**MQ Best Practices**

1. **Containerized approach for MQ:**

This section describes about how MQ can be designed for containerized deployment.

The adoption of containers in the context of MQ can be separated into three aspects:

**1. Foundational:** Many organizations have a traditional software on-premise

deployment, and are keen to cloud enable. For some organizations this may mean

moving to a public cloud provider, while others will take a more gradual approach

containerizing their on-premise deployment. Once running within containers, this

will facilitate component portability, as containers can be copied to any platform

that supports containers. The infrastructure team will streamline their operations

across software products, as all software can be handled in a standard approach,

providing operational consistency. As the footprint of new containers is low, this

increases the number of running containers a machine can handle, compared to

virtual machines, providing the infrastructure optimization. Finally the container

orchestration technology will assure the containers are available, which provides a

level of high availability and fine grained resilience.

**2. Decentralized:** New development architectures are being embraced within

organizations, which facilitate greater team agility, by decoupling dependencies

between teams. To succeed, this needs to include resources such as IBM MQ. Within

the organization there is normally an IBM MQ Administration team that owns and

maintains the messaging platform, however individual development teams are eager

for more control and access. To provide the additional flexibility, individual IBM MQ

instances can be provided to the development teams, so they can be modified

independently of the wider IBM MQ estate. Although the development teams will

have additional access, the IBM MQ instance will normally be integrated into the

wider messaging backbone, and may be administered by the central MQ team.

**3. Optimized:** Container orchestration technology provides the ability to provide

scalability and continuous availability. Although IBM MQ is infrequently the

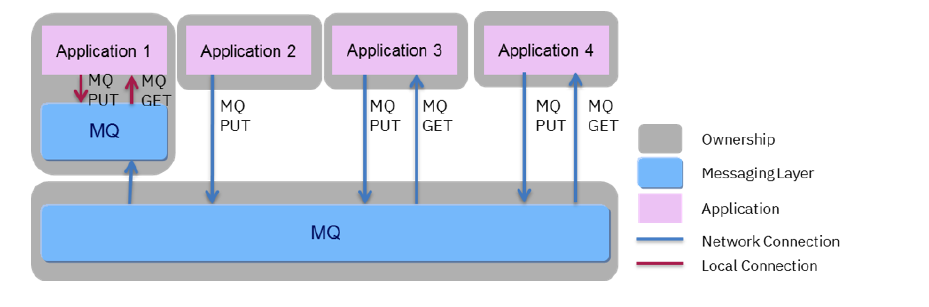
bottleneck, it is reassuring to know that IBM MQ can utilize the same container

scaling principles.

**Aspects: Foundational**

Let’s consider the Foundational aspect. To discuss the changes we will start with a high level

representation of a typical client’s IBM MQ estate as shown below:



An existing IBM MQ estate may include MQ instances (one or more Queue Managers),

logically associated with application teams (as shown with application 1), and also central

MQ instances controlled by a dedicated central MQ team. Interestingly, although the

application teams (application 1) have a MQ instances, the administration of these instances

may be completed by the same administrators who control the central MQ (due to

skills). This could be due to the particular requirements that the application team had for

MQ, requirement for additional isolation, performance, security, need for global transactions

or a drive to decouple for agility. Regardless many organizations will have a starting position

where they have multiple instances of MQ. In addition all or some of these instances may or

may not, be included in a MQ Cluster to assist with scalability, availability and routing.

The existing IBM MQ estate may also include different types of connectivity between the

applications and IBM MQ. IBM MQ supports two options for connecting to a queue

manager:

• **Local Connections**: applications running on the same system can connect using interprocess

communication. This is called a server binding.

• **Network Connections**: applications running on remote or the same system can

communicate using a network connection, using MQ channels support. This is called

a client binding.

