Achieving scaling and high availability with IBM MQ for z/OS queue sharing groups

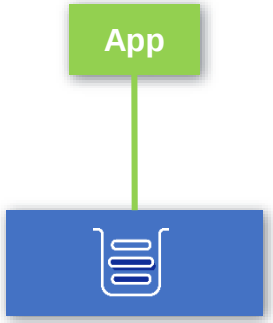
Gwydion Tudur
IBM MQ Development

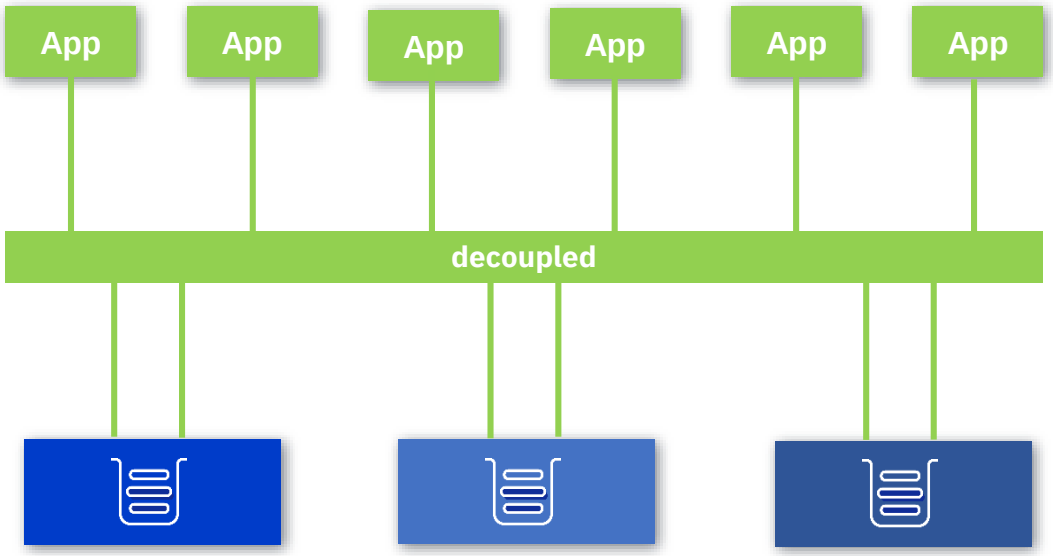
November 2019
Session JN




Agenda

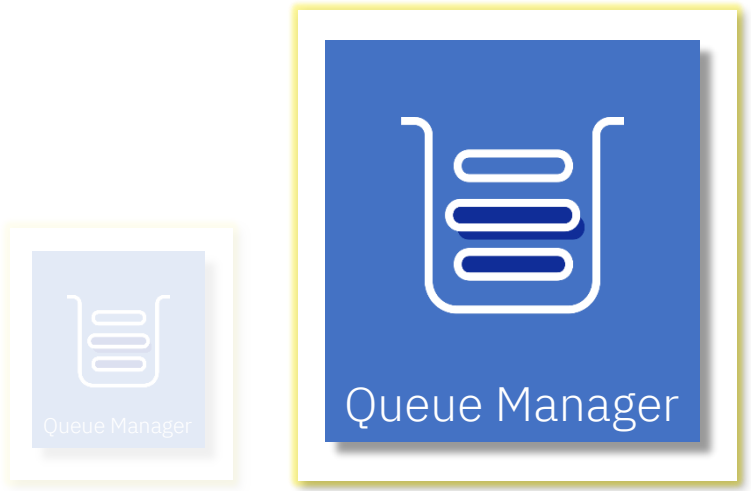
- Why you need scalability and resilience
- Queue sharing groups
- Achieving consistent configuration
- Connectivity
- Application considerations
- Resilience
- Putting it all together
- A comparison with uniform clusters





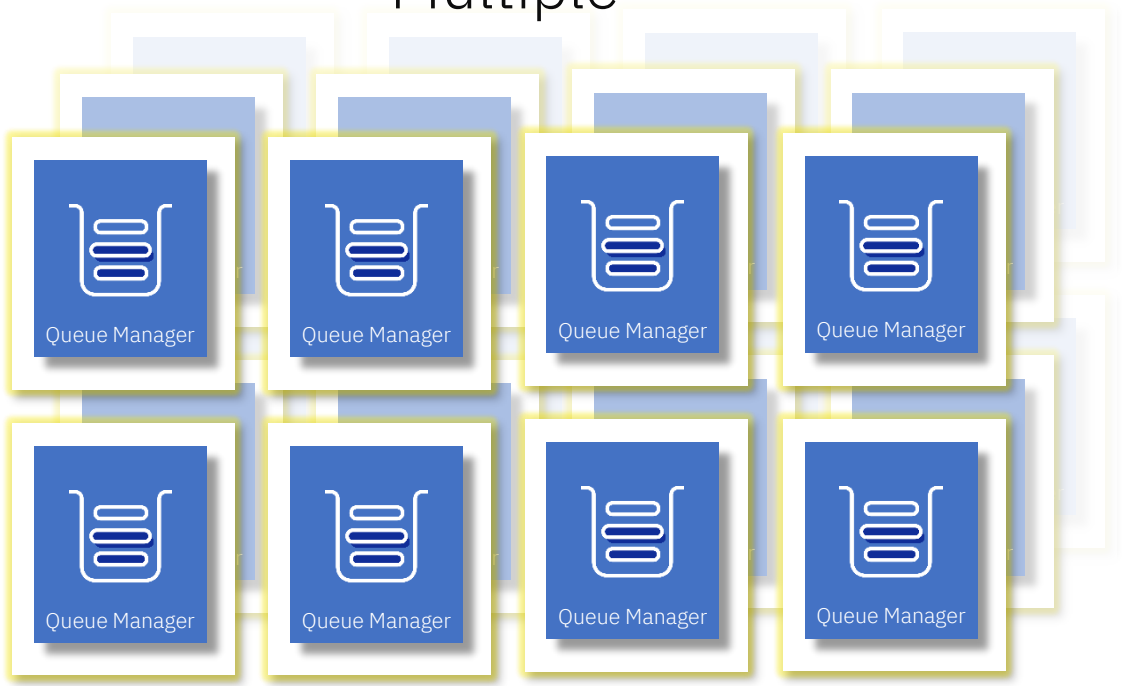
Scaling

Single



x4

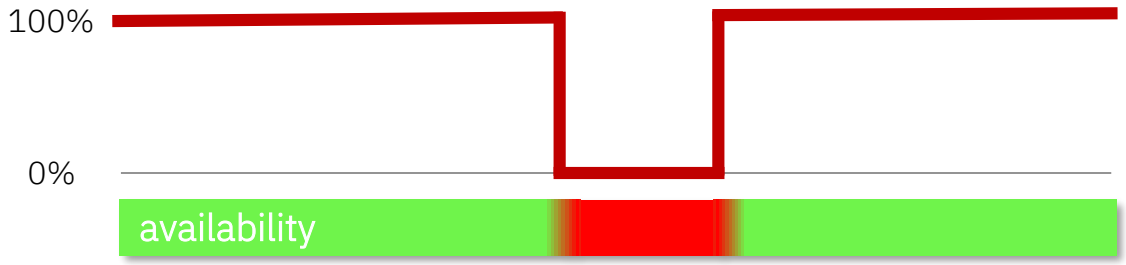
Multiple



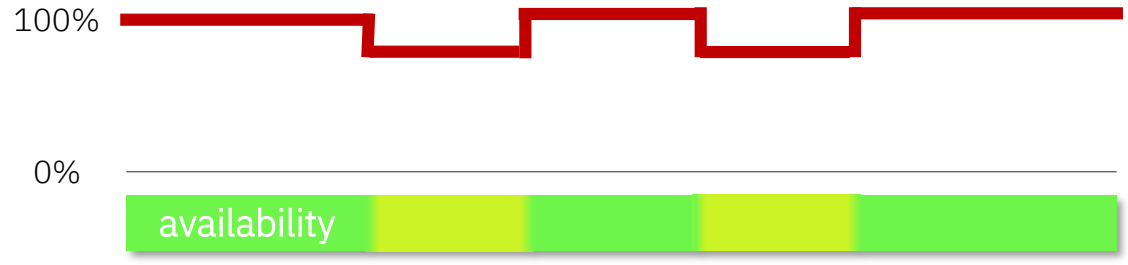
xn

Fault Toleration

Single



Multiple

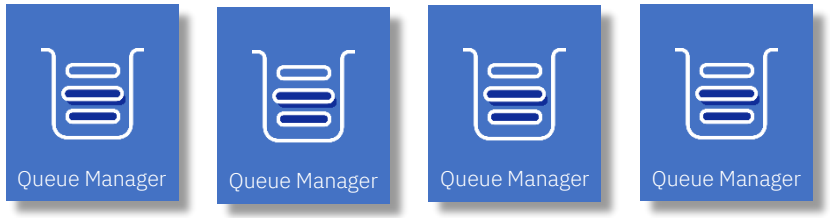


Single vs. Multiple

Single

- Simple
- Invisible to applications
- Limited by maximum system size
- Liable to hit internal limits
- Not all aspects scale linearly
- Restart times can grow
- Every outage is high impact

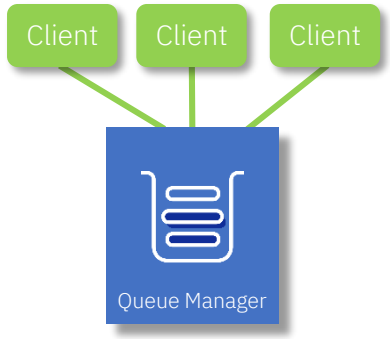
Multiple



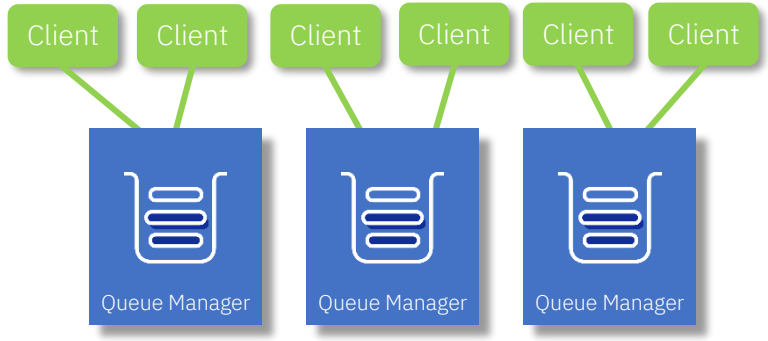
- Unlimited by system size
- All aspects scale linearly
- More suited to cloud scaling
- Reduced restart times
- Enables rolling upgrades
- Tolerate partial failures
- Potentially more complicated
- Needs to, and should be, planned for

It’s not just the queue managers…

Step 1
Horizontally scale the application into multiple instances, all performing the same role
A queue manager works better when there are multiple applications working in parallel



Step 2
Horizontally scale the queue managers
Create multiple queue managers with the ‘same’ configuration
Distribute the application instances across the queue managers



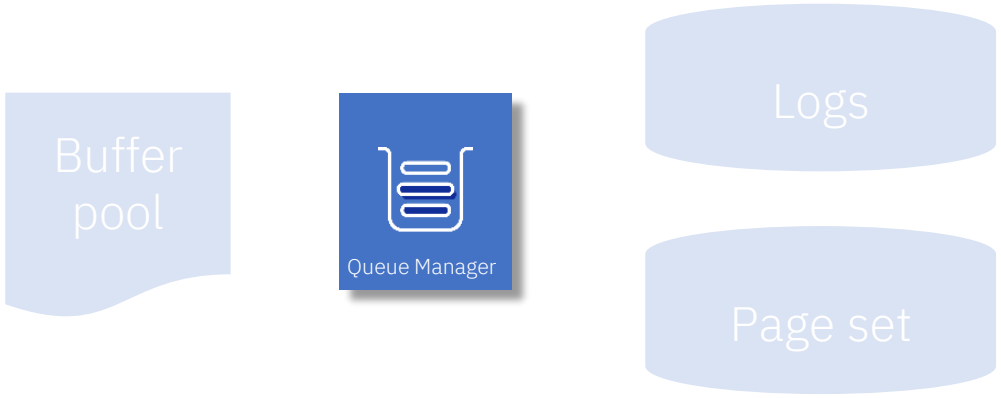
Queue sharing groups

Private queues – a recap

Private queues are defined to individual queue managers

On z/OS messages are held in memory in buffer pools and where necessary stored on disk in a page set

Persistent messages are logged

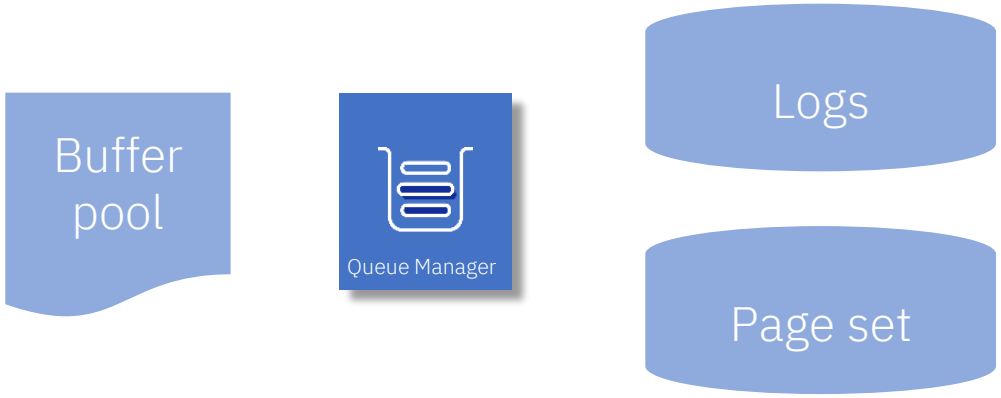


Private queues – a recap

Messages can only be accessed by the owning queue manager

If the queue manager fails messages are lost if non-persistent, or unavailable until the queue manager restarts if persistent

On distributed MQ all queues are implicitly private…



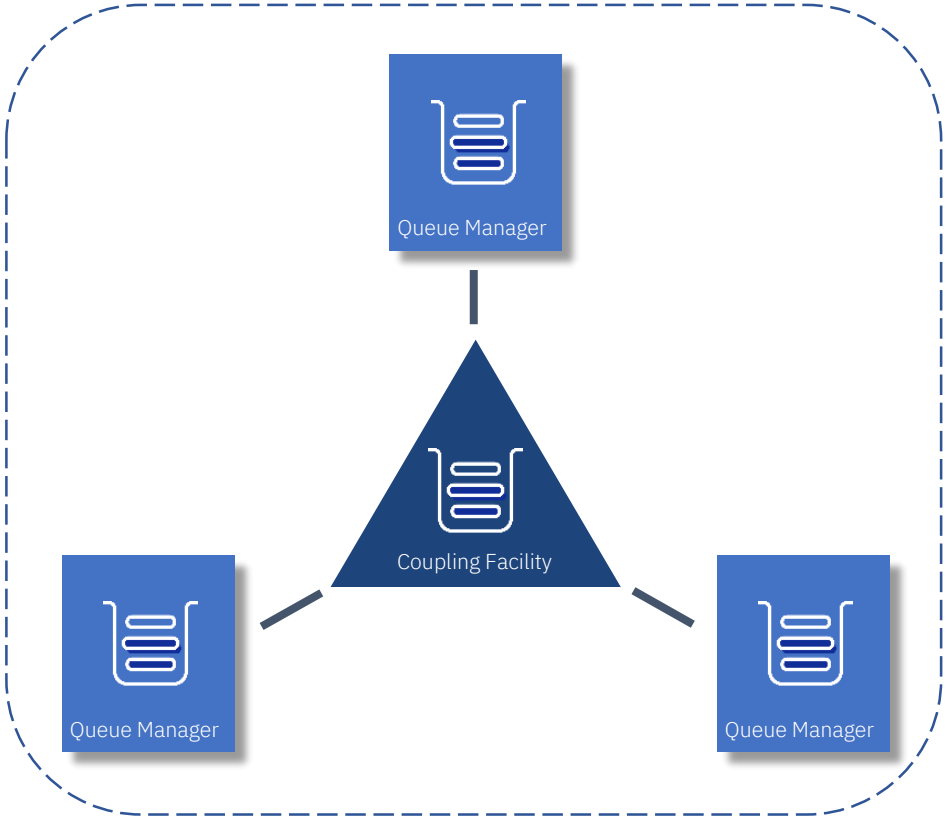
Shared queues

Shared queues are defined to a set of related queue managers – a queue sharing group

Messages are stored in a coupling facility, in a memory structure called a list structure

As before, persistent messages are logged

The messages can be accessed by any queue manager in the group

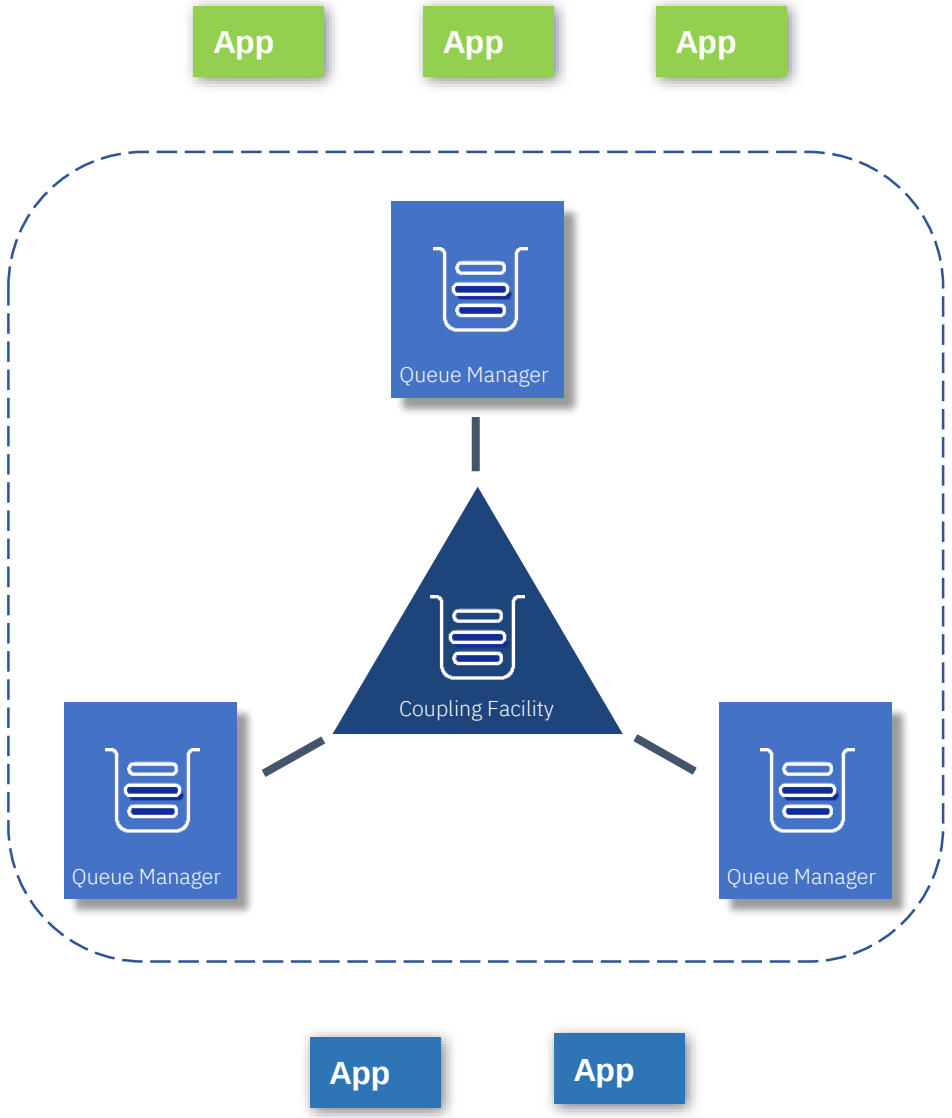


Shared queues

If an individual queue manager in the group isn't running, messages in the shared queue are still available to apps via the remaining queue managers in the group

