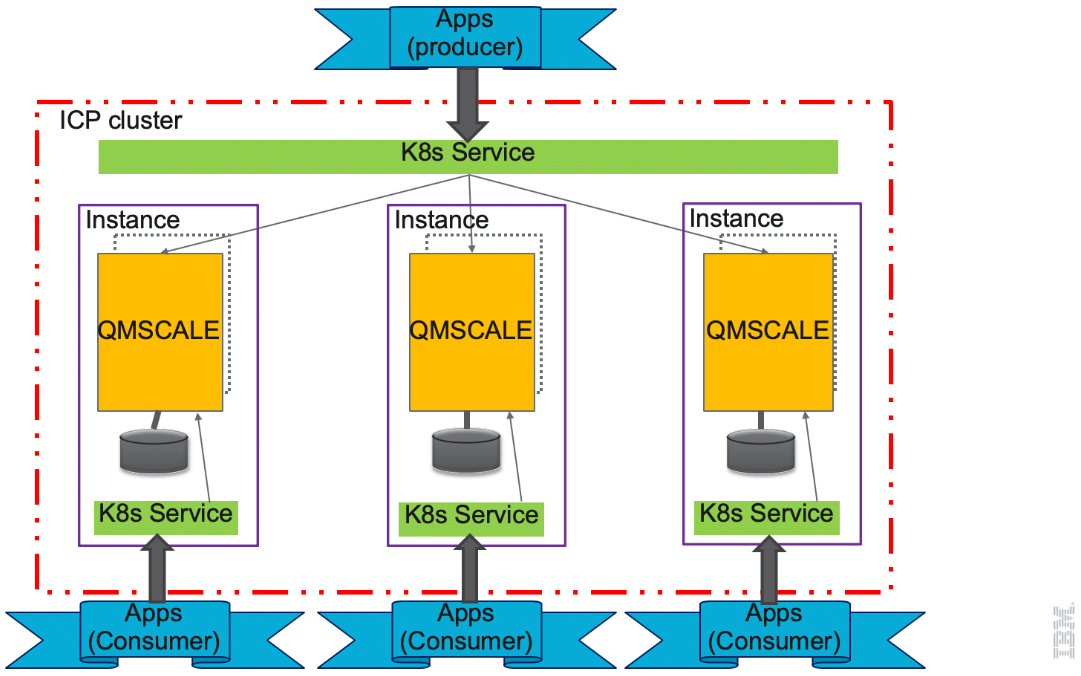
Certified container scaling part 1: The fixed deployment

RobParker
Published on 11/06/2019 / *Updated on 25/06/2019*

Containerisation and specifically solutions such as Kubernetes brings with it the expectation that you can easily scale your systems, not vertically but horizontally. Basically, if you need more, scale what you have to multiple things of a fixed size rather than one big thing that grows. This is a pattern that many have been using with IBM MQ for a long time. IBM CloudPaks give you a great way of easily provisioning IBM MQ queue managers as containers into IBM Cloud Private. This blog will show you how to set up a simple horizontally scaled queue manager topology to demonstrate the concept.

Design

In this blog we will be working with the following design:

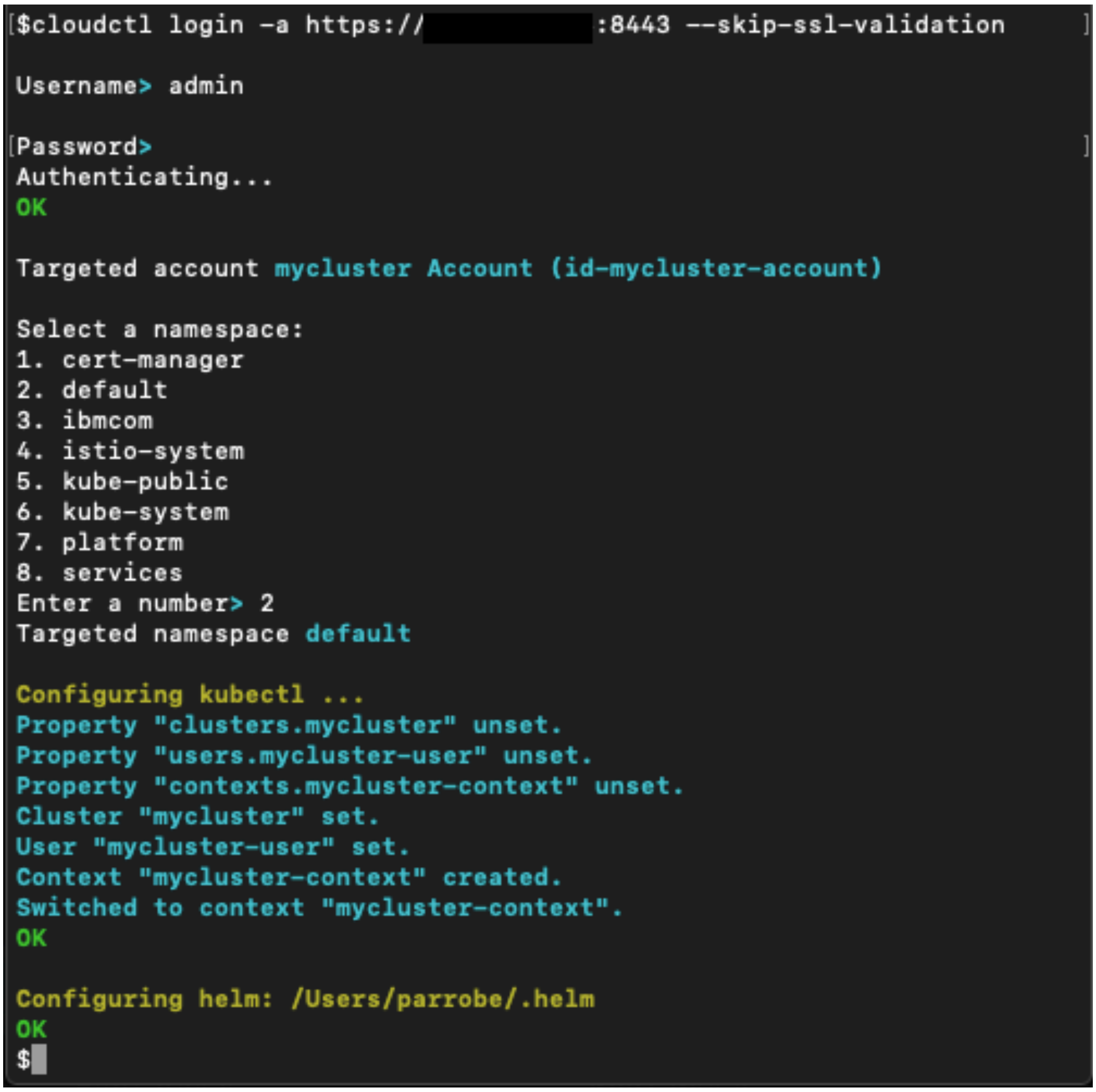


For this scenario we'll be deploying three instances of the IBM MQ CloudPak. This creates three individual queue managers in a repeatable way. We will configure each queue manager with the same set of MQ resources, such as queues and channels. Matching those three queue managers are three instances of the consuming application, one to consume from each queue manager. With this topology, a producing application can connect to any of the queue managers to put messages for processing. You'll see how we configure the K8s system so that producing applications are automatically workload balanced across all three queue managers, ensuring all three systems are fully utilised.