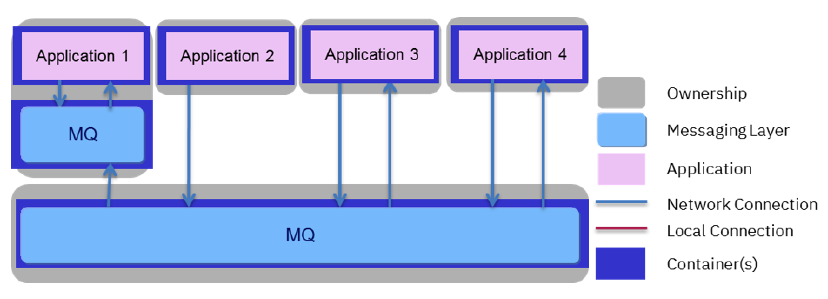
As shown in the diagram it is common for a MQ estate to have different mechanisms being

used. Often for simplicity (and sometime for transactional and performance reasons) the

server binding option was used.

**New Containerized Architecture**

The high level target architecture is shown below:



There are three changes that are important to highlight:

• **MQ containerization**: the IBM MQ installations have been placed into containers,

but the original high level topology remains the same. The diagram deliberately does

not show the detailed configuration of IBM MQ from an availability and scalability

view point, as we will discuss the considerations here in more detail later.

• **Application containerization**: the application logic has also been containerized. This

is deliberate and an important factor to be considered. If you are wanting to realise

operational consistency across software, then the scope of your containerization

project will need to be wider than simply MQ. If it is not then you will lose the

benefit of operational consistency.

• **Separation of MQ and Application logic**: as illustrated above, Application 1 and the

associated MQ are deployed into two separate containers. These two containers can

be deployed separately, or logically linked together at the time of deployment into a

Kubernetes Pod. A pod provides the ability to group together containers that were

previously relatively tightly coupled and run on the same machine. Within a pod you

share several resources, such as the IP address and port space, which can simplify

the move towards containers. Regardless of the deployment mechanism a network

connection between the application and MQ container is encourage, to allow

flexibility. This removes the ability to use local connections (Server Bindings) when

communicating with IBM MQ.

As the new architecture is adopted, and the applications and MQ are moved into separate

containers, there are several aspects to consider:

• Verify the performance characteristics – As you are moving to a containerization

strategy you will need to re-validate the performance characteristics of the entire

solution, and the MQ part is no different.

• Evaluate the security requirements for network connectivity – As a network

connection is being established, this may change the network security requirements for the communication, and also the security checks the MQ completes on accepting

a connection.

• Verify MQ is not acting as a global transactional coordinator – If MQ is acting as the

transactional coordinator, then server bindings would either need to be maintained,

or a new approach taken. It is important to highlight that you can still use MQ in a

transactional manner, and as part of a global transaction, but MQ would not be the

transactional coordinator.

• Separate lifecycles between the application and MQ – Previously people may have

assumed MQ and the Application were available if one of the two were available (for

instance if they were residing on the same machine), moving forward this already

risky assumption becomes less appropriate.

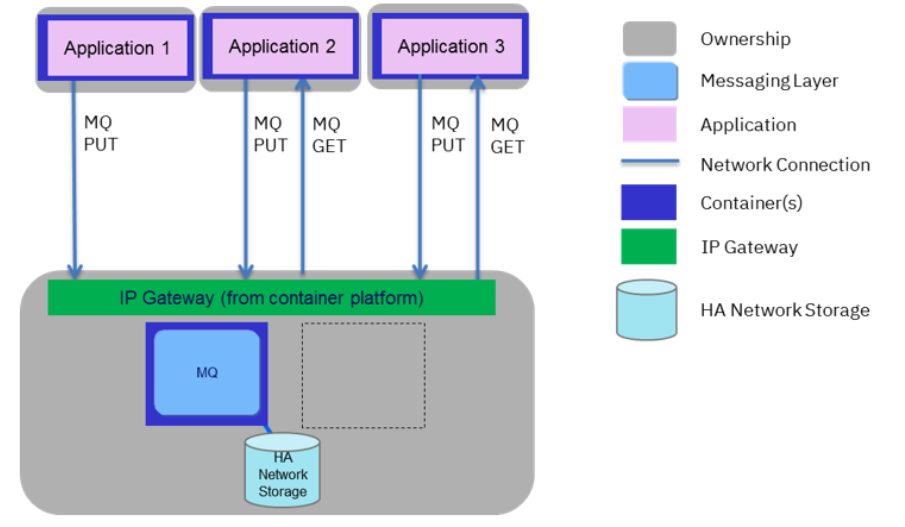
**Deep dive into the IBM MQ design**

Within a container environment clients often start with a single resilient queue manager as

