Keeping WebSphere MQ Channels Up and Running

Morag Hughson
hughson@uk.ibm.com

SHARE in Boston

Preface

N O T E S

This document is provided by IBM as recommendations based on IBM's expert knowledge of WebSphere MQ and MQSeries and experience with customers using these products and their environments and goals. Acceptance and implementation of these suggestions is at the reader's discretion. You may hear different opinions from other experts. There are several areas within WebSphere MQ that are subject to varying opinions and may vary according to the specific environment in which WebSphere MQ is implemented.

After you, the reader, have become familiar with WebSphere MQ you will be able to determine which suggestions you want to follow or you may arrive at opinions that are appropriate for your specific environment.

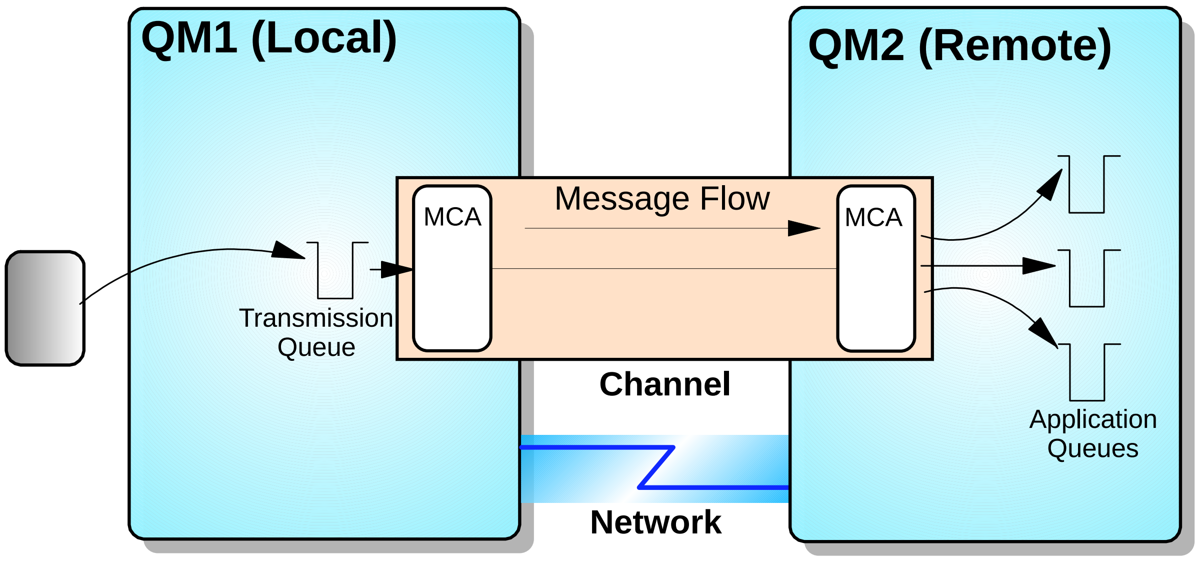
This document is intended for use by Systems Administrators and any others that will support WebSphere MQ networks. Hopefully it will provide the following benefits:

- Give consistency to administration processes
- Provide maximum availability of applications
- Help in avoiding common mistakes that beginners make
- Assist personnel in the early stages of their becoming WebSphere MQ experts
- In general assure a smooth start and success for WebSphere MQ projects

Most of the information presented here relates to versions of both MQSeries and WebSphere MQ. Check the documentation for your particular version to ensure that the parameters are supported.

SHARE in Boston

Channel Architecture



Channel Architecture

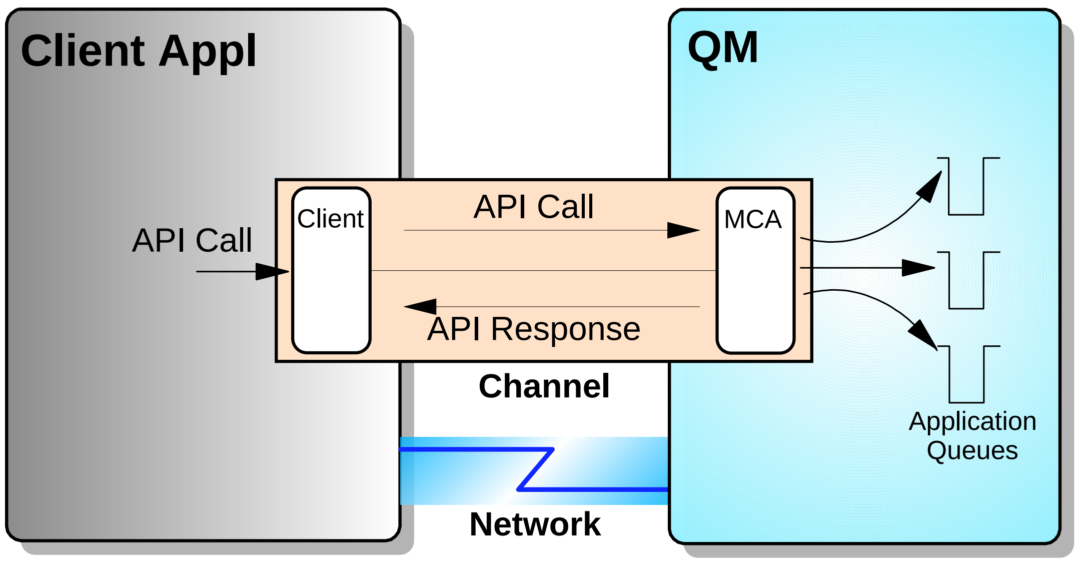
N O T E S

This diagram represents the WebSphere MQ channel architecture. We are going to use this diagram throughout the presentation to illustrate the different channel topics.

The main part of the diagram shows that applications can put messages (either via remote queue definitions or by explicit addressing) which will land on a transmission queue to be delivered to the remote system. QM1 is our local system in this diagram and QM2 is our remote system. The sending MCA servicing our transmission queue gets the messages off the queue and sends them to the remote system via, the communications protocol specified on the channel definition, where they are received by the receiving MCA and placed on the appropriate target application queue. This pair of MCAs is known as the channel.

That's all very well, but what can go wrong?

Client Architecture



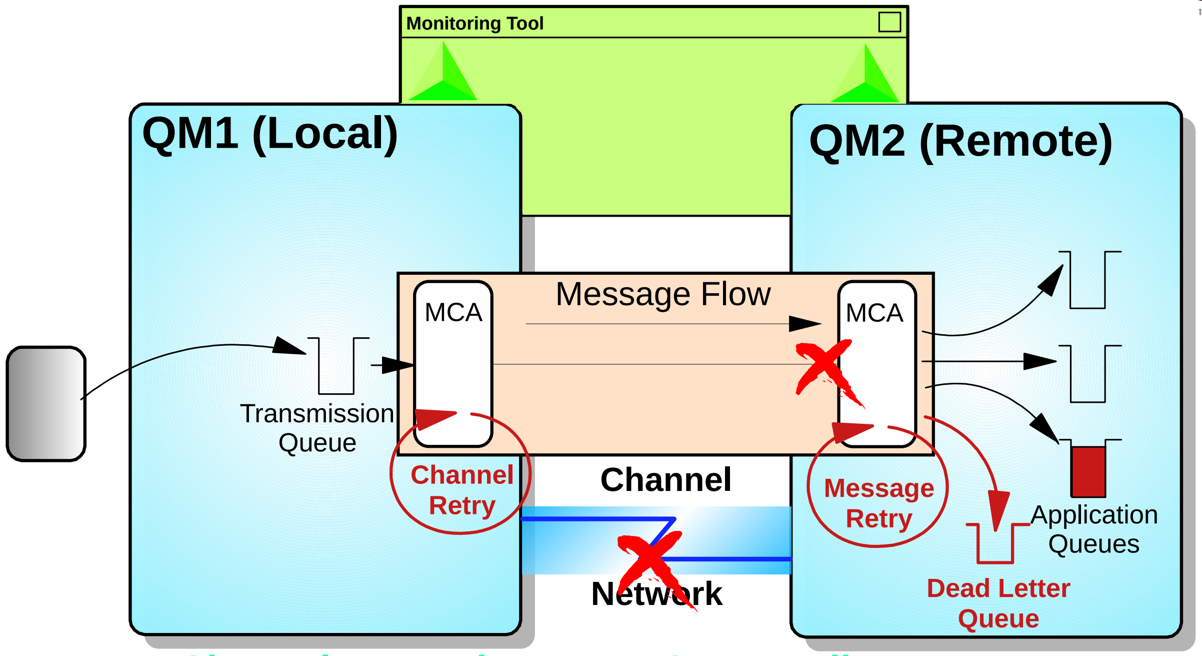
Client Architecture

N O T E S

This diagram represents the WebSphere MQ client channel architecture. We are going to use this diagram throughout the presentation to illustrate the different client channel topics.

The main part of the diagram shows that a client application can make API calls, for example to put or get messages from queues on the queue manager. These API calls are packaged up and flowed across the client channel to the queue manager where the server connection MCA acts as a proxy for the client application and issued the API call to the queue manager.

What can go wrong?



- Channel can end - remote QMgr ending
- Message delivery not possible - Queue invalid or full
- Network failure
- Performance
- How Monitoring can help

What can go wrong?

N O T E S

This foil acts rather like an agenda of what we are going to discuss today. We will look briefly at several different things that can go wrong and then we will go through each one in more detail on later foils.

The channel can end. For example, because of the remote Queue Manager terminating. To recover from this type of problem, you can make use of the channel initiator to retry the channels.