Apps connect to the group rather than an individual queue manager

Queue managers in a queue sharing group can also have private queues

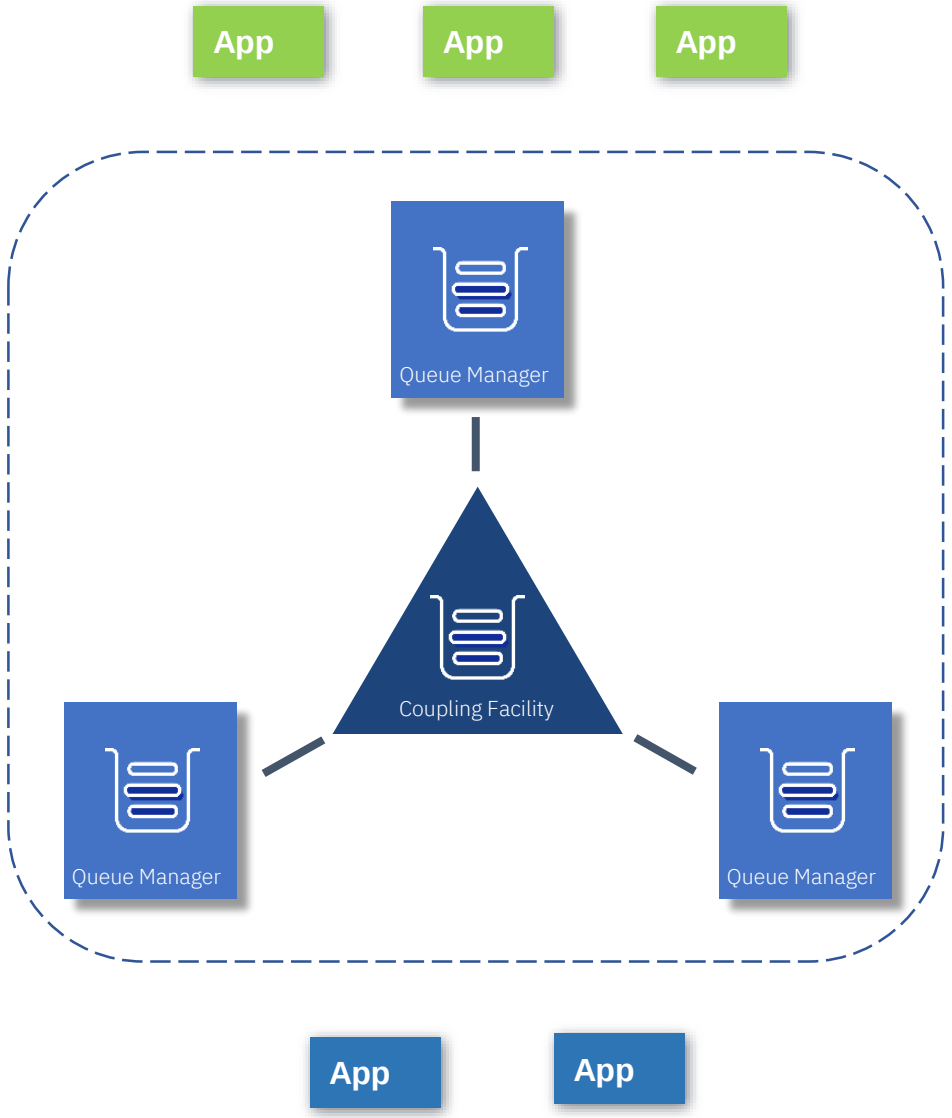


Shared queues - benefits

Resilient: existing messages are available to apps, and new messages can be put, for as long as one queue manager is available

Scalable: throughput can push past the capacity of a single queue manager

Simple: no need to balance messages, and applications, across a set of private queues, or worry about marooned messages



Achieving consistent configuration

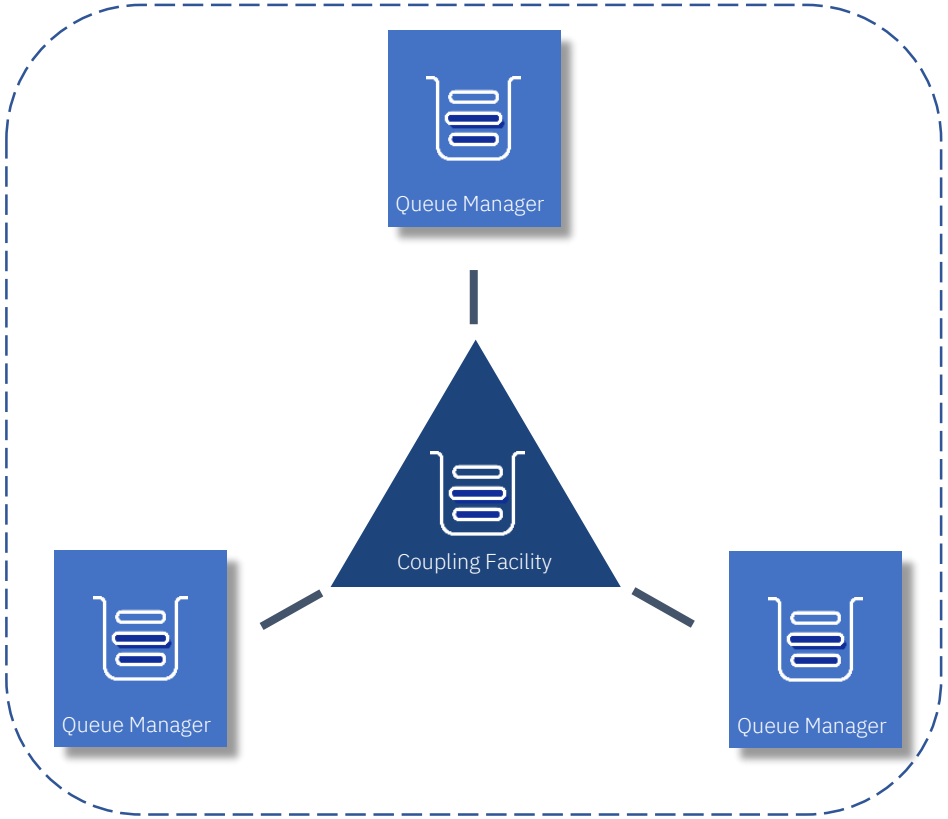
Consistent configuration

A queue sharing group environment **should** consist of a set of identically configured queue managers

Consistent configuration

A queue sharing group environment **should** consist of a set of identically configured queue managers

Luckily this is easy to achieve with queue sharing groups



Consistent configuration - steps

When defining your logs, page sets, BSDS, etc. use the same set of JCL for each queue manager, ensuring that data set names contain the name of both the queue sharing group and the queue manager

The only thing you should be changing in the JCL when moving to another queue manager is the queue manager name

```
DEFINE CLUSTER
       (NAME(MQM.QSG1.QM1.PSID00)
        RECORDS(1000 500)
        LINEAR SHAREOPTIONS(2 3) )
     DATA
       (NAME(MQM.QSG1.QM1.PSID00.DATA) )
```

Consistent configuration - steps

Keep the JCL in version control, it makes it easier to find, back up and track changes

Having the queue sharing group name in the data set name makes it easy to apply consistent policies for zHyperWrite, compression, and data set encryption across the whole queue sharing group

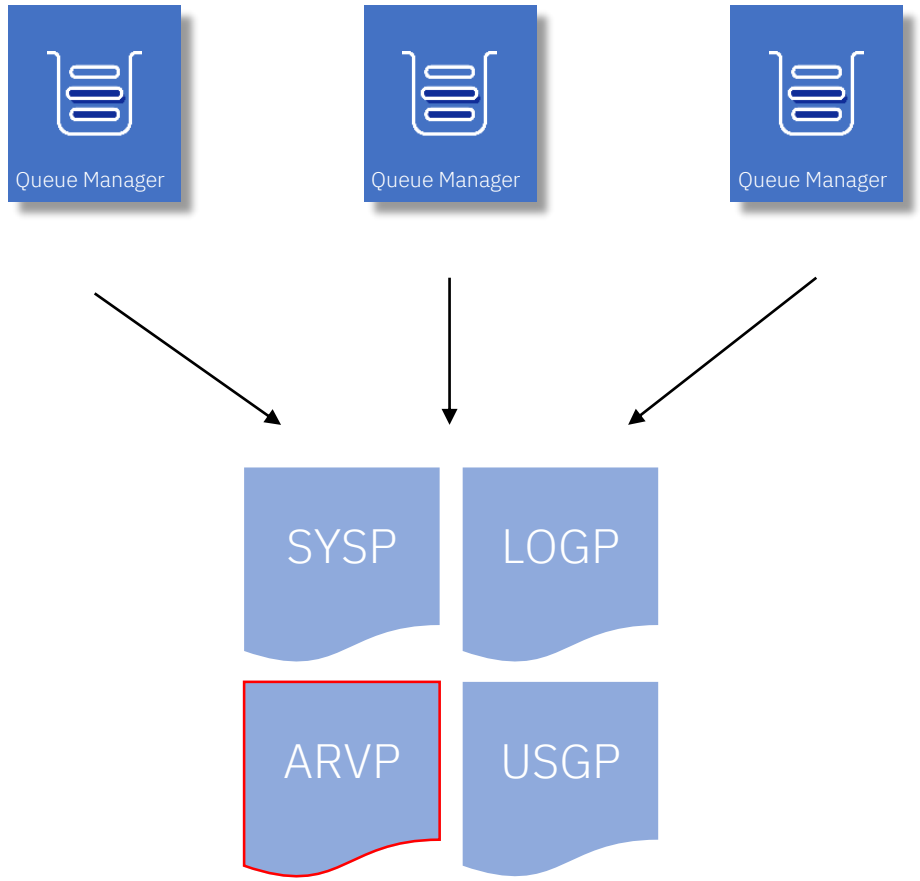
```
DEFINE CLUSTER
       (NAME(MQM.QSG1.QM1.PSID00)
        RECORDS(1000 500)
        LINEAR SHAREOPTIONS(2 3) )
     DATA
       (NAME(MQM.QSG1.QM1.PSID00.DATA) )
```

Consistent configuration - steps

Use a common source for the various CSQ6* macros when building the system parameters for all your queue managers

Keep the source in version control

Watch ARCPFX1 and ARCPFX2, in CSQ6ARVP, which need to be different for each queue manager



Consistent configuration - steps

Use a common source for the CSQINP* data sets you use to configure your queue managers

Keep the source in version control

Use shared queues instead of private queues

DEF QL(APP1.Q) QSGDISP(SHARED) CFSTRUCT(APPSTRUC1)

Consistent configuration - steps

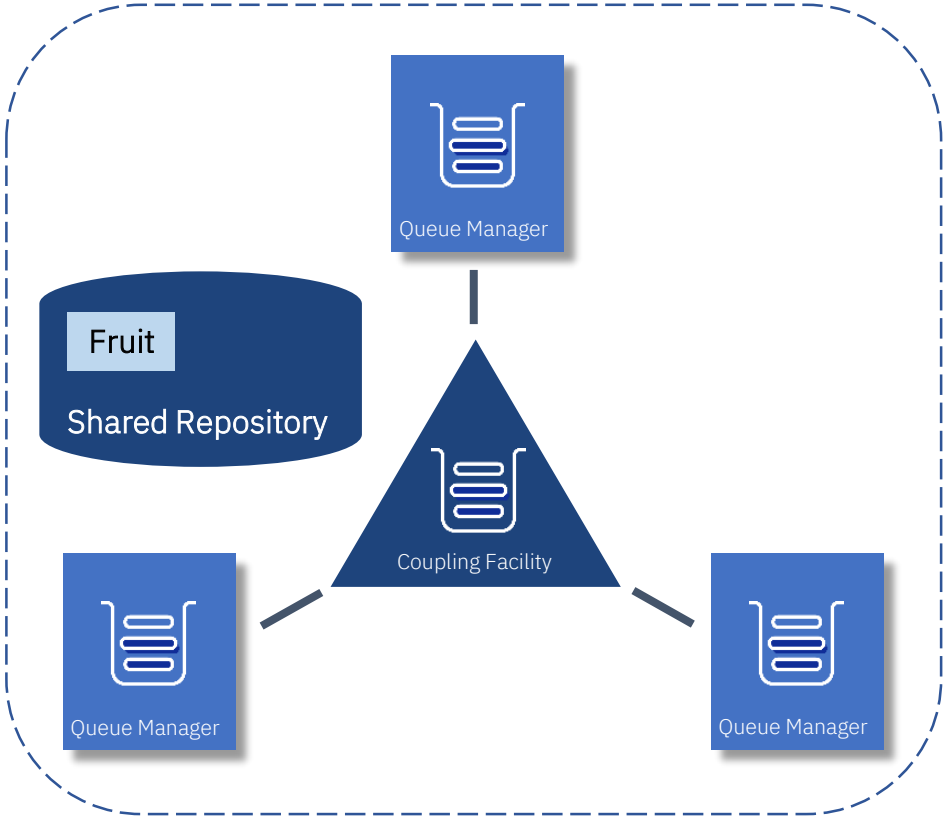
Define other object types to the queue sharing group rather than to individual queue managers

This is done using the QSGDISP(GROUP) parameter

The definition of the object is then held in a shared repository

Each queue manager caches a local definition

DEF TOPIC(FRUIT) QSGDISP(GROUP) TOPICSTR('FRUIT')



Consistent configuration - steps

Define other object types to the queue sharing group rather than to individual queue managers

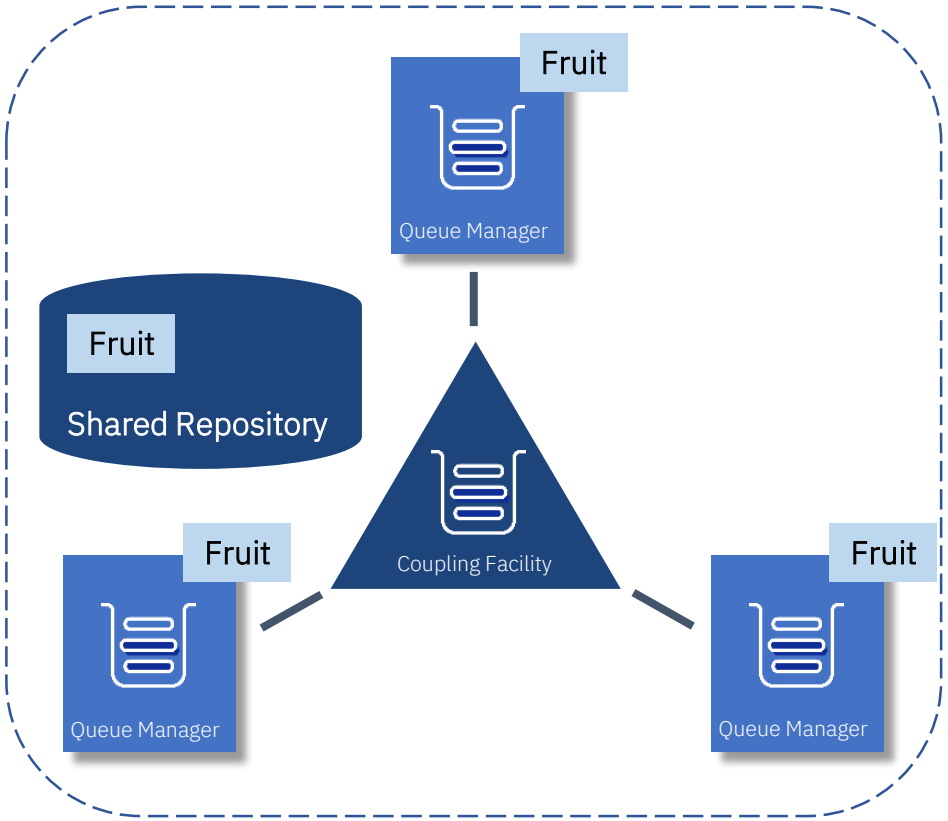
This is done using the QSGDISP(GROUP) parameter

The definition of the object is then held in a shared repository

Each queue manager caches a local definition

NB: Can't do this with DEFINE SUB so use CMDSCOPE(*) instead

DEF TOPIC(FRUIT) QSGDISP(GROUP) TOPICSTR('FRUIT')



Consistent configuration - steps

When configuring security profiles use the queue sharing group name in RACF profiles instead of the name of the individual queue manager