the foundational building block for deployments. The single resilient queue manager

topology is built on the Stateful Container pattern mentioned previously and illustrated

below:



**Single Resilient Queue Manager**

This pattern has the advantage that a degree of high availability is provided automatically by

the container orchestration platform. The MQ container will be automatically restarted in the

case of a failure, or if the container is detected to be unhealthy. The container orchestration

platform will provide an IP Gateway to allow the routing of requests to the location of the

active container. The Queue Manager data and logs will be stored on shared storage, so any

running container can attach. Later we will discuss how this pattern can be used for

scalability and additional availability scenarios. Depending on your requirements for high

availability, your choice of container orchestration platform and its corresponding

configuration, the above approach may or may not be adequate for your needs.

Alternative approaches are available and a brief summary are discussed below:

• **Multi-instance queue manager**: this has an active and standby MQ Queue Manager,

where the standby instance is ready to take over in the case of a failure. The failover

logic is provided by the product, but the routing to the active instance is the

responsibility of the client or an external IP Gateway. In addition you need to

manage and appropriately license the active and standby instances. The failover

time of the multi-instance queue manager can be lower than the single resilient

queue manager, especially when we consider an entire node failure.

• **Replicated data queue manage**r: unfortunately, it is generally not suitable for use

with containers, due to the use of Linux kernel modules. In most cases, users running

containers will not have sufficient access to the host server to manage kernel

Modules.

**Aspect: Decentralized**

One driver for a containerization strategy, is to move towards a decentralized and modern

architecture, where application logic moves away from a monolithic application, and is

separated into application components (perhaps using microservices). This allows the

development of the components to be decoupled and provides greater agility. It also

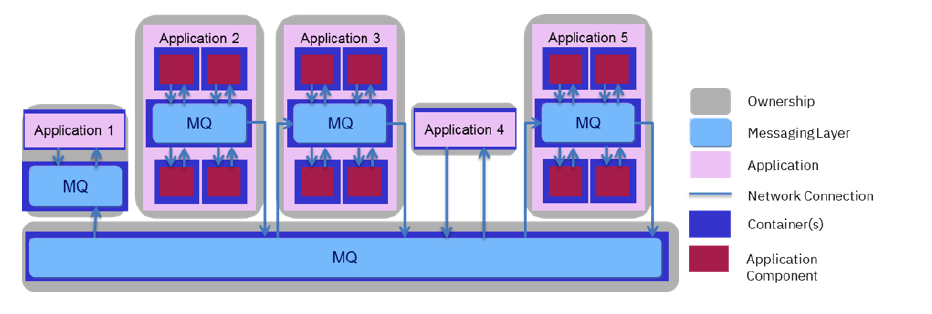
introduces a new challenge and a new requirement for decoupled communication between

components. For synchronous communication this would-be HTTP, while with asynchronous

this would be Messaging.

Applying this principle to our architecture, we start to introduce additional IBM MQ

instances within the application boundary, as illustrated below:



**Decentralized IBM MQ containerized Architecture**

Taking a closer look, we can see that a number of the applications have been modernized,

moving away from a monolithic application into a modern multi-component architecture

(applications 2 & 3). While other applications (applications 1 & 4) have remained as before,

this is likely to be the case in many organizations, where application modernization only

occurs after a cost benefit analysis. The new application (application 5) shows that we expect

new applications to be built using the modern architecture approach.

