Agenda

• **Queue Managers**
  – Queue Manager Clusters
  – Hub and Spoke arrangement
  – Pub/Sub Topologies
  – Queue Sharing Group

• **Clients**
  – MQ Clients
  – MQTT Clients

• N.B. Some of the features we will look at were introduced in recent releases of WebSphere MQ. We will highlight the releases in question as we go through.
Why is Topology important?

When you first start using WebSphere MQ, you probably have one queue manager, and connecting applications together is easy as they just connect to that queue manager. Maybe you even think about High Availability at this point and make sure that this single queue manager is not a Single Point of Failure (SPoF).

As your use of WebSphere MQ grows, you start to add in more queue managers, and as the number increases you have to think about how to join these all together - the definitions to create links between the queue managers; the security of the overall system; the arrangement of the queue managers and what connectivity you will have between them.

The arrangement of these queue managers and the clients that connect into these queue managers is your topology and it is worth understanding what options you have in this area.
Multiple Queue Managers

- **Capacity**
  - Although some users do not use their queue managers enough!

- **High availability**
  - Keeping the service provided by your applications available even with an outage of one machine

- **Localisation**
  - Queue managers near the applications

- **Isolation**
  - One application can’t impact another application

- **Span-of-control**
  - I want my own queue manager!

---

Multiple Queue Managers - Notes

- There are many different reasons why you end up with multiple queue managers.
- First there is perhaps the most obvious reason and that is capacity. One queue manager simply cannot go on forever holding the queues for more and more applications. Eventually the resources on the machine will be exhausted and you need to have more queue managers to cope. It is worth noting at this point that some users of WebSphere MQ have gone to the other extreme though, and are not loading up their queue manager’s enough, but that is a discussion for another time!
- Another very good reason is for availability. Having more than one single point means that there is redundancy in the system, and if one queue manager or machine is unavailable, other queue managers in the system are able to carry on running your work.
- Localisation is also an often quoted reason for having more queue managers. That is, ensuring that there is a queue manager geographically close to where the applications run.
- Isolation of one application from another is sometimes a reason for multiple queue managers, whether ensure things like a full DLQ from one application doesn’t impact another, to any regulatory requirements for separation.
- And finally, another common, but perhaps not so well advised reason is the parochial-ness that occurs in some departments who want to own and run their own queue manager even when there is a department who’s job it is to do this for them. Although of course, when the span-of-control differs, this can be advised.
Goals of Clustering

- Multiple Queues with single image
- Failure isolation
- Scalable throughput
- MQI applications to exploit clusters transparently
- Definition through usage (MQOPEN)
- MQGET always local

Consider a client using the black queue that is available in the cluster on three server queue managers. A message is MQPUT by the client and is delivered to "one" of the servers. It is processed there and a response message sent to a ReplyToQueue on the client queue manager.

In this system, if a server becomes unavailable, then it is not sent any further messages. If messages are not being processed quickly enough, then another server can be added to improve the processing rate.

It is important that both these behaviors are achieved by existing MQI applications, i.e. without change. It is also important that the administration of clients and servers is easy. It must be straightforward to add new servers and new clients to the server.

We see how a cluster can provide a highly available and scalable message processing system. The administration point in processing is MQOPEN as this is when a queue or queue manager is identified as being required by an application.

Note that only one message is sent to a server; it is not replicated three times, rather a specific server is chosen and the message sent there. Also note that MQGET processing is still local, we are not extending MQGET into the network.
The purpose of clustering

- **Simplified administration**
  - Large WMQ networks require many object definitions
    - Channels
    - Transmit queues
    - Remote queues
- **Workload balancing**
  - Spread the load
  - Route around failures
- **Flexible connectivity**
  - Overlapping clusters
  - Gateway Queue managers
- **Pub/sub Clusters**

---

How can we process more messages?

- It would be nice if we could place all the queues in one place. We could then add processing capacity around this single Queue manager as required and start multiple servers on each of the processors. We would incrementally add processing capacity to satisfy increased demand. We could manage the system as a single entity. A client application would consider itself to be talking to a single Queue manager entity.
- Even though this is highly desirable, in practice it is almost impossible to achieve. Single machines cannot just have extra processors added indefinitely. Invalidation of processor caches becomes a limiting factor. Most systems do not have an architecture that allows data to be efficiently shared between an arbitrary number of processors. Very soon, locking becomes an issue that inhibits scalability of the number of processors on a single machine. These systems are known as "tightly coupled" because operations on one processor may have a large effect on other processors in the machine cluster.
- By contrast, "loosely coupled" clusters (e.g. the Internet) have processors that are more or less independent of each other. Data transferred to one processor is owned by it and is not affected by other processors. Such systems do not suffer from processor locking issues. In a cluster solution, there are multiple consumers of queues (client queue managers) and multiple providers of queues (server queue managers). In this model, for example, the black queue is available on multiple servers. Some clients use the black queue on both servers, other clients use the black queue on just one server.
- A cluster is a loosely coupled system. Messages flow from clients to servers and are processed and responses messages sent back to the client. Servers are selected by the client and are independent of each other. It is a good representation of how, in an organization, some servers provide many services, and how clients use services provided by multiple servers.
- The objective of WebSphere MQ clustering is to make this system as easy to administer and scale as the Single Queue Manager solution.
Simplified administration

- Large WMQ networks require many object definitions
- Manually defining the network
  - For each Queue Manager you connect to:
    - Transmission Queue
    - Sender Channel
    - Remote Queue (Optional) – so we won’t count it
    - And a single generic Receiver Channel
- Cluster network
  - Cluster-Sender channel to two full repositories
  - Cluster-Receiver channel as the model back to me.

#Objects = ((#QMgrs-1) * 2 +1) * #Qmgrs
Which is:
#Objects = 2 x #Qmgrs

When creating a fully connection mesh of queue managers using distributed queuing, you need to create the sender channel and transmission queues for connectivity to each queue manager you wish to connect to, plus you’ll need a single generic receiver channel for them to connect back.

