Mission:Messaging: WebSphere MQ cluster design and operation

Skill Level: Intermediate

T.Rob Wyatt
Senior Managing Consultant
IBM

09 Dec 2009

IBM® WebSphere® MQ clusters provide ease of administration and workload distribution, plus they play an important part in making messaging solutions highly available. But the cluster must be healthy in order to realize the full benefits. This installment of Mission:Messaging provides some tips on building a healthy cluster and how to keep it that way.

Cluster health check

As a consultant who specializes in IBM WebSphere MQ, I’m often called on to troubleshoot production outages on an emergency basis. In many cases, there is an unhealthy cluster at the root of the problem. With experience, I became able to recognize recurring patterns in these broken systems and devised a set of recommendations for avoiding these problems and keeping the cluster healthy.

Of course, WebSphere MQ cluster design and management is still a mix of art and science. There is no single path to success, and I’ve been to many shops that successfully maintain their cluster despite breaking some of my rules. But these are often narrow and specialized cases where success depends in large part on local considerations -- which could mean the presence of mitigating factors, like advanced monitoring or advanced administrator skills, or the absence of complicating factors, like unusually low incidents of system change or a very small cluster.

But over time, all systems change and dependencies on the local environment can make the system brittle in the face of such change. As a consultant, the methods I gravitate toward are the ones with the broadest applicability. When it comes to
WebSphere MQ clusters, the methods I propose here are the ones that I have found to be applicable in nearly all cases.

What a cluster is -- and what it is not

Stripped to the essentials, a WebSphere MQ cluster is just a collection of queue managers that share a common namespace, delegation of certain administrative functions, and dense any-to-any connectivity. All of the benefits of MQ clustering derive from one or more of these three things. The ease of administration is a direct result of having delegated a subset of administrative tasks to an internal queue manager process. The ability to have multiple instances of a queue is an artifact of the common namespace combined with dense connectivity. This in turn enables workload distribution and dynamic routing.

But a WebSphere MQ cluster never rises to be more than a collection of individual queue managers, nor does it try to. There is no concept of "connecting to the cluster" with WebSphere MQ. Connections are always to a specific node within the cluster. Similarly, the functionality available to the application remains the same regardless of whether the queue manager participates in a cluster or not. Just as with point-to-point messaging, an application connected to a clustered queue manager can still get messages only from local queues, and can still put to any queue whose name can be resolved locally, regardless of where that destination queue is hosted.

This can sometimes be confusing because of other uses of the term "cluster." For example, a hardware cluster is a logical entity that is composed of multiple physical servers. Things connecting to the hardware cluster see the logical entity and not the physical components underneath. Similarly, many application servers, databases, and other software platforms can be configured in such a way that a collection of them appears to be a single entity. This usage is so common that for many people the word cluster implies a single logical entity comprised of many physical components acting as one.

No wonder, then, that WebSphere MQ clusters are often misunderstood. When first confronted with the term **WebSphere MQ cluster**, many people intuitively picture a large virtual queue manager comprised of multiple physical queue managers all acting as one. Similarly, the term **clustered queue** often evokes images of multiple queue instances acting collectively as a single logical queue to which an application can connect and then put or get messages. But this outside-looking-in paradigm is not at all how WebSphere MQ works, and that misunderstanding can result in poor design or operation choices.

WebSphere MQ clustering is not about how applications talk to the queue manager. It is about how queue managers talk amongst themselves.
Benefits of a WebSphere MQ cluster

When clustering was introduced to WebSphere MQ, the retronym point-to-point network was used to differentiate between a cluster and the classic MQ network topology. The primary distinction between them is that in a point-to-point network, queued messages travel from a point of origin to a single destination. Routing in this network is determined per-queue and embedded in the network definition at build time.

By contrast, messages in a cluster travel from a point of origin to one of several possible destinations. In this case, the selection of the destination and the routing both occur per-message at run time. The result is that a clustered WebSphere MQ network is flexible, dynamic, and resilient.