Many experienced IBM MQ clients may see similarities between this decentralized pattern

and those of a traditional architecture, where the Application and MQ are co-located on the

same machine.

**Aspect: Optimized**

The optimized aspect delivers two benefits, availability and scalability:

**Continuous Availability:** In messaging when required we separate out the availability to

PUT (send) a message, from GETting (retrieving) a message. Many use cases want to assure

an application is able to offload work (PUT a message), and therefore has a higher level of

availability compared to the ability to GET (retrieve) the message. We call this higher level

of availability for offloading work, continuous availability.

A **single resilient queue manager** will be automatically restarted in the event of a failure.

During the restart there will be a small period of time that the queue manager is not available

to receive new messages. This can range from a second or two to a few minutes under some

situations (for example where millions of messages have built up on queues). This affects the

overall availability of the messaging system so to achieve continuous availability we create at

least two single resilient queue managers, which both have queues that can store messages.

In this configuration it is expected that at least one queue manger is available at any one time

to receive inbound messages.

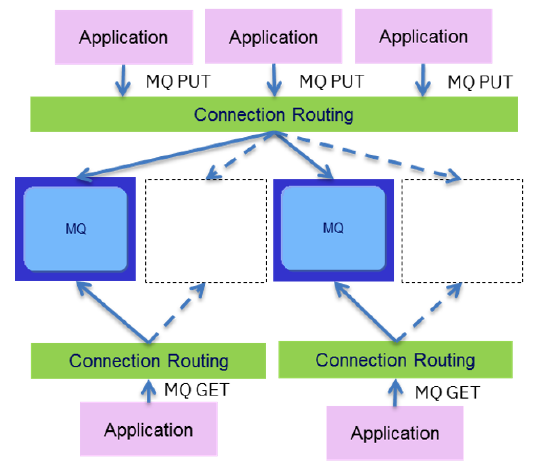
For applications storing messages, these connections will be distributed across the available

MQ instances. While connections for applications retrieving messages will be directly to a

MQ instance. This is due to the need to connect to the MQ instance hosting the individual

queue to retrieve messages. We will discuss later the options for this connection routing. This

is illustrated in the figure below:



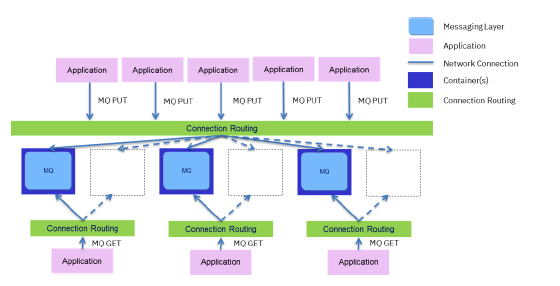
Multiple resilient Queue Managers

**Scalability:** is focused on the ability to scale the MQ instance horizontally and vertically. It is

assumed that vertical scaling has been completed, and horizontal scaling is required. This is

logically completed in a similar manner to the continuous availability part, as illustrated in

the figure below:



Scaling MQ Containers

While discussing the availability and scalability, we included the term Connection Routing,

this provides the ability for the application to have its MQ traffic directed to one of multiple

possible network destinations. We recommend for a production environment using a

capability called Client Connection Definition Tables (CCDT) when providing connection

routing across multiple possible MQ instances, while using the built in IP Gateway (such as a

Kubernetes Service) of the container orchestrator platform when determining the active

location of the container.