For each queue manager you want to connect to, that is a minimum of two objects – let’s say we don’t count queue remote definitions as they are optional per connected queue manager, plus the one receiver. This ends up with a logarithmic algorithm which grows exponentially.

Whereas for the cluster example, we just need two definitions on each queue manager, regardless of how many queue managers are in the network, so it scales nice and linearly.

For more details see
Split cluster transmit queue

- Much requested feature for various reasons...
- Separation of Message Traffic
  - With a single transmission queue there is potential for pending messages for cluster channel 'A' to interfere with messages pending for cluster channel 'B'
- Management of messages
  - Use of queue concepts such as MAXDEPTH not useful when using a single transmission queue for more than one channel.
- Monitoring
  - Tracking the number of messages processed by a cluster channel currently difficult/impossible using queue monitoring (some information available via Channel Status).
- Not about performance...

Split cluster xmit q background - Notes

- This has been a very long standing requirement from a number of customers
- All the reasons on this slide are valid, but the number one reason often quoted in requirements was ‘performance’
  - In reality splitting out the transmit queue does not often buy much here, hence often other solutions (e.g. improving channel throughput) were really needed.
- Main reason for delivery now is to allow application separation
Split cluster transmit queue - automatic

- New Queue Manager attribute which effects all cluster-sdr channels on the queue manager
  - ALTER QMGR DEFCLXQ( SCTQ | CHANNEL )

- Queue manager will automatically define a PERMANENT-DYNAMIC queue for each CLUSSDR channel.
  - Dynamic queues based upon new model queue “SYSTEM.CLUSTER.TRANSMIT.MODEL”
  - Well known queue names: “SYSTEM.CLUSTER.TRANSMIT.<CHANNEL-NAME>”

- Authority checks at MQOPEN of a cluster queue will still be made against the SYSTEM.CLUSTER.TRANSMIT.QUEUE even if CLUSSDR is selected.

Splitting out the S.C.T.Q. per channel
Split cluster transmit queue - manual

- Administrator manually defines a transmission queue and using a new queue attribute defines the CLUSSDR channel(s) which will use this queue as their transmission queue.
  - `DEFINE QLOCAL(APPQMGR.CLUSTER1.XMITQ)
    CHLNAME(CLUSTER1.TO.APPQMGR) USAGE(XMITQ)`

- The CHLNAME can include a wild-card at the start or end of to allow a single queue to be used for multiple channels. In this example, assuming a naming convention where channel names all start with the name of the cluster, all channels for CLUSTER1 use the transmission queue CLUSTER1.XMITQ.
  - `DEFINE QLOCAL(CLUSTER1.XMITQ) CHLNAME(CLUSTER1.*) USAGE(XMITQ)`
  - Multiple queues can be defined to cover all, or a subset of the cluster channels.

- Can also be combined with the automatic option
  - Manual queue definition takes precedence.

Splitting out by cluster (or application)
Hub and Spoke Queue Managers
The arrangement of queue managers shown here is a common one, where one, or sometimes more than one, but still a relatively small number of queue managers are considered the Hub, and the remaining queue managers are spokes. All traffic between Spokes goes via a Hub, and so it is clear that the Hub queue manager(s) is a important point in the Topology and should be made as Highly Available as possible.

Sometimes the spokes are client connections, or each of the Spoke queue managers could be acting as a Hub for clients. Either way, we will look at Clients a little later.
Introduction to Failover and MQ

• **Failover is the automatic switching of availability of a service**
  – For MQ, the “service” is a queue manager

• **Traditionally the preserve of an HA cluster, such as HACMP**

• **Requires:**
  – Data accessible on all servers
  – Equivalent or at least compatible servers
    • Common software levels and environment
  – Sufficient capacity to handle workload after failure
    • Workload may be rebalanced after failover requiring spare capacity
  – Startup processing of queue manager following the failure

• **MQ offers two ways of configuring for failover:**
  – Multi-instance queue managers
  – HA clusters

### Requirement to access data

– Shared disks – for an HA cluster, usually “switchable” between the servers
– Mirrored disks – must be truly synchronized mirror for integrity

### Requirement for client connectivity

– IP address takeover (IPAT) is generally a feature of failover environments
– If a queue manager changes IP address, intelligent routers can hide this or MQ network configuration can be defined with alternative addresses

### Servers must be equivalent

– Common software levels – or at least compatible, to allow for progressive upgrade of the servers
– Common environments – paths, userids, security

### Sufficient capacity to handle workload

– Often, workload will be redistributed following a failover. Often, the systems are configured for mutual takeover where the workload following failover is doubled since the surviving servers must handle the traffic intended for both.
Multi-instance Queue Managers

- **Basic failover support without HA cluster**

- **Two instances of a queue manager on different machines**
  - One is the “active” instance, other is the “standby” instance
  - Active instance “owns” the queue manager’s files
    - Accepts connections from applications
  - Standby instance monitors the active instance
    - Applications cannot connect to the standby instance
    - If active instance fails, standby performs queue manager restart and becomes active

- **Instances are the SAME queue manager – only one set of queue manager data**
  - Queue manager data is held in networked storage

---

“Basic failover”: no coordination with other resources like disks, IP addresses, databases, user applications. There is also no sophisticated control over where the queue managers run and move to (like a 3-node HACMP cluster, for example). Finally, once failover has occurred, it is necessary to manually start a new standby instance.

Architecturally, this is essentially the same as an existing HACMP/VCS setup, with the data shared between systems. It does not give anything “stronger” in terms of availability – but we do expect the typical takeover time to be significantly less. And it is much simpler to administer.