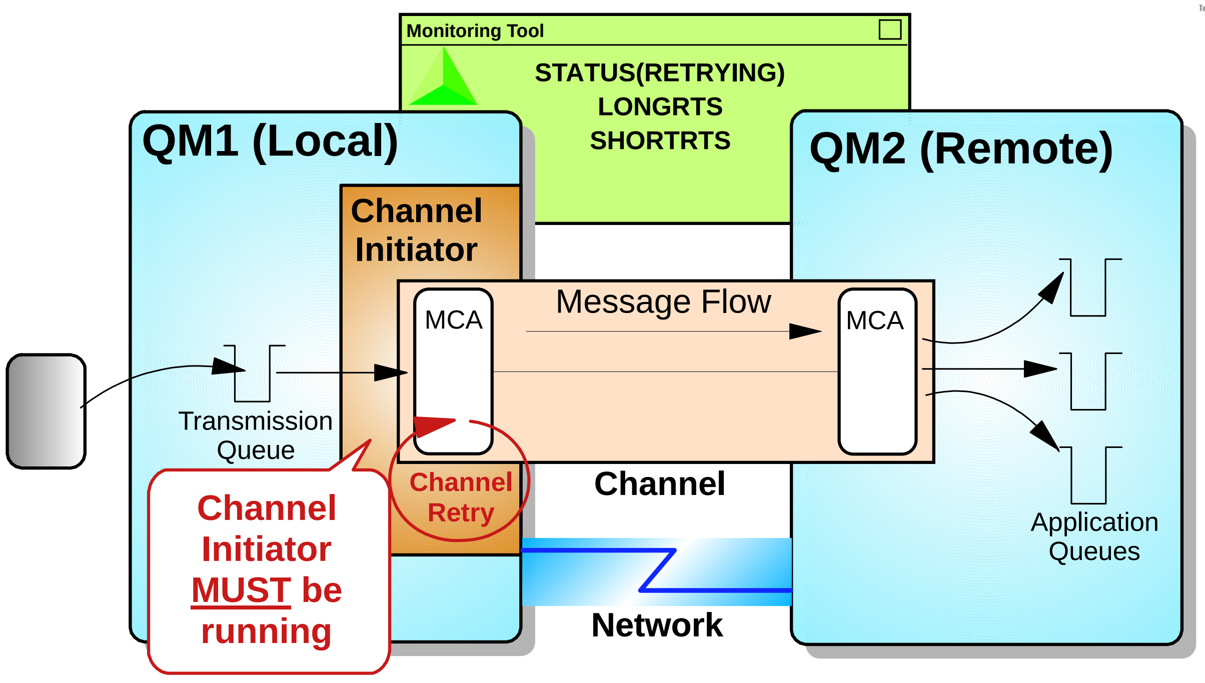
Message delivery might not be possible. This might be because of an application error, for example the specified queue is invalid, or because of a system problem, for example the queue is full. To continue after this type of problem, you can make use of the Dead-Letter Queue or you can use the Message Retry facility.

You might suffer a network failure, and depending on the type of failure, there are a variety of ways for your channel to detect this and recover.

Performance might be an issue, so we will also look at some performance considerations.

There are quite a number of fields in channel status and WebSphere MQ V6 adds several new channel monitoring fields. These are helpful for monitoring the health of your channels, and also for problem determination if things go wrong. A number of Channel Monitoring parameters collect data that is expensive to collect on some platforms, mainly due to taking extra timestamps, therefore it must be turned on. Channel Monitoring can be turned on for all channels by setting a queue manager parameter, MONCHL. It can be turned off for specific channels to override the queue manager setting or turned on for specific channels if it not set at a queue manager level using the channel parameter also called MONCHL. We will look at a selection of the relevent fields available to you in channel status as we go through ths presentation.

Channel Retry



DEF CHL ....
SHORTRTY(10) SHORTTMR(60)
LONGRTY(999 999 999) LONGTMR(1200)

Channel Retry

N O T E S

The 'catchall' mechanism for ensuring that channels are operating is for WebSphere MQ to restart the channel after a failure. This is known as channel retry. Channel retry can be used on sending types of channels, that is senders, servers or cluster-senders.

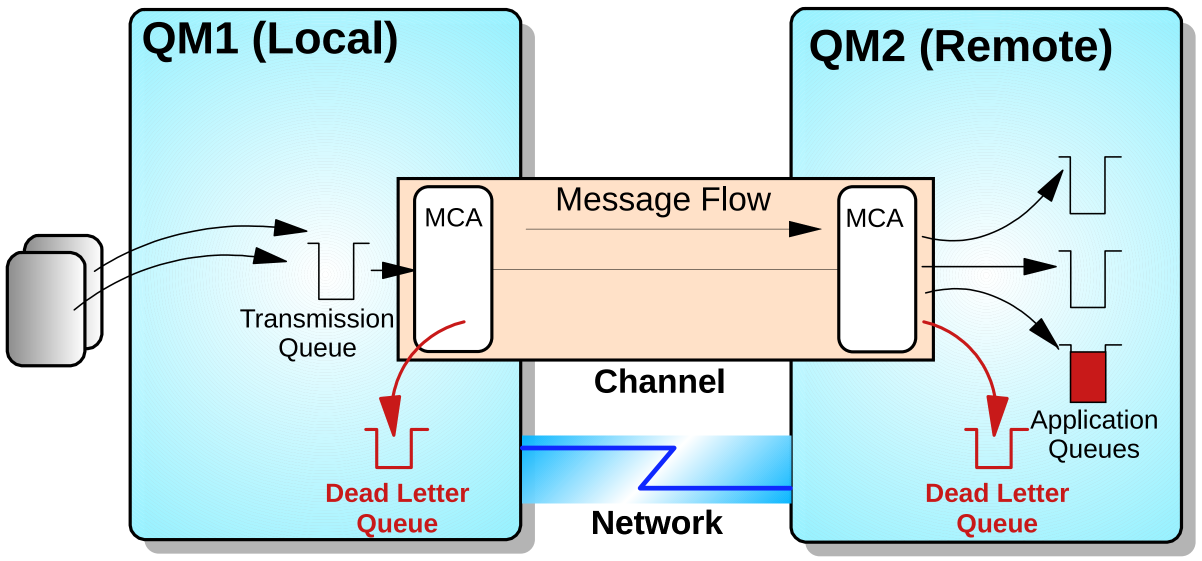
All sender channels should be configured with suitable retry values and the Channel Initiator must be running in order to use the retry mechanism.

Retry values should ensure sufficient time for network problems to be resolved, so for example, retrying for 2 minutes is unlikely to be sufficient! There are two different sets of retry values available for your use, a short retry and a long retry. The short retry values tend to be used for a small number of frequent retries, in other words, to attempt to reconnect to the partner after a temporary blip of some sort. The long retry values tend to be used for a larger number of less frequent retries, in other words, if the reason for the channel ending was a problem that is not simply a small blip.

Be aware that the numbers specified in the long retry values must be enough to cover any outage because if the retry sequence is completed without a successful transfer of messages the channel will require manual intervention to restart it.

The channel will have status RETRYING while it is waiting for the retry interval to pass, and the channel status fields SHORTRTS and LONGRTS will show you how many short or long retry attempts are remaining.

Dead-Letter Queue



- Storage area for messages which are, for example :-
  - **Wrongly addressed**
  - **Unauthorised**
  - **Too large**

Dead-Letter Queue

N O T E S

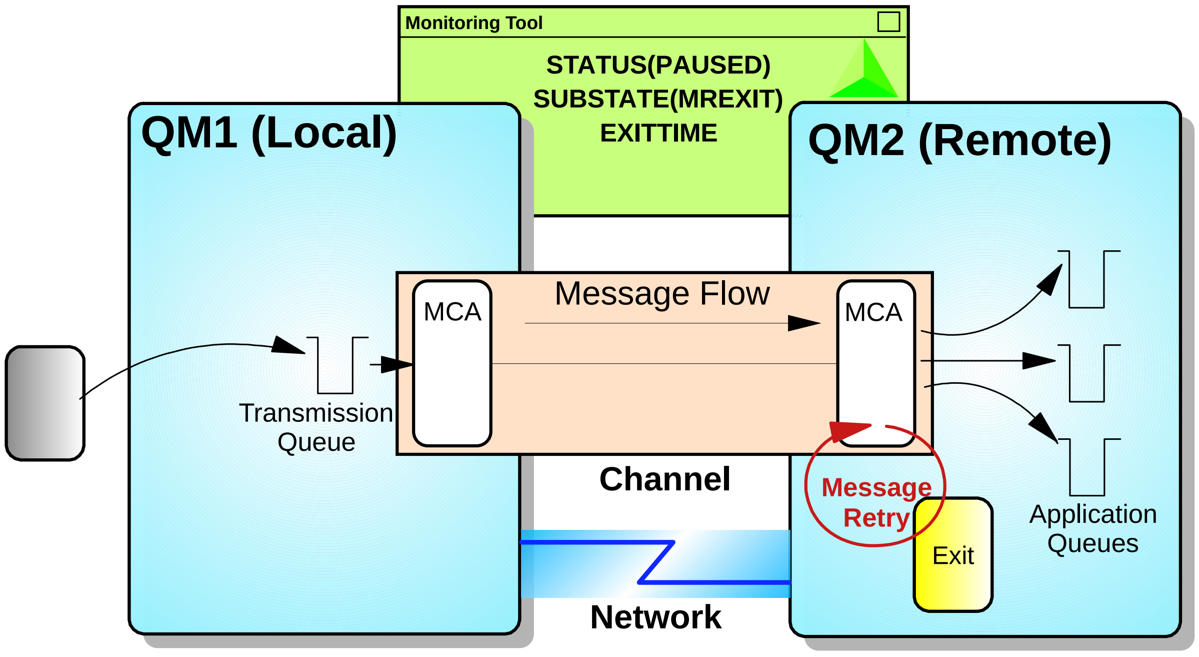
It is strongly recommended that Queue Managers have a Dead-Letter Queue. Without a Dead-Letter Queue a simple programming error in the application, by specifying an invalid Queue Name, can be sufficient to bring the Channel down. If several different applications are utilising the same channel, this is clearly unacceptable.

Note that the DLQ might be used at either end of the channel. At the sending end, the DLQ will be used if a particular message is too large to be sent down the channel. This occurs if the message size exceeds the size of message defined for the channel even though it can be accommodated by the transmission queue. At the receiving end, the DLQ will be used if the message cannot be placed (PUT) onto the target queue. This might occur for any number of reasons, some of which may be transitory. The message retry parameters (defined for a channel) and the Message Retry exit are designed to handle these cases. We will look at Message Retry in a moment.

It should be noted that the use of the DLQ - at either side of a channel - may affect the order of delivery of messages. If there is some problem with a message which is a member of some sequence then this message will be DLQ'd and the message sequence will be interrupted.