These features make WebSphere MQ clusters a natural fit for service-oriented architectures (SOA). But, more importantly, the WebSphere MQ cluster does not displace the point-to-point network. When a point-to-point interface is required (for example, a traditional batch interface), it is easily implemented within the WebSphere MQ cluster. This ability for point-to-point and clustered topologies to coexist provides a smooth transition path for an SOA implementation.

However, the cluster needs to be healthy in order to fully realize the benefits. The remainder of this article focuses on the methods I have picked up over the years to build a healthy cluster and make sure it stays that way.

My recommendations for repositories

Before I offer recommendations, here are some thoughts to put us on the same page.

Any clustered queue manager can advertise some of its queues or topics to the cluster and have others that are known only locally. Of the objects that are advertised to the cluster, certain state information must be tracked so that the cluster can properly route messages. In addition to queues and topics, the cluster also tracks the state of the channels and queue managers in the cluster. The names and state of all these cluster objects is the cluster metadata.

Two queue managers in the cluster are designated to maintain a complete and up-to-date copy of all metadata for the cluster. In a healthy cluster, each of these nodes contains a mirror copy of all information relating to the state of the cluster. Since they have the complete set, these nodes are called full repositories.

All of the other cluster members maintain a subset of cluster metadata and thus are referred to as partial repositories. Each of these nodes maintains the real-time
state of each cluster object that it hosts. Any change to these cluster objects is
immediately published to the two full repositories. In addition, applications connected
to the partial repository queue managers need to put messages to clustered queues
which might be hosted on some other node. The partial repository subscribes to
updates about these remote objects and maintains a local copy of the last known
state of each.

This terminology of full and partial repositories is somewhat unwieldy and a little
confusing. In common usage, "full repository" is often shortened to "repository" and
all other nodes are simply queue managers participating in the cluster. This is
consistent with the WebSphere MQ command set, which uses REPOS as a
parameter.

For example, you can issue a REFRESH CLUSTER(*) REPOS(YES) or you can
ALTER QMGR REPOS(cluster name). This is the convention that I will use
throughout the rest of this article. When I say "repository" I'm referring to one of
the two queue managers designated as a full repository. Everything else is just a cluster
member.

There are my recommendations:

- **The magic number is two**
  Earlier I said that two queue managers are designated as repositories.
  This is not mandatory but strongly recommended. Clusters can function
  with a single repository and even limp along without any repository for
  short periods of time. But using two repositories provides higher
  availability. Eventually, it will be necessary to apply maintenance to the
  repositories, and having two enables the cluster to function normally while
  one repository is offline.

  But if two are good, isn't three better? Not necessarily. Each cluster
  member will publish two updates when one of its objects changes state. If
  there is only one repository, it receives both updates. If there are two
  repositories, each receives one of the two update messages. In both of
  these cases, all cluster nodes report changes to all repositories, and the
  topology provides built-in assurances that the state will be maintained
  consistently across the cluster.

  In the case where there are three or more repositories, each cluster
  member updates only two. Achieving consistency across all of the
  repositories depends on them successfully replicating updates amongst
  themselves. With two repositories, 100% of the metadata is delivered
directly from cluster members. With three repositories, 1/3 of the
metadata on average is delivered through replication. With four
repositories that number is ½, and with five repositories the proportion
rises to 3/5. The more replication that exists, the greater the chance of
replication errors. Extrapolate far enough and that chance of error becomes a virtual certainty.

This is a case where more is definitely not better. All of my cluster designs are based on exactly two repositories, no more and no less. There are very rare exceptions to this rule. For example, in the case of transcontinental clusters I will sometimes use local repository pairs to overcome latency. But this does not mean using four repositories is good, only that it is less bad than running two repositories with extreme latency.

- **Location, location, location!**
  A common practice when designing applications is to lay out the topology as it will exist in production, and the disaster recovery site becomes an exact duplicate. This is often termed active/passive. Although this worked well in the point-to-point world, where everything tended to fail over at once, in the SOA world, services are widely distributed and must be able to fail over individually. This is why the active/active model is increasingly common.

