# Rebalancing JMS applications within a MQ Uniform Cluster in MQ 9.1.3

LaurenceBonney

Published on 06/08/2019

## Whats new in uniform clusters in MQ 9.1.3

Following on from the uniform cluster pattern support added to MQ 9.1.2 - enabling C based clients to be automatically rebalanced across the members of the cluster - MQ 9.1.3 now expands the client coverage in this area to include Java SE JMS applications.

In addition to JMS rebalancing support, a number of other areas have changed since MQ 9.1.2 which will be covered to some extent within this blog namely:

- The addition of a new runmqsc command DISPLAY APSTATUS to query the spread of applications across a uniform cluster see blog Display Application Status On a Uniform Cluster
- Rebalancing is now more intelligent and moves some MQ connections together, such as in the case of MQ applications that comprise at least 2 MQ connections
- New behaviour for MQ CONNTAGs on distributed platforms to relate associated MQ connections and ensure application connections are rebalanced correctly. This is partly in support of the more intelligent rebalancing for JMS applications.

The updates to the use of CONNTAG on distributed platforms is important particularly for JMS applications as each JMS connection and any sessions created from that connection (this applies equally to JMSContexts in applications using JMS 2) **each** result in an MQ connection to the queue manager. As such in the most basic case, where a single connection and session is used within an application, that application will have 2 MQ connections. By using the same CONNTAG on all of the MQ connections created by a JMS connection or JMSContext, rebalancing calculations can be applied based on the spread of JMS connections rather than MQ connections.

This means that a) a JMS connection and its sessions will be rebalanced as a single unit so connections and sessions will always rebalance to the same queue manager, and b) the 'weight' of a single JMS connection application (with its associated session(s)) is the same as a single C connection application with respect to rebalancing. For more information about the way MQ CONNTAGs are used by MQ on distributed platforms, see KC topic ConnTag on Multiplatforms.

The behaviour described in the CONNTAG KnowledgeCenter topic makes it important to set the SHARECNV value for server-conn channels to a suitable value, typically 1 for non-JMS applications. This is because MQ will prevent MQ connections on the same TCP socket from using different MQ CONNTAG values. For JMS applications, the MQ JMS client will automatically ensure that different JMS Connections use separate TCP sockets, but if you have your own MQ C applications that share a TCP socket today, you will need to change the SHARECNV value to 1 to prevent this in a uniform cluster.

## Rebalancing with JMS

This blog will take you through the basic configuration of your JMS applications within this environment and show application rebalancing in action using a basic 2 queue manager uniform cluster. Follow the steps outlined in the MQ 9.1.2 uniform cluster pattern walkthrough blog, to configure your pair of clustered queue managers using the uniform cluster pattern.

### Sample JMS application

For this walk through we'll be using a basic JMS application which creates a single connection, session and message consumer, which will wait on messages arriving on a predefined queue. The application programmatically creates a connection factory and configures said connection factory with the basic properties required to perform rebalancing.

```
import java.text.SimpleDateFormat;
import java.util.Calendar;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;

import com.ibm.msg.client.jms.JmsConnectionFactory;
import com.ibm.msg.client.jms.JmsFactoryFactory;
import com.ibm.msg.client.wmq.WMQConstants;

public class JMSApp
{

  public static void main(String[] args) throws Exception
  {
    JMSApp testApp = new JMSApp();
    testApp.run();
  }

  public void run() throws Exception
  {
    String applName = null;
    String ccdtURL = null;
    Connection connection = null;
    Session session = null;
    Destination destination = null;
    MessageConsumer consumer = null;
    Message msg = null;
    String queueName = "SYSTEM.DEFAULT.LOCAL.QUEUE";

    // Use the MQAPPLNAME if set in the environment
    if(System.getenv("MQAPPLNAME") != null
&& !System.getenv("MQAPPLNAME").isEmpty())
    {
      applName = System.getenv("MQAPPLNAME");
    }

    // Use the MQCCDTURL if set in the environment
    if(System.getenv("MQCCDTURL") != null
&& !System.getenv("MQCCDTURL").isEmpty())
    {
      ccdtURL = System.getenv("MQCCDTURL");
    }

    try
    {
      log("Programmatically creating JMS connection factory");
      JmsFactoryFactory ff =
JmsFactoryFactory.getInstance(WMQConstants.WMQ_PROVIDER);
      JmsConnectionFactory cf = ff.createConnectionFactory();

      log("Setting WMQ_QUEUE_MANAGER to CCDT group definitions for '*ANY_QM'");
      cf.setStringProperty(WMQConstants.WMQ_QUEUE_MANAGER, "*ANY_QM");

      log("Setting client reconnect option");
      cf.setIntProperty(WMQConstants.WMQ_CLIENT_RECONNECT_OPTIONS,
WMQConstants.WMQ_CLIENT_RECONNECT);

      log("Setting client connection mode");
      cf.setIntProperty(WMQConstants.WMQ_CONNECTION_MODE,
WMQConstants.WMQ_CM_CLIENT);

      if (applName != null){
        log("Setting custom application name: '" + applName + "'");
        cf.setStringProperty(WMQConstants.WMQ_APPLICATIONNAME, applName);
      }

      if (ccdtURL != null)
      {
        log("Setting CCDTURL to "+ccdtURL);
        cf.setStringProperty(WMQConstants.WMQ_CCDTURL, ccdtURL);
      }

      log("Creating JMS connection");
      connection = cf.createConnection();

      log("Creating JMS session");
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

      log("Creating JMS destingation for "+queueName);
      destination = session.createQueue("queue:///"+queueName);

      log("Creating JMS consumer for "+queueName);
      consumer = session.createConsumer(destination);

      connection.start();

      log("Waiting for message on "+queueName);
      msg = consumer.receive(900000);

      if (msg == null) {
        log("Message receive timed out with no message received");
      } else {
        log("Message received: " + ((TextMessage)msg).getText());
      }
    }
    catch (JMSException jmsex)
    {
      log("JMSException thrown");
      jmsex.printStackTrace();
    }
    finally
    {
      try {
        if (consumer != null) {
          log("Closing message consumer");
          consumer.close();
        }

        if (session != null) {
          log("Closing session");
          session.close();
        }

        if (connection != null) {
          log("Closing connection");
          connection.close();
        }
      } catch (JMSException jmsex) {
        log("JMSException thrown");
```

```
        jmsex.printStackTrace();
      }

    }
  }
  private static void log(String text)
  {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
    String timeStamp = sdf.format(Calendar.getInstance().getTime());
    System.out.println(timeStamp + " " + text);
  }
}
```