Some 'lazy' applications require messages to always be delivered in the same sequence as they were sent. These applications require that there is no DLQ. The consequence of not having a DLQ is that a 'bad' message on a channel will terminate the channel and therefore prevent any application from receiving their messages. For this reason it is strongly recommended that installations do have a DLQ and that applications are modified to cope with messages to occasionally be delivered out of sequence. This modification is not difficult and there are various techniques which can be employed such as using the Correlation Id as a counter, having a secondary queue or using message grouping.

Message Retry



DEF CHL ....
MRRTY(10) MRTMR(1000)
MREXIT(*msg-retry-exit-name*) MRDATA()

Message Retry

N O T E S

Consider using Message Retry, particularly if you are running without a Dead-Letter Queue.

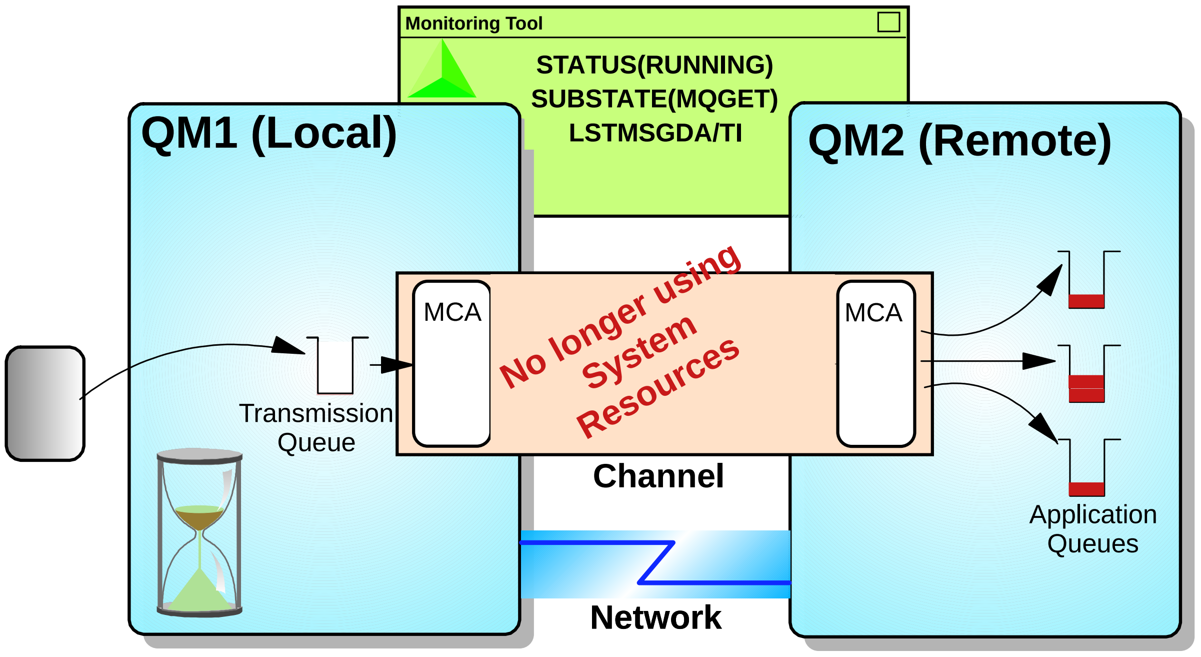
The message retry mechanism can be used with or without an exit. Without an exit the receiving MCA simply retries the failing put on each interval. The values for the time between retries and the number of retries are taken from the channel definition. If used with an exit, the exit is passed these numbers but can change them to be any value. Using an exit with the message retry mechanism allows a more intelligent processing of the failure. You could, for example, redirect messages if they can not be put to target queue.

The use of an exit allows the retry decision to be made on a per message basis.

Messsage Retry is supported on z/OS from V6 onwards.

The receiving end of the channel with have STATUS(PAUSED) while it is waiting for the retry interval to pass. If you are using an exit as part of this processing, you will see SUBSTATE(MREXIT) while the exit is processing and EXITTIME will record the time spent in exits for a message which will include message retry exits.

Disconnect Interval



DEF CHL .... DISCINT(6000)

- **Combine with Triggering**

Disconnect Interval

N O T E S

The Disconnect Interval or DISCINT is the channel attribute on a sending type channel used to control how long a channel should remain running even though there are no messages to send.

The value you set depends on several criteria.

How many channels you have and how critical system resource is

How reliable your network is

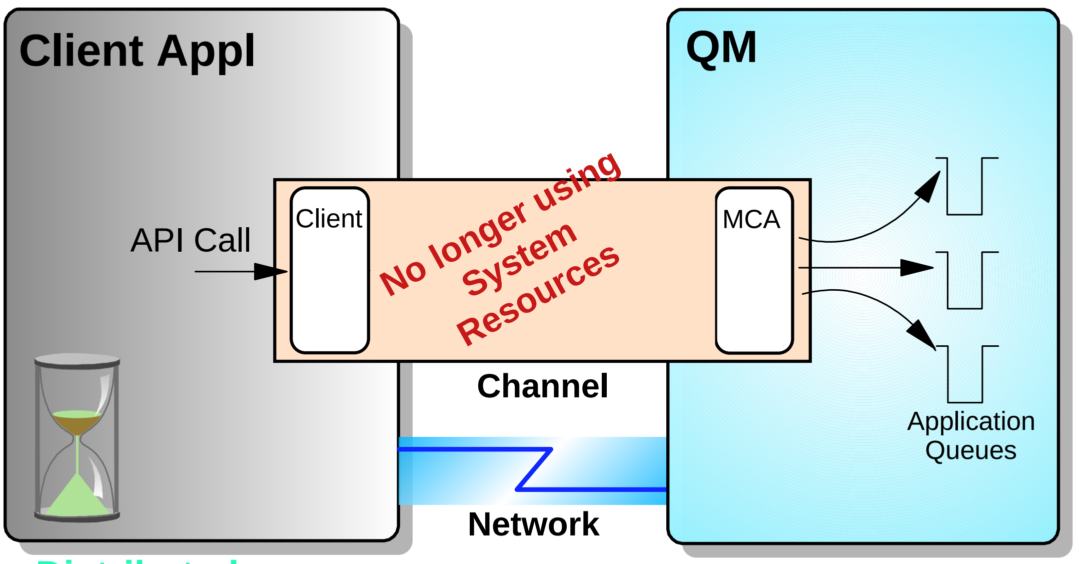
How expensive the communications session is

The ideal value should be longer than the average time between messages to that partner. This means that the channel will disconnect when there is no work to be delivered to that partner.

You should combine the use of a disconnect interval of your channel with channel triggering to restart the channel. This way you channel is started automatically when there is work for it to do, and ended automatically when there is no more work for it to do.

The channel will show STATUS(RUNNING) while it is running (and therefore using resources on the system). It will show SUBSTATE(MQGET) when it is waiting for the disconnect interval to pass since it is also waiting for any new work to arrive on the transmission queue to be sent. You can also look at the LSTMSGDA and LSTMSGTI fields to see exactly when the last message was sent by the sending MCA.

Client Disconnect Interval



- Distributed
  - QM.INI - ClientIdle
- z/OS
  - ALTER CHL CHLTYPE(SVRCONN) DISCINT(500)

Client Disconnect Interval

N O T E S

The Disconnect Interval or DISCINT is the channel attribute on a server-connection type channel used to control how long the client channel should remain running even though there are no API calls currently being issued by the client application.

The value you set depends on several criteria.

How many channels you have and how critical system resource is

How reliable your network is

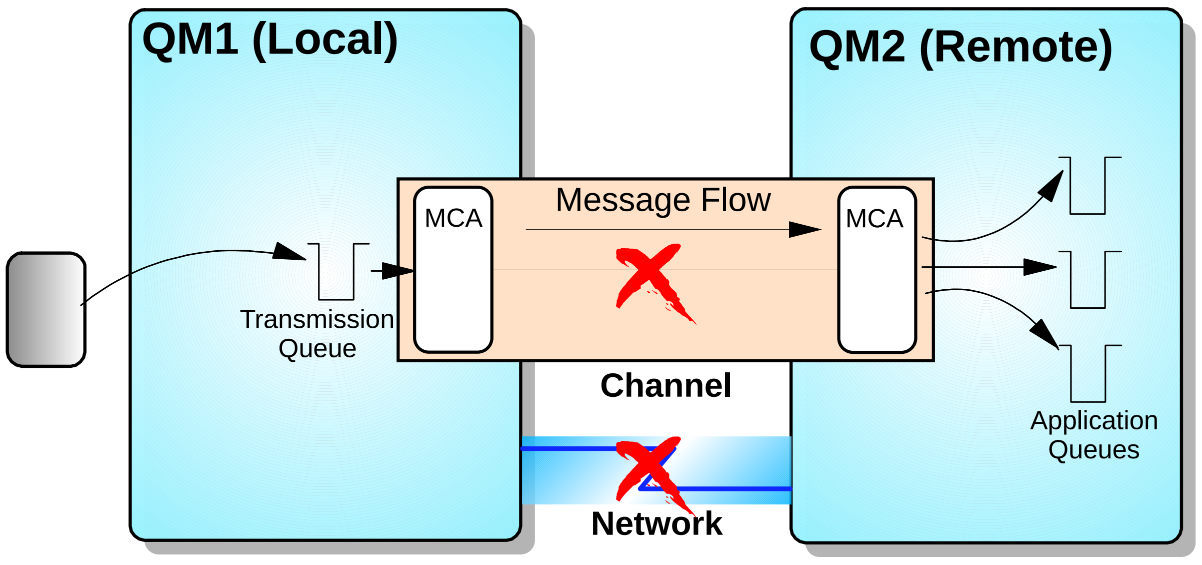
How expensive the communications session is

Whether the client application holds the connection (an thus the client channel) open whilst not making use of it to issue MQ API calls.

This attribute should be set to a value that is longer than the longest design time between API calls because if shorter, the client application's connection will be broken when it shouldn't be.

This can be used to clean up server-connection channels that have become orphaned and so are not receiving any more API calls from the client (althought we'll see later that client heartbeats are even better at this), or to break the connection of an application that is mis-behaving, perhaps looping in non MQ API parts of the client application, in a way it shouldn't be, and as a result holding a channel open when resources ought to be closed off.