CCDT is a file that determines the connection information used by a client to connect to a

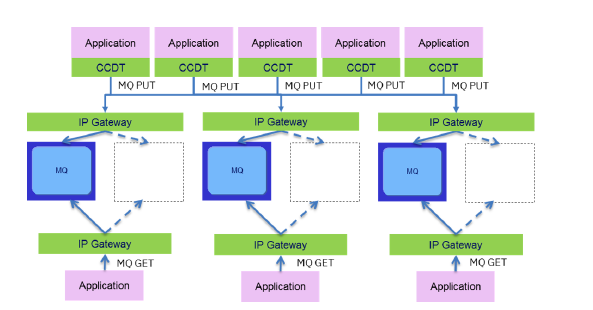
Queue Manager. A CCDT file can contain multiple entries, for a single logical connection,

allowing it to distribute traffic across several queue managers. The CCDT can be

configured so that channels in a group are either sequentially tried (for availability), or

randomly tried based on specified weightings (for workload balancing). The following figure

shows the connection routing using these recommendations:



Routing MQ traffic within containers

The CCDT could be replaced with a container orchestration load balancing component,

however this introduces a number of limitations on the applications which are documented

within the IBM MQ as a Service Redbook

(http://www.redbooks.ibm.com/redpapers/pdfs/redp5209.pdf) section 7.3.

1. **Scalability and Availability of MQ:**

## **Types of availability**

It is important to separately consider “message” and “service” availability. With IBM MQ on distributed platforms, a message is stored on exactly one queue manager, so if that queue manager becomes unavailable, you temporarily lose access to the messages it holds. To achieve high “message” availability, you need to be able to recover a queue manager as quickly as possible. You can achieve “service” availability by having multiple instances of queues for client applications to use, for example, by using multiple MQ cluster queues with the same name on different queue managers. Therefore, horizontal scaling is useful to improve scalability and service availability.

## **MQ availability technologies**

There are three main ways to make MQ highly available in cloud environments:

1. **Multi-instance queue manager**, which is an active-standby pair, using a shared, networked filesystem.
2. **Replicated data queue manager**, which replicates data under control of MQ
3. **Single resilient queue manager**, which offers a simple approach for HA in the cloud, using networked storage.

### **Multi-instance queue managers**

Multi-instance queue managers are the “traditional” way of thinking about high availability with MQ, and involves an “active” and a “standby” system. A multi-instance system consists of two servers where a queue manager could run, and a single shared filesystem. The queue manager’s data is held on the shared filesystem. The queue manager is only active on one server at a time, with the other server waiting in a standby mode. This system has a few key disadvantages:

* It’s sensitive to specific filesystems, because of its reliance on file locking to manage take-over by the standby.
* It requires additional resources and MQ license costs for the standby
* There are two components to manage

### **Replicated data queue managers**

A replicated data queue manager system consists of three servers where a queue manager could run, and an MQ-managed block storage device which is synchronously replicated to each server. The queue manager is only active on one server at a time, with the other two servers waiting in a standby mode, while receiving replicated data.

It is possible to use this system with containers, but it is generally not recommended, due to the use of Linux kernel modules. In most cases, users running containers will not have sufficient access to the host server to manage kernel modules.

### **Single resilient queue manager**

There are new ways of thinking about high availability (HA) in cloud environments. A “single resilient queue manager”, is where you have a single instance of a queue manager, and the cloud environment monitors it and replaces the VM or container as necessary. A queue manager can be thought of in two parts: the data stored on disk; and the running processes which allow access to the data. Any queue manager can be moved to a different virtual machine, or run in a different container, as long as it keeps the same data, and the same network address. Most cloud environments provide the ability to keep IP addresses (for example, [Kubernetes Services](https://kubernetes.io/docs/concepts/services-networking/service/)), and also offer highly available network-attached storage.

As with multi-instance queue managers, this system relies heavily on the availability of the storage system. MQ can’t be more available than the storage it is using. If you want to tolerate an outage of an entire availability zone, you need to use cloud storage which replicates to another zone. For example, [Amazon’s EFS](https://aws.amazon.com/efs/) replicates data in this way.

The recovery times for a “single resilient queue manager” can be similar to using a multi-instance queue manager. The “standby” process for a multi-instance queue manager is already running, but it’s not a full queue manager. The time taken to fully start the standby is similar to regular queue manager startup. The main difference with the “single resilient queue manager” system, is that the cloud provider potentially has to download the image to run the queue manager on a new worker node (this could be a virtual machine or container host). This download time is typically short for a small container image, but larger in the case of a VM image. The download time can be improved by pre-pulling images, as described in the following section.

