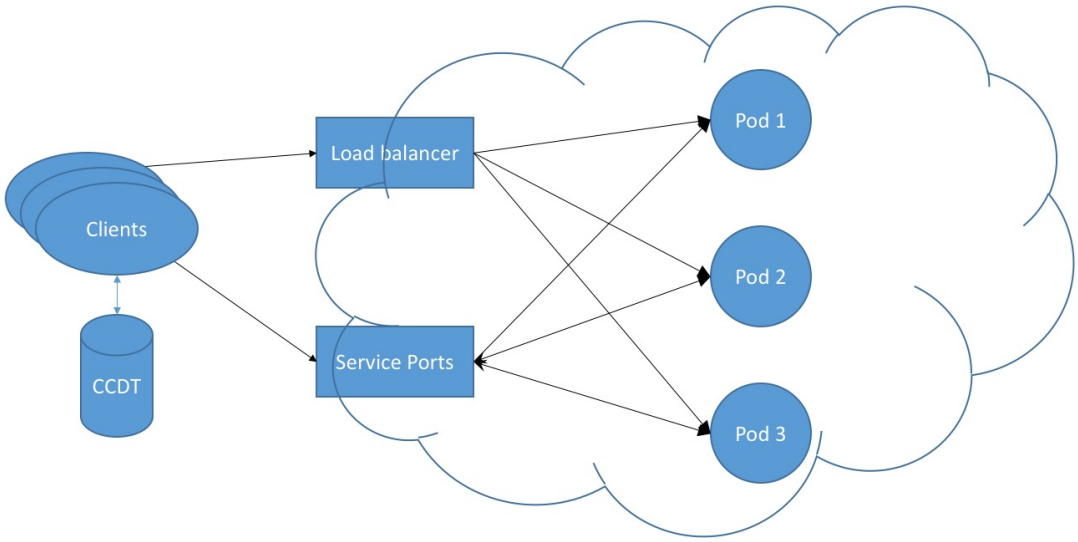
IBM Cloud Private and MQ's Uniform Cluster

ElliotGregory
Published on 25/03/2019 / *Updated on 11/04/2019*

Uniform clustering is new to MQ 9.1.2 and takes an existing MQ cluster and gives it the ability to re-balances all the client connections within a named application. This results in all queue managers having a equal number of connections. For example, if there were three queue managers and the first queue manager had three client connections with the same application name and the others had none, then with uniform cluster it would redistributed the connections so each queue manager had one connection each. Without going into too much detail (see: Building scalable fault tolerant systems with IBM MQ 9.1.2 CD and Walkthrough: Auto application rebalancing using the Uniform Cluster pattern) when a connection is redistributed the client is disconnected from its queue manager with an instruction to connect to a different name queue manager. For this to work the client end will need to use a CCDT (Client Channel Definition Table) which acts as a directory to tell the client where to find the queue manager. The CCDT has been in MQ for some time as a binary file, but now there's a JSON version making it easier to construct. For a client connecting to a uniform cluster, the CCDT will need to have the details of all of the queue managers in that cluster.

The Uniform Cluster constantly monitors the balance applications within the cluster and rebalanced the connections should conditions change. For example queue manager restarting. IBM Cloud Private is essentially a Kubernetes based system for deploying container based software. Kubernetes is an excellent environment for running MQ topologies that consist of multiple queue managers that follow the Uniform Cluster pattern. So how does the Uniform Cluster pattern translate to running MQ in ICP? Well its similar, instead of queue managers it is pods (based on one queue manager per pod), a CCDT referencing the pods within the same application name and an additional reference to a ICP load balancer. So new client connections would use CCDT to reference the ICP load balancer which in turn passes the connection to one of the pods in the cluster. If there is an imbalance of the number of connections to one pod, the Uniform Cluster would instruct a client to reconnect to a different specific pod. The client would reconnect using details from its CCDT.

The rest of this blog will discuss how to create a Uniform Cluster within ICP using the diagram below



Note: This blog will not cover all details such as: how to create a MQ Cluster, whether the solution is suitable for all production models and assumes the reader has an understanding of the K8s commands.

Step 1 – Create three pods containing queue managers

These can be generated using the helm install command followed by K8s service command to open a port to the outside world.

```
helm install --name pod001 --set queueManager.name=QM1 <other-
arguments> <MQHelmChart>

helm install --name pod002 --set queueManager.name=QM2 <other-
arguments> <MQHelmChart>

helm install --name pod003 --set queueManager.name=QM3 <other-
arguments> <MQHelmChart>

kubectl expose pod pod001 --port 1414 --name pod001-service --
type=NodePort

kubectl expose pod pod002 --port 1414 --name pod001-service --
type=NodePort

kubectl expose pod pod003 --port 1414 --name pod001-service --
type=NodePort
```

Where MQHelmChart is your chosen Helm chart to install MQ.

Each pod must have a persistent volume as the pods need to retain their configuration following a queue manager restart and also required if the queue manager itself has persisted messages.