Network Failure



- Heartbeats
- TCP/IP Keepalive
- Adopt MCA

SHARE in Boston

Network failure

N O T E S

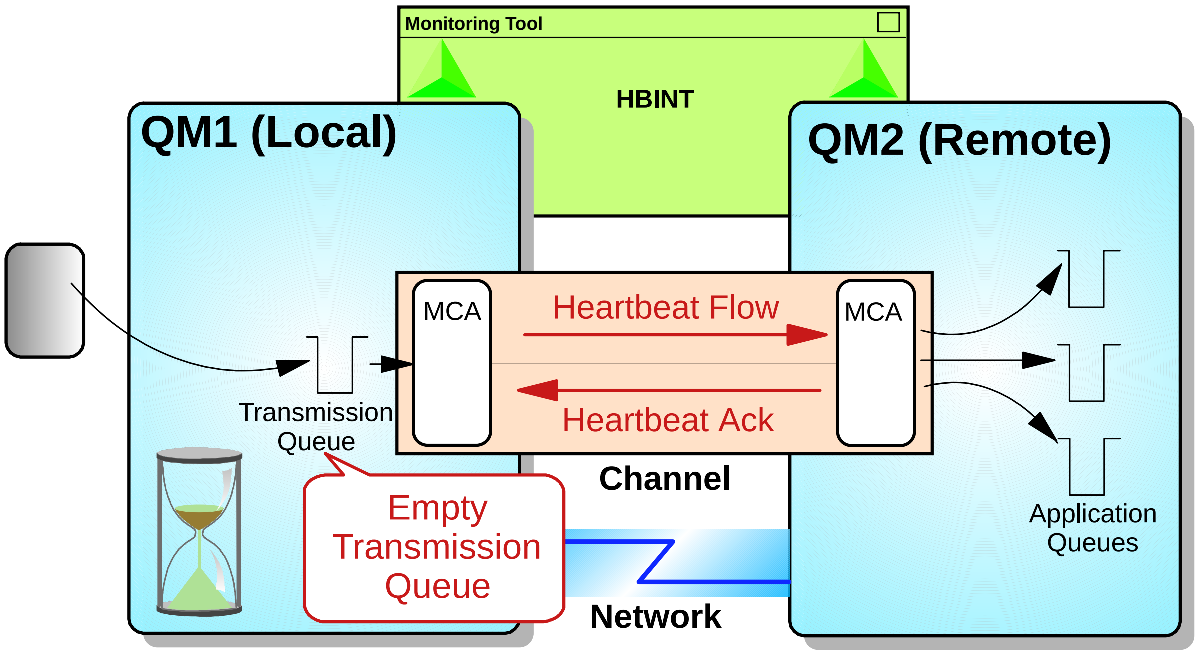
Any network failure reported to the channel will cause the channel to end. If the network failure is not reported to the channel the channel will not end, it may not notice a network failure.

The Channel is reliant on the lower level communications functions to report any network outages. For example, the resting state of a RECEIVER channel is on a recv call. If the recv call does not return the Channel is unaware that connection to the partner is lost. One of the difficulties with MCAs was recognising in a timely way when the network had actually failed, as the comms protocols didn't always return errors at the right times. Typically, LU6.2 has been much better at this than TCP/IP (it's a more complex protocol).

Various enhancements to the MCAs have improved the situation considerably. We will look at a few of these over the next few foils.

SHARE in Boston

Heartbeats



DEF CHL .... HBINT(300)

Free Buffers ✓

Close Queues ✓

Heartbeats

N O T E S

The Heartbeat Interval or HBINT is the channel attribute which controls how often the sending end of the Channel should check that the receiver is ok when there are no messages to send. The value is specified in seconds on both ends of the channel and they negotiate an appropriate value to use at channel startup. This value is the least frequent interval of the two specified. If one or both ends specifies 0 which means no heartbeats, then this value, being the least frequent value possible, is the one chosen. The HBINT field in channel status will show the negotiated heartbeat value being used for this instance of the channel.

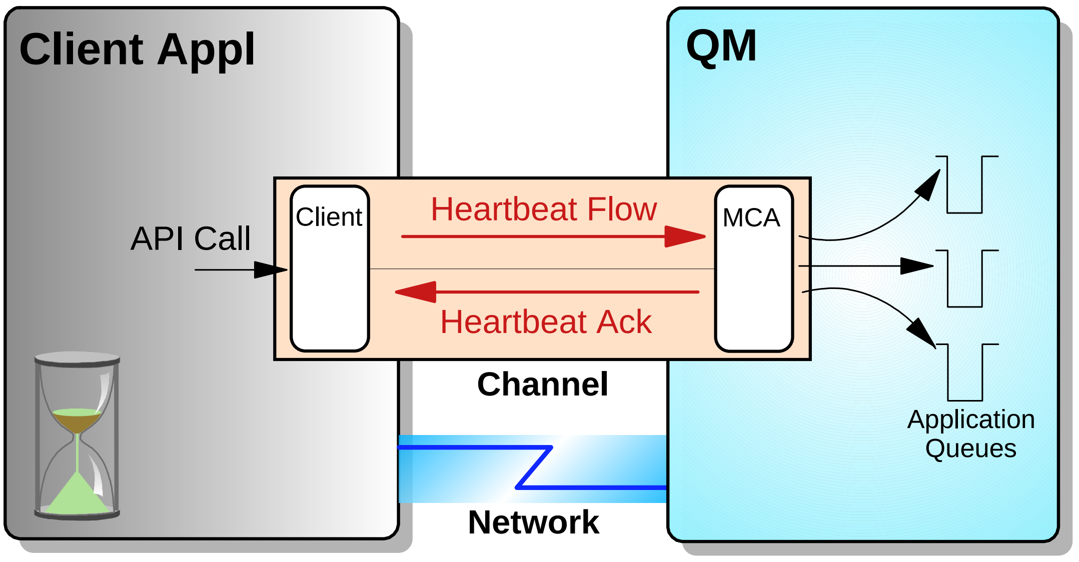
By default channels are defined with a heartbeat interval of 300 seconds (5 minutes). A relatively high value was chosen as the default so that there would be very little impact on the network. In practice a lower value of say 60 seconds is better in a lot of circumstances.

This heartbeat is actually a short message sent to the receiving MCA to check that it is still available. It is only sent when there is no other activity on the channel for at least Heartbeat seconds. These messages solve two problems; sender MCAs may now get an immediate error return from the network, instead of sitting on their MQGET; and they also allow receiver MCAs to do some processing such as checking for the termination of the queue manager before sending their response.

Additionally, since the channel is not transferring messages regularly the channel chooses not to maintain its cache of storage and Queues. So, at heartbeat time a channel will also free any unnecessary storage buffers and message areas and close any cached queues.

So, for performance, the Heartbeat value should not be too small. But note that low heartbeat values will cause no extra network traffic provided the channel remains busy. Consider an application sending a message every two seconds down a channel. Even a low heartbeat value of 3 seconds will never cause a heartbeat to be sent since there will always be an actual message to send.

Client Heartbeats



DEF CHL .... HBINT(300)

- **Heartbeat 'question' can be asked by either end of the channel and at any time (V7 +)**

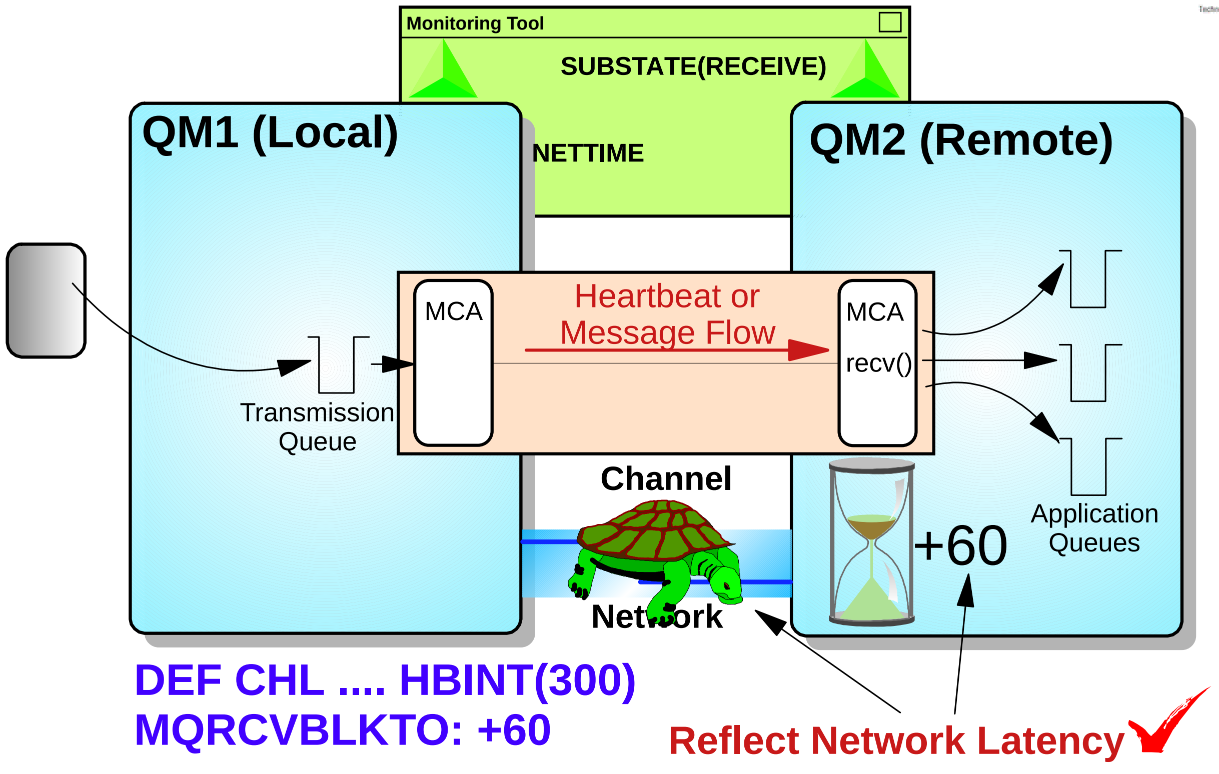
Client Heartbeats

N O T E S

Over client channels heartbeats are sent for the life of the connection. The sever checks the client is still active and the client checks the server is still active. For this heartbeating to be done both ends of the channel must be at least MQ V7.0.0.

In releases prior to MQ V7, however, heartbeats were only sent from the server and only during an MQGET call. As a consequence detection of network problems is much more prompt with MQ V7.