  Although the WebSphere MQ cluster is infrastructure, it is also an SOA application at heart. Cluster members request services from the repositories, which in turn provide run time inquiry and state management. If we approached this as a traditional application, there would be two live repositories in production and two cold standby repositories in the disaster recovery site. The problem with this is that the disaster recovery site needs to hold the current state of the cluster in order to be immediately useful during a failover. This is especially true when the failover includes some subset of the overall production environment.

  Running a live pair of repositories in production and another live pair in disaster recovery solves the problem of currency, but it brings us back to the replication problem. The solution to both of these issues is to run one repository live in production and a second one live in the disaster recovery site.

- **Dedicated servers are best**
  This is the recommendation I tend to take the most heat on because I always specify dedicated servers for the repositories, even for a relatively small cluster. If I have to choose, I would rather put the repositories on low-end, surplus, standalone commodity servers than to put them on highly available, expensive hardware co-located with applications. My position on this is quite opposed to the prevailing advice and I feel it is necessary to go into some detail to support it, so please bear with me.
My fallback position, if I have to put the repositories on servers with application queue managers, is to use a separate, dedicated queue manager. The use cases from best to worst then are:

- Repositories hosted on dedicated servers.
- Repositories hosted on dedicated queue managers but sharing a node with one or more application queue managers.
- Co-locating repositories on application queue managers.

My exception to the dedicated server rule is when the cluster is so small and the applications are forgiving enough that problems can be resolved by wiping the cluster out and starting over. If I have four to six application queue managers of which two are hosting the repositories, I usually maintain a set of scripts that will tear down the cluster and then rebuild it from scratch in a matter of minutes. It is still necessary to take an application outage across the entire cluster, but at least it’s a short outage. The problem is that this does not scale well. Add just a few more queue managers and the ability to tear down and rebuild the cluster from scratch reliably and quickly begins to diminish. Because clusters have a tendency to grow over time, I view this not only as a compromise but also as temporary solution at best.

The problem with co-locating repositories with application queue managers is that it creates dependencies at several levels. One of these is with software versions on the underlying host. When upgrading to a new version of WebSphere MQ, the recommended practice is to upgrade the repositories to the new version first. Although this is not strictly required, using back-level repositories in a mixed cluster can prevent the use of advanced features. In the worst case, this can result in cluster outages. But when the repository is co-located with an application queue manager, the ability to upgrade is dependent on the resources available for the application team to test and certify under the new version. This presents a problem when even one dependent application is present, but more often than not there are several applications sharing the queue manager.

However, upgrades are really the optimistic case here. The worse situation is when there is a cluster problem that requires that a fix pack or patch be applied. In one case, I had a client go for months with a broken cluster and nightly outages because they were unable to apply a fix pack. The decision made five years prior to host repositories on application queue managers seemed like a good idea at the time. The financial impact of their extended outages would easily have offset the cost of dedicated servers, licenses, and ten years of hosting.
But the dependencies even extend to WebSphere MQ's administrative interface, and this is one reason why I'll use a dedicated queue manager when I am unable to obtain a dedicated host. The command used to repair a repository is RESET. The command to repair a cluster member queue manager is REFRESH. These are mutually exclusive. That is, you can't run REFRESH CLUSTER(*) REPOS(YES) on a full repository. In order to run the REFRESH command on a repository, it must first be demoted to a non-repository. It is often necessary at this point to go to the remaining repository and RESET the old repository out of the cluster. When the REFRESH command is run on the old repository, it "forgets" everything it knows about the cluster and rejoins as a regular cluster member. At this point, it can be promoted back to a repository.

