Walkthrough: Auto application rebalancing using the Uniform Cluster pattern

LaurenceBonney
Published on 21/03/2019 / *Updated on 29/03/2019*

What is a Uniform Cluster?

To quote the MQ documentation directly:

IBM MQ Version 9.1.2 introduces uniform clusters, which are a specific pattern of an IBM MQ cluster that provides a highly available and horizontally scaled small collection of queue managers, that an application can interact with as a single group.

The high level aspects of the Uniform Cluster pattern in 9.1.2 are covered in the building scalable fault tolerant IBM MQ systems blog

What this blog will take you through

- Enabling pre-configured MQ clustered queue manager to participate in application rebalancing
- Configuring and running a shipped reconnectable C sample app to show automatic application rebalancing in action
- Restarting a queue manager with active applications to highlight automatic rebalancing back to a restarted queue manager
- Using a more advanced CCDT to use queue manager grouping and client weighting to load balance application connections

Please note: within this blog you will see references to applications being rebalanced, to be more precise it is the MQ connections being rebalanced. However in the context of the example application we use in this blog, each application has only one MQ connection so are used interchangeably.

New capabilities available in MQ 9.1.2 being used

- Uniform Cluster setting on the queue manager
- JSON format CCDT
- Application naming capability

So, before you begin, you need to get hold of MQ 9.1.2 CD.

Step 1: Queue manager configuration

Firstly we'll go through setting up an existing 2 queue manager MQ cluster into a Uniform Cluster. I've assumed you already know how to create a basic pair of MQ clustered queue managers with running listeners for your applications to connect via. If you dont, its really simple to do through the MQ Explorer interface, or if you're feeling more adventurous - via the command line. Check out the MQ Knowledge center for more information.

- Define an MQ Cluster of 2 qmgrs QM1 & QM2
- Add the UniformClusterName qm.ini tuning parameter to each queue manager: `TuningParameters: UniformClusterName=<MQ_CLUSTER_NAME>` Where <MQ_CLUSTER_NAME> is the name of your MQ cluster.
- Restart the queue manager and check for the "AMQ9883I: Joining Uniform Cluster" message in each queue manager's AMQERR log
  If you got this far you should now have a pair of queue managers that are ready to participate in application rebalancing.

Step 2: Showing application rebalancing in action between two active queue managers

Next we'll show how using a Uniform Cluster, applications can be automatically and dynamically rebalanced across active queue managers. To do this we'll be using the reconnectable amqsghac sample. We're using this sample because it is a client connected app that connects using the CNO option MQCNO_RECONNECT. In terms of Uniform Clusters this CNO option should be treated as the 'magic switch' to tell the queue manager that this application instance is willing to be rebalanced to another queue manager in the cluster. Before we get ahead of ourselves, to make best use of application rebalancing the amqsghac sample will need to use a CCDT which defines routes to all queue managers in the Uniform Cluster.

Generating a CCDT

Generating a CCDT for your application instances to use can be done either using the traditional binary format via runmqsc:
```
$ /opt/mqm/bin/runmqsc -n: DEFINE CHANNEL('QM1.APP.CHL') CHLTYPE(CLNTCONN) - TRPTYPE(TCP) CONNAME('localhost(1414)') QMNAME(QM1) REPLACE DEFINE CHANNEL('QM2.APP.CHL') CHLTYPE(CLNTCONN) - TRPTYPE(TCP) CONNAME('localhost(1415)') QMNAME(QM2) REPLACE
```
Or using a new JSON format CCDT introduced in MQ 9.1.2: `{ "channel": [{ "name": "QM1.APP.CHL", "clientConnection": { "connection": [{ "host": "localhost", "port": 1414 }], "queueManager": "QM1" }, "type": "clientConnection" }, { "name": "QM2.APP.CHL", "clientConnection": { "connection": [{ "host": "localhost", "port": 1415 }], "queueManager": "QM2" }, "type": "clientConnection" }] }`

Note: in both the binary and JSON format CCDT examples I've assumed you have created the associated SVRCONN channels of the same names and have listeners running on the referenced ports

In this initial example we're using a very basic CCDT definition just defining direct routes to the two queue managers. We will revisit later where you can build upon this initial CCDT to work around a number of limitations. We'll go through this in more detail in step 5. If you plan on moving into step 5, I would recommend you use the JSON CCDT style as we'll only be going through examples of further modifications using the JSON CCDT.

Configuring and launching the application instances

Now we have a CCDT defined which references routes to both queue managers in your Uniform Cluster, we need to do some final configuration to optimize our application for rebalancing in a Uniform Cluster then we're ready to launch the amqsghac sample.