Receive Wait Time Out



DEF CHL .... HBINT(300)
MQRCVBLKTO: +60

ALTER QMGR
RCVTTYPE(ADD) RCVTIME(60)

Receive Wait Time Out

NOTES

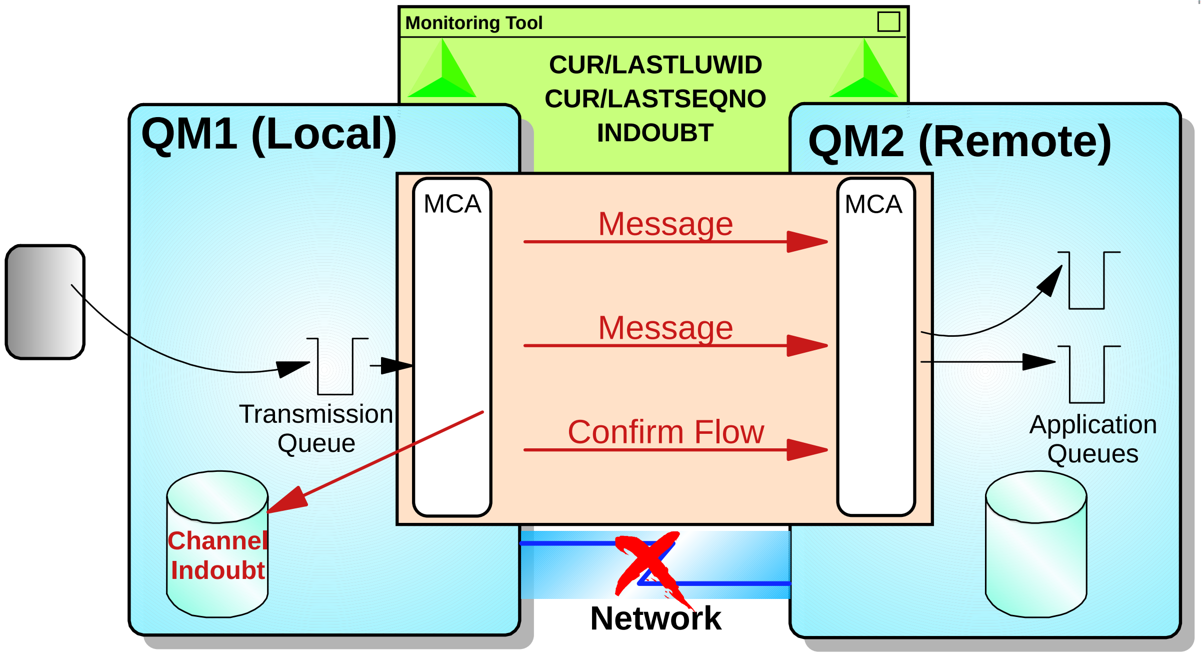
As we have already mentioned, the heartbeat interval used by the sending end of the channel to periodically send this heartbeat message, is a negotiated value, so both ends know what the value is. Without a heartbeat the receiver end of the channel can not predict when the next piece of data should arrive down the channel since the arrival of messages is non-deterministic. However, with a heartbeat the receiver 'knows' that within the heartbeat interval either a message or a heartbeat should arrive. If it doesn't then the communications session must have broken. This is used as the basis for TCP/IP network failure recognition. For CLNTCONN channels, this time-out is also used while waiting for a response to an MQI call (except when the SVRCONN is on z/OS at a version prior to V6 which does not support the client variant of heartbeats. Client heartbeats are supported in WebSphere MQ for z/OS V6).

The receiver will actually time-out if no data is received within twice the Heartbeat interval if the negotiated Heartbeat Interval is less than 60 seconds, or 60 seconds beyond the negotiated heartbeat interval if it is greater than or equal to 60 seconds, by default, before assuming there has been a communications failure. This qualifier can actually be tuned to reflect your network latency, on the distributed platforms, by changing the environment variable MQRCVBLKTO. There is also another environment variable MQRCVBLKMIN which allows you to set a minimum value to be used when calculating the timeout based on the heartbeats. The behaviour can be set on WebSphere MQ for z/OS V6 by the queue manager parameters RCVTTYPE to qualify the timeout (and previously in V5.3.1 by the use of XPARM attributes) RCVTIME to set the timeout and RCVTMIN to set the minimum value.

So.. a small value will detect failures more quickly but may also be more expensive in terms of network traffic and CPU resource if your channel is lightly loaded.

In channel status you see that a receiving type chanel, not surprisingly spends it's idle time in SUBSTATE(RECEIVE). If you monitor the value in NETTIME, this will give you an indication of your network latency.

Assured Delivery



- Synchronisation Data written to disc at both ends
  - Allows rescynchronisation after a network failure
  - Messages in an in-doubt batch cannot be reallocated by clustering algorithm

Assured Delivery

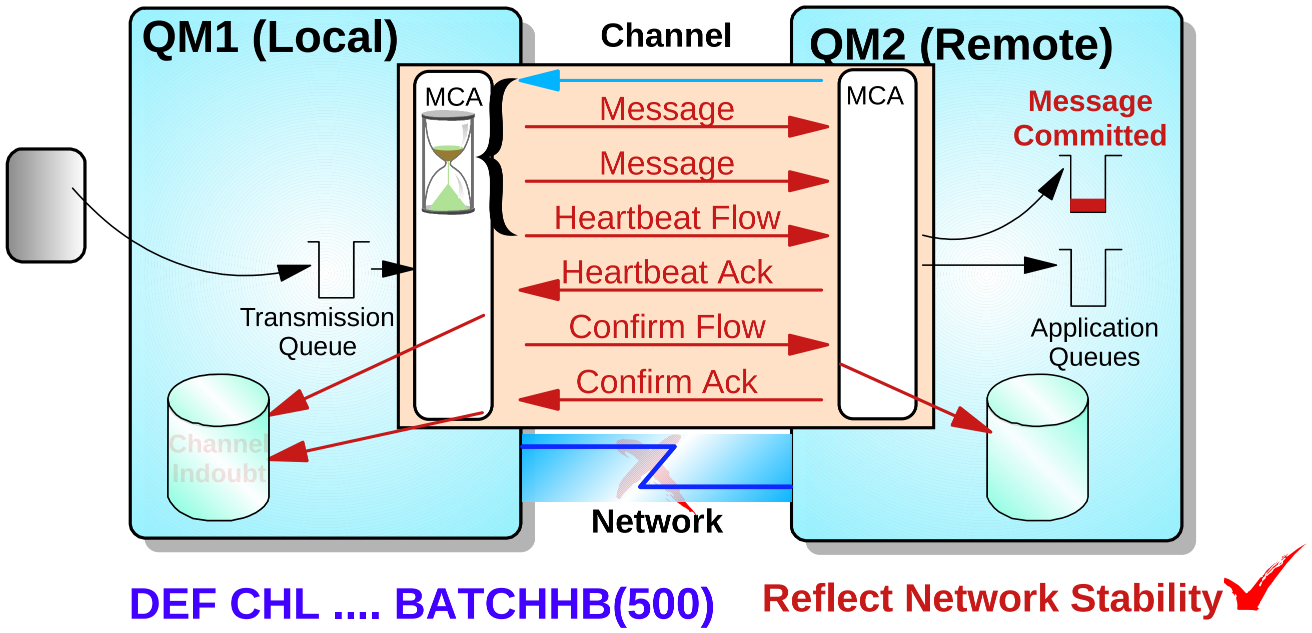
NOTES

Here we can see the two ends of the channel transferring a batch of messages. Note how the MCA's write data to disk at the end of the batch. This is only done for recoverable batches. The data written contains the transaction id of the transaction used at the sending end to retrieve all the messages. Once the sender end has issued a 'confirm' flow to its partner it is 'indoubt' until it receives a response. In other words, the sender channel is not sure whether the messages have been delivered successfully or not. If there is a communications failure during this phase then the channel will end indoubt. When it reconnects to it's partner it will notice from the data store that it was indoubt with its partner and so will ask the other channel whether the last batch of messages were delivered or not. Using the answer the partner sends, the channel can decide whether to commit or rollback the messages on the transmission queue.

This synchronisation data is viewable by issuing a DIS CHSTATUS(*) SAVED command. The values displayed should be the same at both ends of the channel.

Note that if the channel is restarted when it is indoubt it will automatically resolve the indoubt. However, it can only do this if it is talking to the same partner. If the channel attributes are changed or a different Queue Manager takes over the IP address or a different channel serving the same transmission queue is started then the channel will end immediately with message saying that it is still indoubt with a different Queue Manager. The user must start the channel directing it at the correct Queue Manager or resolve the indoubt manually by issuing the RESOLVE CHANNEL command. Note that in this case the user should use the output from DIS CHS(*) SAVED to ensure that the correct action COMMIT or BACKOUT is chosen.