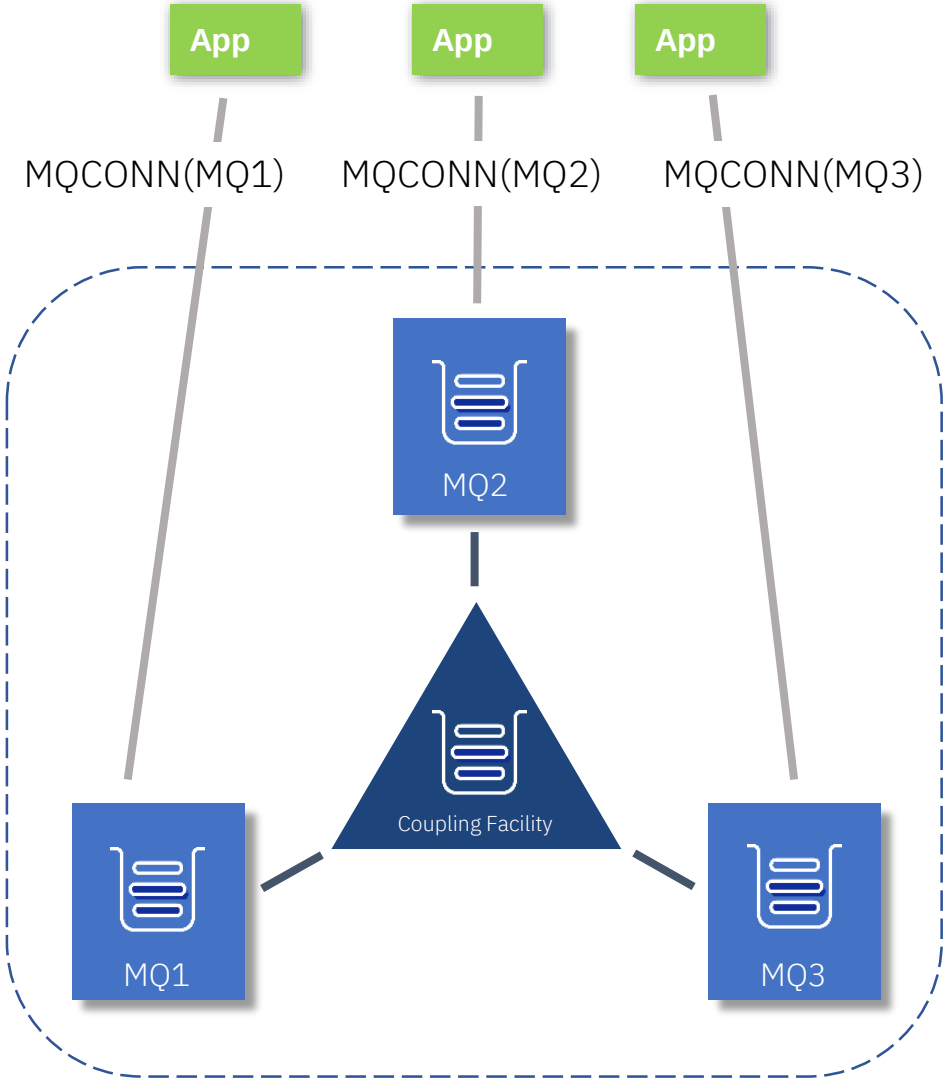
RDEFINE MQQUEUE **QSG1**.APP1.Q

Connectivity

Connectivity

Apps should never specify a specific queue manager name when connecting to MQ

Doing so requires a recompile if the queue manager changes, making your environment brittle, and making it hard to adopt technologies such as queue sharing groups

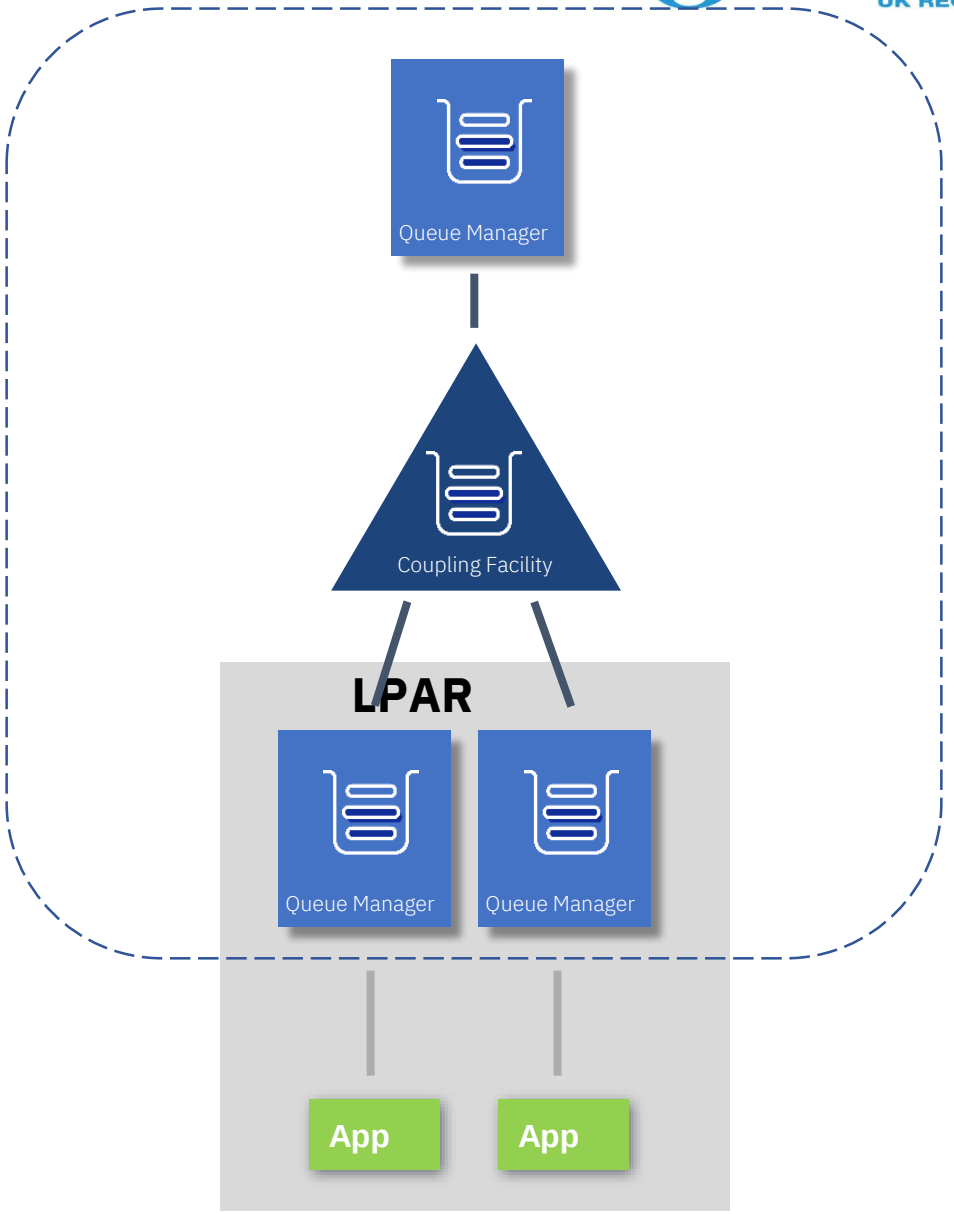


Local apps

Apps should connect to the queue sharing group rather than to a specific queue manager

For apps which make cross-memory connections you should ensure that there are at least two queue managers in the queue sharing group available on each LPAR, providing availability should one of the queue managers fail

This capability is available for apps running in batch, CICS and JEE environments

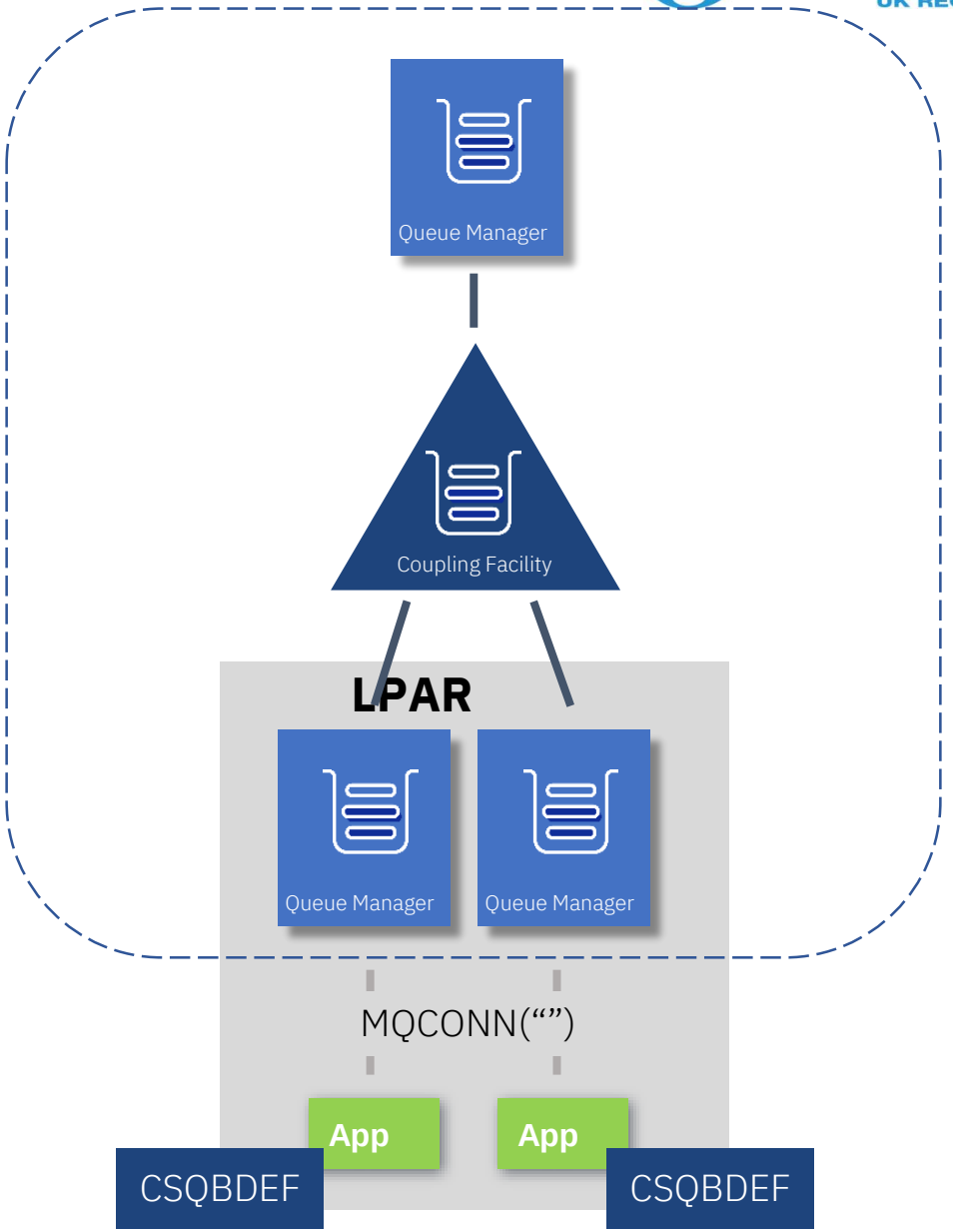


Local apps
Default queue manager

The following options are available to prevent hardcoding of queue manager names for local apps

Apps running in batch environments can connect to the default queue manager. The default queue manager can be specified for each app using the CSQBDEF macro

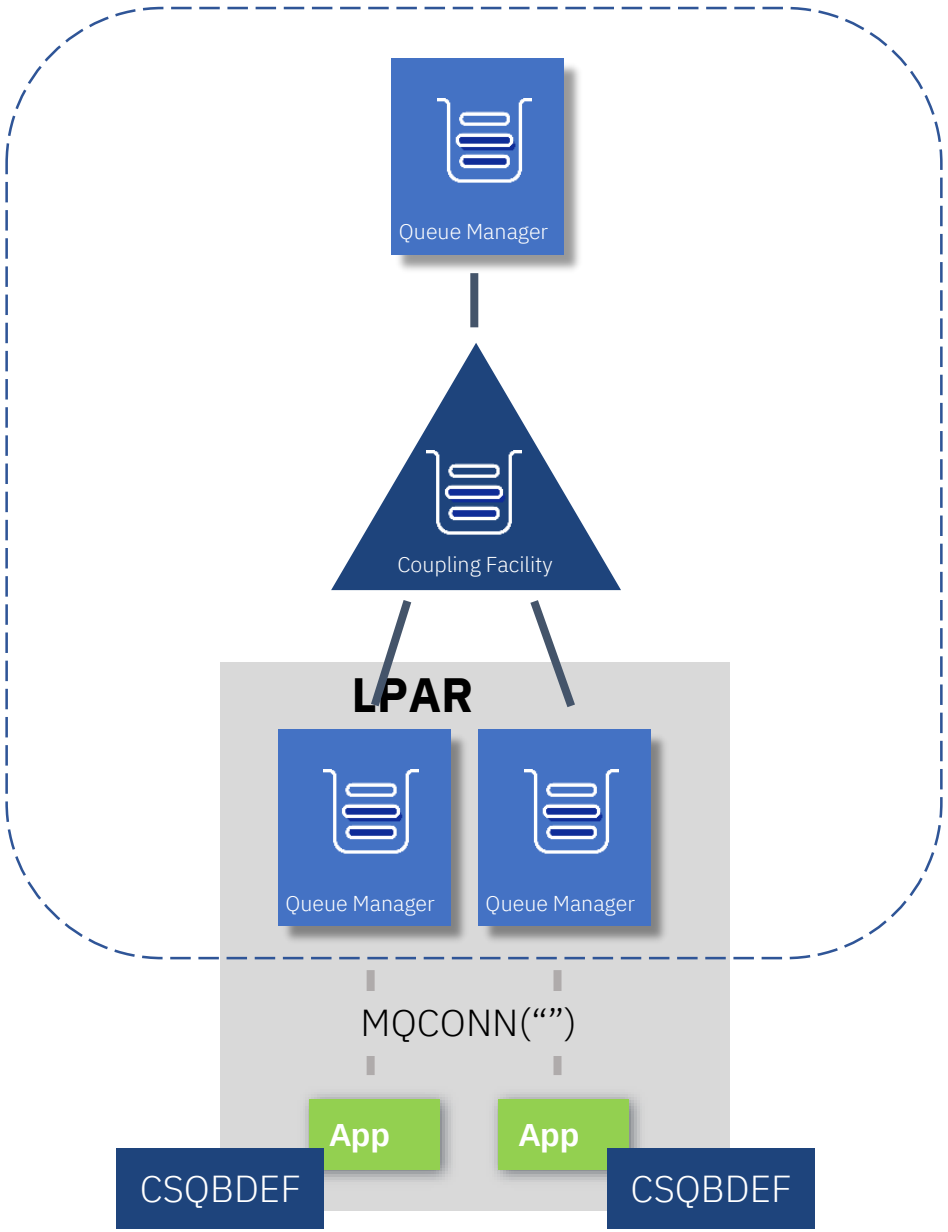
JMS apps can hold connection factory definitions in JNDI allowing the definition to be changed without changing the app



Local apps
Default queue manager

In CICS the MQCONN resource holds the name of the queue manager or queue sharing group to connect to so apps don't need to specify it anyway

In IMS the CSQQDEFV module allows a default queue manager to be specified, but this can't be a queue sharing group



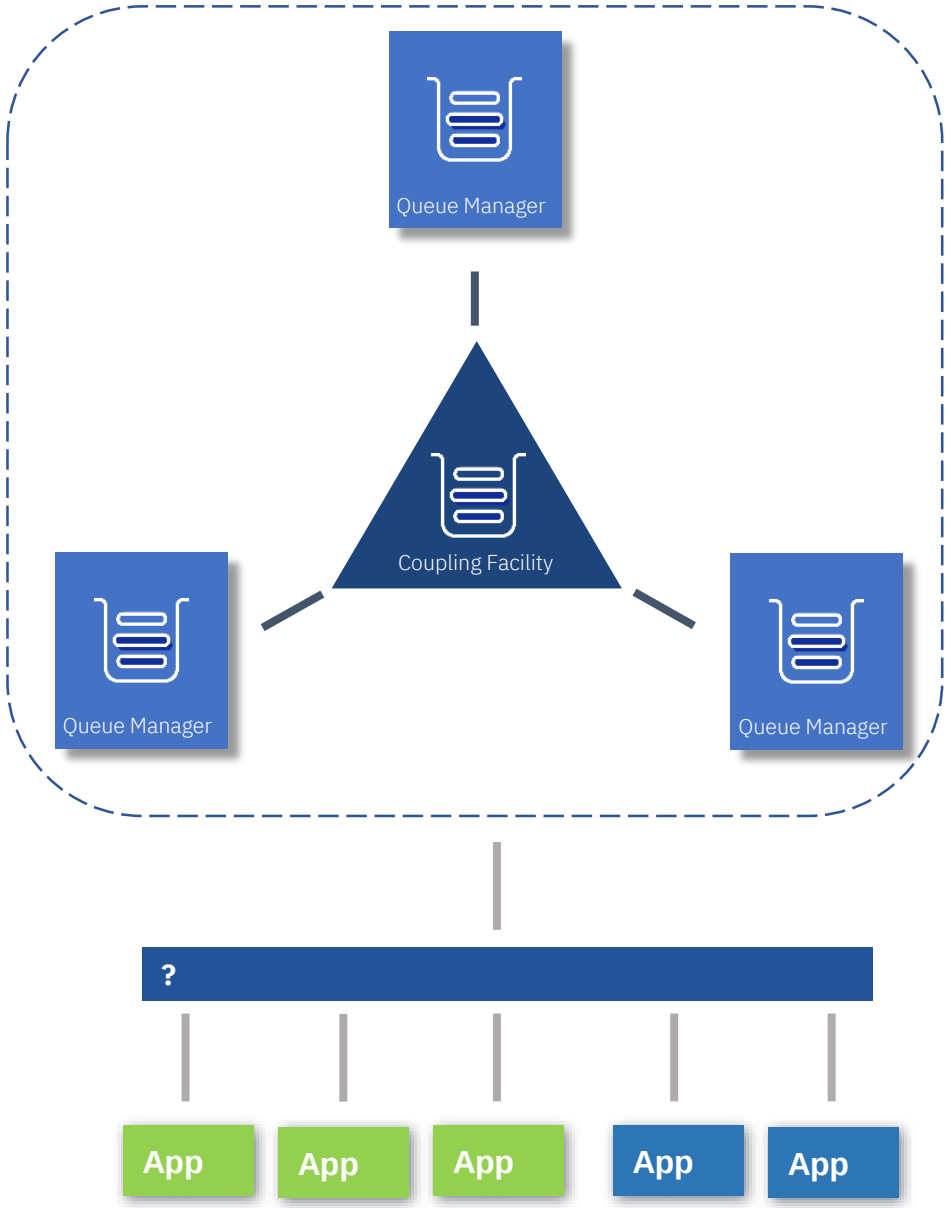
Remote apps

Apps should connect to the queue sharing group rather than a specific queue manager

For apps which connect to MQ over a network something is needed to spread connections across the available queue managers

Several approaches are available:

- Sysplex Distributor
- A workload balancer / IP sprayer
- CCDTs

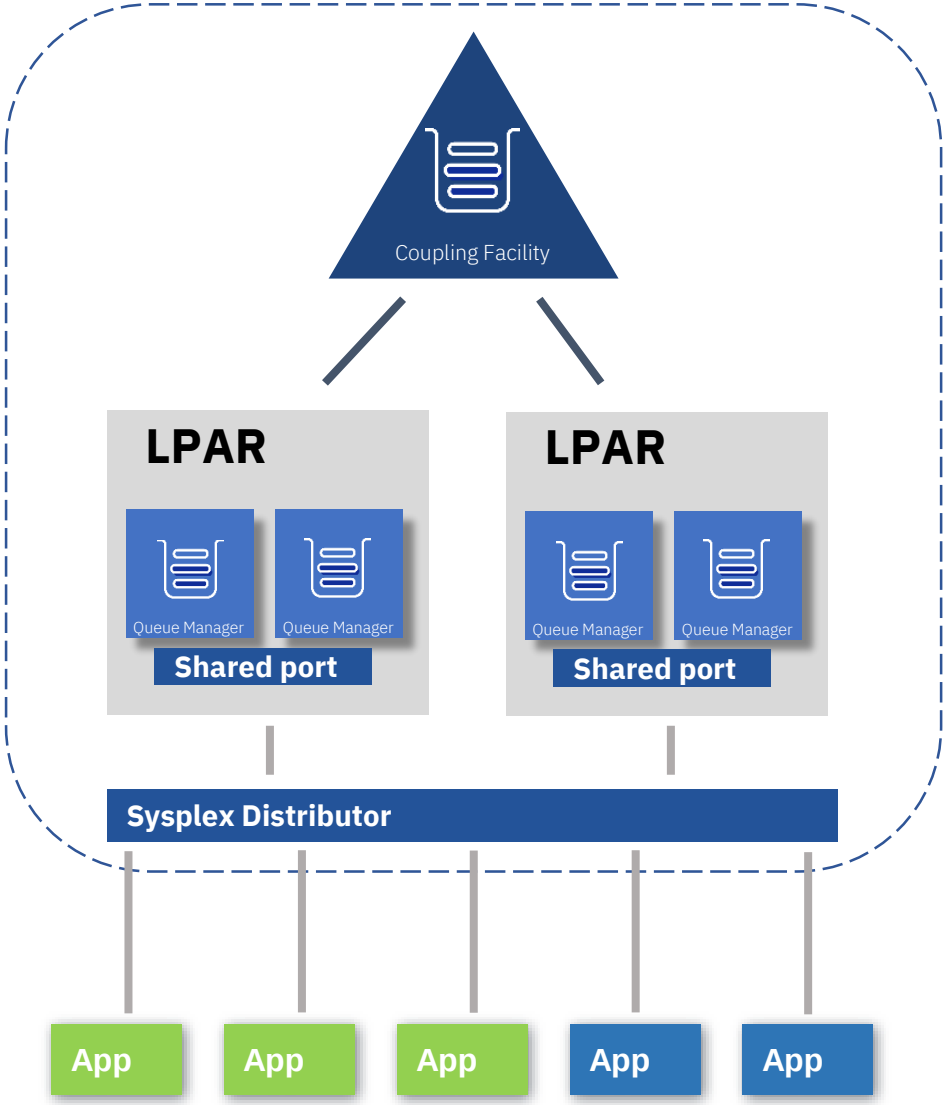


Sysplex Distributor

Sysplex Distributor allows each queue manager in the queue sharing group to listen on the same IP address and port

If multiple queue managers exist on the same LPAR then they can use a shared port

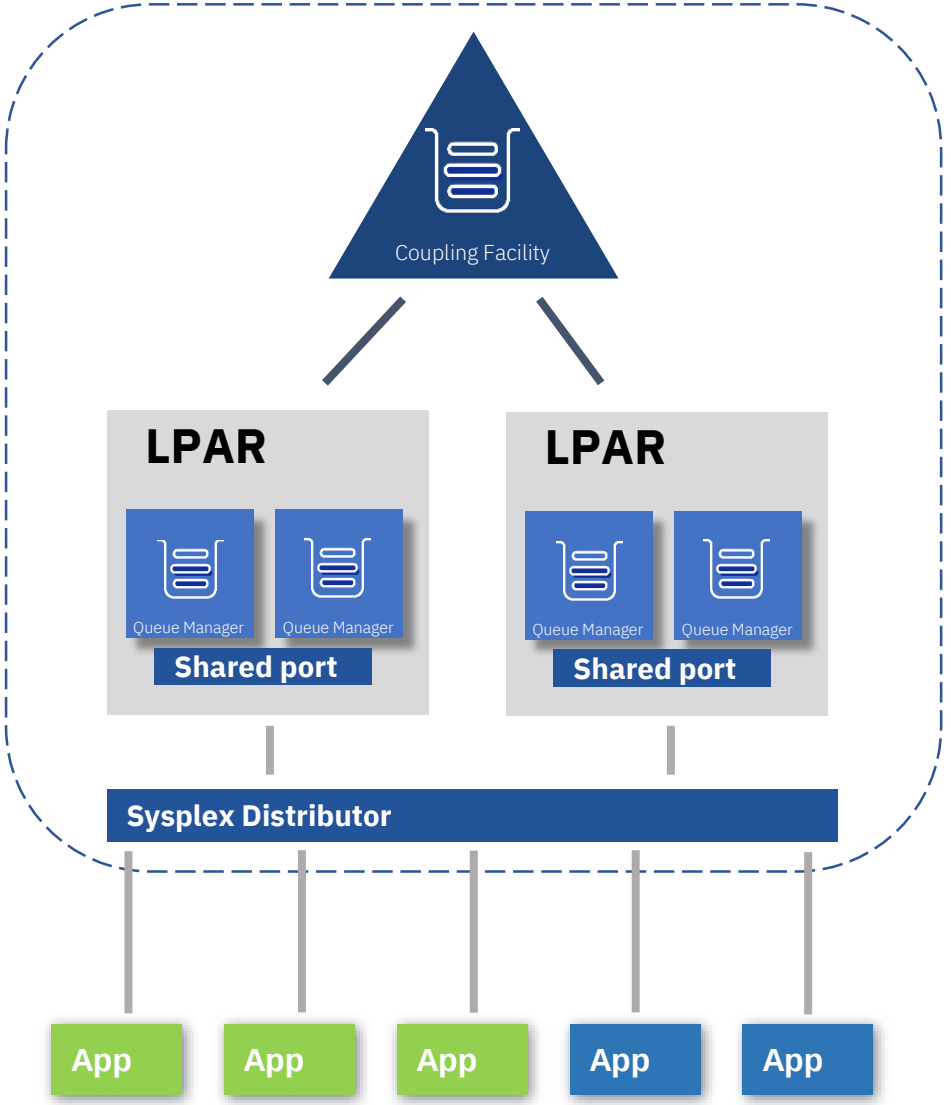
Sysplex Distributor spreads connections across the queue managers based on configured workload management policies



Sysplex Distributor

Sysplex Distributor relies on a “routing” network stack on a single LPAR distributing the connections. If the “routing” stack fails, the responsibility is moved to another LPAR

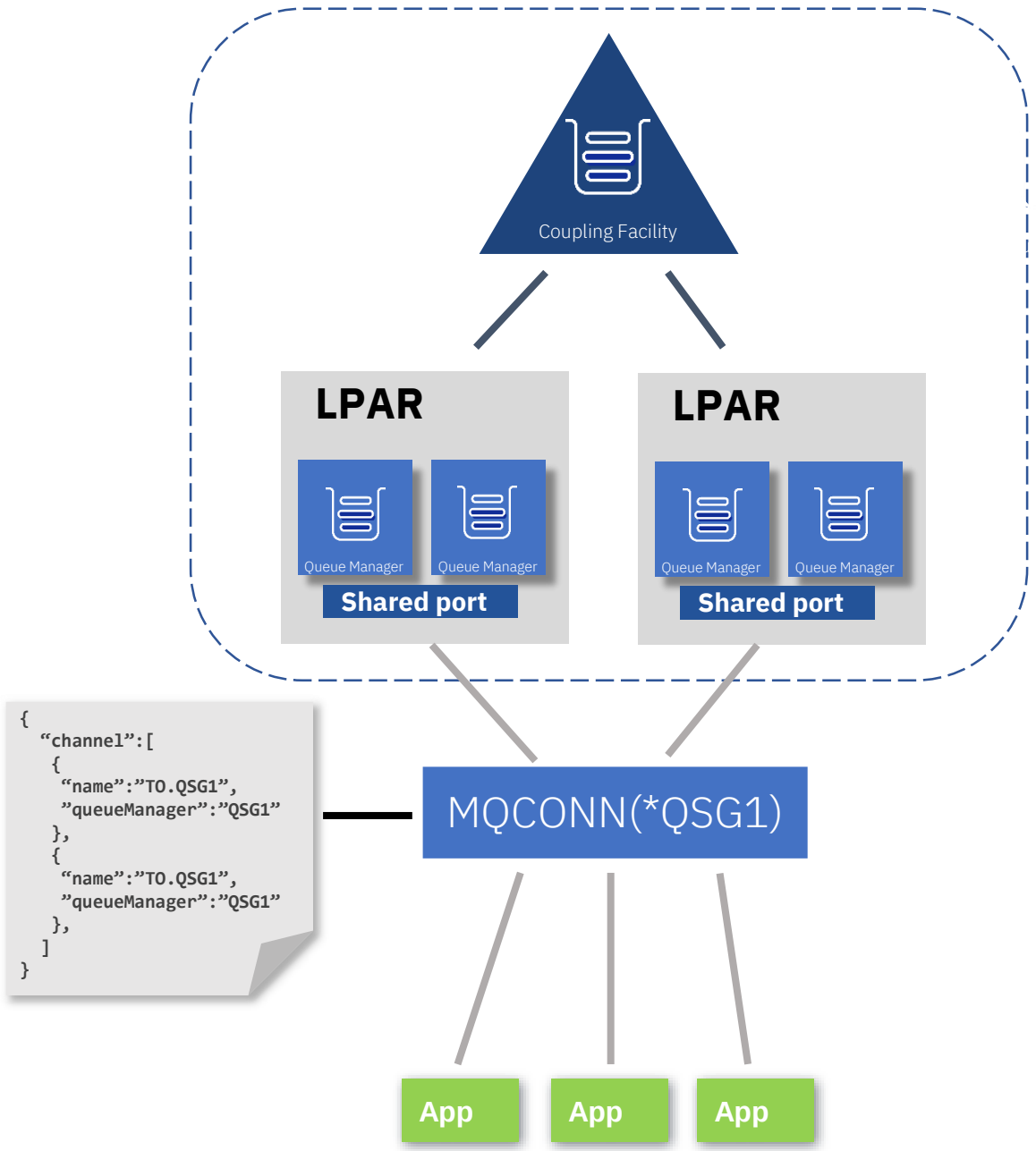
This is transparent to existing connections!



CCDT

CCDTs contain configuration information which is used to define a client channel

MQ clients use this information to connect to a single queue manager, or a group of queue managers

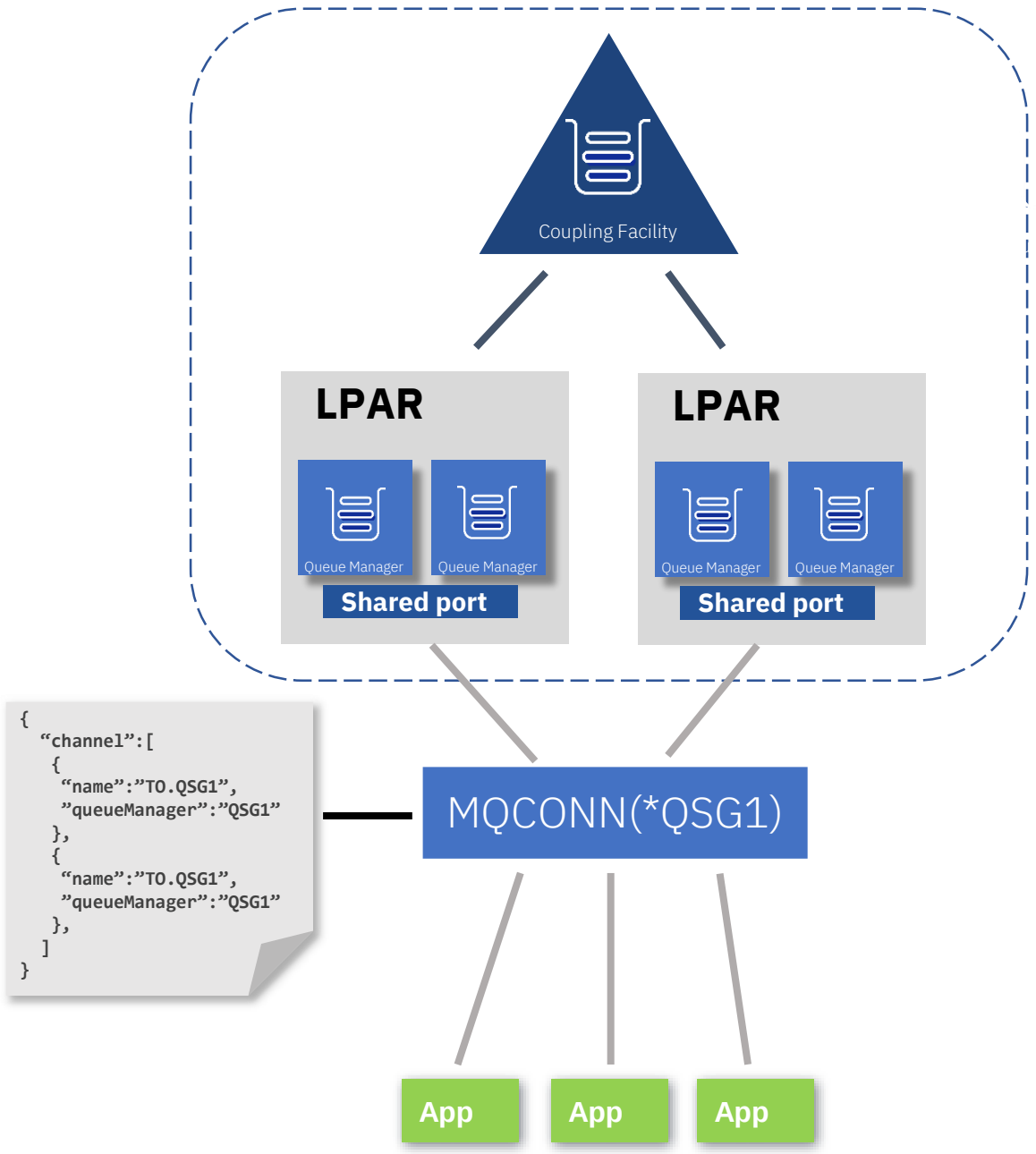


CCDT

CCDTs also provide the ability to randomize where connections are made

This provides the ability to spread work over a queue sharing group

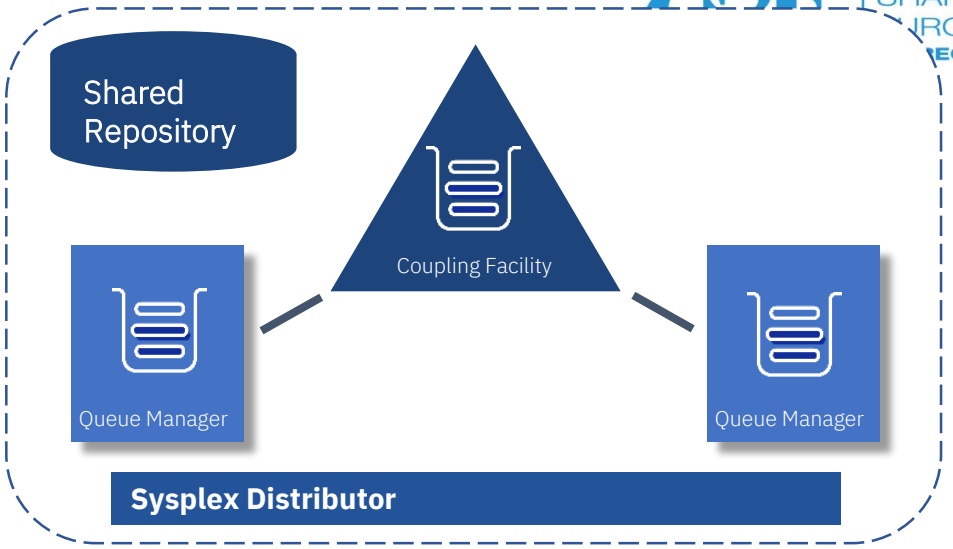
In MQ 9.1.2 this was made easier with JSON format CCDTs as each queue manager can be configured with the same server-connection channel definition

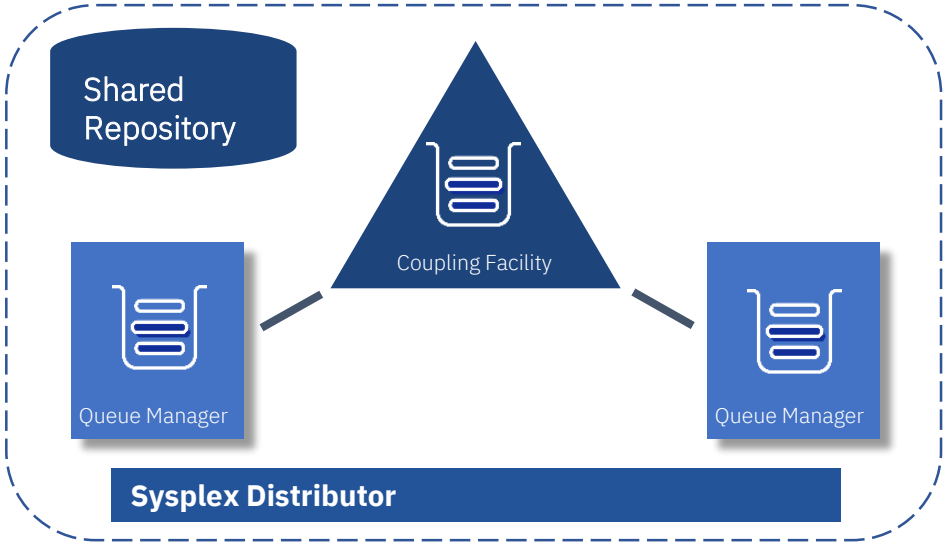


Shared channels

Connectivity concerns aren't just limited to clients connecting into a queue sharing group

