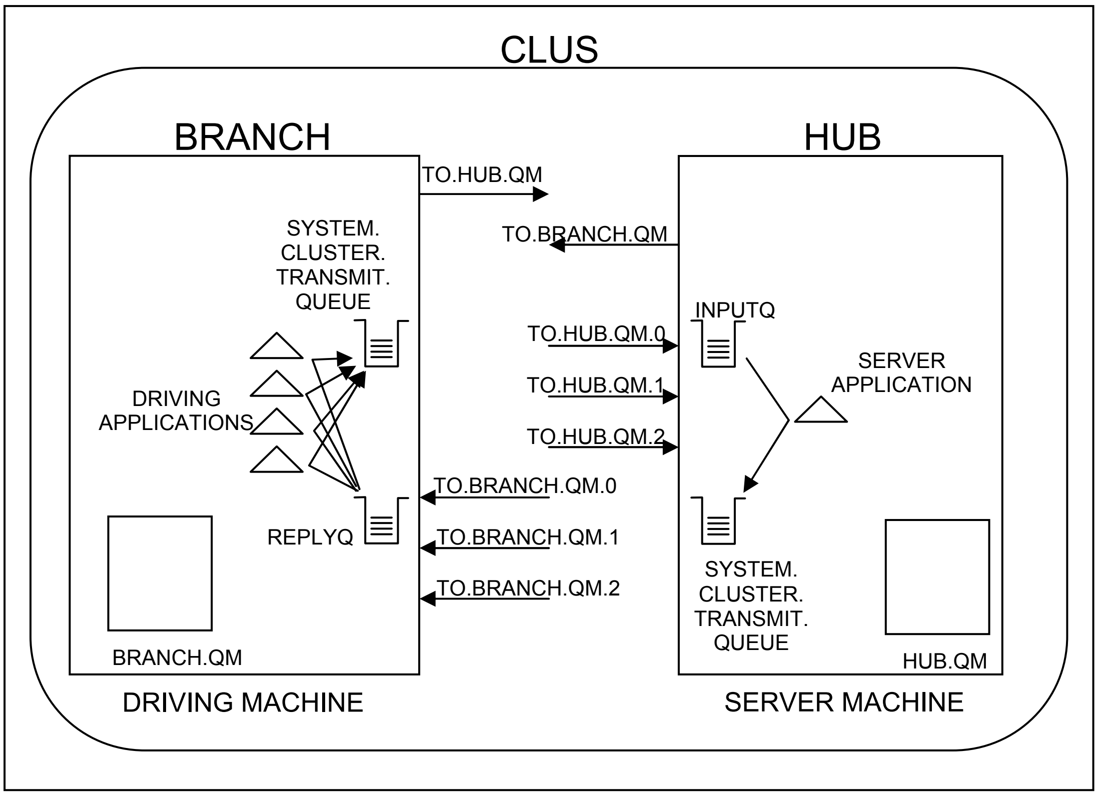

Figure 2 - Scenario tested in this report

All the branch machines in the first scenario were therefore reduced to a single machine. Since the server machine hosting the hub queue manager was the system under test it was not necessary to have more than one reply queue or queue manager on the driving machine. A CLUSRCVR channel was defined on both the driving machine and the server machine for every branch that was being simulated in the test. This was done to ensure that there were as many channel pairs between the two machines as there were branches. To clarify terminology a cluster channel is defined by a CLUSRCVR channel definition and its corresponding (auto-defined) CLUSSDR channel definition. A 'channel pair' refers to two CLUSSDR-CLUSRCVR channel definitions acting in opposite directions to allow two-way communication between two machines.

[1] A branch machine would also have to host one or more auto-defined CLUSSDR channels to any of the other branch queue managers with which it wishes to communicate; but the number of these branch-to-branch channels is assumed to be negligible here.

To ensure that all the channels were exercised the cluster queues INPUTQ and REPLYQ shown in Figure 2 were defined using the DEFBIND(NOTFIXED) attribute. The round-robin workload balancing algorithm internal to clustering technology thereby ensured that successive messages were sent down successive running channels, regardless of the number of driving applications used.