Batch Heartbeats



DEF CHL .... BATCHHB(500) **Reflect Network Stability**

- **If time since last communication is greater than Interval, heartbeat flow is sent**

Batch Heartbeats

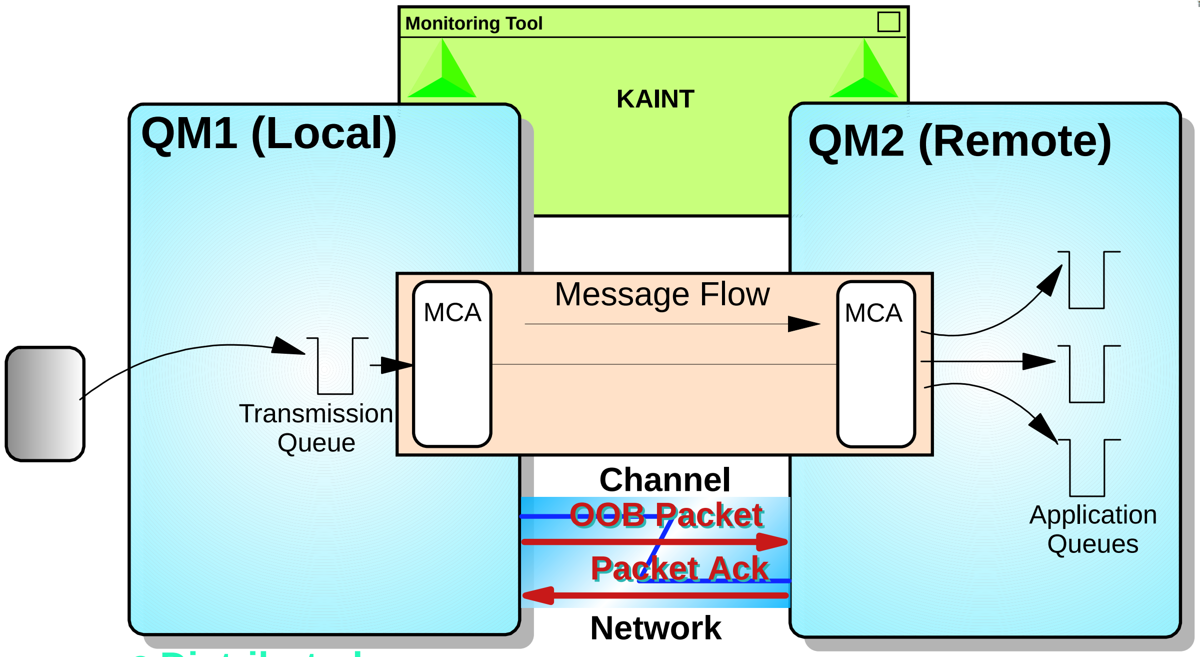
N O T E S

The problem with channels that end indoubt is seen with the clustering algorithm. If a clustering channel ends abnormally, any messages that were to be delivered by this channel are reallocated to another channel and another instance of the target cluster queue. They cannot however be reallocated if they are part of a batch of messages that the sender channel is indoubt about whether the receiver channel has got them.

To reduce the likelihood of a channel ending in indoubt state, an extra flow has been added to the channel protocol. This flow is known as a batch heartbeat. This flow is sent after the messages for the batch have been sent, but before the channel marks itself to be indoubt and sends the confirmation flow. Having received a response from the receiving end of the channel, we can be fairly sure that we can complete the batch. There is now an extremely small window where the receiver channel can end and leave us stuck in indoubt state. Without the batch heartbeat flow this window is much larger (up to the time since the last flow received from the partner).

To determine whether this batch heartbeat flow is sent, we note the time we last received a flow from our partner. If this time was longer ago than the interval specified in the BatchHeartbeat or BATCHHB parameter on the channel, then we will send the flow, otherwise we will not. The interval specified is therefore a reflection of the stability of your network. This interval is specified in milliseconds.

TCP/IP Keepalive



- **Distributed**
  - **QM.INI - KEEPALIVE=yes**
- **z/OS**
  - **ALTER QMGR TCPKEEP(YES)**
  - **DEF CHL ... KAINT(360)**

TCP/IP Keepalive

NOTES

TCP/IP also has its own heartbeat protocol, called KeepAlive, allowing it to recognise network failures. The use of KeepAlive is recommended.

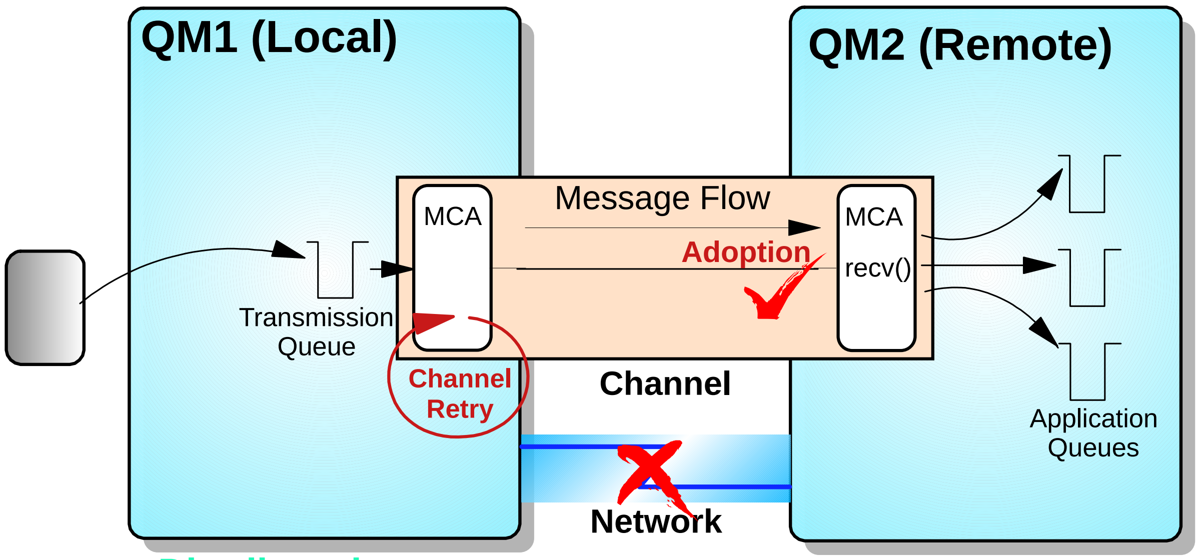
On z/OS only, this can be configured as a channel parameter, allowing different channels to have specific time-out values, or simply use the system values. This channel parameter still requires KeepAlive to be switched on in TCP/IP before it can be used. KAINT can be specified as a number of seconds; or the value AUTO which means the actual KAINT value used is 60 seconds more than the negotiated heartbeat. The actual value used for this channel instance is displayed in channel status.

On all platforms, this can be configured on a Queue Manager basis, on Distributed in the qm.ini file or equivalent and on z/OS using ALTER QMGR. The actual values used for the timers and counters in this case, are set for the entire TCP/IP stack on the machine. Typical values are for network failure to be noticed after a default 2 hours. This is probably too high for a WebSphere MQ environment, but other applications on the same machine may also be wanting to use KeepAlive and so values may need to be a compromise between the requirements of these different applications.

You need to read your TCP/IP manuals to discover how to configure KeepAlive. For AIX, use the "no" command. For HP-UX, use the "nettune" command. For Windows NT, you need to edit the registry. On z/OS, update your TCP/IP PROFILE dataset and add/change the INTERVAL parameter in the TCPCONFIG section.Note: On z/OS TCP/IP APAR PQ75195 is recommended

The use of heartbeats and receive wait time out remove the need for KeepAlive when both ends support both options, however KeepAlive can still be used as well. When communicating with WebSphere MQ implementations which do not support both heartbeats and the non-blocking reads, you should still use KeepAlive. Keepalive is strongly recommended for SVRCONN channels since even if client heartbeats are available, they are only used during MQGETs.

Adopt MCA



- Distributed
  - QM.INI - AdoptNewMCA, AdoptNewMCATimeout, AdoptNewMCACheck
- z/OS
  - ALTER QMGR ADOPTMCA(ALL) ADOPTCHK(ALL)

Adopt MCA

N O T E S

If a channel fails, for example a communications error, there is no guarantee that both ends of the channel will detect the failure at the same time. Often the sending end of the channel detects the failure first since it is the end trying to send messages down the socket. What will happen therefore is that the sender will try to reconnect to the target Queue Manager. When the connection comes into the target machine the Queue Manager sees that it already has a channel running with that name and from that location. Consequently the new connection is rejected.

By configuring AdoptNewMCA you can tell the Queue Manager that if a new connection arrives and there is already a channel with that name running from the same network address and from the same Queue Manager then the existing channel should be ended and replaced with this new instance. In other words, the channel instance should be 'adopted' by the new connection.

The various configuration parameters in the channels stanza of QM.INI or equivalent (or on z/OS use ALTER QMGR) allow the behaviour of Adopt to be changed slightly. Set the list of channel types which may be adopted in this manner to turn on adoption

```
AdoptNewMCA=SVR,SDR,RCVR,CLUSRCVR,ALL,FASTPATH
ALTER QMGR ADOPTMCA(ALL/NO) (on z/OS)
```

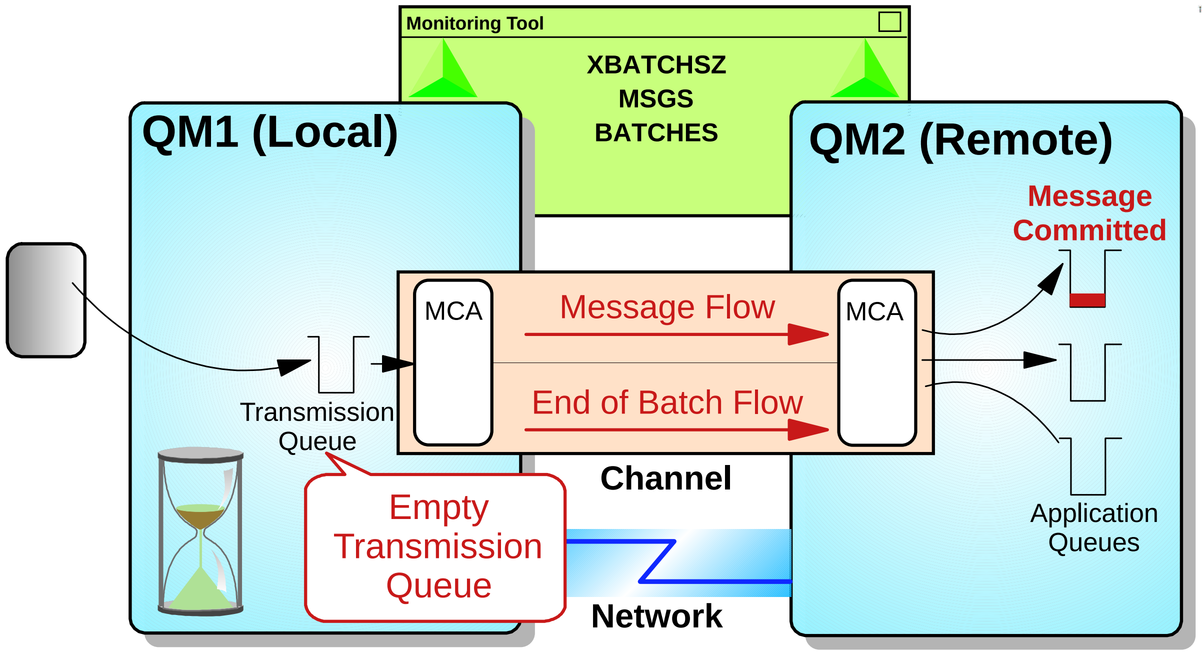
Set the wait interval between ending the 'adopted' channel nicely and ending it immediate.

```
AdoptNewMCATimeout=60
```

Clearly the channel name must match but additionally you can specify that the network address must be the same and also the remote Queue Manager name. This prevents a rogue connection from a different machine on the network from adopting already running channels. Set which attributes must match between the Adopted and new connections.

```
AdoptNewMCACheck=QM,ADDRESS,NAME,ALL
ALTER QMGR ADOPTCHK(NONE/QMNAME/NETADDR/ALL) (on z/OS)
```

Batch Interval



- <u>Minimum</u> lifetime of Batch of Messages

DEF CHL .... BATCHINT(100)

- **Recoverable messages will arrive no sooner than BATCHINT**

Batch Interval

N O T E S

A recoverable batch of messages is ended as soon as the transmission queue is empty. This can lead to more line-turnarounds and disk activity than necessary. Consider an application putting 3 persistent messages to a transmission queue. These is a good chance that the first message will be retrieved by the channel before the second message is put. Consequently the channel will complete the batch and the second and third message will actually go in a second batch.