Between the RESET and REFRESH commands, all other nodes in the cluster lose any information they had about the repository being repaired. This would not be a problem if the repository does not host any application objects, but since it does there is application impact in the cluster. Similarly, the local applications lose all knowledge of the rest of the cluster during the REFRESH process. Promoting the node to a repository again restores knowledge of all the cluster queues, but prior to that, applications can experience errors because the queues they are using suddenly fail to resolve.

There are also dependencies at the operational level. Messages in the cluster are delivered directly from the sending node to the receiving node. There are no cases in a healthy cluster where messages inside the cluster hop through one or more intermediate nodes before arriving at their destination. Therefore, hosting the repositories on dedicated queue managers results in a topology where application data and cluster data never traverse the same channels. Depending on the size of the cluster and the nature of the applications, each can produce a volume of messages sufficient to negatively impact the other. The use of dedicated repository queue managers reduces the volatility attributable to these interactions.

Of course, there are many other interactions when applications and the cluster repository compete for limited resources. Although the cluster repository is a native MQ process, it is still subject to resource limitations like any other application. For example, an application that consumes many channel connections can leave the repository process bumping into the MAXCHANNELS limitation. Starved for connections, it will be unable to publish cluster updates in a timely fashion. Similarly, the repository can compete with applications for transactions under syncpoint, transaction log space, MAXDEPTH of the cluster transmit queue, and many other resources. In all of these cases, the cluster and the applications are in direct competition when they are hosted on the same queue manager.
Similarly, server level contention can arise for disk space, memory, CPU, and other system resources when repositories and application queue managers share a server.

The conventional wisdom says to place the repositories on the most reliable hardware available in the WebSphere MQ cluster. Although I prefer to use a decent server, I will prioritize isolation of the repositories over resilient hardware every time. When you are trying to coordinate a fix pack across a dozen different applications, hardware reliability is the least of your problems.

My recommendations for cluster members

- **Only one explicitly defined CLUSSDR**
  Although there are two repositories in the cluster, it is not necessary to explicitly define cluster sender channels to both of them in order to join the cluster. When the first CLUSSDR channel is defined and the queue manager successfully joins the cluster, it will immediately be informed of the other repository and build channels to communicate with it. A second explicit CLUSSDR definition is unnecessary.

  If the second CLUSSDR channel is not needed, the question then is whether there is any negative impact in defining it. There is.

  Cluster members always publish changes to two repositories. The decision of which two depends in part on the CLUSSDR channels that have been defined. Since an explicitly defined CLUSSDR is weighted heavier, publications will always prefer these over auto-defined channels. If there are two explicitly defined CLUSSDR channels, publications will always use these two channels, regardless of whether the queue managers at the other end of the channels are available, and regardless of the availability of any alternative destinations.

  In the normal course of operations, this is not necessarily a bad thing. But if you ever need to host a new repository, whether temporarily or as a permanent migration, having two explicitly defined CLUSSDR channels will come back to haunt you. The only way to get a cluster member to see the new repository will be to delete one of the explicit CLUSSDR definitions, possibly accompanied by a REFRESH CLUSTER(cluster name) REPOS(YES) command.

  Using more than two explicitly defined CLUSSDR channels is even worse. In that case, it is impossible to tell which two channels will be published to and, in the event both repositories selected are offline, the
cluster member’s publications will sit stranded on the cluster transmit queue.

The most reliable method is to use a single explicitly defined CLUSSDR. This enables the queue manager to find the second repository using normal cluster workload management algorithms. For example, you could bring a third repository online and then suspend one of the primary repositories prior to performing maintenance. The cluster members would publish once to whichever repository their explicit CLUSSDR points to -- whether it was online or not -- and once to one of the two remaining repositories, depending on which one was available.

My recommendations for all nodes

• **Unique CLUSRCVR names per cluster**
  The product manuals illustrate clustering using channel names like TO.<QMgr name>. For example, a queue manager named QM1 would create a cluster receiver channel called TO.QM1 and every other cluster member would use that channel to connect to it. The concern with this is that the naming convention tends to result in the same channel being reused across overlapping clusters, and this could lead to problems.