For full details of how the driver and server machines were configured, and a description of the workload that was applied, refer to *Appendix 1: Details of the testing methodology.*

Results

Tests were conducted at 1000, 2000 and 3000 channel pairs or 'branches' (Table 1). In all the tests 20 driving applications were used on the driving machine. Each application submitted 4 KB persistent messages at the rate of 5 messages/s, giving an overall expected throughput rate of 100 messages/s.

We were unable to achieve recovery from network failure with 1000 or more channel pairs unless specific channel error messages were suppressed. This is because each channel that fails and attempts to restart writes a number of messages to the error log files. When there are many channels the error logs are updated very rapidly thereby placing a heavy load on the disk holding the queue files.

A new feature to WebSphere MQ 5.3 CSD03 plus efix 72615.1 is the option to suppress specified channel error messages. Two new environment variables have been added to control this feature. The first, MQ_CHANNEL_SUPPRESS_INTERVAL defaults to a value of 60,5 (the minimum value allowed is 0,0). This means that after the first 5 occurrences of a given message in a 60 second interval, any further occurrences of that message will be suppressed (not written to the error log). At the end of the interval a summary message is written for each of the suppressed message types indicating how many messages were written and how many were skipped[2]. The message types to be suppressed are listed in the second environment variable, MQ_CHANNEL_SUPPRESS_MSGS. Up to 20 channel error message codes can be included in a comma-separated list. Only channel error messages may be suppressed in this way. Examination of the error logs should confirm that message suppression is occurring as specified. Note that if MQ_CHANNEL_SUPPRESS_MSGS is not defined then no message suppression will occur.

In the tests described here:

- MQ_CHANNEL_SUPPRESS_INTERVAL was left undefined, and MQ_CHANNEL_SUPPRESS_MSGS was set to 9001,9002,9202,9203,9206,9208,9209,9213,9514,9558,9999. This meant that any more than 5 occurrences of any of the corresponding channel error messages (e.g., AMQ9002) would be suppressed in a one minute interval. Suppressing channel error messages greatly reduced the writing to the error logs during and after network failure, and facilitated recovery.
- The channel attributes HBINT=60 and SHORTRTY=1000 were specified. Reducing HBINT reduces the time it takes for the number of running channels to drop to zero; and the 1 minute 1 second it took for the number of running server channels to drop to zero for all the tests in Table 1 bore this out. Increasing SHORTRTY meant that the channels would attempt to retry for more short retry intervals before switching to the longer retry interval.

[2] The summary message feature was not available at the time these tests were conducted.

Table 1 - Results of network recovery tests

No. of channel pairs	Trusted channels[a]	amqmdain[b]	Additional queue manager tuning parameters[c]	Observed throughput (messages/s)	Total CPU% on the server machine[d]	Channel recovery time[e] (min:sec)	Throughput recovery time[f] (min:sec)	FDCs[g]
1000	N	Y	N	98.4	22	2:21	2:29	N
2000	N	Y	N	98.4	52	N/A	N/A	Y
2000	Y	Y	N	98.4	37	2:54	3:38	N
3000	Y	N	Y	96.3 approx.	60-80	8:34	8:49	N

Notes:

b. Indicates whether channels were run trusted (Y) or not (N).

c. Indicates whether amqmdain crtlsr and amqmdain start were used to start the listener and the queue manager (Y) instead of runmqlsr and strmqm respectively (N).

d. Indicates whether additional tuning parameters were required for the queue manager. These are the tuning parameters given for the 3000 channel pair test in Appendix 1.

e. Server CPU utilization figures are quoted for the period prior to the server's network card being disabled.

f. Channel recovery time – the time taken for the original number of running channels on the server machine to be restored after the network card on the server was re-enabled. N/A (not applicable) means that the original number of channels never recovered.

g. Throughput recovery time – the time taken for the original rate of throughput of messages to be restored after the network card on the server was re-enabled. N/A (not applicable) means that the original number of channels never recovered. Note that each test was only run once and therefore the times quoted for the channels and throughput to recover are not necessarily repeatable to the nearest second.

h. Indicates whether any FDC or FFST files were observed on either the server machine or the driver machine.

i. MQ_CHANNEL_SUPPRESS_MSGS was actually set to 9001,9002,9202,9203,9206,9208,9213,9999 for the first three tests in the table (and to 9001,9002,9202,9203,9206,9208,9209,9213,9514,9558,9999 for the 3000 channel pairs test). We do not believe the addition of the 3 extra message codes – AMQ9209, 9514 and 9558 - would have materially improved the recovery times quoted for the first three tests.