Just as with a configuration using an HA cluster, the takeover is in essence a restart of the queue manager, so nonpersistent messages are discarded, queue manager channels go into retry, and so on.
Multi-instance Queue Managers

1. Normal execution

- Machine A
  - QM1 Active instance
  - MQ Client
  - Can fail-over

- Machine B
  - QM1 Standby instance
  - MQ Client
  - 168.0.0.2

Owns the queue manager data

2. Disaster strikes

- Machine A
  - MQ Client
  - 168.0.0.1
  - Networked storage
  - MQ Client
  - Locks freed
  - Client connections broken

- Machine B
  - QM1 Standby instance
  - 168.0.0.2
  - Networked storage
Multi-instance Queue Managers

3. FAILOVER

Standby becomes active

MQ Client

MQ Client

network

Machine B

QM1

Active instance

168.0.0.2

network

networked storage

Owns the queue manager data

MQ Client

MQ Client

network

Machine B

QM1

Active instance

168.0.0.2

network

networked storage

Owns the queue manager data

4. Recovery complete

Client connections reconnect

Client connection still broken
HA clusters

• **MQ traditionally made highly available using an HA cluster**
  – IBM PowerHA for AIX (formerly HACMP), Veritas Cluster Server, Microsoft Cluster Server, HP Serviceguard, …

• **HA clusters can:**
  – Coordinate multiple resources such as application server, database
  – Consist of more than two machines
  – Failover more than once without operator intervention
  – Takeover IP address as part of failover
  – Likely to be more resilient in cases of MQ and OS defects

• **SupportPac MC91 has been withdrawn**
  – Existing MC91 will work, but is not really appropriate any more
  – Product now contains equivalent function, e.g. on crtmqm

---

**HA Clusters – Notes**

- The collection of servers that makes up a failover environment is known as a cluster. The servers are typically referred to as nodes.

- One node runs an application or service such as a queue manager, while the HA cluster monitors its health. The following example is called a cold standby setup because the other nodes are not running workload of their own. The standby node is ready to accept the workload being performed by the active node should it fail.

- A shared disk is a common approach to transferring state information about the application from one node to another, but is not the only solution. In most systems, the disks are not accessed concurrently by both nodes, but are accessible from either node, which take turns to "own" each disk or set of disks. In other systems the disks are concurrently visible to both (all) nodes, and lock management software is used to arbitrate read or write access.

- Alternatively, disk mirroring can be used instead of shared disk. An advantage of this is increased geographical separation, but latency limits the distance that can be achieved. But for reliability, any synchronous disk writes must also be sent down the wire before being confirmed.
MQ in an HA cluster – Cold standby

1. Normal execution

- MQ Client
- MQ Client

Network

HA cluster

168.0.0.1

Machine A

QM1 Active instance

can fail-over

QM1 data and logs

shared disk

Machine B

2 machines in an HA cluster

2. Disaster strikes

- MQ Client
- MQ Client

Network

HA cluster

168.0.0.1

Machine B

IP address takeover

Shared disk switched

QM1 data and logs

shared disk
3. FAILOVER

MQ in an HA cluster – Cold standby

4. Recovery complete

MQ in an HA cluster – Cold standby
MQ in an HA cluster – Active/active

1. Normal execution

- MQ Client
- Machine A: QM1, Active instance
- Machine B: QM2, Active instance
- Network
- Shared disk
- Network

2. Disaster strikes

- MQ Client
- Machine A: QM1, Active instance
- Machine B: QM2, Active instance
- Network
- Shared disk
- Network
### MQ in an HA cluster – Active/active – Notes

- This configuration is also sometimes called a "mutual takeover" system.
- In normal operation, both machines are running independent queue managers. If one of the systems fails, then this configuration can migrate the failed queue manager to the working machine. So it still appears to applications outside the cluster that you have 2 queue managers. The throughput of each queue manager may degrade (depending on how heavily loaded they run) but at least the work is still getting done.
- With this kind of setup, you probably have a failback capability so that the queue manager can be sent back to its original node when the failure has been corrected. Whether the failback is automatic or not may be your choice, but I'd strongly recommend that it's done manually so that applications which have already connected to the running queue manager do not have their connections broken arbitrarily. You probably want to monitor the workload and only failback when there's not too much work that's going to be disrupted by the failback.
- A number of variations on the themes of cold and hot standby are also possible, for example having 3 nodes to host 2 queue managers (an "N+1" configuration). The availability of these options will depend on the facilities available in your cluster software.
- In this configuration, we've shown the IP address associated with each queue manager being migrated. You will also need to keep a port number reserved for each queue manager (the same number on both machines in the cluster), and have appropriate setup for runmqslr or a queue-manager listener object.
Multi-instance QM or HA cluster?

• Multi-instance queue manager
  ✓ Integrated into the WebSphere MQ product
  ✓ Faster failover than HA cluster and MC91
    • Delay before queue manager restart is much shorter
  ✗ Runtime performance of networked storage
  ✗ More susceptible to MQ and OS defects

• HA cluster
  ✓ Capable of handling a wider range of failures
  ✓ Failover historically rather slow, but some HA clusters are improving
  ✗ Some customers frustrated by unnecessary failovers
  ✗ Require MC91 SupportPac or equivalent configuration
  ✗ Extra product purchase and skills required
THINK GLOBAL – ACT LOCAL

Pub/Sub Topologies
Publish/Subscribe Topologies

- Local Queuing -> Distributed Queuing
- Publish/Subscribe -> Distributed Publish/Subscribe
- Application API calls remain the same
- Administration changes have the effect

Publish/Subscribe Topologies - Notes

- In this presentation we have concentrated on the publish/subscribe API and Administration within a single queue manager. Of course, just as queuing can be on a single queue manager or can be moving messages between multiple queue managers – known as distributed queuing – so can publish/subscribe. We call this distributed publish/subscribe.
- This is the concept (and the features to implement it) that an application may be publishing to a topic on QM1 and other applications may be subscribing on that topic on others queue managers, here QM2 and QM3, and the publication message needs to flow to those other queue managers to satisfy those subscribers.
- The application code stays the same, you still call MQSUB or MQOPEN and MQPUT, the difference, as with distributed queuing is in the administrative set-up of your queue managers.
- We are going to look at the different ways you can set up your queue managers to publish messages to another queue manager.
A pub/sub cluster is a cluster of queue managers, with the usual CLUSRCVR and CLUSSDR definitions, but that also contains a TOPIC object that has been defined in the cluster.