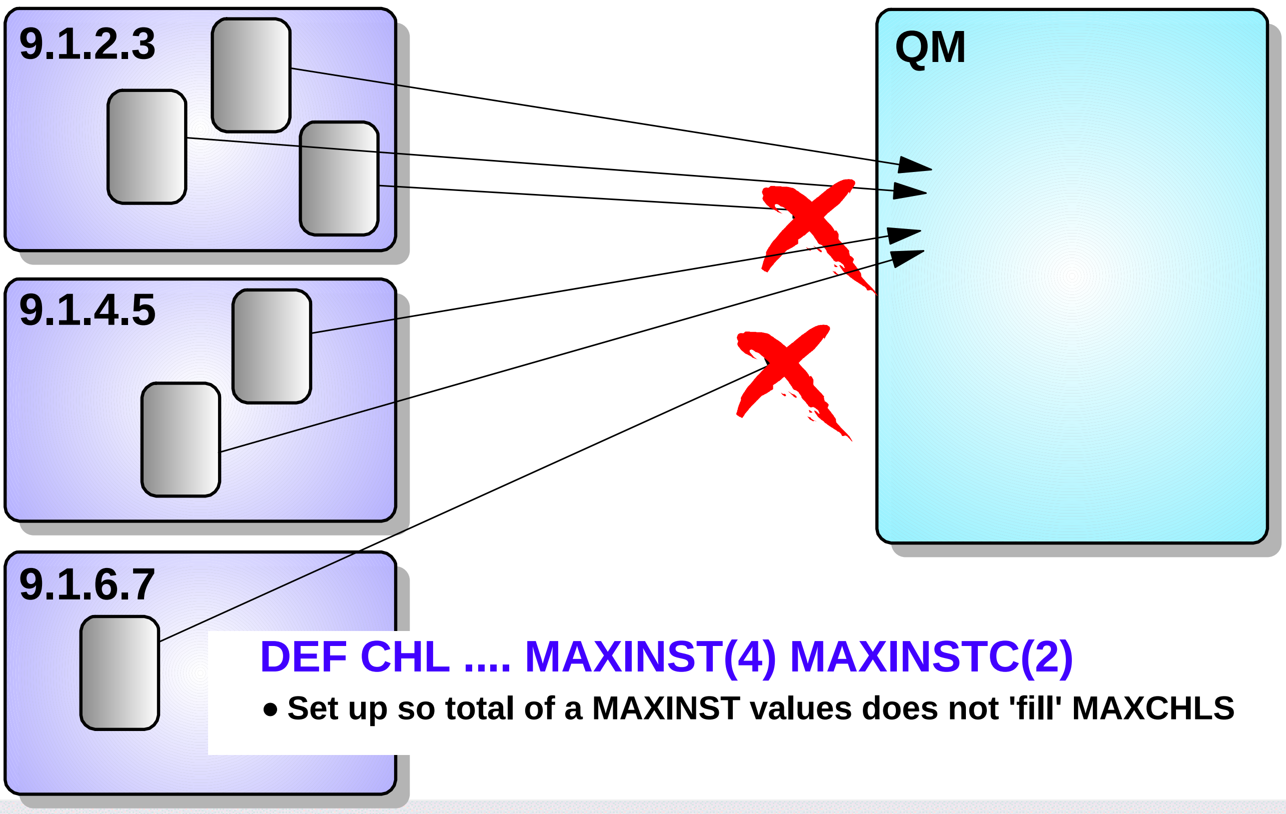
By adding a small delay to the batch, say, BATCHINT(100) then all 3 messages would be sent in the first batch with only half the disk activity. Note that the messages may well appear to take longer to be delivered though since there is a delay of at least 100ms. BATCHINT has no effect on non-persistent messages NPMSPEED(FAST) since these messages are put outside of syncpoint. Usually applications send database queries as non-perisistent which are time critical and database updates as persistent messages which are not so time critical.

The net effect and value of BATCHINT depends greatly on the MQ messaging network and messaging profile. The greater the number of channels on the machine the larger the potential savings.

One way of helping to determine whether there would be much advantage of BATCHINT is to look at the effective batch size. Issue a DISPLAY CHSTATUS and divide the number of messages (MSGS) by the number of batches (BATCHES) or ues the XBATCHSZ indicator in WebSphere MQ V6. If this number is very low, (say between 1 and 2) then the channel is having to commit for each 1 or 2 messages which is inefficient and may benefit from BATCHINT. Note that this is only an indication since if all the messages are non-persistent BATCHINT won't help at all.

Another thing to look for is the disk usage of the channel processes. If you see large disk utilisation then there's a good chance BATCHINT will reduce this cost and improve overall throughput.

Protect Max Channels



DEF CHL .... MAXINST(4) MAXINSTC(2)

- **Set up so total of a MAXINST values does not 'fill' MAXCHLS**

Protect Max Channels

NOTES

New attributes on your server-connection channels allow you to restrict the number of client-connection instances that can connect in. Now you can configure your system so that server-connection instances cannot fill up your maximum number of channels.♂

There are in fact two attributes on your server-connection definition.

MAXINST restricts the number of instances in total for the specific channel name.

MAXINSTC restricts the number of instances from a specific IP address for that channel name.

Performance

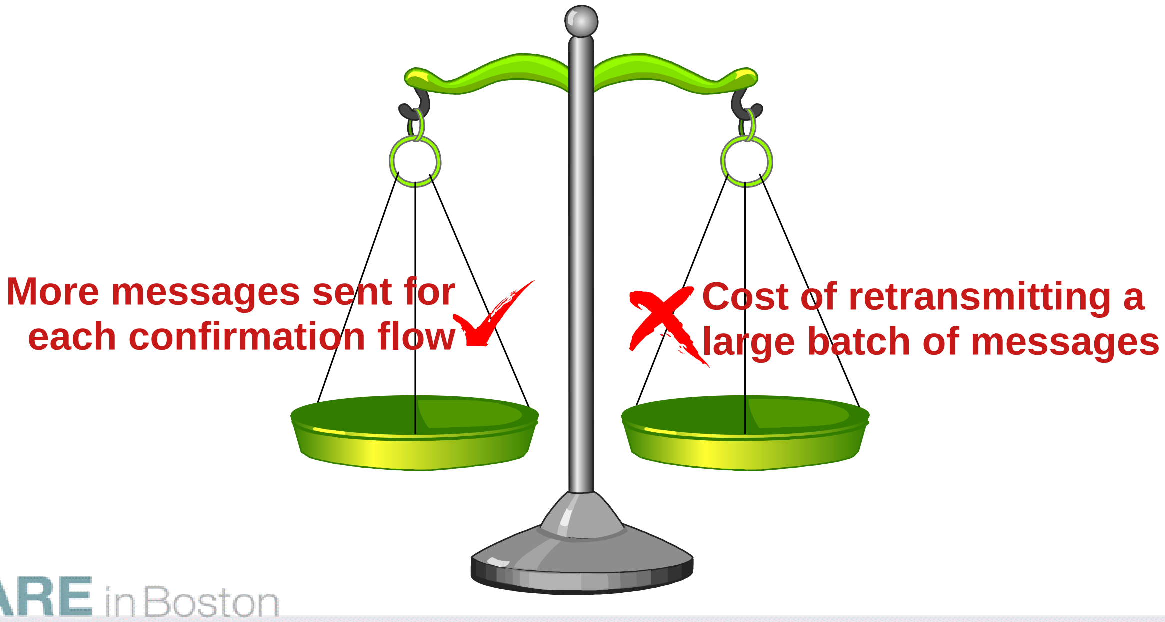
- Some questions to think about
  - What is arrival rate on transmission queue?
  - What are required response times?
  - How long does a batch take to transmit?
  - What is the startup time for a channel?
- Answers depend on your environment
  - Application Usage
  - Operating System/Hardware
  - Network Bandwidth
  - Message Sizes

Performance configuration

- Use BatchInterval to keep batches open
  - if arrival rate is slightly slower than sending complete batch
  - may increase latency, but improve total throughput
- Set DisconnectInterval high enough so channels kept alive
  - Reduce number of times a channel is started/stopped/restarted
  - Unless network charges by "active sessions" or you are hitting MaxChannels limit
- Three Scenarios
- See Capacity Planning Guides
  - SupportPac MP00
  - SupportPac MP16

Scenario 1

- Bulk transfer of pre-loaded transmission queue
  - Use a large BatchSize (default of 50 is good),
    - unless using very large messages (cost of retransmission)
  - BatchInt should be 0


More messages sent for each confirmation flow ✓
✗ Cost of retransmitting a large batch of messages

Scenario 1

NOTES

Our first performance tuning scenario is a bulk transfer of the messages on a pre-loaded transmission queue.

You are aiming here to be able to send the messages in large batches therefore reducing the cost of the end of batch processing per message. So a large Batch Size should be used (the default of 50 would be good here) and Batch Interval should be set to 0. There is no need to hold the batch open in this scenario since all the messages are pre-loaded on the transmission queue.

Your performance balance is that of how many messages you should send in one batch to get a low cost of end of batch processing per message, weighed up against the cost of retransmitting that entire batch should network connectivity be lost part way through. If you are moving very large messages, this balance may need to be achieved with a smaller Batch Size than 50.