Key findings of the tests were:

- Message throughput before network failure was close to the expected value of 100 round trips/s for up to 3000 channel pairs.
- Rapid and reliable recovery from failure of 1000 channel pairs could be achieved in under 3 minutes merely by suppressing specific channel error messages.
- The first test with 2000 channel pairs was configured in the same way as for the test with 1000 channel pairs. However, recovery could not be achieved: an FDC occurred on the server which indicated that additional desktop heap could not be allocated for the agent processes (see the Discussion).
- The second test with 2000 channel pairs was run with trusted instead of non-trusted channels. This relieved demands for both desktop heap and CPU, and this time the system was able to recover from network failure (in under 4 minutes).

- Recovery at 3000 channel pairs was achieved (in under 9 minutes) by making two further changes in addition to running the channels trusted. First, strmqm and runmqlsr were used in preference to amqmdain; and second, additional queue manager tuning parameters were applied. Both changes were aimed at reducing contention for Windows resources still further, although the driver and server machines were still under heavy load as it was not possible to obtain regular samples with Windows performance monitor.
- Although we could achieve message throughput at 4000 channel pairs we could not achieve recovery from network failure (results not shown). This was true both at 20 and 100 round trips/s (with approximately 62% and 69% total CPU utilization on the server machine respectively).

Discussion

- The most important constraint is the number of simultaneously active channels that the hub server can support, coupled with the peak message rate and the application load.
- The use of clustered channels has relatively little effect on performance and the results are expected to be similar for non clustered channels.
- After the network is re-established the rate at which the branches try to recover their connection to the server is governed by the retry interval set on the hub server CLUSRCVR channel definition. The rate at which the hub server attempts to recover connections back to the branches is similarly governed by the retry interval set on the branch systems' CLUSRCVR channel definitions. If the message rates are low and there is no message backlog the channels can restart at a rate which is less than that configured by the retry intervals, because the channels only attempt to restart when there is a message on the cluster transmit queue. Our tests were designed so that we tested the worst case: message rates were high enough to force all of the channels to attempt to restart soon after the network was restored. We used the default short retry interval (SHORTMR) of 60 seconds, so restarting of all of the channels would have taken at least 60 seconds.
- If there is a second hub server in the cluster, unaffected by the failure, the recovery process is not visible to the branch customers as any new request messages are processed by the second server. Making the retry interval shorter will speed up recovery, but it will also place more load on the hub server that is trying to recover, possibly making the recovery unreliable. Be aware that as soon as the channels start to recover then messages will be sent down them due to the workload balancing algorithm. Both servers should therefore be sized to manage the full workload from the cluster in the absence of the partner server.
- We increased the number of short retry attempts (SHORTRTY) from the default of 10 to 1000 attempts in order to ensure that we did not start to use the long retry interval. This would not necessarily be a good choice for a real system where using the long retry interval might be more appropriate, for example when a second hub server has been configured.
- Application messages start to flow soon after the first channels restart, but the overall throughput is initially below that prior to the network failure for at least two reasons. First, not all of the channels will have started; and, second, the server has fewer resources available for processing messages due to the additional power required to start the channels. In all of the tests described here the original throughput was restored in less than one minute after the original number of channels was restored on the server (Table 1).
- In tests simulating 1000 or more branch systems, i.e., where the hub system ran 2000 or more channels simultaneously, reliable recovery from network failure could only be achieved when specific error messages generated by the channels were suppressed. The environment variable MQ_CHANNEL_SUPPRESS_MSGS was used to do this. On

the hardware used here, with 1000 branch systems and some error message suppression, recovery from network failure could be achieved in less than 3 minutes.