With a cluster you have “any-to-any” connectivity. There are direct links between all queue managers in the cluster. This provides good availability for the delivery of messages, if one route is unavailable, there may well be another route to deliver the messages to the target subscription.

With a TOPIC object defined in the cluster, an application connected to one queue manager in the cluster can subscribe to that topic or any node in the topic tree below that topic and receive publications on that topic from other queue managers in the cluster.

This is achieved by the creation of proxy subscriptions on the queue managers in the cluster, so that when a publication to the topic in question happens on their queue manager, they know to forward it to the appropriate other members of the cluster.

You can view these proxy subscriptions through the same commands we saw earlier. By default proxy subscriptions are not shown to you because the default value for SUBTYPE is USER. If you use SUBTYPE(ALL) or SUBTYPE(PROXY) you will see these subscriptions.

There are a few attributes that are specifically related to Distributed Publish/Subscribe. PUBSCOPE and SUBSCOPE determine whether this queue manager propagates publications to queue managers in the topology (pub/sub cluster or hierarchy) or restricts the scope to just its local queue manager. You can do the equivalent job programmatically using MQPMO_SCOPE_QMGR / MQSO_SCOPE_QMGR.

PROXYSUB is an attribute that controls when proxy subscriptions are made. By default it has value FIRSTUSE and thus proxy subscriptions are only created when a user subscription is made to the topic. Alternatively you can have the value FORCE which means proxy subscriptions are made even when no local user subscriptions exist.
Hierarchies

QMGR attributes
PARENT
PSMODE

DISPLAY PUBSUB

A hierarchy is a parent/child related set of queue managers. It has less availability than a pub/sub cluster as the availability of an intermediate queue manager has an impact on the connectivity of other queue managers and thus the delivery of publications to satisfy subscribers.

It is however a solution that scales well due to the fact that there are less connections overall without the all-to-all connectivity that the pub/sub cluster provides.

In order to set up a hierarchy you nominate your parent queue manager using ALTER QMGR PARENT(queue-manager-name). You don’t need to nominate children, this can be worked out from the fact that they have nominated you as their parent. You can display details about your parent and children queue managers using the DISPLAY PUBSUB command.

The same topic attributes we looked at on the pub/sub cluster page also apply to hierarchies.

It is worth noting that arranging a Hierarchy as a Hub + Spoke topology, i.e. a single layer Hierarchy, and then ensuring the parent/Hub is Highly Available is a good way to counteract the inherent SPoF issues with Hierarchies.
Comparison - Notes

- Pub/sub clusters and hierarchies provide different benefits. You should choose the appropriate topology that gives you the benefits you need.

- Pub/sub clusters are very similar to the Message Broker collectives topology in that you have any-to-any connectivity, a fully-connected set. In fact the queue managers are connected together with all-to-all connectivity which means that all the queue managers in the cluster will have channels running to all the other queue managers in the cluster. This may limit the scalability of this solution, with the number of channels being the gating factor. It is however a very available solution since to one queue manager is a bottle-neck or single point of failure for the cluster. It also provides the same benefits of clustering that you get for queued clusters, that of reduces administration definitions.

- Hierarchies are very similar to, and interoperable with, WebSphere MQ V6 Queued Pub/Sub hierarchies. They have interconnection only between parent and child queue managers and so the loss of one queue manager, or connectivity to it, can cause a break in the link for other queue managers. Less availability therefore, but more scalable as less channels are needed for each queue manager. It is less flexible and definitions (channels and transmit queues) need to be set up manually.
THINK GLOBAL – ACT LOCAL

Queue Sharing Groups (z/OS)
What is a Queue Sharing Group?

- Allows high availability of WebSphere MQ resources
- Several queue managers share the same queues and messages
- Messages kept in the Coupling Facility
  - if >63KB then offloaded
- Queue managers that can access the same set of shared queues form a queue-sharing group (QSG)
- Applications can access shared queues from any queue manager in the QSG

- A Queue Sharing Group (QSG) is a co-operating set of queue managers running on z/OS which all have access to Shared Queues held in the Coupling Facility.

- In the Queue Sharing Group environment, an application can connect to any of the queue managers within the QSG. Because all the queue managers in the queue-sharing group can access the same set of shared queues, the application does not depend on the availability of a particular queue manager; any queue manager in the QSG can service any queue. This gives greater availability if a queue manager stops because all the other queue managers in the QSG can continue processing the queue.

- When messages are greater in size than 63K they are offloaded into DB2 or Shared Message Data Sets (SMDS). SMDS is a new feature in WebSphere MQ V7.1.
Advantages of using shared-queues

- Allows WebSphere MQ applications to be:
  - Scalable
  - Highly available
- Allows workload balancing to be implemented
  - Naturally performs pull workload balancing
    - based on processing capacity of each queue manager
- No outages for shared queue applications
  - Can stagger outages of each queue manager in the QSG
- Flexible capacity management
  - Can dynamically add or remove queue managers
- Peer Recovery
  - MQ detects if a queue manager abnormally disconnects from the Coupling Facility
  - Another queue manager in the QSG completes pending units of work (where possible)
  - Uncommitted gets (under sync-point) are backed out
    - Messages can be ‘re-got’ by another queue manager
  - Uncommitted puts (under sync-point) are committed
    - Message made available as soon as possible

Advantages of using shared-queues – Notes

To further enhance the availability of messages in a queue-sharing group, WebSphere MQ detects if another queue manager in the group disconnects from the Coupling Facility abnormally, and completes units of work for that queue manager that are still pending, where possible. This is known as peer recovery.
A history of Large Shared Queue Messages

- Initially only small non-persistent messages
- Added large messages and persistence
- Large messages initially BLOBs in DB2
- Now Shared Message Data Sets (SMDS)
  - With selective offload

## Notes

- In the first version of WebSphere MQ Shared Queues, only small (<63K) non-persistent messages were available. Next persistence was added and then big messages were added being stored initially in DB2 as BLOBs.
- More recently, in WebSphere MQ V7.1, large Shared Queue messages can now be stored in Shared Message Data Sets (SMDS) instead of in DB2, resulting in an improvement in performance.
- Additionally, the point at which offload of these messages is done, can be tuned to lower than 63K, allowed a balance between performance and CF capacity ($$$).
Shared message data set concepts