As you can see the application itself is pretty simple/limited, the important section to note is the programmatic configuration of the connection factory. Of course you could achieve the same result by using a JNDI store and configuring the connection factory up front - but for the context of this short walk through we'll keep things all under one roof.

## Connection factory property considerations for rebalancing

We'll now go through each of the connection factory values being set in the application above and why each is important in the context of your JMS application being able to successfully participate and rebalance within a uniform cluster.

Setting the queue manager name to the name of the group name definitions in the CCDT created in the C application rebalancing walk through. As described in that blog, using a group name rather than a single queue manager name directly, allows for the application to move between each of the available queue managers using the traditional style application reconnect logic rather than being anchored to a particular queue manager - this is covered in more detail is the referenced blog above.

```
log("Setting WMQ_QUEUE_MANAGER to CCDT group definitions for '*ANY_QM'");
cf.setStringProperty(WMQConstants.WMQ_QUEUE_MANAGER, "*ANY_QM");
```

Setting the client reconnect option WMQ_CLIENT_RECONNECT, meaning the application is willing and able to reconnect to any queue manager. Accepting this capability comes with a number of limitations such as, if your application has state tied to a particular queue manager it would not be suitable to use the WMQ_CLIENT_RECONNECT option and therefore would not be a suitable application for application rebalancing within a uniform cluster - with MQ 9.1.3 queue manager capabilities at any rate.

```
log("Setting client reconnect option");
cf.setIntProperty(WMQConstants.WMQ_CLIENT_RECONNECT_OPTIONS,
WMQConstants.WMQ_CLIENT_RECONNECT);
```

Setting the connection mode to WMQ_CM_CLIENT. Rebalancing is only possible with client connected applications.

```
log("Setting client connection mode");
cf.setIntProperty(WMQConstants.WMQ_CONNECTION_MODE,
WMQConstants.WMQ_CM_CLIENT);
```

Although not strictly required, its good practice to make use of the custom application name added in 9.1.2 so that applications that are logically the same can be treated as such by the uniform cluster rebalancing algorithms within the queue manager. Alternatively if you have used a Java class name that makes it easy to identify your application you can skip this step and the MQ JMS classes will use the class name as the applname.

```
log("Setting custom application name: '" + applName + "'");
cf.setStringProperty(WMQConstants.WMQ_APPLICATIONNAME, applName);
```

The final configuration step is setting the WMQ_CCDTURL to the location of the binary or JSON CCDT which references all the queue managers in your uniform cluster. This must include direct references to the queue managers (required for uniform cluster rebalancing), and optionally (recommended) group definitions for the queue managers (for traditional application reconnect).

```
log("Setting CCDTURL to "+ccdtURL);
cf.setStringProperty(WMQConstants.WMQ_CCDTURL, ccdtURL);
```