You have to consider queue manager to queue manager channels too



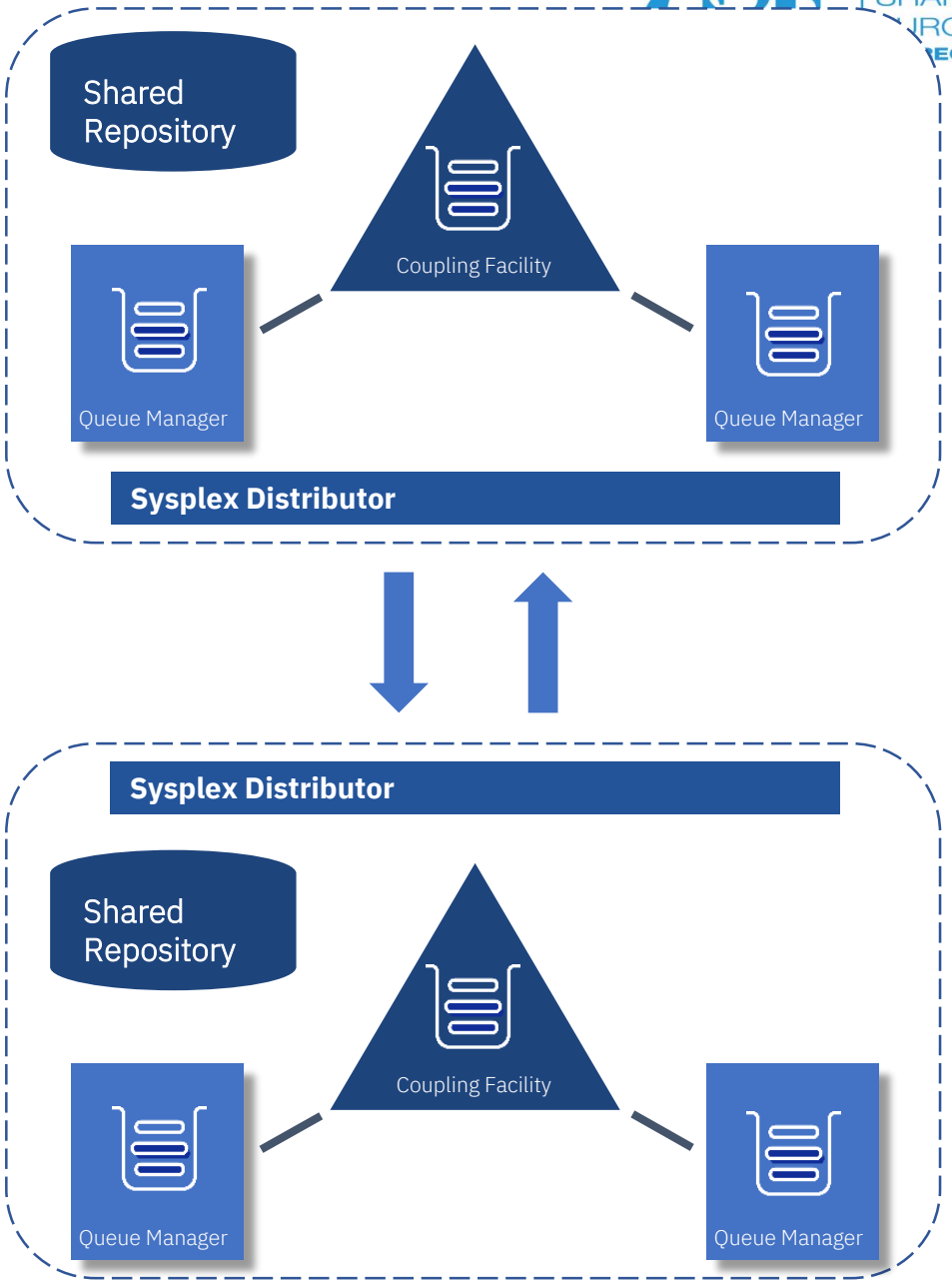


Shared channels

Queue sharing groups support the idea of shared sender and receiver (etc.) channels

These store channel state information at the queue sharing group level rather than in individual queue managers

If a queue manager running a shared channel fails, the channel can automatically be restarted on a remaining queue manager in the group and carry on running where it left off



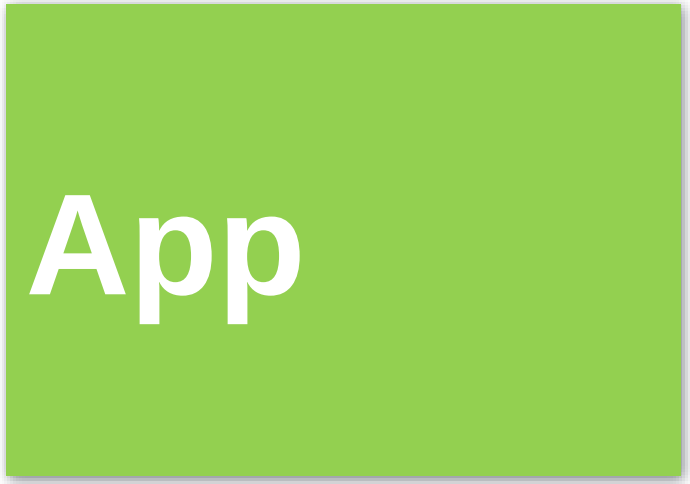
Application considerations

Application considerations

The MQ APIs are generally the same regardless of whether you are connected to a single queue manager, or a queue sharing group

However moving an app to using a queue sharing group is typically not an isolated project, it is often part of a wider effort to improve a system's resilience that includes other subsystems such as CICS, Db2, IMS etc.

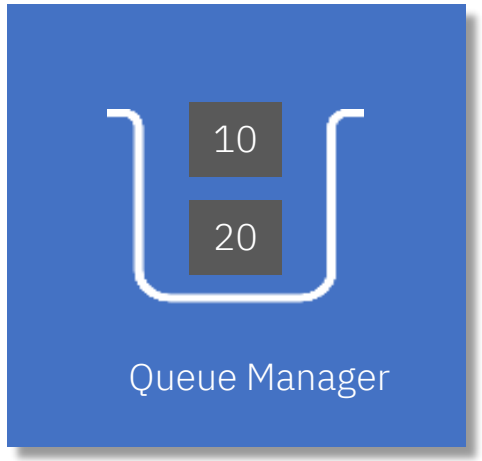
The main challenges are often affinities...

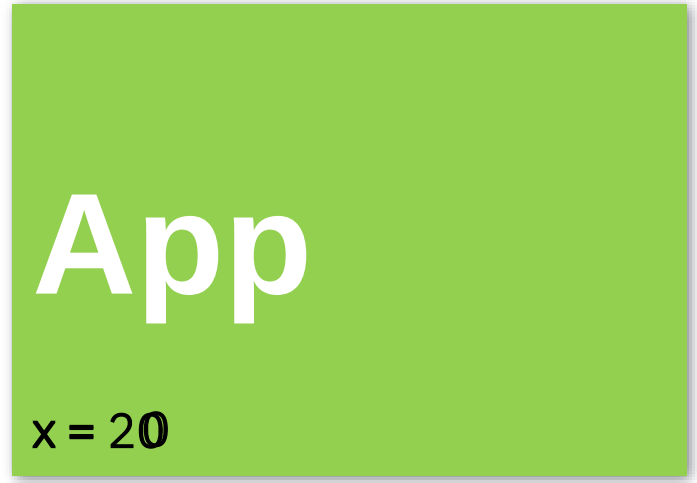


Affinities

An affinity is a coupling between two otherwise separate actions

For example if an app uses the information in one message to generate some state which is stored in local memory, and then updates that state using the information in a second message there is an affinity between the two messages

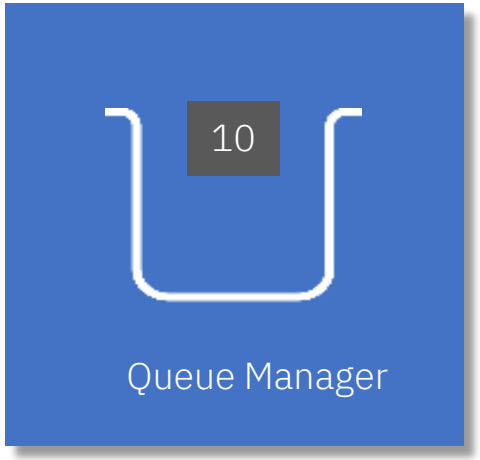


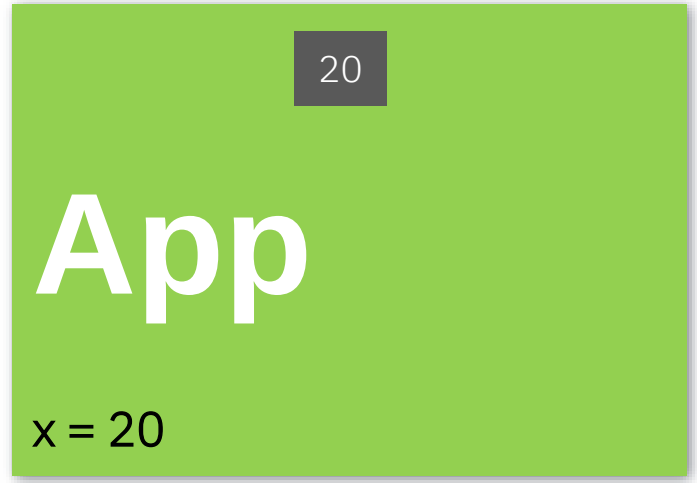


Affinities

An affinity is a coupling between two otherwise separate actions

For example if an app uses the information in one message to generate some state which is stored in local memory, and then updates that state using the information in a second message there is an affinity between the two messages

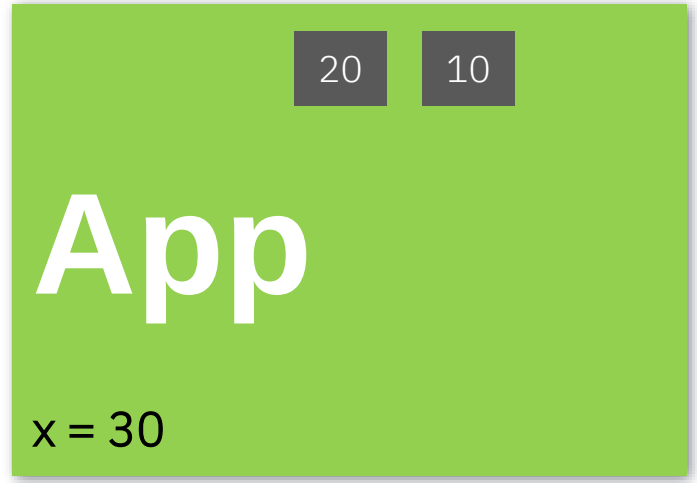




Affinities

An affinity is a coupling between two otherwise separate actions

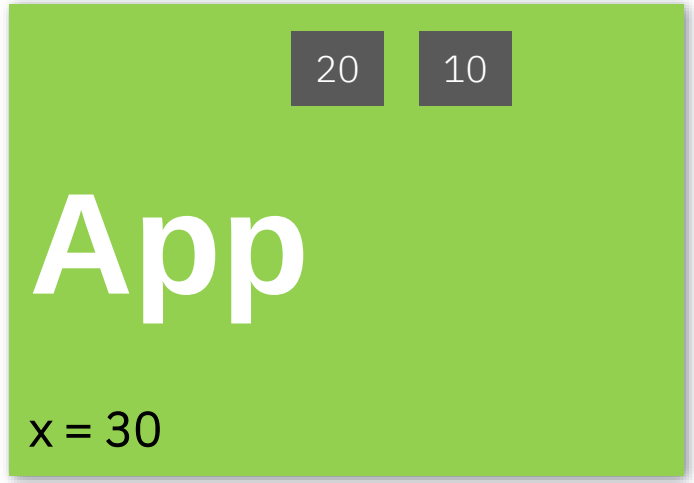
For example if an app uses the information in one message to generate some state which is stored in local memory, and then updates that state using the information in a second message there is an affinity between the two messages



Affinities

Affinities make it hard to use technologies like shared queues as well as parallel sysplex in general

Affinities also cause challenges when using MQ clusters too, so it's a wider challenge!

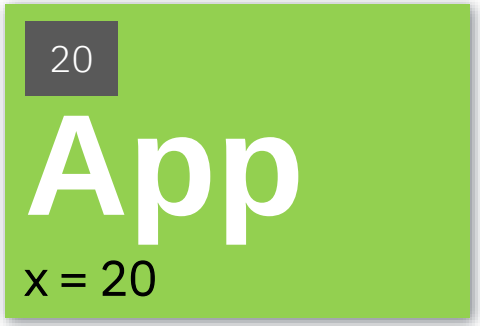


Affinities

In the previous example, what if the first and second messages were got by different app instances on different LPARs?

While this affinity can easily be solved, for example by putting the state in the database, its better to try and avoid affinities as much as possible

Where they are necessary assume that at some point in the future your app is going to need to run multiple (collaborating) instances across multiple LPARs, and design your app with that in mind



Enforcing serialized processing

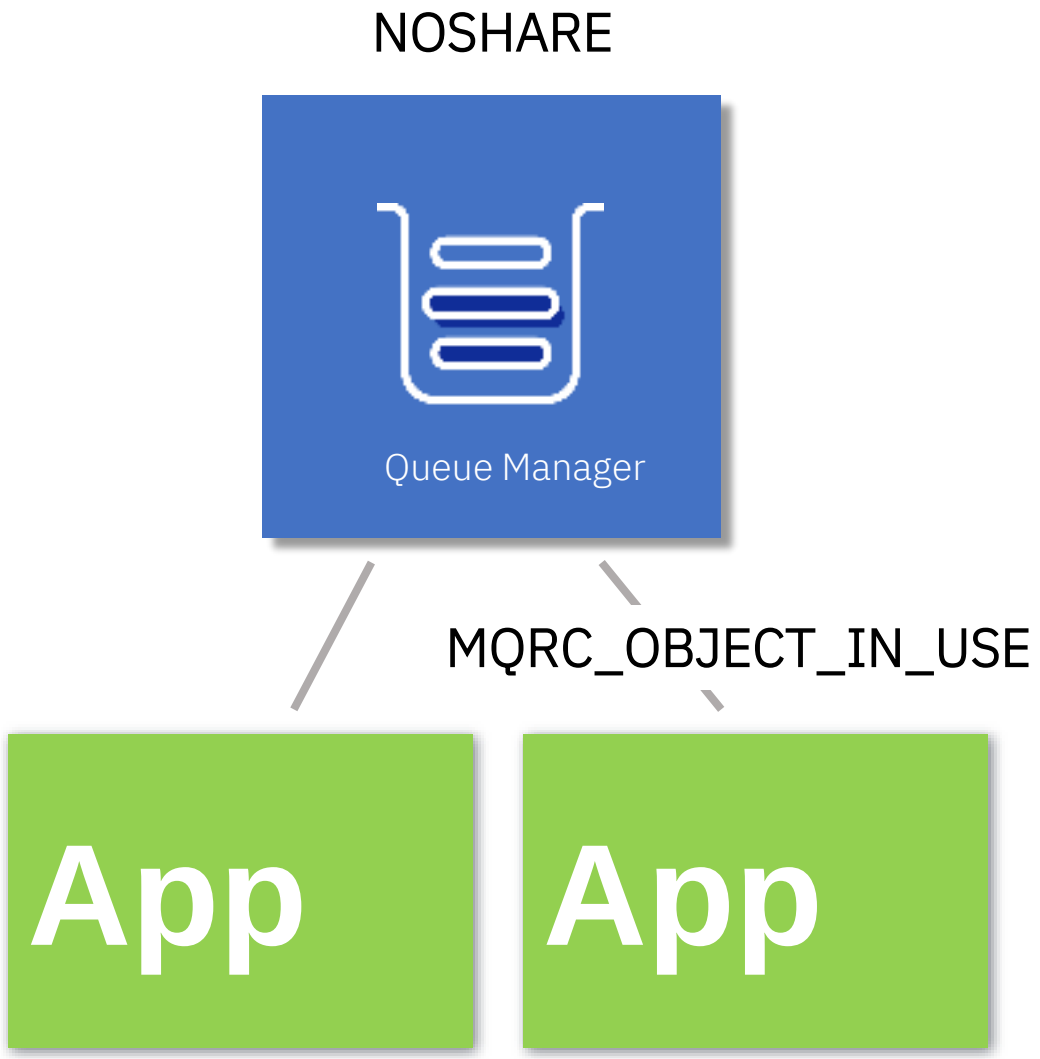
Ideally it would always be possible to have multiple apps processing the messages from a given queue as that gives the greatest ability to scale

However sometimes strict message ordering needs to be maintained

A simple way of achieving this is to define the queue with NOSHARE, or using the MQOO_INPUT_EXCLUSIVE option

However this doesn't guarantee strict message ordering when getting messages under synch-point as the exclusivity is only available until the app closes the queue

It also doesn't help if messages on the queue are partitioned, perhaps by correlation id, or if multiple queues are being used



Enforcing serialized processing

An alternative approach is to use connection tags

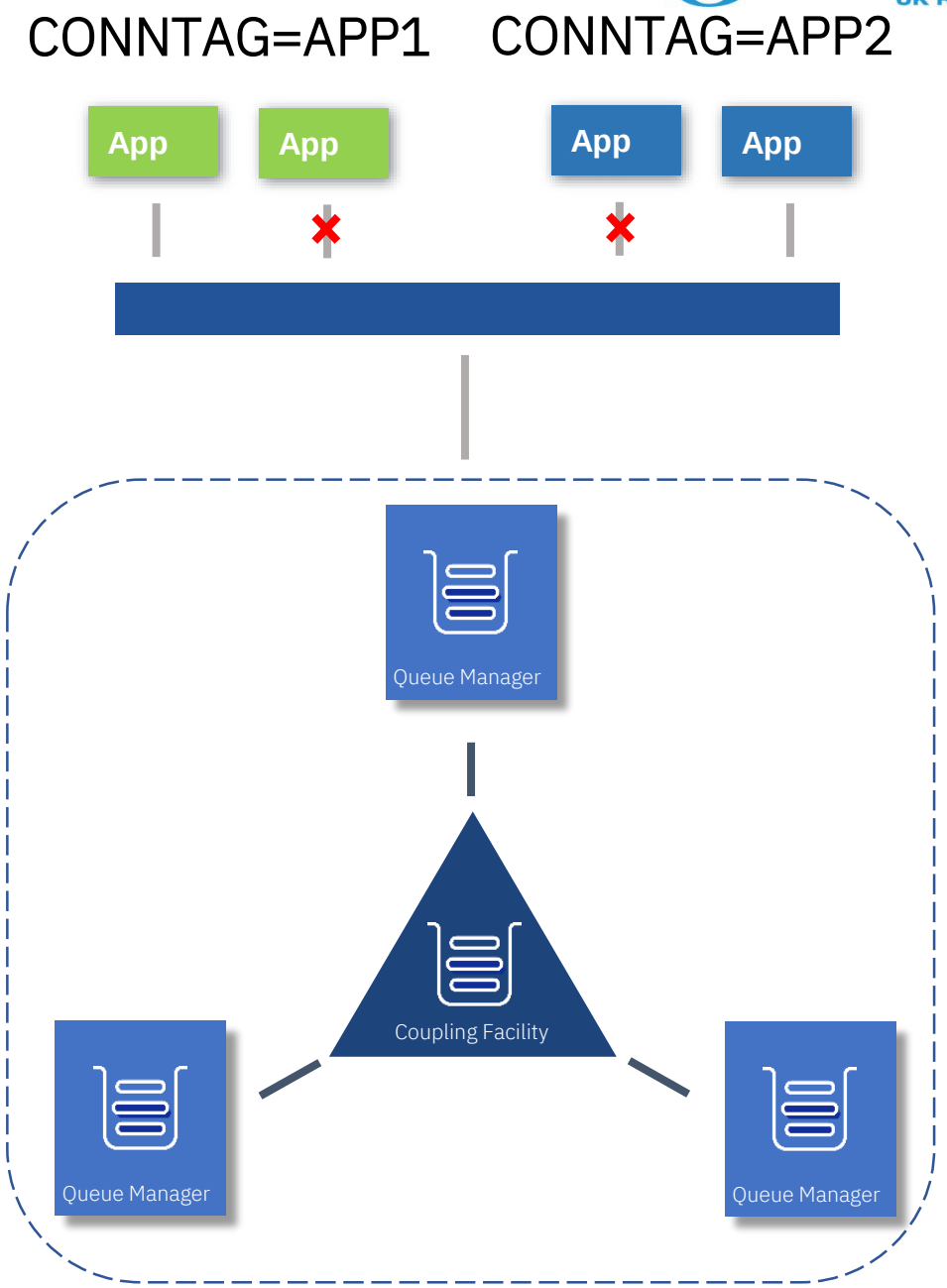
Each app **type** that needs serialization has its own connection tag (an arbitrary byte string)