- To support significantly more than 1000 branch systems the queue Manager is constrained by the allocation of desktop heaps for desktop objects within Windows. Further information is provided in the Microsoft Knowledge Base Article – 184802 (http://support.microsoft.com/default.aspx?scid=KB;en-us;q184802). Briefly, a desktop heap must be allocated for every desktop object that is created in Windows. The system allocates desktop heap from a 48 MB system-wide buffer. As more and more channels are started there comes a point when there is insufficient memory in this system-wide buffer to allocate a new desktop heap for a new desktop object. This will occur even though the total real memory on the system may be far from exhausted.
- Recovery of 2000 branch systems was only achieved by running the channels in trusted mode in addition to suppressing certain channel error messages. Running the channels in trusted mode conserves memory and also CPU workload because the channels bind directly to the queue manager and do not require a separate agent process. The system recovered from network failure in less than 4 minutes. Be advised, however, that by running channels in trusted mode a failure of a channel might impact the whole queue manager.
- Recovery of 3000 channel pairs in under 10 minutes was achieved by considerable tuning:
  1. Additional channel error messages were suppressed.
  2. The channels were run trusted.
  3. Tuning parameters were set on the queue manager to increase the number of threads running in each channel pooling process and the number of threads running in each agent process. This relieves demand for desktop heap in Windows.
  4. strmqm, endmqm and runmqlsr were used in preference to amqmdain start, amqmdain end and amqmdain crtlsr. This also relieves demand for desktop heap. However, using strmqm, endmqm and runmqlsr means that you cannot log off the system without ending the queue manager.
- When more than 3000 branch systems are being recovered the CPU power of our systems became the key constraint. When enough branch systems are simulated there is insufficient CPU power to service each channel within the time constraints needed to start it. Effectively the channels fail before they completely start.
- Rapid and reliable recovery from network failure could not be achieved at 4000 channel pairs on the hardware we used. To get beyond this point more powerful processors than the ones we used are probably needed.

Other considerations

The following were not relevant to the tests we performed but may be relevant to your situation:

- You are advised to use the maximum log file size of 16 384 pages for a full repository queue manager. Each page is 4 KB in size, giving a total size of 64 MB per log file. If you run out of log space later on you can always add more log files and restart the queue manager, whereas the log file size cannot be altered once the queue manager has been created.

- In a large cluster you may wish to consider running the full repository queue managers on dedicated hardware separate from the hub systems. This reduces the risk of application problems impacting the full repository processing and allows the full repository queue managers to be tuned separately.

- The repository manager performs some operations, such as compressing the SYSTEM.CLUSTER.REPOSITRY.QUEUE, as a single transaction. It determines when to do this based on the maximum uncommitted messages setting (MAXUMSGS) of the queue manager. The default value of 10 000 is fine but reducing this significantly on a full repository queue manager might cause it to perform many smaller units of work and hence increase repository processing time.

- To run a large number of channels in the hub systems the value of *MaxChannels* and *MaxActiveChannels* needs to be reviewed as the default value of 100 is unsuitable. These values are set in the *Channels* stanza of the Windows registry. For example:

  ```
  amqmdain reg QMNAME -c add -s Channels -v MaxChannels=10000
  ```

- If your full repository queue managers reside on separate hardware you can set *MaxActiveChannels* to a small value if your DISCINT channel attributes are also short. This has little effect for the majority of the time when the repository manager only has to run a few channels. However it will occasionally need to run channels to all of the queue managers in the cluster if, for example, the definition of a widely used cluster queue on the hub server has changed. Setting *MaxActiveChannels* small, say to 500 in a 1000 branch cluster, would spread the processing over a time of DISCINT+ a few seconds. In this example the first 500 channels would start, then go inactive, and after DISCINT seconds the second 500 would run. This technique puts an upper limit on the amount of work the repository manager is expected to do at any one time and allows it to serve very large clusters.

- Each channel that sends or receives a persistent message will update a scratchpad file. During checkpoint processing this file must be forced to disk. In these tests this expanded the time it took to complete the checkpoint to nearly 180 seconds for 3000 branch systems. This in itself is not a problem so long as all of the transactional activity can fit in the log during the 180 second interval. It also has the side effect of extending the queue manager's restart time by about 180 seconds while this activity is replayed.