- Offloaded message data for shared messages is stored in data sets.
- Each application structure has an associated group of shared message data sets, with one data set per queue manager. Named using DSGROUP parameter on CFSTRUCT definition.
- Each queue manager owns a data set for each structure, opened for read/write access, which it uses to write new large messages.
- Each queue manager opens the data sets for the other queue managers for read-only access, so it can read their message data.
- When a message with offloaded data needs to be deleted, it is passed back to the queue manager which originally wrote it, so that the queue manager can free the data set space when it deletes the message.
- Messages too large for CF entry (> 63K bytes) are always offloaded.
- Other messages may be selectively offloaded using offload rules:
  - Each structure has three offload rules, specified on the CFSTRUCT definition.
  - Each rule specifies message size in Kbytes and structure usage threshold.
  - Data for new messages exceeding the specified size is offloaded (as for a large message) when structure usage exceeds the specified threshold.
  - Default rules are provided which should be sufficient for most cases and can be set to dummy values if not required.
Availability – two servers

This chart shows the availability impact of adding a second server.

The chart assumes that the two servers use shared queues -- in-flight transactions on a failing server are backed-out and can be processed by the other server (the client does not see the outage).

For many installations, unplanned outages are much less frequent than planned outages. WebSphere MQ Shared Queue Support allows one queue manager to take a planned outage without interrupting other queue managers in the QSG. This includes (for example) outages for software upgrades such as new versions and releases of the queue manager.

Chart is only applicable for "normal" unplanned outages (such as caused by software errors or operator errors). It is not applicable for disasters (such as meteorite impacts).

Adding more servers gives even better availability -- in particular the failure of one server has less relative impact on the overall capacity.
Availability – example

<table>
<thead>
<tr>
<th></th>
<th>One server</th>
<th>Two servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planned outages</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Unplanned outages</td>
<td>0.5</td>
<td>0.002</td>
</tr>
<tr>
<td>Total</td>
<td>1.5</td>
<td>0.002</td>
</tr>
</tbody>
</table>

Outages in hours per month

Availability – example – Notes

- This chart is a worked example of the availability impact of adding a second server.

- The example is highly simplified and does not take into account many factors that affect real systems, for example:
  - It only considers server outages – many other factors affect the availability of real systems
  - It assumes that failures are independent – in real systems a single event can cause more than one server to fail (common-mode failure)
  - It ignores the impact of reduced capacity.

- 0.002 hours is about 7 seconds – a reduction in total outage time of some three orders of magnitude.
### Comparison of Technologies

<table>
<thead>
<tr>
<th>Technology</th>
<th>Access to existing messages</th>
<th>Access for new messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Queues, HP NonStop Server</td>
<td>continuous</td>
<td>continuous</td>
</tr>
<tr>
<td>MQ Clusters</td>
<td>none</td>
<td>continuous</td>
</tr>
<tr>
<td>HA Clustering, Multi-instance</td>
<td>automatic</td>
<td>automatic</td>
</tr>
<tr>
<td>No special support</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

- This picture shows one view of the different capabilities. However you also need to consider factors such as total hardware/software price, the requirement for nonpersistent message availability (remember that they are discarded by a failover-restart), and the requirement for persistent message availability (not if you're using the shared queue support).
How to connect a client to a server

• Which channel should be used?
• How to specify the client’s connection to a queue manager:
  – Explicitly on the MQCONNX verb
  – MQSERVER variable
  – Client channel definition table
  – Pre-Connect exit

• Techniques for changing the CONNAME
  – Comma-separated list
  – Connection Sprayer
  – Sysplex Distributor (z/OS)

A channel is a logical communication link. Clients communicate with a server using channel called a client connection (CLNTCONN). On the server there must be a server connection (SVRCONN) channel available to connect to.

The client identifies which channel should be used when the application issues an MQCONN/MQCONNX call.

The choice about which channel to use is made by checking the following (in this order):
  – The ClientConnOffset or ClientConnPtr in the MQCNO structure (if an MQCONNX was issued).
  – The MQSERVER environment variable.
  – The client channel definition table. This can be found as described by the MQCHLLIB and MQCHLTAB environment variables or using the Active Directory on Windows.
  – The channel definition table found in the default path.
  – The pre-connect exit if defined

The CONNAME in the channel definition found by any of the above means, could also be affected by other availability techniques:
  – It could contain a comma-separated list of connection names (or IP addresses) to try
  – It could be an address used by a network sprayer, such as Sysplex Distributor, to balance connections
Client Channel Definition Table (CCDT)

- **A channel definition table is:**
  - A binary file (not editable by a user)
  - Created as AMQCLCHL.TAB when client channels are defined
  - Use CSQUTIL MAKECLNT function on z/OS

```
RUNMQSC
DEFINE CHANNEL(...) +
CHLTYPE(CLNTCONN) ....
```

```
<mq root>/qmqr/QMGRNAME/@ipcc (Win)
<mq root>/qmqr/QMGRNAME/@ipcc (Unix)
```

```
copy
c:\mqm\qmqr\qmgrname\@ipcc\AMQCLCHL.TAB
```

```
to
z:\mytable.tbl
```

Notes:
- Choose one of your MQ server machines to define all your CLNTCONN definitions. You cannot append channel definition tables together. If you want to define multiple client connection channels then you need to define all the channels on one of the servers. To make a client channel definition table on z/OS you use the CSQUTIL MAKECLNT function.
- Find the AMQCLCHL.TAB file and copy it to a location which is accessible by the client machines. Channel definitions can be shared by more than one client. In other words, the client definition table can be located on a file server. The name of the file can be changed if required but you must use the MQCHLTAB environment variable to MQ what you called it.
- Environment variables, MQCHLLIB and MQCHLTAB, can be used to enable the clients to locate the channel table if it is not in the default location or with the default name.
Using Channel Definition Tables: Locate QMgr