When the app connects in it passes a flag indicating how it wants to be serialized:

MQCNO_SERIALIZE_CONN_TAG_QSG tag can't be in use at the same time anywhere in the queue sharing group

MQCNO_RESTRICT_CONN_TAG_QSG tag can only be used on different connections if each connection comes from same address space

If the original app instance fails, the tag remains in place until any associated units-of-work are resolved, preserving message order



Resilience

IBM MQ for z/OS provides capabilities to ensure your connectivity never lets you down

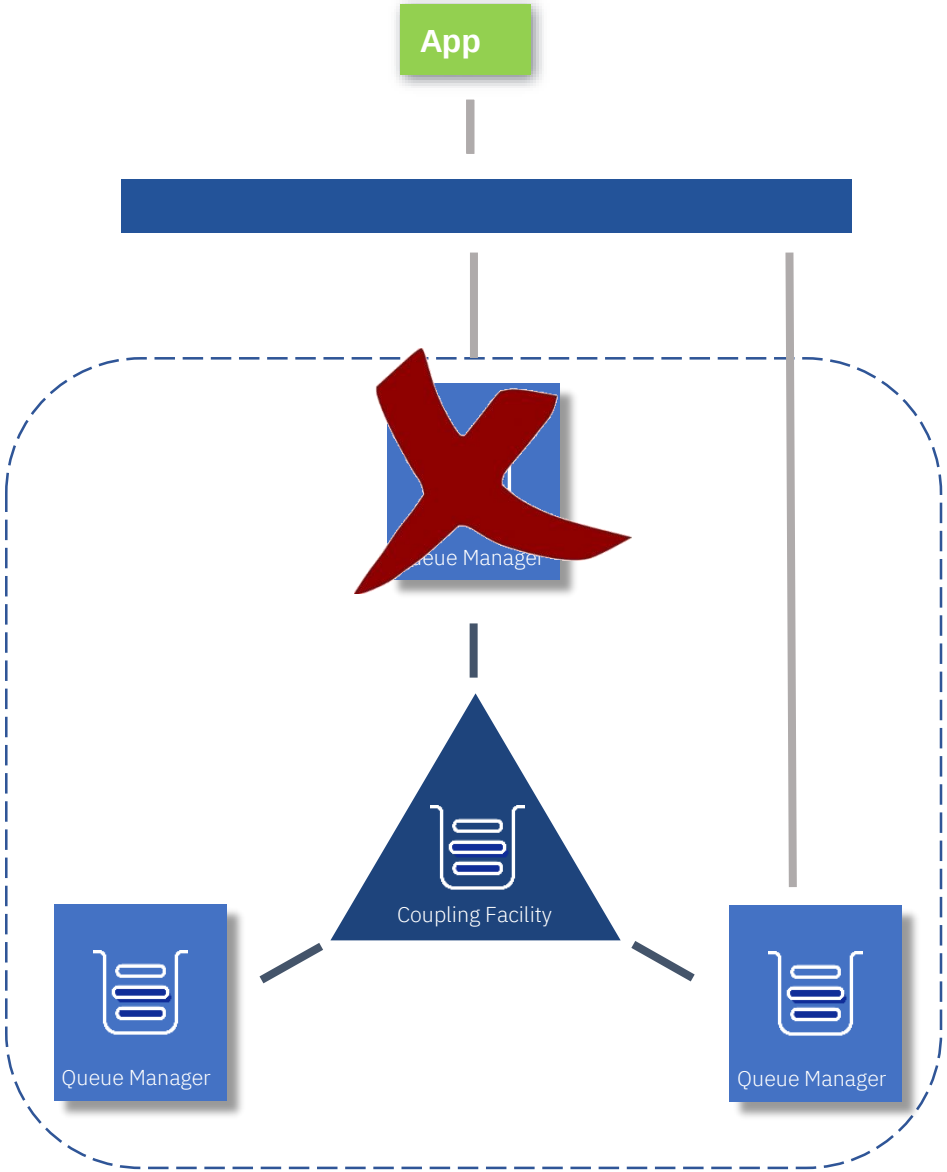
	High Risk and Impact	*Medium Risk and Impact*	*Low Risk and Impact*	*Minimal Risk and Impact*
	Resilient/Reliable	Failover Capable	Fault Tolerant	Continuous Availability
Data High Availability				• Shared Queues – Queue managers are connected via a Coupling Facility. Application requests for a service are held in the CF and available to all connected queue managers. If an individual queue manager is unavailable for any reason, new and existing messages are available to the rest of the queue managers
Service High Availability			• Clustering - Application requests for a service are spread among a cluster of queue managers. If a queue manager fails, new requests are routed to surviving queue managers allowing the service to continue to be available, eliminating SPoFs for message driven services. Existing messages on the failed queue manager aren't available until it is back online	• Clustering - Application requests for a service are spread among a cluster of queue managers. If a queue manager fails, new requests are routed to surviving queue managers allowing the service to continue to be available, eliminating SPoFs for message driven services. Existing messages on the failed queue manager aren't available until it is back online
Enhanced Data Recovery		• Data replication – MQ active log datasets can be mirrored to a secondary storage subsystem using data replication. zHyperwrite support improves performance	• Data replication – MQ active log datasets can be mirrored to a secondary storage subsystem using data replication. zHyperwrite support improves performance	• Data replication – MQ active log datasets can be mirrored to a secondary storage subsystem using data replication. zHyperwrite support improves performance
Data Recovery	• Logging – MQ records all significant changes to persistent data in a recovery log. Dual logging offers protection against data loss. • Archiving – Logs automatically archived to secondary storage (tape or DASD)	• Logging – MQ records all significant changes to persistent data in a recovery log. Dual logging offers protection against data loss. • Archiving – Logs automatically archived to secondary storage (tape or DASD)	• Logging – MQ records all significant changes to persistent data in a recovery log. Dual logging offers protection against data loss. • Archiving – Logs automatically archived to secondary storage (tape or DASD)	• Logging – MQ records all significant changes to persistent data in a recovery log. Dual logging offers protection against data loss. • Archiving – Logs automatically archived to secondary storage (tape or DASD)

Resilience to queue manager failure

Persistent and non-persistent messages on shared queues are available via other queue managers in the group if individual queue managers fail

If client apps are using automatic client reconnect then the failure can be entirely transparent to them

Alternatively, applications can detect the connection error and reconnect to the group and carry on processing



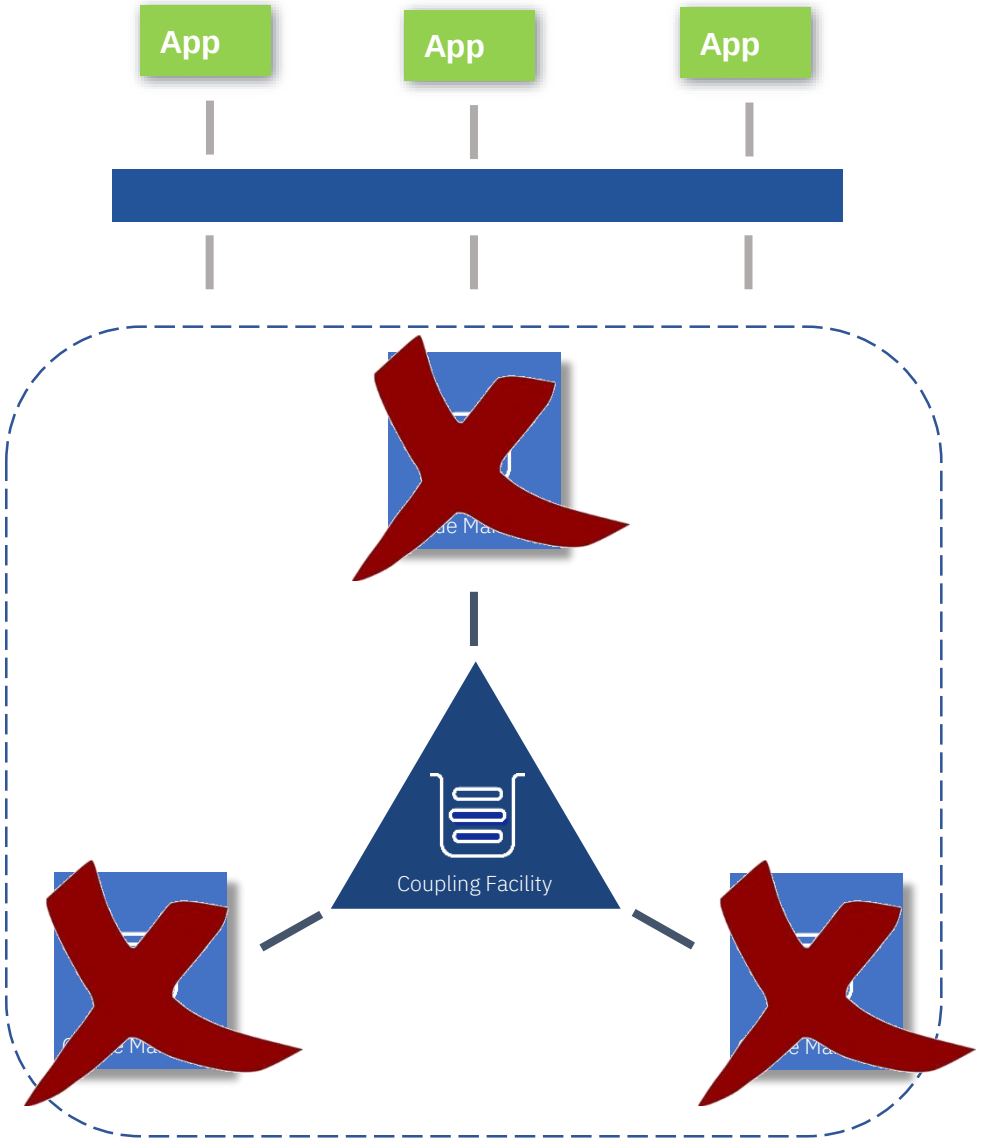
Resilience to queue manager failure

Persistent and non-persistent messages on shared queues are available via other queue managers in the group if individual queue managers fail

If client apps are using automatic client reconnect then the failure can be entirely transparent to them

Alternatively, applications can detect the connection error and reconnect to the group and carry on processing

Note that non-persistent messages are not lost even all queue managers in the group fail!

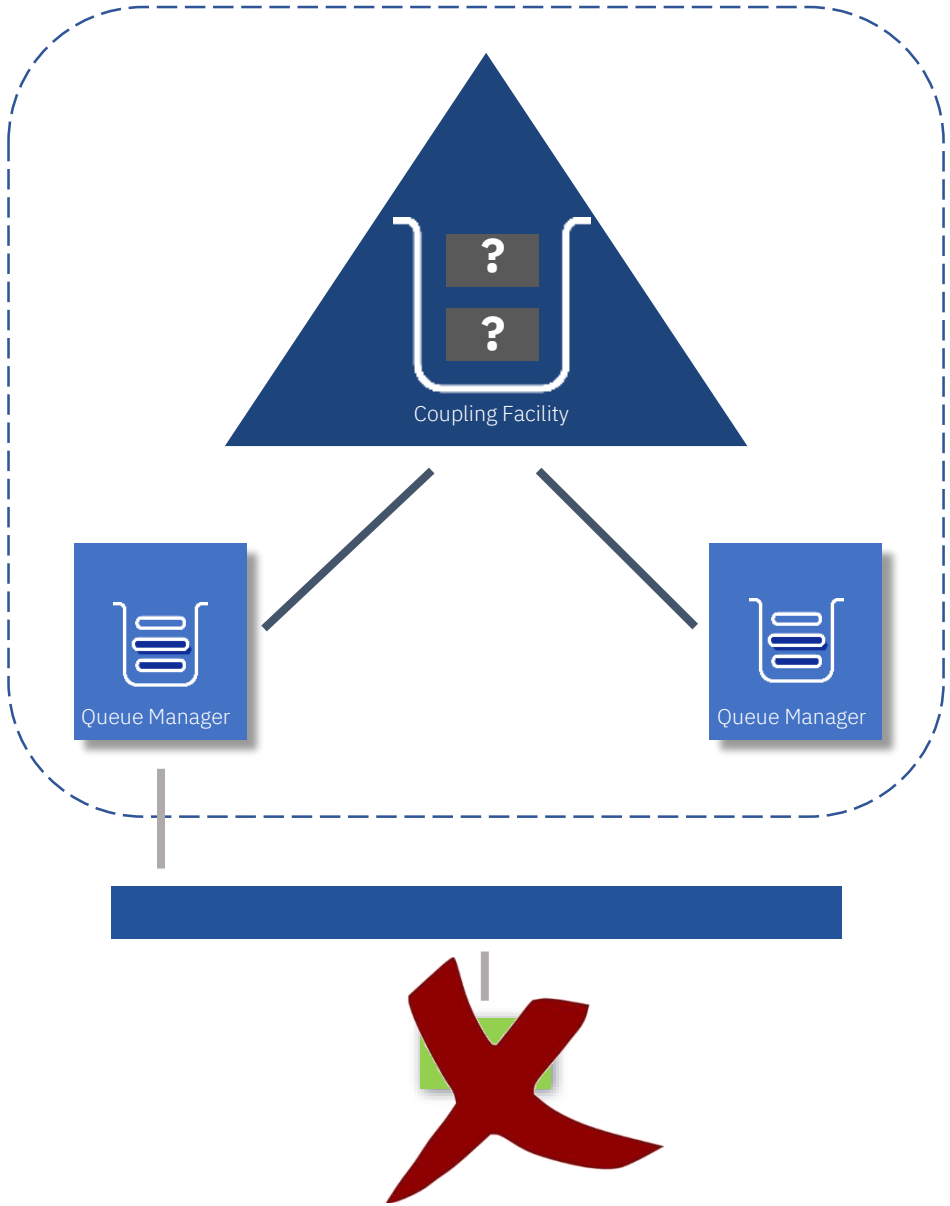


Transaction recovery

When apps fail, the queue manager and app runtime environment cooperate to ensure that inflight transactions are either committed, or rolled back

This happens regardless of whether shared queues are being used or not

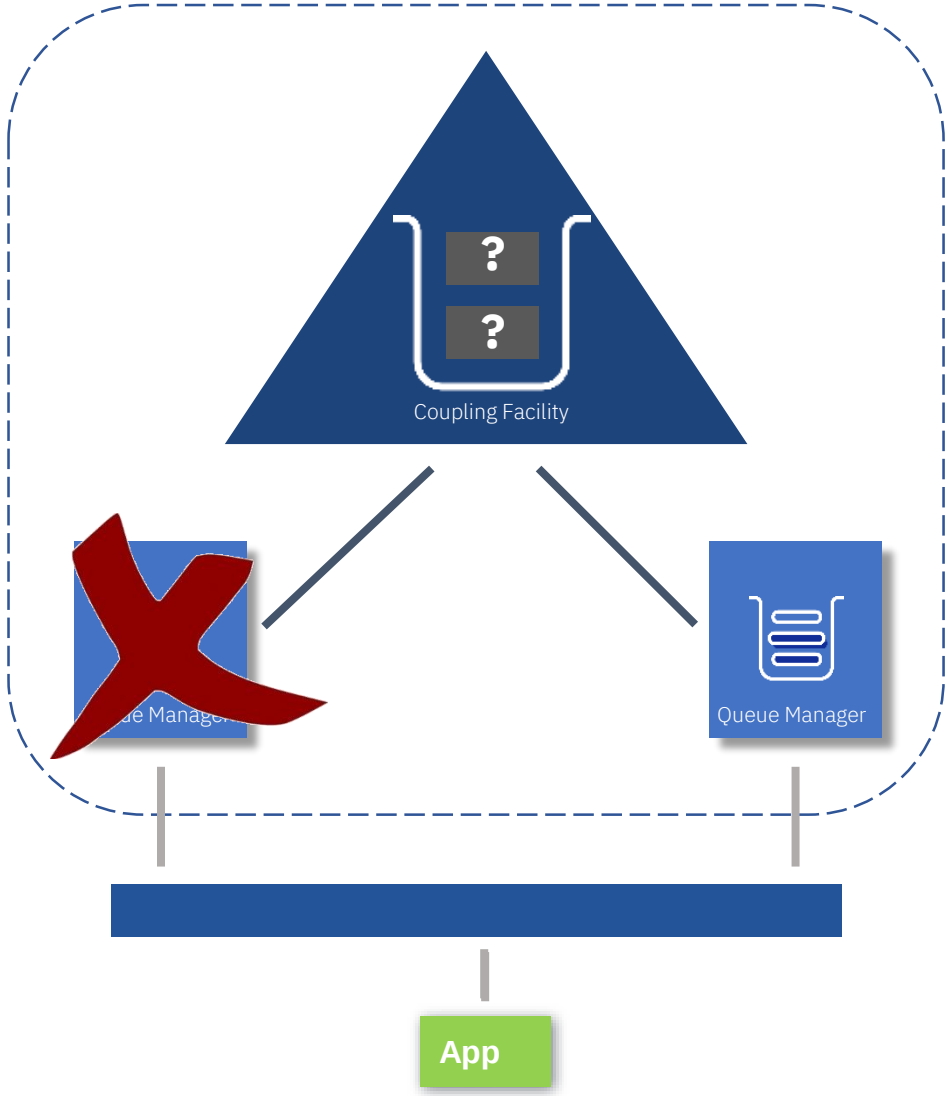
Rolling back the transaction, if necessary, allows other instances of the app to process the messages



Peer recovery

If a queue manager fails while an app was using shared queues the aim is for the app to be able to carry on processing as quickly as possible

This is achieved using peer recovery – other queue managers in the group use information in the coupling facility and the failed queue manager's logs to rollback, or commit messages as appropriate allowing the app to carry on processing