Before you begin

First you should make sure you have a command prompt open which has been configured to access your ICP cluster. You will need both helm and kubectl in order to deploy and create your resources:

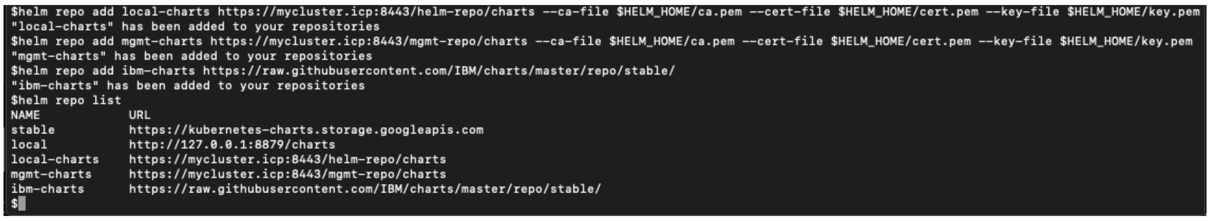
```
cloudctl login -a https://-Cluster IP-:8443
```



```
[$cloudctl login -a https://[redacted]:8443 --skip-ssl-validation          ]

Username> admin

[Password>                                                                 ]
Authenticating...
OK

Targeted account mycluster Account (id-mycluster-account)

Select a namespace:
1. cert-manager
2. default
3. ibmcom
4. istio-system
5. kube-public
6. kube-system
7. platform
8. services
Enter a number> 2
Targeted namespace default

Configuring kubectl ...
Property "clusters.mycluster" unset.
Property "users.mycluster-user" unset.
Property "contexts.mycluster-context" unset.
Cluster "mycluster" set.
User "mycluster-user" set.
Context "mycluster-context" created.
Switched to context "mycluster-context".
OK

Configuring helm: /Users/parrobe/.helm
OK
$
```

Additonally, make sure that your helm client has been configured to use your IBM Cloud Private Helm repo. You should also have added the ibm-charts helm repo to your helm client:

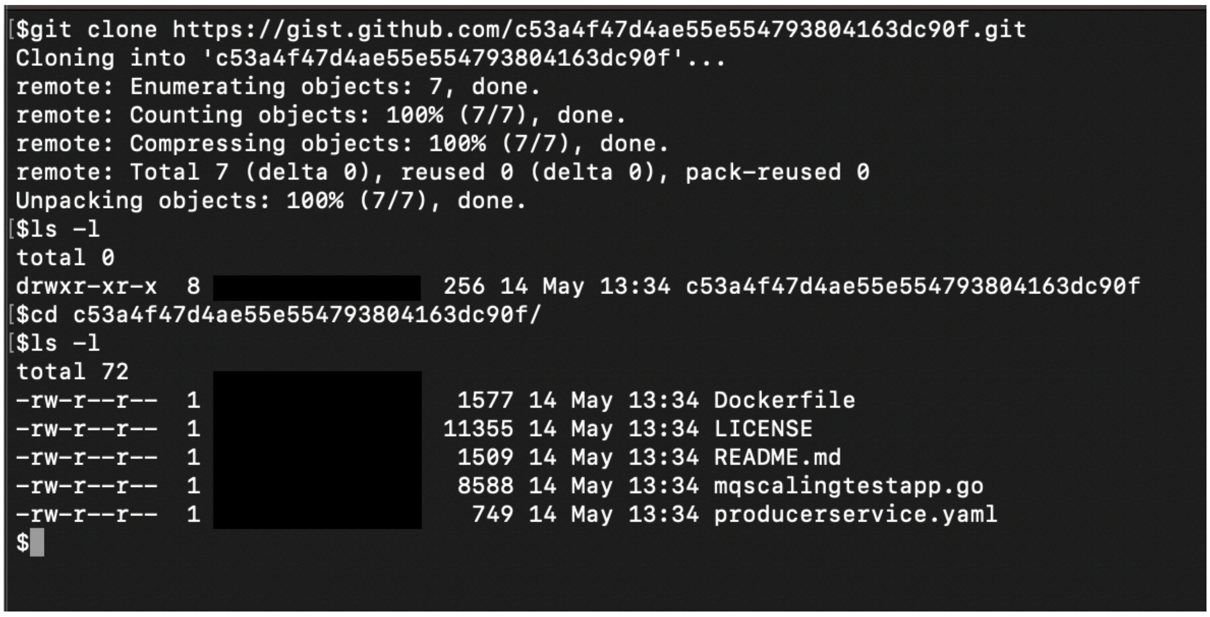
```
helm repo add ibm-charts
https://raw.githubusercontent.com/IBM/charts/master/repo/stable/
```



```
$helm repo add local-charts https://mycluster.icp:8443/helm-repo/charts --ca-file $HELM_HOME/ca.pem --cert-file $HELM_HOME/cert.pem --key-file $HELM_HOME/key.pem
"local-charts" has been added to your repositories
$helm repo add mgmt-charts https://mycluster.icp:8443/mgmt-repo/charts --ca-file $HELM_HOME/ca.pem --cert-file $HELM_HOME/cert.pem --key-file $HELM_HOME/key.pem
"mgmt-charts" has been added to your repositories
$helm repo add ibm-charts https://raw.githubusercontent.com/IBM/charts/master/repo/stable/
"ibm-charts" has been added to your repositories
$helm repo list
NAME            URL
stable          https://kubernetes-charts.storage.googleapis.com
local           http://127.0.0.1:8879/charts
local-charts    https://mycluster.icp:8443/helm-repo/charts
mgmt-charts     https://mycluster.icp:8443/mgmt-repo/charts
ibm-charts      https://raw.githubusercontent.com/IBM/charts/master/repo/stable/
$
```