Note that it is important to have a highly available cloud environment. For example, in Kubernetes, the master components such as the scheduler need to be highly available, as they are responsible for re-instating a failed queue manager.

## **Improving message availability**

You can improve message availability by reducing the recovery time as much as possible. For example:

* **Frequent health checks** — If you can detect a failure quickly, you can take action more quickly. Balance this with performance concerns, because health checks use resources.
* **Pre-pulling images** – when using the “single resilient queue manager” pattern, part of the recovery time in a cloud environment is the time taken to download the software image to the host server. In container terms this is often a “docker pull”. You can remove this delay by making sure the image is already available on an eligible hosts. Pre-pulling images is not a first-class feature in Kubernetes, but it can be accomplished. One way to pre-pull images, is as follows. Run a Daemon Set using your MQ image, on every worker node which is eligble for MQ deployment, but override the “entrypoint” of the container so that it doesn’t run a queue manager. This will cause Kubernetes to pull the MQ image to each node. By running in a Daemon Set, Kubernetes will restart the container immediately if it fails. If you also set the image pull policy to “Always”, then Kubernetes will pull the image again every time it needs to run the container. If you make the container entrypoint (say) sleep for thirty minutes, then the MQ image will be pulled down again (if necessary) every thirty minutes.
* **Quiesce messaging load before a planned outage** – in the case of a planned outage, you can redirect traffic away in advance, to reduce the number of messages which will be held on a particular queue manager. For example, you can do this by suspending the queue manager from an MQ cluster, or by lowering the priority of the channels or queues in the cluster. This has two benefits: Messages held on queues during an outage cannot be processed during that time, so it’s better to have them processed through alternative queue mangers; The time taken to restart a queue manager can be affected when many messages are stored or being processed by a queue manager, so storing fewer messages allows a quicker recovery.
* **Redirect existing messages before a planned outage** – in the case of a planned outage, you could attempt to drain a queue manager which has already been quiesced. One way of doing this, with MQ clusters, is to use the “[amqsclm](https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.0.0/com.ibm.mq.dev.doc/q024620_.htm)” sample to re-send all messages back into the cluster. This will re-distribute the messages to other queue managers.

# **Scalability**

The normal techniques for scaling MQ are not changed when you move to the cloud, but you do get more options. Standard, well proven techniques, such as using MQ clusters to workload balance messages across multiple queue managers, continue to work as normal. Running in containers also makes it easier to “vertically” scale, by changing the resources used by a container, and potentially re-scheduling to another worker node if necessary. However, you may be interested in using your cloud platform to “horizontally” scale MQ, by using multiple MQ servers with similar or identical configuration. This has potential benefits both to scalability, and for availability (as a so-called “active-active” availability solution). Horizontal scaling is often used in cloud environments, particularly for stateless workloads, but requires careful architecture when applied to stateful workloads. This is particularly true when you’re sending high value messages using MQ.

## **Dynamic versus static scaling**

It is easier to deploy a horizontally scaled set of MQ servers, which is fixed in sized, than to dynamically change the number of MQ servers. Scaling IBM MQ up by adding more queue managers is fairly straightforward, by using either an IBM MQ cluster of queue managers, or a load-balanced set of identical queue managers, depending on what messaging patterns you use. Scaling down is more complex because you almost always want to remove a queue manager in a controlled manner to be sure no messages remain on the queue manager at the time it is deleted.

Scaling down in many cloud environments is especially difficult, because you need to distinguish between a server which is shutting down but will be restarted elsewhere, and a server which is shutting down because it is being deleted. This isn’t a problem with stateless workloads, but is usually a problem with stateful workloads like MQ.