- If the heartbeat interval, HBINT, is long relative to the channel retry interval consider setting AdoptNewMCA=ALL to accelerate recovery from a network failure. AdoptNewMCA is an attribute in the *Channels* stanza of a queue manager. A CLUSRCVR channel will not detect it has failed until it has detected a failed heartbeat flow. Until then a CLUSRCVR will reject restart attempts by the CLUSSDR end of the channel. Specifying AdoptNewMCA=ALL, however, ensures that a CLUSRCVR channel will terminate and restart as soon as the CLUSSDR end of the channel attempts to restart. See the *WebSphere MQ System Administration Guide* for more information about AdoptNewMCA and the related attributes AdoptNewMCATimeout and AdoptNewMCACheck.

- Use DISCINT to reduce the number of active channels. Some of the branch queue managers may be inactive for long periods of time. The DISCINT parameter on the hub and spoke system's CLUSRCVR channel definition controls how long the channel will stay running if no messages flow over it. The default is 6000 seconds. The extra latency to start a channel is not normally visible to interactive applications. Leaving this value long means the server has to maintain the state of a larger number of inactive channels and this burden increases with the number of channels. Making this value too short may mean that the queue manager spends more time stopping and starting channels than it would spend maintaining inactive channels. A value of 600 seconds (10 minutes) is a good compromise when starting to tune this channel attribute.

- There are some workloads where a much shorter DISCINT can be considered for the CLUSRCVR channel definitions:
    i. One example is a system which stores up till activity and periodically transfers it to a hub server. In this case the messages from each branch system flow in predictable bursts separated by long periods of inactivity. Setting the DISCINT short will shut down the channel as soon as the burst is complete and reduce the load on the system in between the bursts.
    ii. Another example is a system which is a dedicated full repository for your cluster and which runs no other workload. This is a good candidate for a short DISCINT value because the repository messages flow in short bursts of one or two messages with long gaps, possibly of many days, in between. A DISCINT of 10 seconds is a good choice for this type of system.

Appendix 1: Details of the testing methodology

An overview and rationale of the testing methodology have already been given (see *Testing methodology* and especially Figure 2). This section provides the details of how the server and driver machines were configured and also describes the workload used to drive the system. Two IBM Netfinity 8500R machines running Windows 2000 Advanced Server were used (see *Appendix 2: Measurement environment* for full details). Each machine had its own cluster queue manager with a full repository.

Configuration of the server machine

The server machine, HUB, was configured as follows:

1. The system environment variable MQ_CHANNEL_SUPPRESS_MSGS was set to 9001,9002,9202,9203,9206,9208,9209,9213,9514,9558,9999:

   Right-click on the 'My Computer' icon on the desktop, select 'Properties', 'Advanced' tab, 'Environment Variables'. Under 'System variables' click on 'New'. Enter CHANNEL_SUPPRESS_MSGS for 'Variable name' and 9001,9002,9202,9203,9206,9208,9209,9213,9514,9558,9999 for 'Variable value'. Click OK, OK and OK to close all the windows.

   Note that MQ_CHANNEL_SUPPRESS_INTERVAL was left undefined.

2. The queue manager HUB.QM was created:

   ```
   crtmqm –q –h 50000 –lf 16384 –lp 12 –ls 2 –u
   SYSTEM.DEAD.LETTER.QUEUE HUB.QM
   ```

3. The registry entries for HUB.QM were edited as follows:

   - amqmdain reg HUB.QM -c add -s Log -v LogBufferPages=512
   - amqmdain reg HUB.QM -c add -s Channels –v MaxChannels=20000
   - amqmdain reg HUB.QM -c add -s Channels –v MQIBindType=FASTPATH

   NB In the test with 3000 channel pairs the following queue manager registry entries were also edited:

   - amqmdain reg HUB.QM -c add -s TuningParameters -v AgentClassMap=(1,1,1)
   - amqmdain reg HUB.QM -c add -s TuningParameters -v AgentClassLimit1=(150,250,999)
   - amqmdain reg HUB.QM -c add -s Channels -v PPOptThreads=250
   - amqmdain reg HUB.QM -c add -s Channels -v PPMaxThreads=250
   - amqmdain reg HUB.QM -c add -s Channels -v PPOptProcesses=100
   - amqmdain reg HUB.QM -c add -s Channels -v PPMaxProcesses=-1

   The first two registry entries ensured that all the agents would run with a large number of threads (a maximum of 150 threads per agent process up to 999 agents, and then 250 threads per agent thereafter). The last four entries ensured that the channel pooling processes would run with up to 250 threads each. All the changes were therefore aimed at reducing the number of agent and channel pooling processes to reduce demand for desktop heap (see the Discussion).