  To understand why this could be an issue, you first need to understand why an overlapping cluster would be used. There are many reasons, but what they tend to have in common is they provide granularity of management of some aspect of the cluster. But the CLUSRCVR is inextricably bound to cluster management operations. If you REFRESH the cluster, the CLUSRCVR must be stopped for all clusters it participates in. Because the naming convention encourages you to reuse cluster channels across overlapping clusters, it robs you of the very functionality the overlapping cluster was supposed to provide.

  The naming convention I try to use instead is <Cluster name>.<QMgr name> because it enforces channels that are unique per cluster. With this naming convention, it is possible to perform maintenance on one cluster without impacting any other clusters in which the queue manager participates.

• **MCAUSER that restricts administrative access**
  Any inbound channel with a blank MCAUSER value will permit whatever is connecting to administer the local queue manager. In the case of a CLUSRCVR channel, a blank MCAUSER means that any other queue manager in the cluster can administer the local queue manager. This is
usually not what you want since a legitimate administrator will usually connect directly or administer the queue manager from the command line at the underlying host.

Setting the MCAUSER of the CLUSRCVR channel to a service account forces it to use the privileges of that account. You can use setmqaut to specify which queues the channel can put messages on. A simple configuration will enable access to all queues except SYSTEM.ADMIN.COMMAND.QUEUE as well as any initiation queues, transmit queues, and most SYSTEM.* queues. A more secure option is to deny access to all queues except those specifically granted, and then limit these to a handful of authorized destinations.

In either case, the channel must be able to place messages onto SYSTEM.CLUSTER.COMMAND.QUEUE in order for the queue manager to be a member of the cluster.

Of course, in order for this to be effective, all inbound channels would need to be secured against administrative intrusions. By "inbound channel" I mean every RCVR, RQSTR, CLUSRCVR or SVRCONN channel, including the ones named SYSTEM.DEF.* and SYSTEM.AUTO.*, even on queue managers where channel auto-definition is disabled.

Summary

There is a lot more to clustering than what I've discussed in this article, but these are the things I consider to be the most important. Most of what I have recommended here are policy or process changes. The one recommendation that requires funding is to host repositories on dedicated servers and I'm often challenged on that point, especially when the cluster consists of a small handful of nodes.

The most common question is, "Can't I wait until the cluster is bigger and then buy the new nodes?" Of course you can. However, if it is at all possible to implement the target topology immediately -- while the cluster is still small -- there are many benefits, one of which is never having to decide what criteria are used to determine when the right time to upgrade will be. Unless there is impact, any decision is somewhat arbitrary. After all, you've lived with it this long, right? The result in most cases is that the decision is put off until there are problems and then change is no longer optional. When I hear the question, it translates in my mind to "Can I defer this change until there is guaranteed impact?"

I can't guarantee that you will never have problems with your cluster, but if you adopt these recommendations, I'm confident that you will have fewer problems than you
would otherwise, and any problems that do arise will be easier to resolve.
Resources

Learn

• Education: Join webcast on WebSphere MQ Queue Manager Clusters
• IBM developerWorks WebSphere
• Podcast: The Deep Queue
• Author's Web page: T-Rob.net

Get products and technologies

• IZ61338: WEBSPHERE MQ (WMQ V7.0.1.0) CLUSRCVR CHANNELS REPORT

Discuss

• IBMers' Blog on Messaging
• The Vienna WebSphere MQ List server
• MQSeries.net
• developerWorks WebSphere MQ forum

About the author

T.Rob Wyatt

T.Rob Wyatt is a Senior Managing Consultant with IBM Software Services for WebSphere who assists customers with the administration, architecture, and security of WebSphere MQ. Recently he has focused on WebSphere MQ security, publishing in the IBM WebSphere Developer Technical Journal and presenting at the IMPACT and European Transaction and Messaging conferences. T.Rob also hosts The Deep Queue, a monthly WebSphere MQ security podcast.