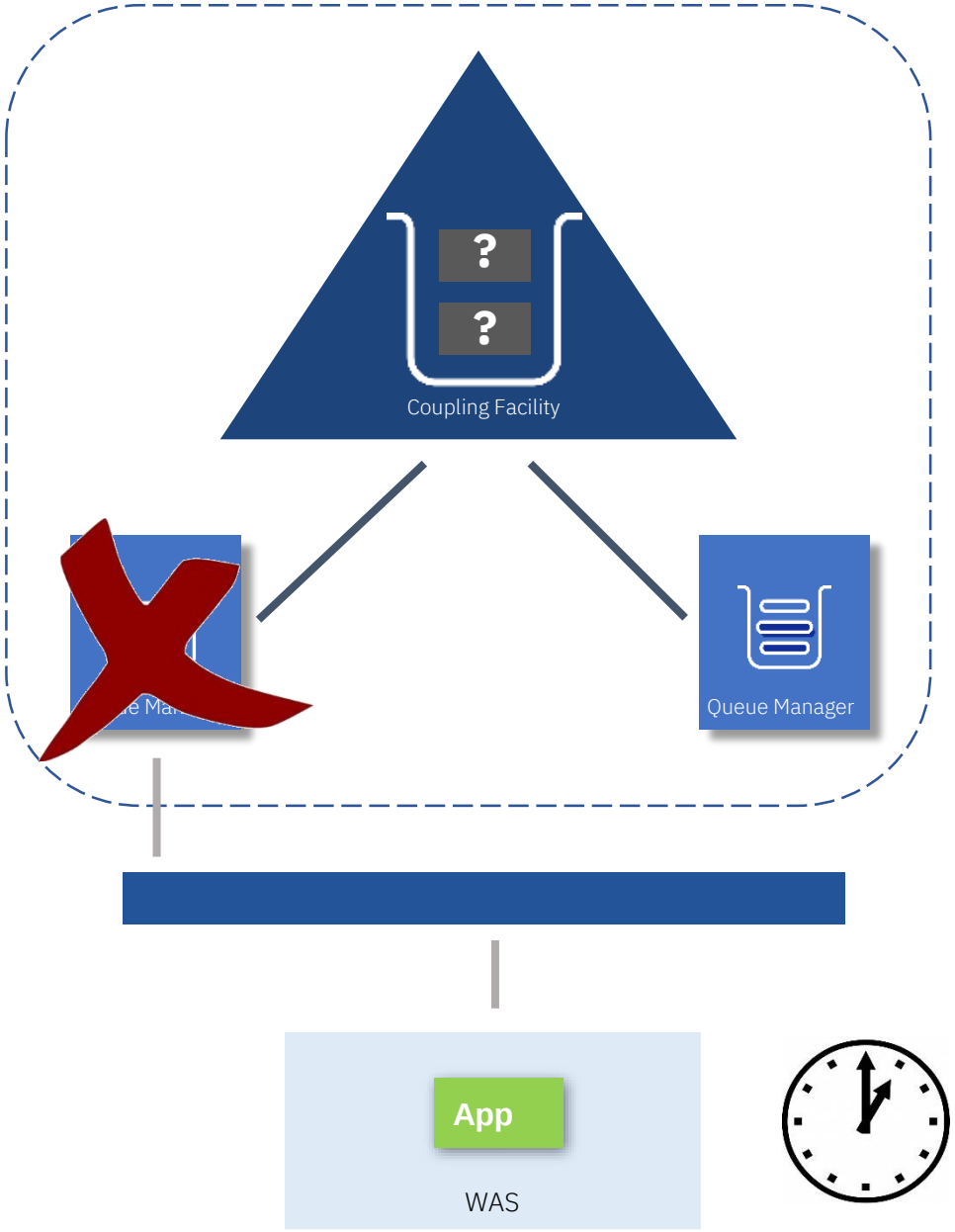


Group UR

In some cases the app runtime environment (CICS or WAS) own the transaction state and must cooperate with MQ to correctly resolve the transaction

Traditionally the queue manager that the app was interacting with owned this information, which meant it needed to be restarted to recover the transaction

This can delay recovery



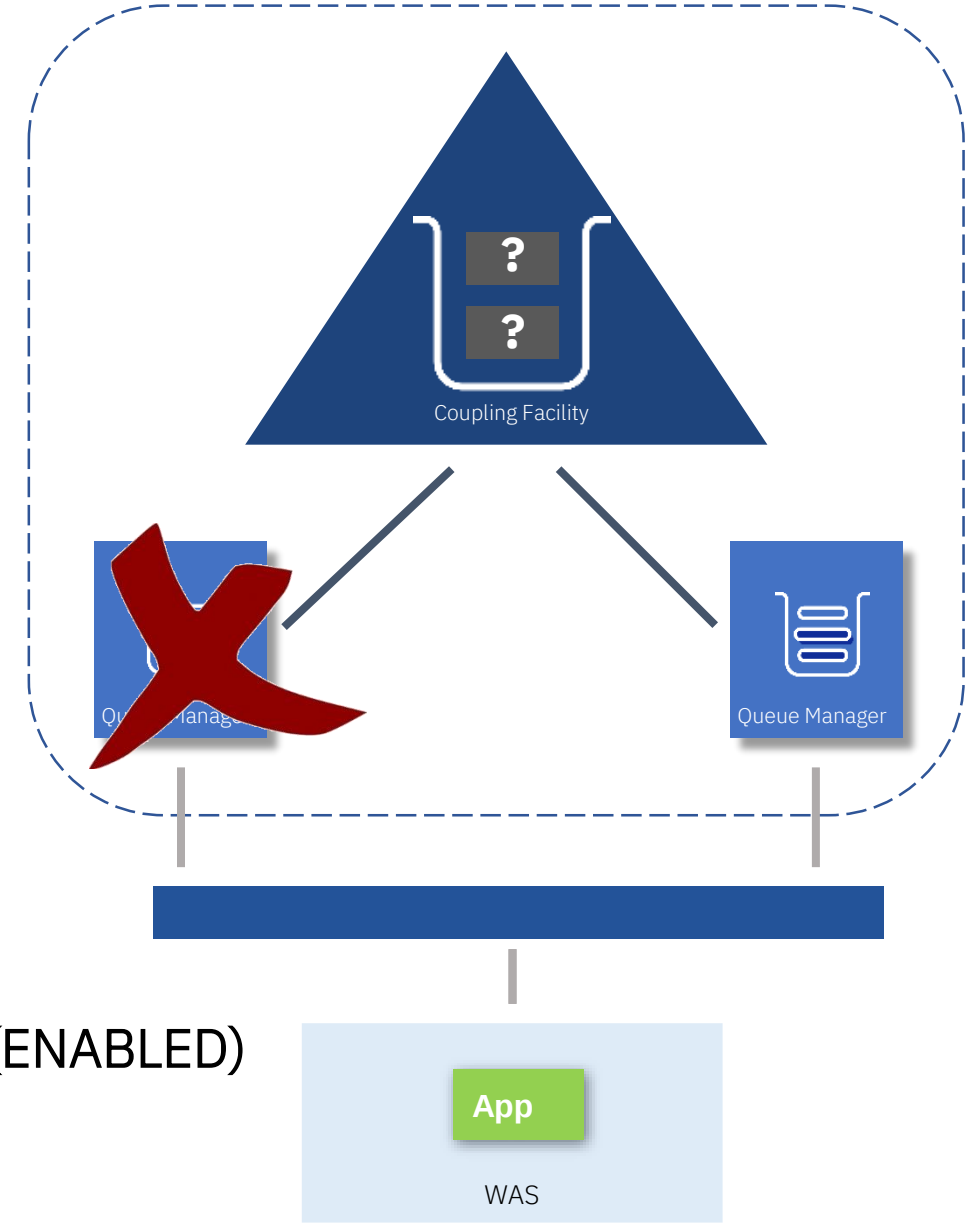
Group UR

Group units of recovery can be enabled on the queue managers in the group

This allows transaction state to be owned by the queue sharing group, instead of the individual queue manager

In the event of a queue manager failure the app runtime environment can then connect to another queue manager in the group and resolve the transaction

Allowing the app to carry on processing



ALTER QMGR GROUPUR(ENABLED)

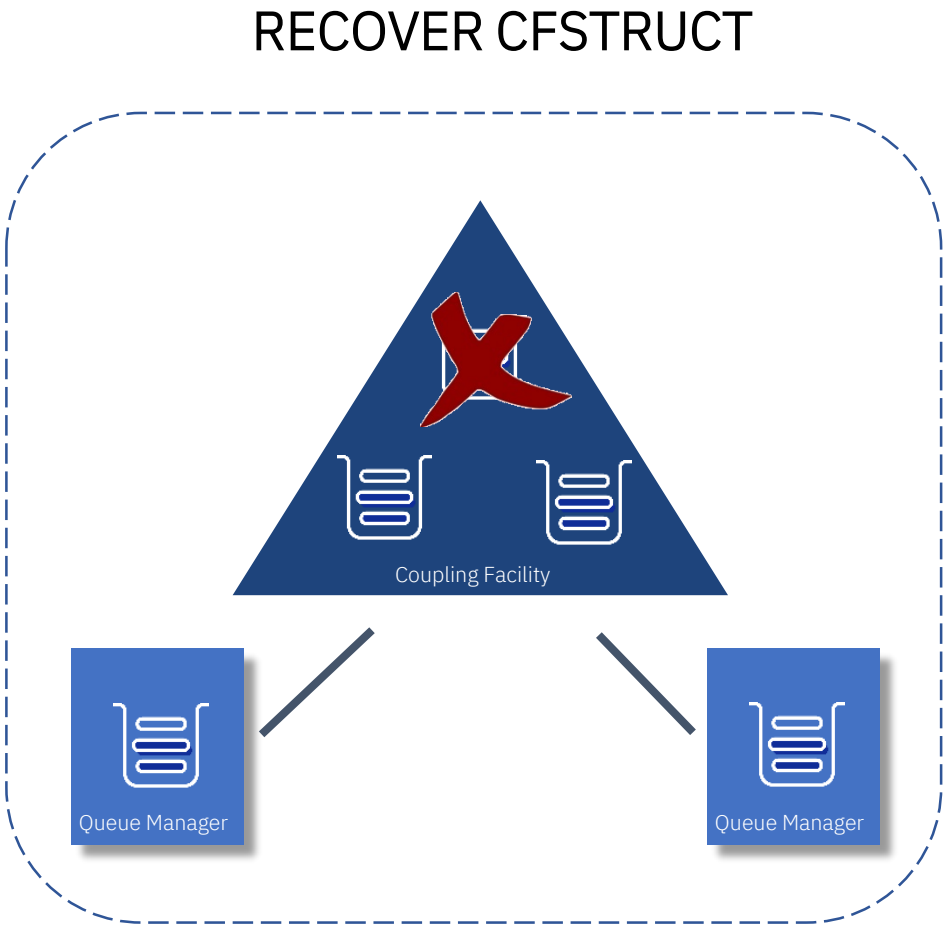
Resilience to structure failure

The structures used to store shared queues are highly resilient to failure as they run in a coupling facility, and coupling facilities don't contain any application code

It is possible to increase this failure isolation by using dedicated hardware for coupling facilities

If a structure does fail then any non-persistent messages stored in the structure are lost

Persistent messages can be restored from a backup in a queue manager's logs using the RECOVER CFSTRUCT command, this can be done manually or automatically by MQ



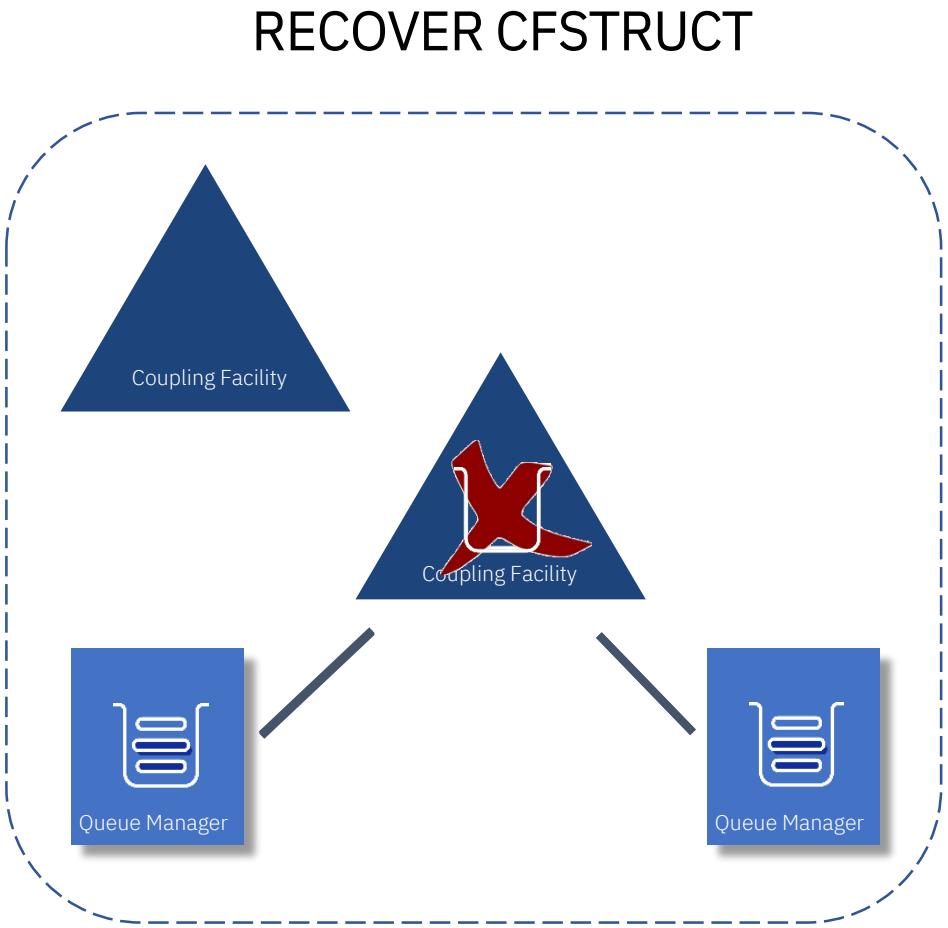
Resilience to coupling facility failure

The network links between queue managers and coupling facilities can fail, or indeed the coupling facility itself can fail

A total failure of network links is treated the same as a failure of a coupling facility by MQ

If this situation is detected then MQ will attempt to reconnect to the coupling facility which might result in an alternate coupling facility being used

As before, persistent messages can be restored from a backup in a queue manager's logs using the RECOVER CFSTRUCT command, this can be done manually or automatically by MQ

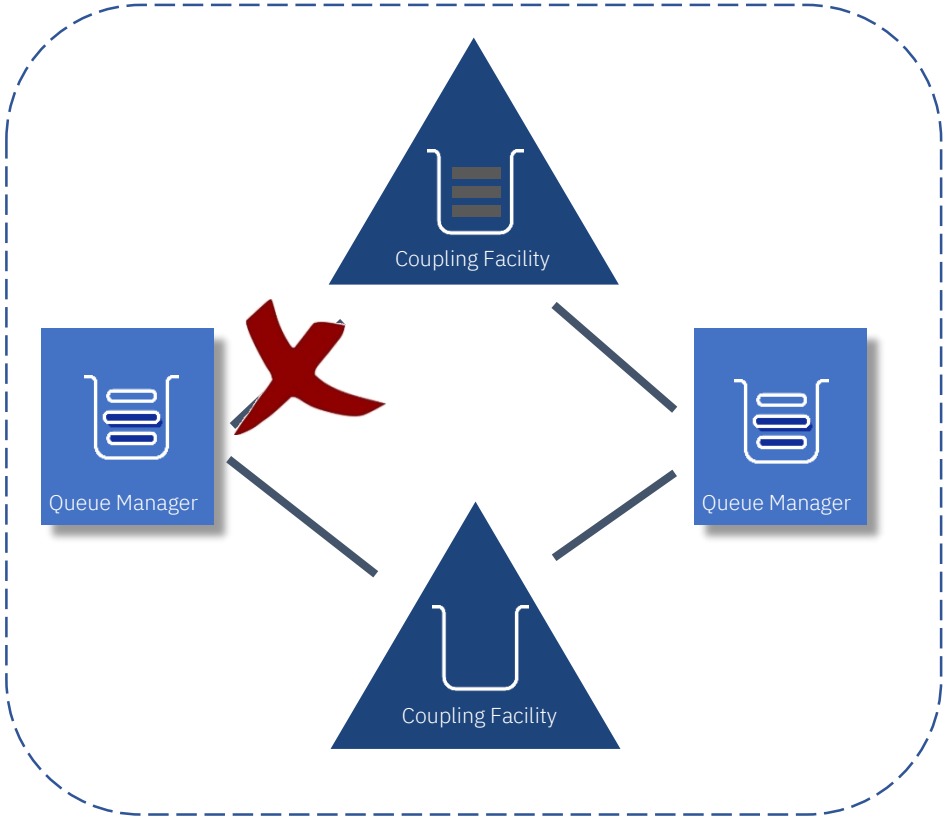


Resilience to coupling facility failure

The network links between queue managers and coupling facilities can fail, or indeed the coupling facility itself can fail

If some network links fail, and only a subset of queue managers loose access to the coupling facility the queue managers will request that the contents of the shared queues are copied to a coupling facility to which all queue managers can access

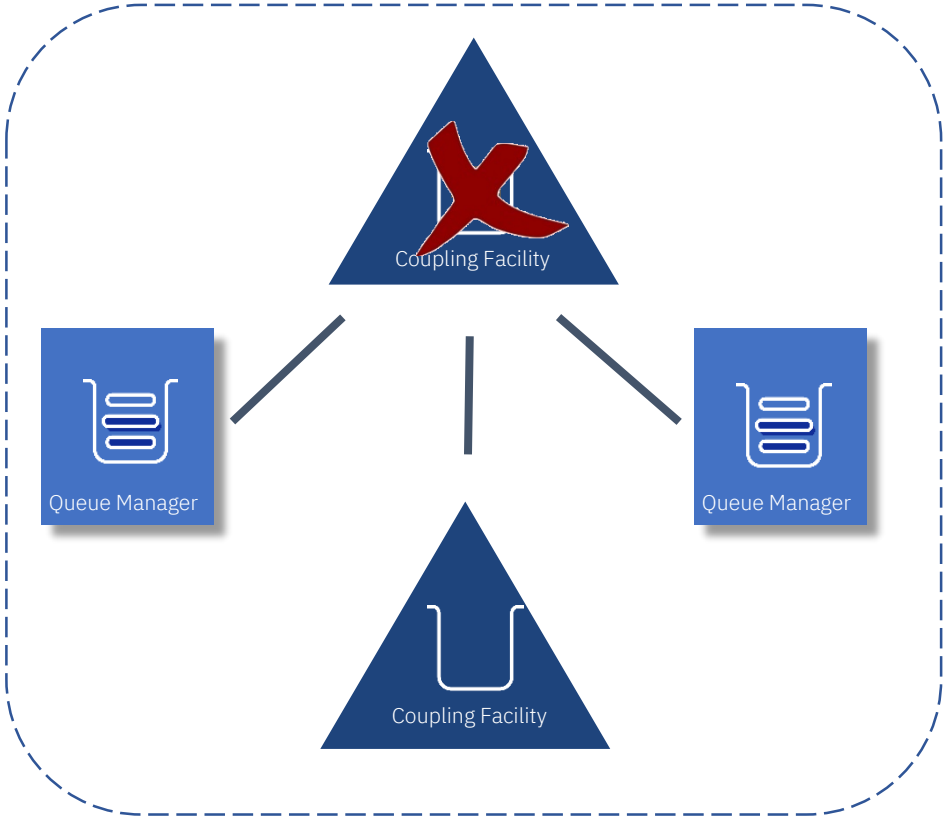
In this case non-persistent and persistent messages are preserved, and apps just experience a brief delay while the messages are copied