Give your application instances a unique name using the new MQ 9.1.2 application naming support (it will default to the executable name if not). *Why is this important?* The name of the application is what identifies instances of an application to the queue managers in a Uniform Cluster, applications with the same name are treated as the same and are balanced across the queue managers accordingly.

It is **strongly** recommended you use unique application names either through using the custom application identifier or by naming your executable accordingly, for each type of application you connect to the Uniform Cluster. This is so that only true instances of that application are considered together when identifying which should be rebalanced around the cluster. Naming disparate applications in the Uniform Cluster as instances of the same application is not recommended.

An example of using a custom application name using the system environment variable is shown below, these can also be set programmatically in all supported client languages:

```
$ export MQAPPLNAME=MY.SAMPLE.APP
```

Using custom application names has other advantages outside of just Uniform Clusters as it allows you to better identify the applications connected to your queue managers using the existing runmqsc DISPLAY CONN command, which we will go into later.

Optionally set your CCDT location (Note: this is required if not using the traditional binary CCDT location & name):

```
$ export MQCCDTURL=file:///var/mqm/AMQCLCHL.TAB $ export
MQCCDTURL=file://<JSON CCDT dir>/<JSON CCDT filename>
```

With the above environment variables set, launch 2 instances of your app connecting to the same qmgr:

```
$ /opt/mqm/samp/bin/amqsghac SYSTEM.DEFAULT.LOCAL.QUEUE QM1
$ /opt/mqm/samp/bin/amqsghac SYSTEM.DEFAULT.LOCAL.QUEUE QM1
```

Rebalancing in action

As you can see in the previous examples above both applications have been started against QM1 opening queue SYSTEM.DEFAULT.LOCAL.QUEUE and sitting in a waiting get for messages on that queue. At this point the Uniform Cluster will have identified it has two applications called 'MY.SAMPLE.APP' connected to QM1 and zero connected to QM2. QM2 will eventually request an application instance from QM1 to correct the application imbalance and QM1 will instruct one of the application instances to rebalance to QM2. Because the amqsghac sample is using an event listener, when this rebalance occurs one instance of the sample will report a reconnect event having occurred:

```
Sample AMQSGHAC start 15:22:20 : EVENT : Connection Reconnecting (Delay:
```

```
120ms) 15:22:20 : EVENT : Connection Reconnected
```
Displaying connections against each qmgr in RUNMQSC will show the number of connected app instances on each queue manager:
```
$ /opt/mqm/bin/runmqsc QM1 5724-H72 (C) Copyright IBM Corp. 1994, 2019.
Starting MQSC for queue manager QM1. DISPLAY CONN(*) WHERE (APPLTAG EQ
MY.SAMPLE.APP) 1 : DISPLAY CONN(*) WHERE (APPLTAG EQ MY.SAMPLE.APP)
AMQ8276I: Display Connection details. CONN(971B4F5C02924622)
EXTCONN(414D5143514D31202020202020202020) TYPE(CONN) APPLTAG(MY.SAMPLE.APP)
$ /opt/mqm/bin/runmqsc QM2 5724-H72 (C) Copyright IBM Corp. 1994, 2019.
Starting MQSC for queue manager QM2. DISPLAY CONN(*) WHERE (APPLTAG EQ
MY.SAMPLE.APP) 1 : DISPLAY CONN(*) WHERE (APPLTAG EQ MY.SAMPLE.APP)
AMQ8276I: Display Connection details. CONN(27214F5C01A51B25)
EXTCONN(414D5143514D32202020202020202020) TYPE(CONN) APPLTAG(MY.SAMPLE.APP)
```
The quick fingered amongst you may be able to display the connections on QM1 prior to the rebalancing to show both application instances connected to QM1. In the post rebalance state shown above, each queue manager shows 1 instance of the application connected. As you can see from the output by specifying a custom application name the MQ admin is able to identify what applications are connected in a more understandable way than purely relying on the default executable name.

Step 4: Application rebalancing after qmgr restart

One of the big advantages of using a Uniform Cluster to manage and "level out" your connected application instances is when either adding additional queue manager horsepower into your cluster or when having to stop and start a queue manager, say for applying maintenance. The issue faced with existing topologies performing one of these actions is how to get your existing applications to connect/reconnect to this new/restarted queue manager. In a traditional configuration the best you could do is having to self-manage new application instances towards this underused queue manager. With a Uniform Cluster all application instances which are capable of rebalancing are eligible to be reconnected to address the imbalance.