Next, download a GitHub Gist which contains a test application we will use in a later section. It also contains a yaml file that is used to create the kubernetes service the producer applications connect to. You can find the gist here and download it using the following command:

```
git clone https://gist.github.com/c53a4f47d4ae55e554793804163dc90f.git
```



```
[$git clone https://gist.github.com/c53a4f47d4ae55e554793804163dc90f.git
 Cloning into 'c53a4f47d4ae55e554793804163dc90f'...
 remote: Enumerating objects: 7, done.
 remote: Counting objects: 100% (7/7), done.
 remote: Compressing objects: 100% (7/7), done.
 remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 0
 Unpacking objects: 100% (7/7), done.
[$ls -l
 total 0
 drwxr-xr-x  8 [redacted]  256 14 May 13:34 c53a4f47d4ae55e554793804163dc90f
[$cd c53a4f47d4ae55e554793804163dc90f/
[$ls -l
 total 72
 -rw-r--r--  1 [redacted]   1577 14 May 13:34 Dockerfile
 -rw-r--r--  1              11355 14 May 13:34 LICENSE
 -rw-r--r--  1               1509 14 May 13:34 README.md
 -rw-r--r--  1               8588 14 May 13:34 mqscalingtestapp.go
 -rw-r--r--  1                749 14 May 13:34 producerservice.yaml
 $
```

Creating your MQ instances

To create each instance we can either use the IBM Cloud Private catalog or the Helm command line tool. For this example we will use the Helm command line tool. Each instance that is deployed has specific options configured identifcally:

- An additional label has been set called **mqscalinggroup** with the value **MQSCALEGROUP1**.
- To allow connection outside of the cluster the default service is set to NodePort.
- We use the IBM MQ Advanced for Developers certified container as this versions comes with default objects we can use to quickly and easily connect our test applications.