Resilience to coupling facility failure - duplexing

z/OS also provide coupling facility duplexing where each write operation to a primary coupling facility is synchronously replicated to a backup

If the primary coupling facility fails, the backup transparently becomes the primary without MQ, or apps, being aware



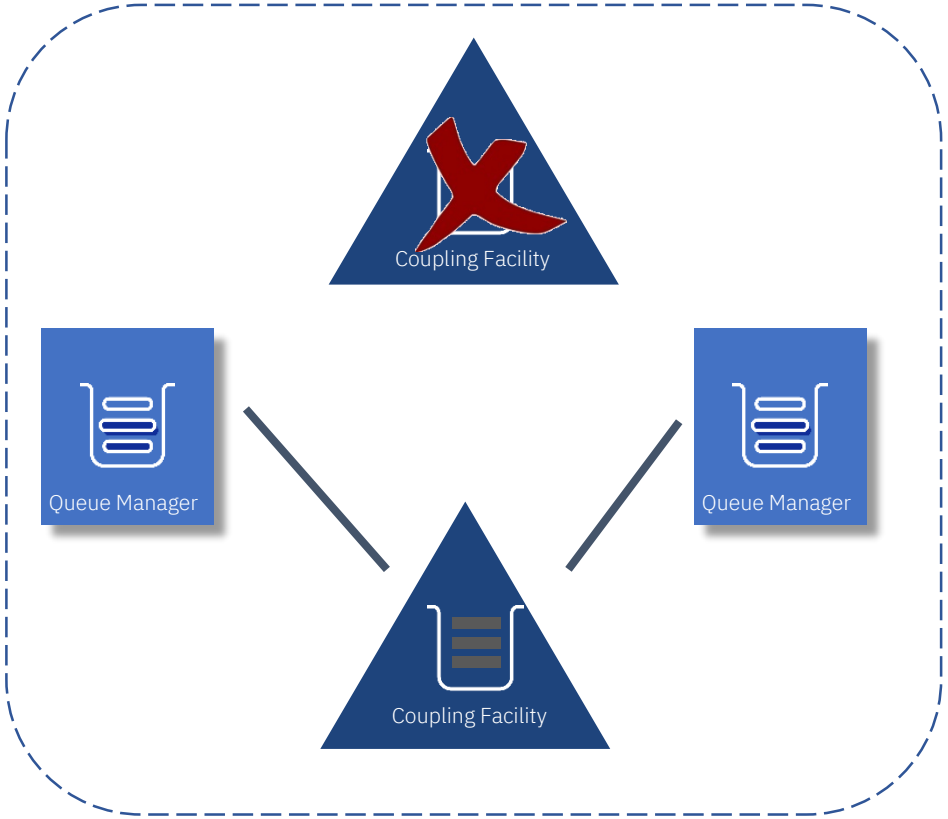
Resilience to coupling facility failure - duplexing

z/OS also provide coupling facility duplexing where each write operation to a primary coupling facility is synchronously replicated to a backup

If the primary coupling facility fails, the backup transparently becomes the primary without MQ, or apps, being aware

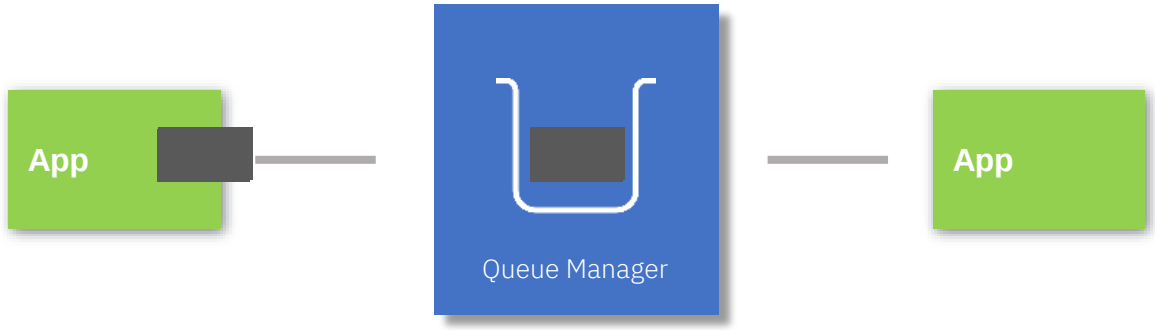
While providing an extremely high level of resilience this approach needs to be balanced against the higher CPU cost and extra latency it introduces on every single write operation

In many cases the automatic recovery, and connectivity loss tolerance, provided by MQ is sufficient



Resilience to application failure

Queues are designed for temporary storage of data being exchanged between apps, allowing the apps to be decoupled and scale independently



Resilience to application failure

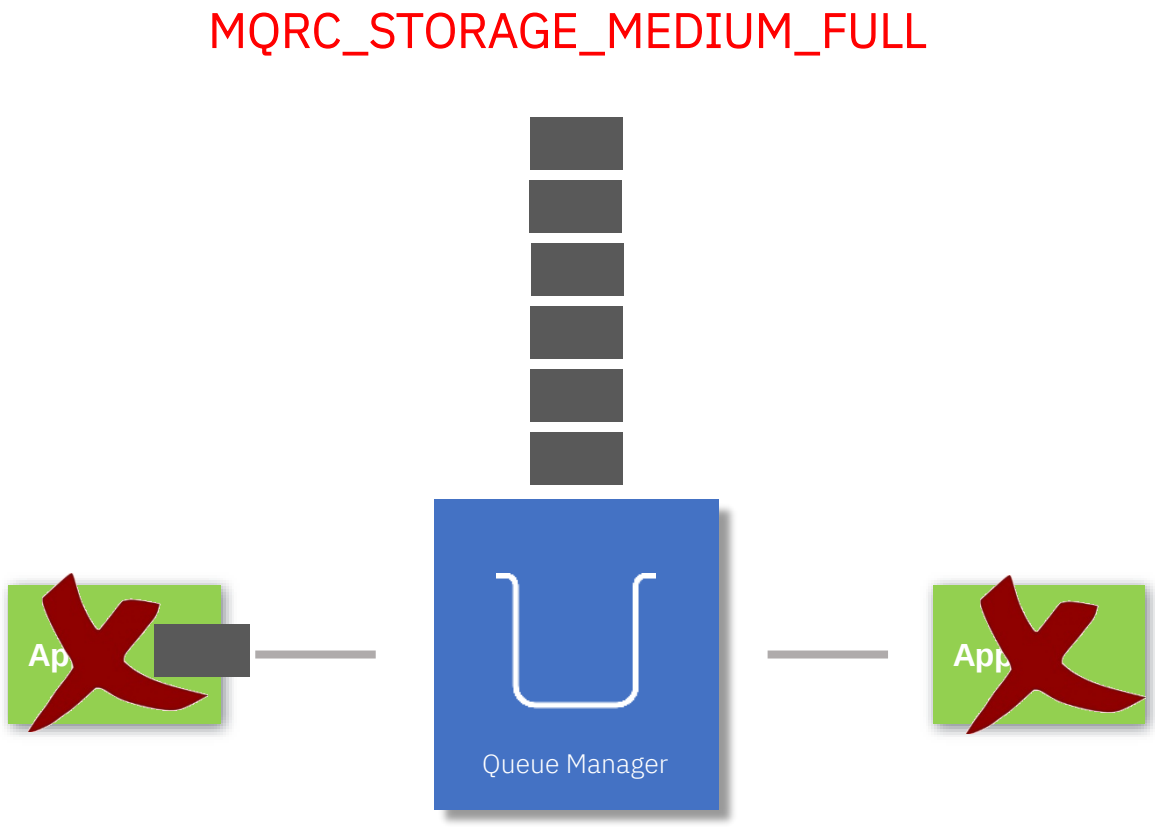
Queues are designed for temporary storage of data being exchanged between apps, allowing the apps to be decoupled and scale independently

Should a getting app fail the queue can act as a buffer for messages sent by the putting app

However queue storage is finite and will eventually fill up if the getting app isn't started in time. This can potentially lead, in the worst cases to a putting app failure

Private queues can hold at most 64GB of message data

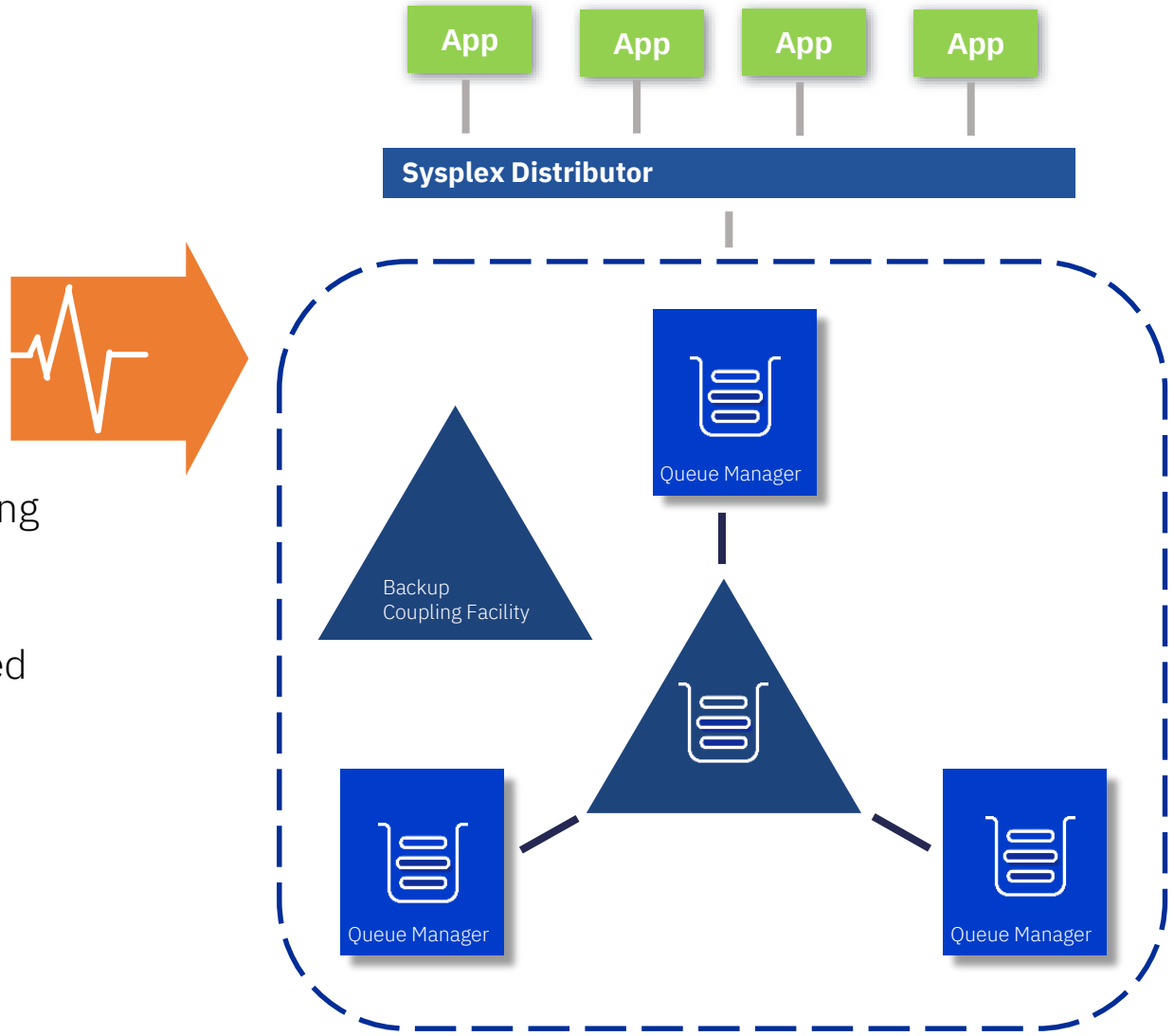
Shared queues configured to offload data to SMDS can hold much more data – many TBs, providing extra time to resolve app outages, and a more resilient solution



Putting it all together

Bringing it all together

- Design your apps so they have no, or minimal, affinities
- Build a set of queue managers using a single configuration template
- Create a queue sharing group and add the queue managers to it
- Configure backup coupling facilities
- Define the necessary MQ objects to the queue sharing group rather than individual queue managers, using shared queues where necessary
- Ensure your queues and storage are sufficiently sized for resilience in the case of app failure
- Use technology like Sysplex Distributor to provide a resilient workload balancing layer
- Connect your apps
- Monitor your environment, including your apps, for failure



Remember that equivalent steps need to be taken for the rest of your environment

A comparison with uniform clusters

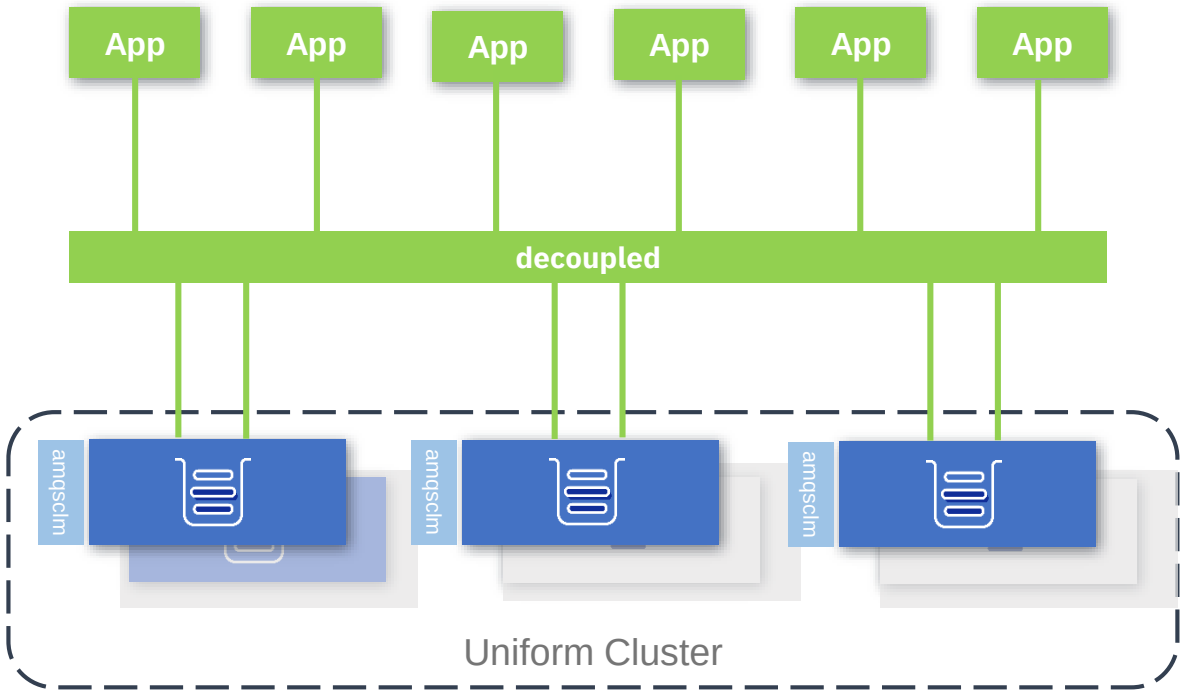
Uniform clusters – a recap

Introduced in MQ 9.1.2, and enhanced in later CD releases, uniform clusters make it easier to build scalable, fault tolerant solutions on distributed MQ

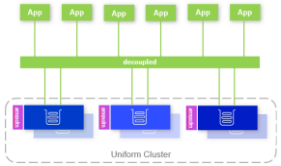
Uniform clusters build on the existing capabilities of MQ clustering and CCDTs to spread work over a set of identically configured queue managers

One significant benefit of uniform clusters, compared with regular clusters, is that MQ works to ensure that apps are spread over the available queue managers in the cluster to prevent messages building up when there is no connected app to consume them

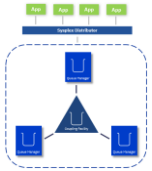
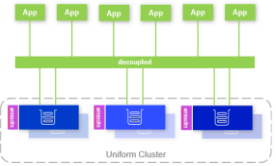
Uniform clusters are not available on z/OS. Use queue sharing groups instead.



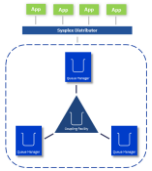
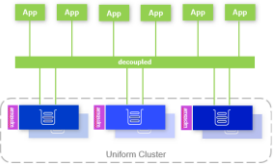
A comparison

Capability	Queue sharing groups 	Uniform clusters 
Ability to simply apply common configuration across all queue managers at startup	Yes	The administrator is currently responsible for this
Simple and efficient distribution of apps across available queue managers	Yes using Sysplex Distributor, a CCDT or an IP sprayer	Yes using JSON CCDTs
Automatic app rebalancing to ensure an even spread of apps at all times	No. However with shared queues apps can access all messages regardless of where they are connected	Yes
Ensuring all messages are processed	Yes. With shared queues apps can access all messages regardless of where they are connected to	Yes using AMQSCLM and automatic app rebalancing

A comparison

Capability	Queue sharing groups 	Uniform clusters 
High availability	Yes, queue sharing groups provide a range of capabilities to provide best in class high availability of messages regardless of queue manager, hardware or app failure	Yes, uniform clusters provide high availability of the services using clustered queues and can be combined with technologies such as RDQM for queue manager high availability
Scaling up	Queue managers can easily be added to the queue sharing group, Sysplex Distributor ensures an even spread of new workload, existing workload remains where it is	Queue managers can easily be added to the uniform cluster, automatic app rebalancing ensure an even spread of workload, both new and existing

A comparison

Capability	Queue sharing groups 	Uniform clusters 
Scaling down	Queue managers can easily be removed from the queue sharing group. Shared queues removes concerns regarding draining queues and Sysplex Distributor allows apps to reconnect to remaining queue managers	Messages need to be drained from existing queues before a queue manager is removed from the cluster. App rebalancing ensures apps are spread over remaining queue managers
Recommended app types	Almost all apps can exploit queue sharing groups, but care needs to be taken regarding affinities	Many apps can exploit uniform clusters but care is required when apps have affinities or use message/correlation id to locate specific reply messages

Summary

- Why you need scalability and resilience
- Queue sharing groups
- Achieving consistent configuration
- Connectivity
- Application considerations
- Resilience
- Putting it all together
- A comparison with uniform clusters

Please submit your session feedback!

- Do it online at http://conferences.gse.org.uk/2019/feedback/JN
- This session is JN

1. What is your conference registration number?

This is the three digit number on the bottom of your delegate badge

2. Was the length of this presention correct?

1 to 4 = "Too Short" 5 = "OK" 6-9 = "Too Long"

1 2 3 4 5 6 7 8 9

3. Did this presention meet your requirements?

1 to 4 = "No" 5 = "OK" 6-9 = "Yes"

1 2 3 4 5 6 7 8 9

4. Was the session content what you expected?

1 to 4 = "No" 5 = "OK" 6-9 = "Yes"

1 2 3 4 5 6 7 8 9