Assuming you've got step 3 complete and you've now got two application instances running, one connected to each queue manager, lets simulate taking a queue manager offline for maintenance.

Firstly end one of the queue managers (QM2) with the -r option to cause the application instance to reconnect to the remaining queue manager (QM1). Note: this is just existing client reconnect behavior in action:
```
$ /opt/mqm/bin/endmqm -r QM2
```
The application instance that was connected to QM2 will display another reconnect event in its output as it reconnects to QM1. Displaying connections against the remaining running qmgr (QM1) in runmqsc will show the number of connected application instances back to 2:
```
$ /opt/mqm/bin/runmqsc QM1 5724-H72 (C) Copyright IBM Corp. 1994, 2019.
Starting MQSC for queue manager QM1. DISPLAY CONN(*) WHERE (APPLTAG EQ
MY.SAMPLE.APP) 2 : DISPLAY CONN(*) WHERE (APPLTAG EQ MY.SAMPLE.APP)
AMQ8276I: Display Connection details. CONN(971B4F5C1E744622)
EXTCONN(414D5143514D31202020202020202020) TYPE(CONN) APPLTAG(MY.SAMPLE.APP)
AMQ8276I: Display Connection details. CONN(971B4F5C02924622)
EXTCONN(414D5143514D31202020202020202020) TYPE(CONN) APPLTAG(MY.SAMPLE.APP)
```

Finally by restarting QM2, as with when originally connecting both applications to QM1, the queue managers will identify an imbalance and cause one of the application instances to rebalance over to QM2. This may not be the same application instance as the one originally request to reconnect in step 2.

Step 5: Using a more advanced CCDT configuration

Up to this point the existing CCDT appears to do the job, however with some fancy footwork we have deftly stepped around some limitations with this approach. Predominantly this centers around existing client reconnect logic and how the client decides which queue manager it is willing to connect to.

Limitations with the current CCDT and direct queue manager reference by the application

By specifying a particular queue manager name on the initial application connection (in our example in step 2 this was QM1) we are instructing the client to connect to this and only this queue manager rather than any queue manager in the CCDT - when using traditional client (re)connect logic. The notable caveat here is when the queue manager participates as part of a Uniform Cluster, its reconnect requests will ask a client to reconnect to a particular queue manager which may not be the queue manager the client originally requested to connect to, and so long as the client has a route to that other queue manager defined in its CCDT the client will attempt to fulfill that request. To reiterate, in cases where we enact a traditional client (re)connect the client will always and only attempt to (re)connect back to its original queue manager - as such this isn't a new issue to Uniform Clusters but highlights the fact you should be using queue manager groups within your CCDT whether using Uniform Clusters or otherwise.

To illustrate this issue, lets revisit the queue manager maintenance scenario in step 4 but rather than stop QM2, lets stop QM1. If you've followed this blog post all the way through, at this point we have one application instance connected to QM1 and one connected to QM2.

```
$ /opt/mqm/bin/endmqm -r QM1
```

This time you will notice the application instance connected to QM1 get stuck in a reconnect loop attempting to try to reconnect to the now stopped QM1.

```
Sample AMQSGHAC start 15:25:48 : EVENT : Connection Reconnecting (Delay:
16ms) 15:25:48 : EVENT : Connection Reconnecting (Delay: 1152ms) 15:25:50 :
EVENT : Connection Reconnecting (Delay: 1986ms) 15:25:52 : EVENT :
Connection Reconnecting (Delay: 4088ms) 15:25:56 : EVENT : Connection
Reconnecting (Delay: 7983ms)
```

Surprised? This is because the queue manager will only try to reconnect back to the queue manager specified on its initial connection (QM1), and due to it now having been stopped, will fail to do so.

Back in our example in step 4, when we ended QM2 the application that was connected to QM2 had specified QM1 in its original connection, so using traditional client reconnect logic, it found QM1 in its CCDT definition and successfully executed a reconnect back to that queue manager.

Defining CCDT queue manager groups

So how do we avoid this situation? The answer comes in the form of CCDT queue manager groups. By specifying a set of queue manager group definitions within your CCDT you can decouple the application instances from a particular queue manager and take advantage of the built in load balancing capabilities available when queue manager groups are used.