Scenario 2

- Trickle transfer for deferred processing
  - Exact numbers depend on arrival rate
  - Try to achieve high batches
  - Set BatchInt to acceptable delay before transmission e.g. 1 minute
  - But don't make it very large: otherwise you get long-running UoW

More messages sent for each confirmation flow ✓
✗ Delay before messages are available at target

Scenario 2

N O T E S

Our second performance tuning scenario is a trickle transfer of messages for deferred processing.

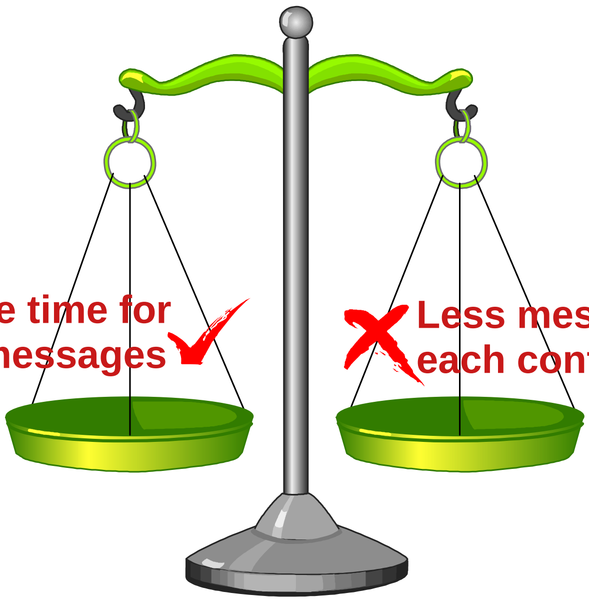
You are aiming here to achieve a reasonable batch size of message therefore reducing the cost of end of batch processing per message. This is a classic use of Batch Interval. Messages are arriving at a slightly slower rate than the channel can process them. Holding the batch open for a little longer allows more messages to be batched up together.

Your performance balance is that of how many messages you can send in one batch, weighed up against the fact that recoverable messages will appear to take longer to be delivered since they are not committed on the target queues until the batch completes, i.e. after at least a Batch Interval.

If delivery time is not an issue, be careful not to set Batch Interval too large, since you then run the risk of generating long running Units of Work.

Scenario 3

- Synchronous Request/Reply
  - May not want to use BatchInt if response time is key
  - If persistent volume is low consider a smaller BatchSize
  - For non-persistent message flows, use NPMSPEED(FAST) and a non-zero BatchInt


Faster response time for messages ✓
✗ Less messages sent for each confirmation flow

Scenario 3

NOTES

Our third performance tuning scenario is a synchronous request/reply model where the response time does matter.

This is a similar situation to scenario 2, except this time the balance is weighted more towards a faster response time rather than more messages in a batch.

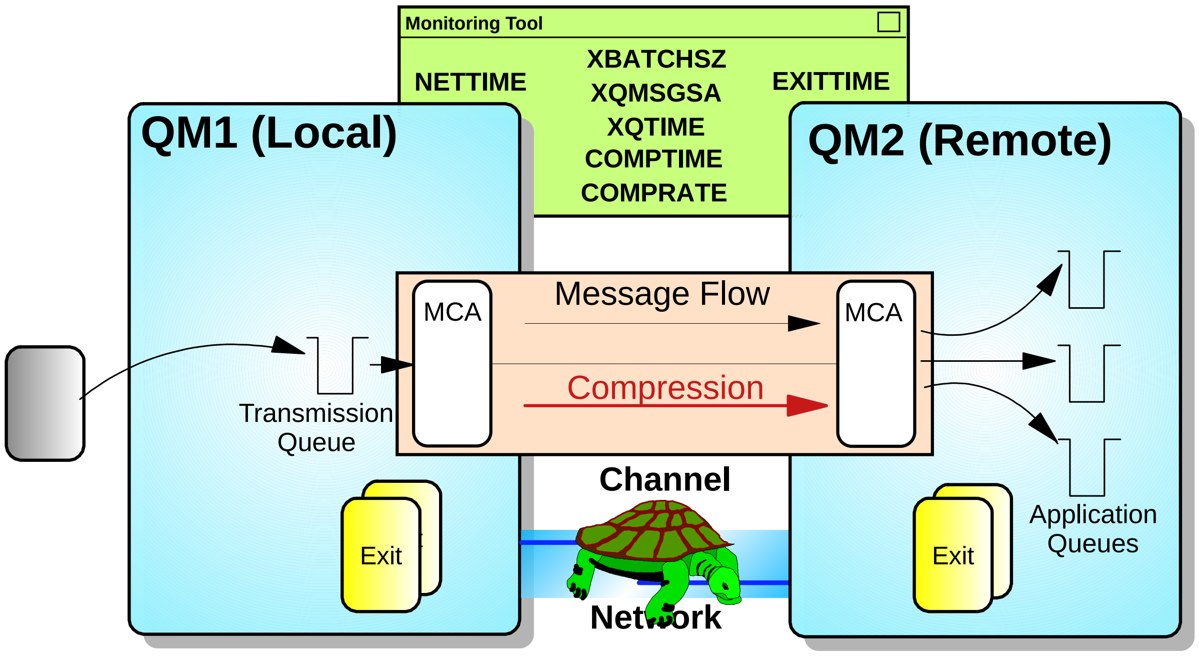
If the persistent message volume is low, you may wish to consider using a smaller batch size and either not using Batch Interval at all, or using a fairly low value in the order of a few seconds to ensure messages are available in a timely fashion.

If you have non-persistent messages, you should use Non-Persistent Message Speed or NPMSPEED of FAST, and set a non-zero Batch Interval, since non-persistent messages delivered over a fast channel are available immediately. The Batch Size value is still used but does not affect the response time.

To give some example figures:

With an expected rate of 30 non-persistent msg/sec, set BatchInt=2000 (2secs) and BatchSize=50 for an effective batch size of 50.

Channel Monitoring



DISPLAY CHSTATUS(*channel-name*) MONITOR

Channel Monitoring

NOTES

We have looked at a number of the channel status fields throughout this presentation, some of which are new in V6. This foil shows all the new monitoring fields in V6. A number of these fields are a pair of indicators, a short term and a long term rolling, weighted average indicator.

We already briefly mentioned EXITTIME earlier. This inidicator shows the amount of time spent in exits per message. If you have various different exits, e.g. message, send, recieve, message retry, this will be a combined amount of time spent in exits for a message.

WebSphere MQ V6 adds compression to channels. This is outside the scope of this presentation, except to mention that channel status provides indicators for the time spent compressing a message and the rate of compression achieved.

NETTIME is an indicator of the time taken to do a round trip on the network. This is done whenever a channel confirms a batch, or sends a heartbeat (the times when an answer is requested from the partner). NETTIME only measures network time, and does not include the MQCMIT for example, that will be done as part of the confirmation of a batch.

XBATCHSZ is an indicator of the achieved batch size, which was discussed earlier.

XQTIME is an indicator of the time messages spend on the transmission queue

XQMSGSA shows how many messages are available to be moved by a cluster-sender channel. This is effectively the current depth of the sub-queue for that channel (cluster-sender channel's MQGET from the SYSTEM.CLUSTER.TRANSMIT.QUEUE by Correl-Id). Since this could become an expensive value to calculate, if the number exceeds 999, we stop counting and display 999.

Channel Status Parameter Reference

Channel Status	Description
MONITOR	Displays the set of Channel Monitoring fields turned on by MONCHL (see table below).
STATUS	Channel state - what the channel is doing
SUBSTATE	More granular status - what the channel is doing in detail
LONGRTS, SHORTRTS	Number of channel retries left to do
LSTMSGDA, LSTMSGTI	When the last message was sent down the channel
HBINT	The negotiated heartbeat interval used on this instance
CURLUWID, LASTLUWID, CURSEQNO, LASTSEQNO	Batch status fields
INDOUBT	Is the sender currently indoubt with its partner
KAINT	The current value of keepalive being used on this channel
MSGS, BATCHES	How many messages, in how manay batches sent so far

Channel Status Monitor	Description
COMPRATE	The compression rate achieved
COMPTIME	The time spent doing compression
EXITTIME	The time spent running exits
NETTIME	The time a network round trip takes
XBATCHSZ	Achieved Batch Size
XQMSGSA	Messages available for CLUSSDR
XQTIME	The time messages spend on the Transmission Queue

Parameter Reference

Channel	Description
HBINT	Heartbeat Interval
BATCHSZ	Maximum number of Messages in a Batch
BATCHINT	Minimum lifetime of a Batch of Messages
DISCINT	Interval after which a channel will end
SHORTRTY, SHORTTMR LONGRTY, LONGTMR	Channel retry counts and timers
MRRTY, MRTMR MREXIT, MRDATA	Message retry counts and timers and exit
MAXINST, MAXINSTC	Restrict the number of inbound clients

QM.INI - Channel Stanza	ALT QMGR	Description
AdoptNewMCA	ADOPTMCA	Adopt Channel types
AdoptNewMCATimeout		Adopt Quiesce time-out
AdoptNewMCACheck	ADOPTCHK	Adopt parameter check
KeepAlive	TCPKEEP	Use Keepalive

Environment Variables	ALT QMGR	Description
MQRCVBLKTO	RCVTIME	Receive Wait Timeout
	RCVTTYPE	Receive Wait Timeout Qualifier
MQRCVBLKMIN	RCVTMIN	Receive Wait Minimum Timeout

Download this presentation

(or something very like it)

- SupportPac MD0C
  - **Translated into a number of different languages**

Thank-you !

Reference

N O T E S

These two sets of tables list the channel status parameters we have looked at through this presentation and those attributes of a channel which affect the performance and recovery capabilities.

Some of the attributes are part of the true channel definition and others are setup in other places such as an INI file.

Full details of the parameters on the channel definition can be found in the InterCommunication book.

On the distributed platforms those parameters that are in the QM.INI file are in the Channels stanza or on Windows they are implemented in the Registry

Key : HKEY_LOCAL_MACHINE/SOFTWARE/IBM/MQSeries/QueueManager/<QMNAME>

Channel Initiator parameters provide some of the same parameters on z/OS that can be found in the QM.INI file on the distributed platforms. These parameters are specified through ALTER QMGR on WebSphere MQ for z/OS V6. Previously they were specified through the XPARMS which are built using the CSQ6CHIP macro.