4. The registry was secured by issuing:

```
amqmdain regsec
```

5. The listener and queue manager were started in all but the test with 3000 channel pairs as follows:

   The listener was started:

```
amqmdain crtlsr HUB.QM –t tcp –p 3604
```

   Then the queue manager was started:

```
amqmdain start HUB.QM
```

   The listener service must be started before starting the queue manager because the listener runs as a service under a parent process for the queue manager. If *amqmdain crtlsr* is invoked after *amqmdain start* then the listener will be defined but it will not run and there will be no throughput of messages.

   NB In the test with 3000 channel pairs *amqmdain* was not used to start the listener and the queue manager. Instead *runmqlsr* and *strmqm* were used as follows to conserve Windows address space:

   The queue manager was started:

```
strmqm HUB.QM
```

   The listener was started (with the environment variable MQ_CONNECT_TYPE having been set to FASTPATH):

```
runmqlsr –m HUB.QM –t tcp –p 3604
```

6. The following definitions were then run in via RUNMQSC:

   1. HUB.QM was made a full repository for the cluster named CLUS:

```
ALT QMGR REPOS(CLUS)
```

   2. A cluster queue was defined to hold the incoming request messages:

```
DEF QL(INPUTQ) DEFBIND(NOTFIXED) CLUSTER(CLUS)
```

   3. A single cluster-sender channel was defined:

```
DEF CHL(TO.BRANCH.QM)  CHLTYPE(CLUSSDR) TRPTYPE(TCP) –
        CLUSTER(CLUS) CONNAME(‘BRANCH(3603)’) –
        MCATYPE(THREAD) DISCINT(0) –
        SHORTRTY(1000) HBINT(60) –
        BATCHINT(0) BATCHSZ(50) –
        NPMSPEED(FAST) REPLACE
```

   4. Then N cluster-receiver channels were defined, where N was the number of imaginary branch queue managers in the cluster to be simulated:

```
DEF CHL(TO.HUB.QM.i) CHLTYPE(CLUSRCVR) TRPTYPE(TCP) –
```

```
CLUSTER(CLUS) CONNAME('HUB(3604)') –
MCATYPE(THREAD) DISCINT(0) –
SHORTRTY(1000) HBINT(60) –
BATCHINT(0) BATCHSZ(50) –
NPMSPEED(FAST) REPLACE
```

For example, if N was 1000 then i in TO.HUB.QM.i would have been 0, 1, 2, ..., 998, 999.

Configuration of the driver machine

The driver machine, BRANCH, was configured as described below. Some of the configuration options would not be required for a branch machine in a production system but were applied to our driving machine for ease of administration and to conserve resources on the machine. We reproduce these additional configuration options to show you how we ran the tests but present them in italics to indicate they would not be required on a branch machine in a production system.

1. *The system environment variable MQ_CHANNEL_SUPPRESS_MSGS was set to 9001,9002,9202,9203,9206,9208,9209,9213,9514,9558,9999:*

   Right-click on the 'My Computer' icon on the desktop, select 'Properties', 'Advanced' tab, 'Environment Variables'. Under 'System variables' click on 'New'. Enter CHANNEL_SUPPRESS_MSGS for 'Variable name' and 9001,9002,9202,9203,9206,9208,9209,9213,9514,9558,9999 for 'Variable value'. Click OK, OK and OK to close all the windows.

   Note that MQ_CHANNEL_SUPPRESS_INTERVAL was left undefined.