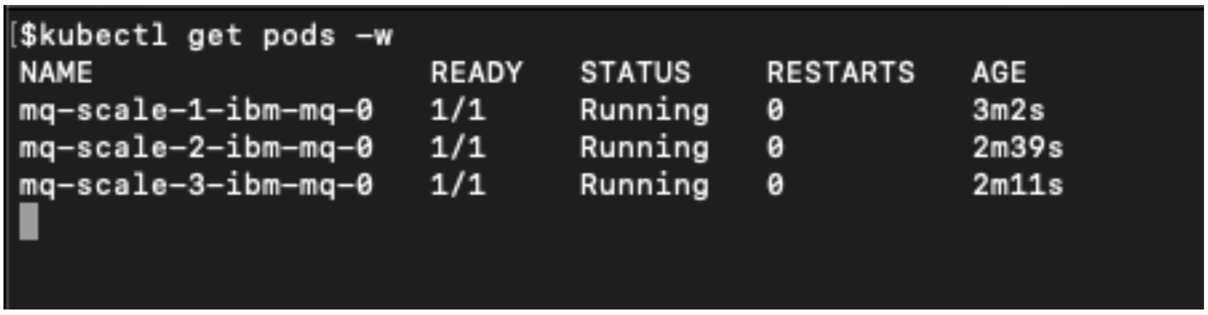
Run the following commands to deploy 3 instances of the IBM MQ certified container with a release name of: **mq-scale-1**, **mq-scale-2** and **mq-scale-3**.

```
helm install --tls --name mq-scale-1 --set license=accept --set
service.type=NodePort --set metadata.labels.mqscalinggroup=MQSCALEGROUP1
```

```
ibm-charts/ibm-mqadvanced-server-dev
helm install --tls --name mq-scale-2 --set license=accept --set
service.type=NodePort --set metadata.labels.mqscalinggroup=MQSCALEGROUP1
ibm-charts/ibm-mqadvanced-server-dev
helm install --tls --name mq-scale-3 --set license=accept --set
service.type=NodePort --set metadata.labels.mqscalinggroup=MQSCALEGROUP1
ibm-charts/ibm-mqadvanced-server-dev
```

Now you have deployed the instances we need to wait until all 3 queue managers are ready and running. You can do this by running the following command and waiting until all 3 show as "*Ready 1/1*":

```
kubectl get pods -w
```



```
[$kubectl get pods -w
 NAME                   READY   STATUS    RESTARTS   AGE
 mq-scale-1-ibm-mq-0    1/1     Running   0          3m2s
 mq-scale-2-ibm-mq-0    1/1     Running   0          2m39s
 mq-scale-3-ibm-mq-0    1/1     Running   0          2m11s
```

The *-w* flag on the command tells kubectl to keep updating the screen. If the command stops early and returns you to the terminal prompt, just re-run the command. It may take some time for the queue managers to start, depending on whether the ibmcom/mq image is pre-pulled. If a deployment is taking a long time, use the following command to see what is preventing it from starting:

```
kubectl describe pod
```

Once all 3 queue managers are running and flagged as ready, move onto the next section where we will create a kubernetes service for the producer applications.

Creating your producer service

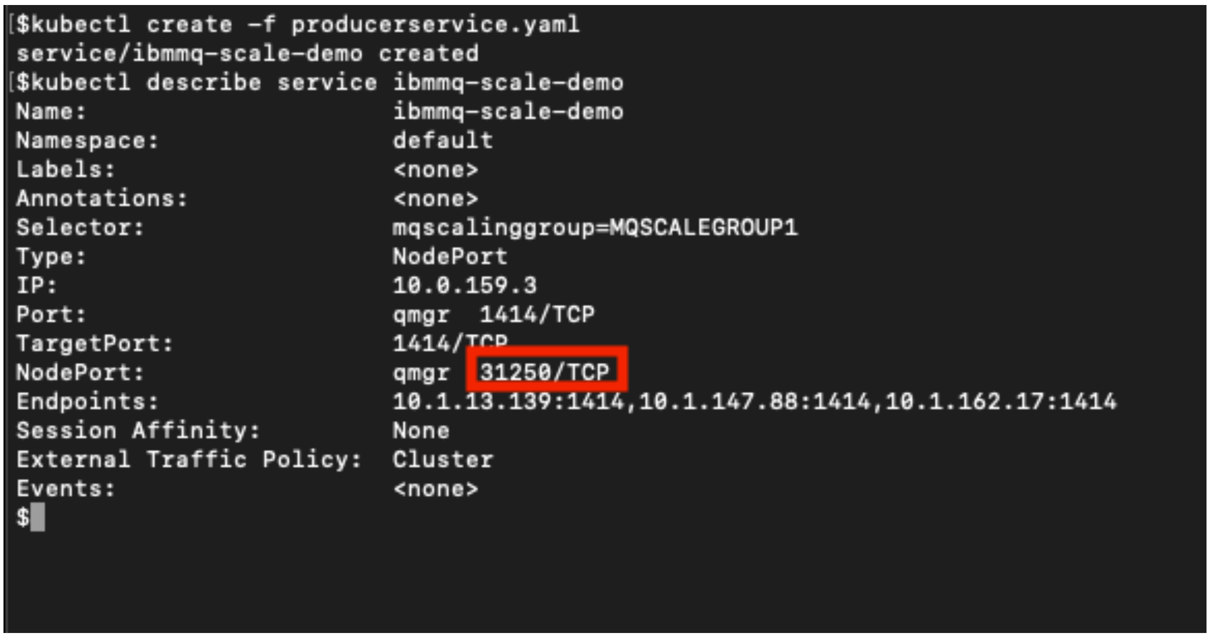
In the GitHub gist that you downloaded earlier you will find a file called *producerservice.yaml*. This file is used to create a kubernetes service called **ibmmq-scale-demo** which will forward connections to any containers that have the label **mqscalinggroup=MQSCALEGROUP1**.