• How is the QMNAME client channel attribute used?
  – def chl(chl1) chltype(clntconn) conname(host1)  qmname(mars)
  – def chl(chl2) chltype(clntconn) conname(host2)  qmname(venus)
  – def chl(chl3) chltype(clntconn) conname(ethernet) qmname(venus)
  – def chl(chl4) chltype(clntconn) conname(dialup)  qmname(venus)

Connected via channel chl2 to "venus"
connected via channel chl3 to “venus”

Using Channel Definition Tables: Locate QMgr – Notes

• In this example the user has defined two client channels.
  • The client searches through the client channels in alphabetical channel name order. It looks for a channel definition with a QMNAME field which matches what the application specified on the MQCONN call. We therefore find channel ‘chl2’. If we did not find any channel definitions which match the application would receive a 2058 (Queue Manager name error) reason code.
  • The transmission protocol and associated connection are extracted from the channel definition and an attempt is made to start the channel to the machine identified (venus). In order for the connection to be successful clearly there must be started listener at the remote machine and the queue manager itself must be started.
  • If the connection can not be established then a reason code is returned to the application which gives more details about why it could not connect. There is also more information in the client error log explaining the reason for the failure.
Using Channel Definition Tables: Availability

**How do we choose from several Queue Managers?**

- `def chl(chl1) chltype(clntconn) conname(ip.mars) qmname(planet)`
- `def chl(chl2) chltype(clntconn) conname(ip.venus) qmname(planet)`
- `def chl(chl3) chltype(clntconn) conname(ip.pluto) qmname(planet)`

MQCONN ("*planet",hConn,cc,rc);

**Notes**

- In this example the client tries to connect to a queue manager first using "chl1" but the communication link is down. Secondly it tries "chl2" but the queue manager is not currently running. Finally the client tries to connect using channel "chl3". The communications link is running and the queue manager is running.

- However, if we had specified the name “planet” without the asterisk, we would discover that the name of the queue manager "pluto" did not match the one specified on the MQCONN call “planet” and so this connection fails. We use the asterisk to tell MQ that we, the application, don’t really care what the actual Queue Manager name is.

- The * specifies that the client does not care if the actual name of the Queue Manager does not match the name given.
Pre-Connect Exit: Dynamic Connections

- **Look up in a directory such as LDAP**
  - Sample exit in SupportPac MA98 does just that
- **Make choice based on any accessible criteria**
  - Is system running?
  - Workload?
  - Where is the application running now?
- **No “*” needed**

```c
MQCONN ("planet",hConn,cc,rc );
```

**NOTES**

The Pre-Connect exit was introduced in WebSphere MQ V7.0.1.4 and is an exit point that operates in a very similar manner to the CCDT. In fact it can replace your use of the CCDT.

A sample exit which looks up client channel definitions in a central LDAP server is provided in Support Pac MA98

This exit could of course be extended to make decisions based on all sorts of criteria, not just the definitions.
Problems of Client Channel Definition Table (CCDT):

- CCDT file has to be distributed to all client machines
- File-based access: all records read to client before looking for records required
- Regenerate and redistribute file if changes are made
- Applications must use '*' notation

Pros of Pre-Connect Exit:

- Clients can connect to central LDAP server to find information
- Only records required are returned to client machine
- Changes are made in central server and picked up on next client connect
- Applications just code name they are given, not aware whether it is queue manager name or not

Notes:

- The Pre-Connect exit was designed to address some of failings of CCDT.
- The CCDT is a file which must be distributed to all client machines for the client to use. This can be mitigated somewhat by placing the CCDT on a central file server that can be shared by multiple clients.
- The CCDT is a file-based system, so when looking for the record to use, first all records must be transferred to the client machine (if hosting on a shared file server).
- Applications must use an asterisk '*' to indicate that this is not the real queue manager name – why expose this to the application at all?
- If a change is needed to one of the definitions, or another one is added, then the CCDT must be regenerated and redistributed to all clients again.
- These various issues have been addressed when using the Pre-Connect exit sample that stores client channel definitions in a central LDAP server. However, you do have the more complex task of setting up the LDAP server to start with.
Comma-separated CONNAME

- The IP address of a Multi-Instance queue manager changes when it moves
  - MQ channel configuration needs list of addresses unless you use external IPAT or an intelligent router
  - Connection name syntax extended to a comma-separated list
    - In MQ 7.1, extended to LOCLADDR too

```define channel (to . qm2)
  chltype (sdm)
xmitq (qm2)
  conname ('168.0.0.1,168.0.0.2')
```

- Available on all platforms and all channel types
  - including SENDER on z/OS for example

---

Comma-separated CONNAME - Notes

- The comma-separated list of hostnames/IP addresses in the CONNAME field for channels was introduced in WebSphere MQ V7.0.1 along with Multi-Instance queue managers since they are not a full-blown HA Manager and so do not move the IP address when the queue manager moves. Instead the connecting channels and clients can be configured to provide both possible IP addresses for the Multi-Instance queue manager.

- It was provided on all platforms in V7.0.1, even those which were not providing the Multi-Instance queue manager support natively (for example z/OS) because they may well have channels connecting *TO* a MIQM. Support for this feature is both for client connections and for any other channel types that use a CONNAME to initiate a connection, for example a SENDER channel, however, using a list of targets in a SENDER channel must be done with care. The target should be a MIQM or a QSG shared channel.

- Of course, this feature may also be useful for other network connection routing even when not using Multi-Instance queue managers.
The MQ client can only commit a unit of work carried out on the queue manager it is connected to. The client cannot be used to manage work carried out on other resource managers. Therefore the MQBEGIN call is not available within MQ clients.

- Work between different resource managers can only be loosely coordinated, as shown by the slide, where different resource managers are accessed using their native interfaces and under different units of work.
- However, this is often sufficient. In the example shown the MQGET operation is not committed until a successful commit of the database update. This means that if the application crashes the request message is rolled back to the queue and can be reprocessed. If the application crashes after the database update but before the MQ transaction is committed then the transaction would merely be replayed.
The function provided by the Extended Transactional Client allows a client to participate in units of work coordinated by an XA transaction manager. Externally coordinated transactions can now work with queue managers located on different machines from where the transaction coordination takes place.