In addition, you need to consider how use of dynamic scaling can affect MQ clusters. If you delete a queue manager that is a member of an MQ cluster without first performing the recommended steps to remove that queue manager, information about that queue manager will be held by the MQ cluster for up to 90 days, but shouldn’t have any adverse effects. However, the same is not true for MQ objects such as cluster queues, which can cause problems if they are not removed carefully. For example, messages may stay on cluster transmission queues waiting for deleted queue managers to return

In summary, it is not recommended to reduce the size of the “set” after initial deployment, which will cause the deletion of queue managers, and can result in loss of messages. In Kubernetes terms, this means that you should not change the number of replicas in a [Stateful Set](https://www.google.com/url?q=https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/&sa=U&ved=0ahUKEwj8n-PQhOfaAhVDFiwKHezYD6AQFggEMAA&client=internal-uds-cse&cx=013288817511911618469:elfqqbqldzg&usg=AOvVaw3FOaOg1fXI7HIS1WkDKKdr) after initial deployment. If you want to change the number of replicas, you must have a proven process in place to ensure that new replicas are correctly integrated into your solution.

## **Message ordering**

If you use horizontal scaling, messages can be processed concurrently, which means messages might be received out of sequence. IBM MQ provides features to allow groups of messages to be handled in small ordered batches, when being sent between queue managers. You can also choose to manage this yourself in your application.

## **Client versus server bindings**

IBM MQ client applications should generally be scaled separately from IBM MQ servers, which means running them in separate virtual machines or containers. Having single-purpose containers is also considered a best practice, and brings the maximum benefits of containerization, including dependency isolation, resource isolation, security isolation, and ease of access to logs. For MQ, this means that cloud topologies typically use IBM MQ client connections to communicate between applications and queue managers.

There are some cases where server bindings may be required, though, for example applications that are using a queue manager as the transaction coordinator, of global (XA) transactions are required to connect using server bindings. In addition, there are a number of cases where server bindings may be desirable, for improved performance, or simply because applications have been developed in that way.

There are several IBM-provided solutions which require server bindings, either for the reasons given above or due to a need for access to locally held MQ information:

* IBM Integration Bus uses server bindings to provide global transaction support on distributed platforms, as well as for [certain nodes](https://www.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/bb28660_.htm).
* IBM Application Performance Monitoring agent for MQ uses server bindings
* IBM MQ web server uses server bindings to connect to the queue manager to administer.

You can use any of these technologies from within the same container. If you want to use them from a separate container (for example, for increased isolation, or easier error log management), then there are some technical restrictions:

* A shared IPC namespace (shared memory), because in server bindings mode, MQ clients connect to the server using shared memory.
* A shared process (PID) namespace, because in server bindings mode, the MQ server monitors the process ID (pid) of the client, to determine whether it is still running.

## **Load balancing**

In order to horizontally scale, you need more than one queue manager which is able to offer the same messaging service (such as a queue). Once you have this, you need a way to assign clients to each queue manager in a balanced way. The simplest load balancing is *connection load balancing*, where a server (queue manager) is chosen at connection time. MQ also provides more sophisticated load balancing, on a per-message basis, through the use of MQ clusters.

### **Discovering and connecting to a queue manager**

How do MQ clients locate a queue manager to use? There are three main approaches for an initial connection:

* **Manual/fixed** – Clients can be configured at deployment time to use a specific queue manager for the lifetime of that deployment. For example, this might be done by having a queue manager and client application in the same Kubernetes Pod. It could also be as simple as supplying an environment variable, Kubernetes ConfigMap, or via VCAP\_SERVICES in CloudFoundry.
* **Client Channel Definition Table (CCDT)** – Clients can be configured to retrieve queue manager information using an MQ [CCDT](https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.0.0/com.ibm.mq.con.doc/q016730_.htm). CCDTs offer a rich set of connection information, including channel identifiers, TLS cipher specs, maximum message size, and more. The CCDT needs to be updated whenever new queue managers are created, or old ones deleted. The CCDT can be provided as a file (for example, using a Kubernetes ConfigMap), or accessible on the network via HTTP or FTP (for example, located using a Kubernetes Service). Note that if you use a “[queue manager group](https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.0.0/com.ibm.mq.dev.doc/q027490_.htm)” feature of CCDTs, and your client application connects using a “\*” in the queue manager identifier, this is similar to using an L4 load balancer, except it avoids the problems with JMS.
* **L4 load balancer** – Clients can be configured to use a network load balancer, which selects the destination for TCP/IP connections. This is often known as an “OSI layer 4” or “L4” load balancer, and is commonly offered in cloud environments. For example, a typical Kubernetes Service will spray TCP/IP connections to one or more servers, in round-robin order. OSI layer 7 load balancing is discussed later, as it doesn’t apply to the initial connection.