Now this is where one particular aspect of JSON CCDTs becomes very useful - the ability to specify duplicate channel names. This saves us having to create yet another set of SVRCONN channels on the queue managers for the queue manager group definitions in addition to those we're using for the direct queue manager references.

```
{ "channel": [ { "name": "QM1.APP.CHL", "clientConnection": { "connection":
[ { "host": "localhost", "port": 1414 } ], "queueManager": "ANY_QM" },
"connectionManagement": { "clientWeight": 1, "affinity": "none" }, "type":
"clientConnection" }, { "name": "QM2.APP.CHL", "clientConnection":
{ "connection": [ { "host": "localhost", "port": 1415 } ], "queueManager":
"ANY_QM" }, "connectionManagement": { "clientWeight": 1, "affinity":
"none" }, "type": "clientConnection" }, { "name": "QM1.APP.CHL",
"clientConnection": { "connection": [ { "host": "localhost", "port":
1414 } ], "queueManager": "QM1" }, "type": "clientConnection" }, { "name":
"QM2.APP.CHL", "clientConnection": { "connection": [ { "host": "localhost",
"port": 1415 } ], "queueManager": "QM2" }, "type": "clientConnection" } ] }
```

What the example above now gives you is a queue manager group definition with a route to both QM1 and QM2 using the group queue manager name of 'ANY_QM', and the original set of direct references to the queue managers - required for the new application rebalancing.

The more observant of you will notice we've added a couple of additional options in the group definitions, these being 'clientWeight' and 'affinity'. By specifying a non-zero client weight the order in which clients will attempt to use connection details for the group definitions will not be fixed as such allowing for a rudimentary form of load balancing. Further to this by setting affinity to 'none' we build up our ordered list of group connections to attempt to try in a random order - for any clients on a particular named host. If you were to run your clients on different hosts this option would not be required, but it doesn't hurt!

In actual fact you can further refine your JSON CCDT and queue manager configuration by using the same SVRCONN channel name for each of your queue managers, thus making them more uniform and simplifying scripting configuration and management, but as we've already used unique names on each of our queue managers lets stick with what we have.

Note: It is not possible to specify the same channel name in a binary CCDT due to the fact the channel name is the primary key. You can work around this by having unique channels for both your CCDT group definitions and your direct named queue manager definitions, but its ugly and requires twice the number of SVRCONN channels defined on your queue managers and as such twice as much to manage and keep consistent.

Connecting application instances using CCDT queue mananger groups

Now we have an updated CCDT for our application instances to use and gone through the nuances of the new options, lets put it to the test! If you haven't already, stop your application instances. With the original CCDT and APPNAME environment variables still defined, restart the applications now specifying the queue manager group 'ANY_QM'.

```
$ /opt/mqm/samp/bin/amqsghac SYSTEM.DEFAULT.LOCAL.QUEUE "*ANY_QM"
$ /opt/mqm/samp/bin/amqsghac SYSTEM.DEFAULT.LOCAL.QUEUE "*ANY_QM"
```

Now your application instances will attempt to connect to one of the queue managers defined in the queue manager group, and with the client weight and affinity options defined as above, we should see each application instance connect to one of the queue managers.

Now lets try ending QM1 to force the one application connected to QM1 to reconnect to QM2:

```
$ /opt/mqm/bin/endmqm -r QM1
```

Rather than the application getting stuck in a reconnect loop trying to connect to one queue manager as we saw using the old CCDT, the application now tied to the queue manager group 'ANY_QM' will go through each of the definitions for 'ANY_QM' and when it is able to successfully connect to one of the underlying queue managers, it will do so. Again that application will notify you of the reconnect having happened:

```
Sample AMQSGHAC start 15:28:20 : EVENT : Connection Reconnecting (Delay:
120ms) 15:28:20 : EVENT : Connection Reconnected
```

At this point you can bring QM1 back up and wait for the Uniform Cluster application rebalancing logic to kick in and rebalance one of the applications back to QM1, which will use the direct CCDT definition for QM1 rather than the queue manager group definitions.

Closing

We've now gone through the basic functionality of using the Uniform Cluster pattern as of the capabilities present in MQ 9.1.2, highlighting the benefits of allowing your queue managers to manage your connected application instances. This is likely to be the first step of a much longer journey, as such there are a number of limitations in its current form.

- JMS and .NET clients do not make use of the queue manager name provided by the rebalance request, only the C client and MQ clients backed off of the C client (Node, Go) will act on this information and attempt to rebalance directly to the requested queue manager.
- Queue managers participating in a Uniform Cluster are expected to exhibit some levels of uniformity relating to application access (common TLS/authority rules) and queue manager objects used by your application instances (queues, topics etc). This is not policed in MQ 9.1.2.
- The Uniform Cluster pattern is not available for queue managers on the mainframe platform.

So, make sure you keep up to date with our future CD releases to see how this capability evolves. And please give us your feedback along the way!

by LaurenceBonney