The Extended Transactional Client still does not support the MQBEGIN call - all units of work must be started using the XA interface by an external transaction manager. Potentially allows simpler administration by separating the machines with the databases on from the machines with the queue managers on.

Note: you will only find benefit in the Extended Transactional Client if you use another resource manager other than WebSphere MQ!

Wider access to the Extended Transactional Client for all customers:

- With the availability of WebSphere MQ V7.5, the Extended Transactional Client is available for use in all client deployments without additional licensing.
**XA Clients – The problem**

- Client App connects via front-end routing and starts UOW
- All the examples we have shown already could be this front-end routing
- If there is a failure, TM will reconnect via generic port to inquire what transactions need resolving
- If TM reconnects to QM2 it will only be told what is in-doubt on QM2, meaning that it will throw away any information about in-doubts on QM1

---

**XA Clients – The problem – Notes**

- Standard MQ clients that do not participate in global transactions are stateless. This means that you can use any of the previously mentioned techniques to route them to a queue manager and no matter at what point in the protocol the network connection is lost, it is safe to route them to a different queue manager the next time.

- XA Clients, on the other hand, do have state. This means that when there is a network failure, if there is the possibility to be routed to a different queue manager then the transactionalilty of the global transaction may be compromised. For this reason it is not supported to run XA Clients with any kind of front-end routing except in two very clear cases:-
  - Multi-Instance queue manager
  - Queue Sharing Group with GROUPUR
You may use an XA Client with front-end routing if the only possible alternative network routes are still pointing to the same queue manager, for example because it is a Multi-Instance queue manager. The comma-separated CONNAME was created for exactly this purpose, but other techniques can be used so long as you are sure that they will only target the addresses that the Multi-Instance queue manager resides on.
When using the Extended Transactional Client, or the JMS transactional client (under WAS), it is possible to use 2-phase commit applications in a QSG. When specifying the connection options to the Transaction Manager (TM) it is necessary to provide the QSG name rather than the QMGR name, and also configure the client channel to be routed to a suitable (V7.0.1 or higher queue manager) in the QSG. When using this configuration, any Unit of Recovery (UR) that is created will have a GROUP disposition. This means that it can be inquired and resolved on any queue manager in the QSG.

If a connection fails for some reason, and the TM reconnects to the QSG, it can inquire and resolve the transactions no matter which qmgr it is now connected to, and where the transactions were originally started.
Automatic Client Reconnection

- Client library provides reconnection logic on detection of a failure
- Tries to hide queue manager failures by restoring state automatically
  - Can reconnect to the same or or another, equivalent queue manager
  - Re-opens queues, re-establishes subscriptions
  - All MQI handles keep their original values
- MQI or JMS calls block until connection is remade
  - By default, will wait for up to 30 minutes
  - Long enough for a queue manager failover (even a really slow one)

Automatic Client Reconnection: What causes reconnection?

- Only explicit ends or failures
  - Communications failure
  - Queue Manager or Listener failure
  - STOP CONN
    - endmqm –s or endmqm –r
- The following will not cause reconnect
  - STOP CHANNEL
  - Any other endmqm
Automatic Reconnection: MQI Considerations

• Some MQI options will fail if you have reconnection enabled
  – Using MQPMO/MQGMO_LOGICAL_ORDER, MQGET gives MQRC_RECONNECT_INCOMPATIBLE

• New MQCONNX options
  – MQCNO_RECONNECT
  – MQCNO_RECONNECT_Q_MGR
  – MQCNO_RECONNECT_DISABLED

• MQPUT of PERSISTENT message outside of syncpoint
  – May return MQRC_CALL_INTERRUPTED

• Event handler notified of reconnection ‘events’

• MQSTAT may return only a ‘subset’ of information
  – Reconnection may have occurred during a message sequence
  – Can be used to query reconnection status

Automatic Client Reconnection

• In an ideal world all applications would have sensible, appropriate retry logic to cope with all connection failures and Automatic Client Reconnection would not be needed. One could still argue that reconnection is better done by the application since the application itself knows what state is important. However, for a large number of applications the Automatic Client Reconnection provides a ‘good enough’ reconnection service and requires virtually no effort to use.

• The application must merely state at connect time that it’s willing for the connection to be reconnected automatically and whether, if that happens, the connection must go back to the same Queue Manager. Clearly it is better if the application can avoid this type of affinity but for some conversational applications, which contain a lot of ongoing state, getting back to the same Queue Manager may be paramount.

• When an application is in reconnect mode it can very easily look like an application hang since the MQI call be issued at the time of the connection failure just doesn’t return. If the application is a user facing application it may be useful to show to the user that the application is attempting to reconnect. This is one of the reasons the application may wish to register an event handler so that the application gets control when reconnects occur. The application can also affect the reconnect process by adjusting the reconnect timeouts or cancelling the reconnect altogether.
When using a client channel definition table (CCDT) or Pre-Connect exit to configure the client connectivity used by your client applications, you can provide a number of destination queue managers to choose from in order to provide redundancy and alternate destinations when one fails.

You can define these destinations with a weighting so that the spread of client connections between the group of queue managers is as you require.

You can then use the same set of definitions with all your clients – no need to produce different copies to spread out your client connections across all the back-end servers.

The default value of CLNTWGHT is 0 – which retains the V6 behaviour of primary then secondary choices chosen by alphabetical order.