There are a number of restrictions you face when using an L4 load balancer (or a CCDT with a “\*” in the queue manage identifier), which need to be carefully considered before you use this approach:

* **Manual/fixed use of multiple channels** – In most cloud environments, addressing information for services consists of simply a TCP/IP address and port. MQ provides additional flexibility: behind a single TCP/IP address and port, you have one or more channel definitions. This is typically used to configure security settings for certain connections, such as TLS ciphers and certificates. The CCDT mechanism allows you to define channel information as part of the connection information, but if you use an L4 load balancer, you need to specify channels in a manual/fixed way (for example, with an environment variable at deployment time). This is fine if you have a small number of different channel configurations to deal with, but becomes cumbersome if you configure lots of different channels.
* **Avoid JMS APIs when using a L4 load balancer** – MQ’s Java Messaging Service (JMS) implementation uses multiple TCP/IP connections, which could be incorrectly distributed across different queue managers. Using a CCDT resolves this problem.
* **Restricted use of request/response pattern** – A common messaging pattern is where a client sends a message and waits for a response before continuing. This creates state in the application, and can result in messages which don’t get consumed if/when the client fails and gets restarted (which can be common in cloud environments). If you want to use request/response messaging, then you should either persist any information required to process a response (for example, in a database), or place all information needed to process the response in the message itself.
* **Avoid server affinity** – See the next section

### **Server affinity**

You have an architecture with “server affinity” if you have any situation where if a client were disconnected for some reason, that it would need to reconnect to the same queue manager. Some MQ features which can create server affinity include:

* Global (XA) transactions
* Durable subscriptions
* Specifying a queue manager name (!) in the client

See Chapter 7 of the [IBM MQ as a Service Redpaper](https://www.redbooks.ibm.com/redpapers/pdfs/redp5209.pdf) for more details.

If you use features which require server affinity, and you want to use an L4 load balancer or a CCDT with a “\*”, then the client is responsible for persisting the information it needs to reconnect to the correct queue manager. For example, the client might query the queue manager name, and write it to a database or key-value store.

### **Heterogeneous or homogeneous sets of queue managers**

Many cloud environments make it easy to “horizontally scale” multiple replicas of a container or VM. This gives you a “homogeneous set” of queue managers, with an identical configuration. This typically makes it easy for the cloud to provide L4 load balancing, and also allows for cloud-managed rolling upgrades of your queue managers. For example, Kubernetes offers a Kubernetes Stateful Set, and will manage upgrading each replica (queue manager) in turn.

### **MQ clustering — L7 load balancing**

MQ clusters offer more sophisticated load balancing, by choosing a cluster queue when it is opened, on a per-message basis, or when creating a message group. This can happen within a single TCP/IP connection. This load balancing is part of the MQ clustering feature, and requires the queue managers to be interconnected. It is often used in conjunction with one or more “gateway” queue managers, which act as a front-end for routing messages to back-end queue managers.

This load balancing is based on many things, but is primarily based on a set of weightings and priorities, and whether queue managers are known to be available or not.

1. **Deploy MQ instance in CP4I using Operator:**

IBM® MQ 9.1.5 and later are deployed to Red Hat® OpenShift® using the IBM MQ Operator.

Refer the link <https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.2.0/com.ibm.mq.ctr.doc/deploy_ctr.htm?view=embed> for deployment of MQ using operator in Openshift cluster