2. The queue manager BRANCH.QM was created:

   ```
   crtmqm –q –h 50000 –lf 16384 –lp 12 –ls 2 –u
   SYSTEM.DEAD.LETTER.QUEUE BRANCH.QM
   ```

3. The registry entries for BRANCH.QM were edited as follows:

   - amqmdain reg BRANCH.QM -c add -s Log -v LogBufferPages=512
   - *amqmdain reg BRANCH.QM -c add -s Channels –v MaxChannels=20000*
   - *amqmdain reg BRANCH.QM -c add -s Channels –v MQIBindType=FASTPATH*

   NB In the test with 3000 channel pairs the following queue manager registry entries were also edited:

   - *amqmdain reg BRANCH.QM -c add -s TuningParameters -v AgentClassMap=(1,1,1)*
   - *amqmdain reg BRANCH.QM -c add -s TuningParameters -v AgentClassLimit1=(150,250,999)*
   - *amqmdain reg BRANCH.QM -c add -s Channels -v PPOptThreads=250*
   - *amqmdain reg BRANCH.QM -c add -s Channels -v PPMaxThreads=250*
   - *amqmdain reg BRANCH.QM -c add -s Channels -v PPOptProcesses=100*
   - *amqmdain reg BRANCH.QM -c add -s Channels -v PPMaxProcesses=-1*

4. The registry was secured by issuing:

```
amqmdain regsec
```

5. The listener and queue manager were started in all but the test with 3000 channel pairs as follows:

   The listener was started:

   ```
   amqmdain crtlsr BRANCH.QM –t tcp –p 3603
   ```

   Then the queue manager was started:

   ```
   amqmdain start BRANCH.QM
   ```

   NB In the test with 3000 channel pairs *amqmdain* was not used to start the listener and the queue manager. Instead *runmqlsr* and *strmqm* were used as follows to conserve Windows address space:

   The queue manager was started:

   ```
   strmqm BRANCH.QM
   ```

   The listener was started *(with the environment variable MQ_CONNECT_TYPE having been set to FASTPATH)*:

   ```
   runmqlsr –m BRANCH.QM –t tcp –p 3603
   ```

6. The following definitions were then run in via RUNMQSC:

   1. BRANCH.QM was made a full repository for the cluster named CLUS:

   ```
   ALT QMGR REPOS(CLUS)
   ```

   2. A cluster queue was defined to hold the reply messages:

   ```
   DEF QL(REPLYQ) MAXDEPTH(99999) DEFPSIST(YES) –
           DEFBIND(NOTFIXED) CLUSTER(CLUS) REPLACE
   ```

   3. A single cluster-sender channel was defined:

   ```
   DEF CHL(TO.HUB.QM)  CHLTYPE(CLUSSDR) TRPTYPE(TCP) –
           CLUSTER(CLUS) CONNAME(‘HUB(3604)’) –
           MCATYPE(THREAD) DISCINT(0) –
           SHORTRTY(1000) HBINT(60) –
           BATCHINT(0) BATCHSZ(50) –
           NPMSPEED(FAST) REPLACE
   ```

   4. Then N cluster receiver channels were defined, where N was the number of imaginary branch queue managers in the cluster to be simulated:

   ```
   DEF CHL(TO.BRANCH.QM.i) CHLTYPE(CLUSRCVR) TRPTYPE(TCP) –
           CLUSTER(CLUS) CONNAME('BRANCH(3603)') –
           MCATYPE(THREAD) DISCINT(0) –
   ```

```
SHORTRTY(1000) HBINT(60) –
BATCHINT(0) BATCHSZ(50) –
NPMSPEED(FAST) REPLACE
```

For example, if N was 1000 then i in TO.BRANCH.QM.i would have been 0, 1, 2, ..., 998, 999.

Points to note about the configuration of the server and driver machines:

1. In the test with 3000 channel pairs the runmqlsr process on each machine was started before the channels were defined. This gave the system more opportunity to start smoothly since the two queue managers could exchange information about channels as soon as they started to be defined. Had the listeners been started after the channels had been defined the driving applications might have tried to send messages before all the necessary information had been exchanged.
2. Because the driver and server machines were actually configured simultaneously a one minute sleep interval was introduced between the starting of the listener and the running in of the channel definitions on both machines. This increased the likelihood that both queue managers would have been created and the listeners running before the channel definitions were run in.
3. Points 1 and 2 were only necessary because of the way we configured the testing systems and are not recommended as being necessary for production systems. Effectively the entire branch network in the test system was being established within a few minutes. This is not recommended practice for a production situation.
4. It was only necessary to define one CLUSSDR channel explicitly on each machine because the clustering technology automatically created further CLUSSDR channels as required. Thus if 1000 CLUSRCVR channels had been defined between the two machines, 1000 corresponding CLUSSDR channels were created automatically.
5. Although we used the same attributes for both the manually defined single CLUSSDR channel and all the CLUSRCVR channels, it is the attributes of the CLUSRCVR channels that are used to define the attributes of the corresponding auto-defined CLUSSDR channels. In other words, concentrate on specifying the CLUSRCVR channel attributes correctly in your cluster configuration; those of the manually defined CLUSSDR channel are only used the first time it starts successfully.
6. Notice that HBINT, the interval between heartbeat flows flowing down each channel, was set to 60 seconds rather than the default of 300 seconds. This was done to accelerate the rate at which the channels stopped running when the network card was disabled on the server machine. The number of running channels on the server machine would fall to zero approximately one minute following network failure compared to approximately 5 minutes when the default HBINT value was used. This was done purely to facilitate testing; you do not need to alter HBINT in your configuration. A short HBINT value will permit rapid detection of failure but will incur an increased load of heartbeat messages flowing over the network. The converse is true for a long HBINT value. The default value of 300 seconds (5 minutes) is likely to be adequate for most situations.
7. To end the queue manager at the end of each test *amqmdain end <QUEUE_MANAGER_NAME>* was used instead of *endmqm –w <QUEUE_MANAGER_NAME>* for all but the test with 3000 channel pairs.

Workload configuration

Once all the channels had been defined on both machines a non-trusted server application was started on the server machine, HUB. The server application had 40 threads, each of which retrieved a request message off the local input queue, INPUTQ, and put a reply to the remote queue, REPLYQ (specified in the *ReplyToQ* field of the request message). Each thread in the server application retrieved a request message and put the reply message inside a single unit of work.

On the driver machine many single-threaded, trusted driving applications put request messages on the INPUTQ. Both the putting of the request message and the getting of the corresponding reply message were done inside a single unit of work. Because a single reply queue was shared among all the driving applications the reply messages were retrieved using the *CorrelId* field of the message descriptor. Each driving application waited indefinitely for a reply message and, once it had received one, put another request message after a specified think time had elapsed since it had sent the previous request message. For example, if the think time was one second but the reply was received 0.6 s after sending the request then the next request would not be sent until 0.4 s later. Conversely, if the reply was received after more than one second had elapsed then another request would be sent immediately.

In the tests described here 20 single-threaded driving applications were used to put 4 KB persistent messages at a specified rate of 5 messages/application/s, giving an overall rate of 100 messages/s. In a system comprising 1000 branches this workload would be equivalent to one message being sent by every branch every 10 seconds. This was thought to be a demanding load compared to many scenarios in production. Recall from the *Testing methodology* section that the round-robin workload balancing algorithm internal to clustering technology automatically ensured that all the channels were utilised regardless of the number of driving applications used.

Network failure and recovery was simulated by disabling and re-enabling the network adapter card on the server machine, HUB. This was achieved by right-clicking on ‘My Computer’, selecting ‘Manage’ and then ‘Device Manager’, ‘Network adapters’. By right-clicking on the network adapter card listed underneath ‘Network adapters’ it can be disabled or enabled as required.

A separate program was used to monitor the number of running channels on both the driver and server machines. The network card was not re-enabled until all of the running channels on both machines had fallen to zero.

Appendix 2: Measurement environment

The driver and server machines both had the following specification:

Hardware

Machine model	IBM Netfinity 8500R
Processor	Intel Pentium 3 Xeon 700 MHz, 2 MB L2 cache
Architecture	4-way SMP
Memory (RAM)	8 GB
Disk	2 internal 10,000 rpm SCSI disks – 18 GB and 9 GB; 1 external 10,000 rpm SCSI disk – 9 GB
Network	1 Gigabit Ethernet

Software

Operating System	Microsoft Windows 2000 Advanced Server with Service Pack 3 (Build 5.00.2195)
WebSphere MQ	WebSphere MQ for Windows, Version 5.3 CSD 03. Efix 72615.1, which includes the summary message feature for suppressed channel error messages, was not available at the time the tests were conducted. Note: the queue files and log files were located on separate, dedicated physical disks.
Compiler	Microsoft Visual C++ 6.0 Professional Edition

***** END OF REPORT *****