## Launching your JMS application instances

Assuming you've successfully compiled your JMS application, and created your CCDT to include your direct queue manager and queue manager group definitions, lets start up a couple of application instances:

```
$ export MQAPPLNAME=JMSAPP1 $ export MQCCDTURL="file:///tmp/CCDT.json"
$ /opt/mqm/java/jre64/jre/bin/java -cp
/opt/mqm/java/lib/com.ibm.mq.allclient.jar:/tmp/ JMSApp 2019/07/24 10:48:07
Programmatically creating JMS connection factory 2019/07/24 10:48:09
Setting WMQ_QUEUE_MANAGER to CCDT group definitions for '*ANY_QM'
2019/07/24 10:48:09 Setting client reconnect option 2019/07/24 10:48:09
Setting client connection mode 2019/07/24 10:48:09 Setting custom
application name: 'JMSAPP1' 2019/07/24 10:48:09 Setting CCDTURL to
file:///tmp/CCDT.json 2019/07/24 10:48:09 Creating JMS connection
2019/07/24 10:48:10 Creating JMS session 2019/07/24 10:48:10 Creating JMS
destingation for SYSTEM.DEFAULT.LOCAL.QUEUE 2019/07/24 10:48:10 Creating
```

```
JMS consumer for SYSTEM.DEFAULT.LOCAL.QUEUE 2019/07/24 10:48:10 Waiting for
message on SYSTEM.DEFAULT.LOCAL.QUEUE
```

```
$ export MQAPPLNAME=JMSAPP1 $ export MQCCDTURL="file:///tmp/CCDT.json"
$ /opt/mqm/java/jre64/jre/bin/java -cp
/opt/mqm/java/lib/com.ibm.mq.allclient.jar:/tmp/ JMSApp 2019/07/24 10:48:10
Programmatically creating JMS connection factory 2019/07/24 10:48:10
Setting WMQ_QUEUE_MANAGER to CCDT group definitions for '*ANY_QM'
2019/07/24 10:48:10 Setting client reconnect option 2019/07/24 10:48:11
Setting client connection mode 2019/07/24 10:48:11 Setting custom
application name: 'JMSAPP1' 2019/07/24 10:48:11 Setting CCDTURL to
file:///tmp/CCDT.json 2019/07/24 10:48:11 Creating JMS connection
2019/07/24 10:48:11 Creating JMS session 2019/07/24 10:48:12 Creating JMS
destingation for SYSTEM.DEFAULT.LOCAL.QUEUE 2019/07/24 10:48:12 Creating
JMS consumer for SYSTEM.DEFAULT.LOCAL.QUEUE 2019/07/24 10:48:12 Waiting for
message on SYSTEM.DEFAULT.LOCAL.QUEUE
```

Using queue manager groups with our CCDT and applications and with the weighing and affinity settings configured in the C application rebalancing walk through, if you launch your applications from different hosts (or containers) you will likely get a natural spread of applications across your queue managers; due to the difference in hostnames on the client systems which is used to seed the randomisation of CCDT lookup for the clients.

However (in my local environment) I'm running my applications on a single system, initially the applications all connect to the same queue manager end point. To simulate this you could end one of the queue managers to force the appications to connect to the other active queue manager - be mindful to use 'endmqm -r' if you have already launched your applications to allow them to automatically reconnect to the other queue manager rather than disconnect and force the application to handle reconnect.

## Showing application spread using runmqsc DIS APSTATUS

The current spread of the JMS application instances can be shown using the runmqsc DIS APSTATUS command new in MQ 9.1.3, on one of the queue managers in the uniform cluster, for example:

```
dis apstatus(JMSAPP1) type(qmgr)
12 : dis apstatus(JMSAPP1) type(qmgr)

AMQ8932I: Display application status details.
APPLNAME(JMSAPP1) ACTIVE(YES) COUNT(2) MOVCOUNT(2) <---- 2 applications
(COUNT(2)) BALSTATE(HIGH)
LMSGDATE(2019-07-24) LMSGTIME(10:48:54) QMNAME(QM1) <---- on QM1
QMID(QM1_2019-07-24_10.43.06)

AMQ8932I: Display application status details.
```

```
APPLNAME(JMSAPP1) ACTIVE(YES) COUNT(0) MOVCOUNT(0) <---- 0 applications
(COUNT(0)) BALSTATE(LOW)
LMSGDATE(2019-07-24) LMSGTIME(10:48:31) QMNAME(QM2) <---- on QM2
QMID(QM2_2019-07-24_10.43.05)
```

Note: the information presented via the DIS APSTATUS command is only a point in time statement based on the information available on the queue manager the command is being run on, as such there may be a short delay in information being presented for remote queue managers.

After a short while QM2 will request an application from QM1 and the application will get rebalanced, for example:

```
dis apstatus(JMSAPP1) type(qmgr)
14 : dis apstatus(JMSAPP1) type(qmgr)

AMQ8932I: Display application status details.
APPLNAME(JMSAPP1) ACTIVE(YES)
COUNT(1) MOVCOUNT(1) BALSTATE(OK)
LMSGDATE(2019-07-24) LMSGTIME(10:55:48) QMNAME(QM1)
QMID(QM1_2019-07-24_10.43.06)

AMQ8932I: Display application status details.
APPLNAME(JMSAPP1) ACTIVE(YES)
COUNT(1) MOVCOUNT(1) BALSTATE(OK)
LMSGDATE(2019-07-24) LMSGTIME(10:55:33) QMNAME(QM2)
QMID(QM2_2019-07-24_10.43.05)
```

## Closing

We've now gone through the basic functionality of using the Uniform Cluster pattern as of the capabilities present in MQ 9.1.3 This represents another step on a longer journey, as such there continue to be a number of limitations in its current form.

- .NET clients do not make use of the queue manager name provided by the rebalance request, only the JMS (Java Platform, Standard Edition), C client and MQ clients based on the MQ 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 9.1.3.

- 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*