Step 2 – Load balancer

Use Kubernetes(K8s) kubectl to create a load balancer. Put the following into a file

```
apiVersion: v1

kind: Service

metadata:

  name: lb

spec:

  type: LoadBalancer

  selector:

    app: "ibm-mq"

  ports:

  - protocol: TCP

    port: 1414

    targetPort: 1414
```

then run the following command

```
kubectl create -f <filename>
```

Step 3 – Create a CCDT

Create a JSON formated CCDT table. First collect the external port values, three pods and one load balancer.

```
kubectl get services --output
jsonpath='{.items[0].spec.ports[0].nodePort}' --field-selector
metadata.name=pod001-service
```

```
kubectl get services --output
jsonpath='{.items[0].spec.ports[0].nodePort}' --field-selector
metadata.name=pod002-service

kubectl get services --output
jsonpath='{.items[0].spec.ports[0].nodePort}' --field-selector
metadata.name=pod003-service

kubectl get services --output
jsonpath='{.items[0].spec.ports[0].nodePort}' --field-selector
metadata.name=lb
```

Make a json file consisting of 4 channels, one for load balancer and three pods similar to below

```
{

    "channel": [

    {

        "name": "<Channel>",

        "clientConnection": {

            "connection": [

            {

                "host": "<Host ICP>",

                "port": <Load Balancer External Port>

            }],

        "queueManager": "*GROUP"

        },

        "type": "clientConnection"

    }

    ,

    {
```

```
        "name": "<Channel>",
        "clientConnection": {
            "connection": [
            {
                "host": "<Host ICP>",
                "port": <Pod External Port 1>
            }],
        "queueManager": "QM1"
        },
        "type": "clientConnection"
    }
    ,
    {
        "name": "<Channel>",
        "clientConnection": {
            "connection": [
            {
                "host": "<Host ICP>",
                "port": <Pod External Port 2>
            }],
        "queueManager": "QM2"
        },
        "type": "clientConnection"
    }
    ,
```

```
    {
        "name": "<Channel>",
        "clientConnection": {
            "connection": [
            {
                "host": "<Host ICP>",
                "port": <Pod External Port 3>
            }],
        "queueManager": "QM3"
        },
        "type": "clientConnection"
    }
    ]
}
```

Where:
<Channel> the name of the service connection channel. Must be the same for all queue managers for the load balancer to function correctly.
<Host ICP> the IP address of the ICP host machine
<Load Balancer External Port> the external port value generated by the K8s service load balancer command earlier
<Pod External Port 1> the external port value generated by the K8s service pod 1 command earlier
<Pod External Port 2> the external port value generated by the K8s service pod 2 command earlier
<Pod External Port 3> the external port value generated by the K8s service pod 3 command earlier
At this stage the individual connections to each pod can be verified using runmqsc

```
export MQCCDTURL=<Path to json file>

runmqsc -c <Queue manager name>
```

Making life easier with CCDT

- Queue manager's configuration, as runmqsc can be perform from a single client location.
- With CCDT stored in a central location, i.e. a server then only one CCDT need to be updated for all client to pick up

Look out for the blog "New JSON CCDT & Uniform Cluster features in IBM Cloud Private" for more information on CCDT.

Step 4 – Build the Uniform Cluster

- Create a standard MQ cluster between the three pods using the external port information collect previously. Check out the MQ Knowledge center for more information.
- Edit the qm.ini file for each queue manager on each pod to include

```
TuningParameters:

   UniformClusterName=<MQ_CLUSTER_NAME>
```

where MQ_CLUSTER_NAME is the name given to the cluster previously created.
Possible command sequence:

```
kubectl cp twinpines/pod001-twinpines-ibm-mq-
0:/var/mqm/qmgrs/QM1/qm.ini .

vi qm.ini

kubectl cp qm.ini twinpines/pod001-twinpines-ibm-mq-
0:/var/mqm/qmgrs/QM1/qm.ini
```

Each queue manager/pod needs to be restarted for this configuration to be read.

```
kubectl exec <pod name> -it -- endmqm -r <queue manager name>
```

Using the endmqm indirectly results in the pod termination, as the liveliness monitor will see the stopped queue manager and trigger its replacement. The replacement pod will automatically start the queue management and use the preceding pods persistent volume and configuration.
This area is well cover in the blog "Walkthrough: Auto application rebalancing using the Uniform Cluster pattern" which gives an in depth view of configuration of MQ and creating the uniform cluster.

Step 5 – Ready to use

Now you have a uniform cluster running any application wishing to benefit from connection rebalancing must use the CCDT we created earlier when connecting to it. Each application should specify "*GROUP" as the queue manager name. This will give initial balancing of connections.

Reducing and expanding

With the working solution of three pods it is possible to terminate a pod and in doing so the connections within the uniform cluster would be redirected to the remaining pods. Similarly, when a pod is put back using its preexisting persisted configuration it will automatically reconnect to the uniform cluster and request to take workload.

Adding an additional pod

Adding a new pod as a partial repository is relatively straight forward.

- Create the pod and externalised port – see step 1
- Update the CCDT with the new pods details – see step 2.
- Configure the queue manager in the new pod as a partial repository and update the qm.ini – see step 4

Closing

This blog is a quick journey through using MQ's Uniform cluster within the ICP environment. There are many different implementations, this being one of them. There are variations in how the pods are managed in ICP and that would depend on the production environment. Look out for other blogs which cover Uniform Cluster and how it works within MQ and how the new JSON CCDT works. The combination of ICP, Uniform Cluster and CCDT makes some interesting powerful combinations.

by ElliotGregory