By default client channels have AFFINITY(PREFERRED) set. This means that any particular client application will attempt to connect to the same queue manager each time. This is the same behaviour that was available in V6 with the mechanism that the primary connection was attempted first, then if it was not available, the secondary connection was attempted, and so on. If it is desired that connections from the same machine are to be workload balanced as above, AFFINITY(NONE) can be chosen.
THINK GLOBAL – ACT LOCAL

MQTT Clients
How do we attach Mobile devices?  
The Smarter Planet Protocol

• WebSphere MQ Telemetry Transport (MQTT)  
  – Messaging optimized for mobile, smart sensors and telemetry devices  
  – Enables intelligent decision-making based on remote real-world events  
  – Remote resource management of static or moving assets, people, locations

• An open standard with Industry leadership & mindshare  
  – MQTT Protocol and client code contributed to open source  
    • see MQTT.org and Eclipse Paho  
    – allows development communities to provide further client code& device support

MQTT protocol – Notes

- The MQTT protocol was originally designed for the constraints of Supervisory Control and Data Acquisition (SCADA). An example of such a system is a pipeline, running thousands of kilometres through inhospitable regions, which is monitored and controlled by thousands of devices along the pipeline. These devices report state, such as temperature and flow rate, and can be directed to open and close valves.

- IBM and a partner Arcom (now Eurotech) were asked to provide a solution for a customer that was struggling to scale. Their incumbent topology incorporated a central system that polled each of the devices in turn over a satellite network, which became constrained because the network became saturated and the central system was running at 100% CPU. Polling is inherently inefficient and wastes a lot of bandwidth, especially if the state has not changed. This was compounded because the satellite communication link is low bandwidth and costly to utilise.

- The problems were solved by changing the system to use an asynchronous messaging approach. WebSphere MQ servers were located at the data centre, which acted as a hub sitting between the devices on the pipeline and the backend processing applications. MQ Telemetry Transport (MQTT) was chosen to provide the communications between the pipeline and the MQ servers because the footprint of the traditional enterprise MQ client was too big to run on the processing units on the pipeline. Also, and more importantly, the traditional MQ protocol was far too heavy for the network that was charged by the volume of data sent.

- MQTT is a bidirectional/duplex messaging protocol that enables messages and events to be pushed from the client to the server and the server to the client. Pushing messages is inherently more efficient than polling. Instead of the server continuously polling the device to find out when a state has changed, a state change event is generated and pushed to the server only when the state change occurs. This change alone dramatically reduced the network usage enabling thousands of additional devices to be added to the system.

- MQTT is a publish/subscribe protocol. This allowed the customer to integrate these messages with other systems, such as for billing and maintenance.
MQTT: Key Features

- **Open**
  - Open published spec designed for the world of “devices”
  - MQTT client code (C and Java) donated to the Eclipse "Paho" M2M project

- **Reliable**
  - Three qualities of service:
    - 0 – at most once delivery
    - 1 – assured delivery but may be duplicated
    - 2 – once and once only delivery
  - In-built constructs to support loss of contact between client and server.
    - “Last will and testament” to publish a message if the client goes offline.
  - Stateful “roll-forward” semantics and “durable” subscriptions.

- **Lean**
  - Minimized on-the-wire format
    - Smallest possible packet size is 2 bytes
    - No application message headers
  - Reduced complexity/footprint
    - Clients: C=50Kb; Java=100Kb

- **Simple**
  - Simple / minimal pub/sub messaging semantics
    - Asynchronous ("push") delivery
    - Simple set of verbs -- connect, publish, subscribe and disconnect.

Included in IBM WebSphere MQ v7.1 and 7.5

---

MQTT: Key Features – Notes

- The MQTT specification is open and royalty free.
- The MQTT specification is being standardized at OASIS.
- Third party and open source servers and clients now number:
  - 16+ MQTT servers
  - 40+ MQTT clients
- MQTT provides three qualities of service (QOS):
  - Once and once only delivery
  - At least once (may be duplicated)
  - At most once
- As the QOS increases so does the overhead and number of bytes ‘on the wire’
- Provides solution designers with a flexible choice - many messages can be delivered at a lower quality of service, e.g. regular status updates
- Mobile, web and other machine-to-machine (M2M) applications have similar requirements on footprint and bandwidth. Mobile applications can be web browser based, native applications, or a hybrid of the two.
WMQ Telemetry V7.1

- **Highly scalable**
  - Includes new Java Listener (MQXR) which can handle up to 100K concurrently connected devices
  - Fully integrated / interoperable with WMQ
    - Publishers and subscribers can exchange messages with MQI and JMS applications
- **Ships with two types of client:**
  - Basic
    - direct connectivity from a device
  - Advanced:
    - Acts as a “concentrator/hub” for mini-networks of Telemetry devices
- **Provides rich security**
  - Network: SSL
  - Authentication: JAAS
  - Authorisation: OAM
- **In addition any third party, opensource or roll your own MQTT client can be used**

WMQ Telemetry V7.1 – Basic Client support

- Basic client must be connected to an MQ server to send and receive messages
- Small footprint
- Ships
- Direct connectivity from device
- Java and C versions
  - Very small footprints
    - 30KB C
    - 100KB Java
- Can be redistributed with application / device
- Other basic clients available:
  - Reference
  - Third party
  - Open source
  - Roll your own
- Eclipse PAHO Client support for -
  - Android (HTC, Samsung, others)
    - Java Client
  - iOS (iPhone)
    - C Client needs some modifications to run on iOS
**WebSphere MQ Telemetry V7.1 – Advanced Client Support**

- Acts as a “concentrator/hub” for mini-networks of Telemetry devices
- Can connect to one or more backend servers
- Can buffer messages

- **MQ Telemetry Daemon for Devices ships with MQ Telemetry**
  - Aimed at embedded / constrained systems
  - Very Small footprint
    - 100KB

- **Micro broker ships with Lotus Expeditor**
  - 100% Java
  - Three versions:
    - For devices (PDA…)
    - For desktop
    - For small servers
  - Others
    - Open source

**IBM MessageSight**

- A secure, easy to deploy appliance-based messaging server that is optimized to address the massive scale requirements of the machine to machine (m2m) and mobile use cases.
- A million connections, and millions of messages per second
- It is designed to sit at the edge of the enterprise and can extend your existing messaging infrastructure or be used standalone
- Extends and complements the existing IBM Connectivity and Integration portfolio
- Demo: Trouble in San Diego - [http://www.youtube.com/watch?v=vtQYskN6jMg](http://www.youtube.com/watch?v=vtQYskN6jMg)