To create the service, run the following command:

```
kubectl create -f producerservice.yaml
```

Because a service is created instantly, we don't need to wait for it to become ready. We will need to know the port number the service has bound to. To find out the port number run the following command:

```
kubectl describe service ibmmq-scale-demo
```



```
$kubectl create -f producerservice.yaml
service/ibmmq-scale-demo created
$kubectl describe service ibmmq-scale-demo
Name:                     ibmmq-scale-demo
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 mqscalinggroup=MQSCALEGROUP1
Type:                     NodePort
IP:                       10.0.159.3
Port:                     qmgr  1414/TCP
TargetPort:               1414/TCP
NodePort:                 qmgr  31250/TCP
Endpoints:                10.1.13.139:1414,10.1.147.88:1414,10.1.162.17:1414
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
$
```

In the picture above you will notice that the endpoints section displays 3 endpoints. The IP addresses and port numbers are the address of the 3 MQ CloudPak deployments. Additionally, you can see that the service is listening for traffic on the cluster port of **31250**. This is the port that we will use in the next section to test our deployment.

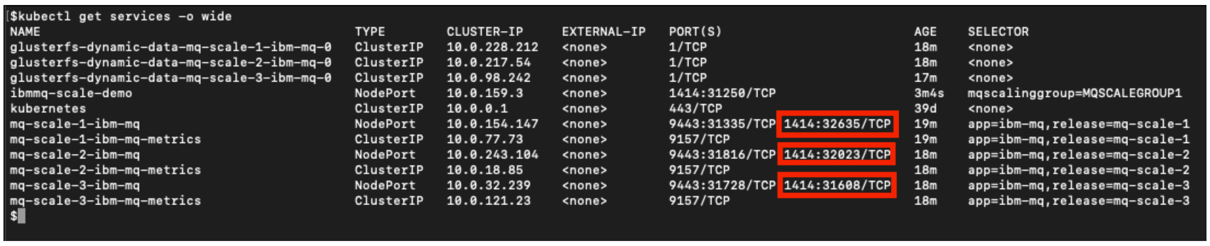
Testing it works

To test our full deployment we will connect 3 consumer applications, one to each queue manager, and then connect a producer application that will repeatadly put messages onto one of the default queues that are supplied with the IBM MQ Advanced for Developers CloudPak.

Finding our port numbers

In the previous sections we found out the port number for the kubernetes service that the producer application will use. We now need to find out the port number each consuming application will use. The consuming applications will connect directly to each queue manager, on a 1:1 basis. Thus, we need to find out the port numbers of the kubernetes service for each queue manager deployment. The following command will list all of the services that have been created, but also display the ports they are listening on (when they are of type NodePort):

```
kubectl get services -o wide
```



```
$kubectl get services -o wide
NAME                                       TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)                         AGE    SELECTOR
glusterfs-dynamic-data-mq-scale-1-ibm-mq-0 ClusterIP   10.0.228.212   <none>        1/TCP                           18m    <none>
glusterfs-dynamic-data-mq-scale-2-ibm-mq-0 ClusterIP   10.0.217.54    <none>        1/TCP                           18m    <none>
glusterfs-dynamic-data-mq-scale-3-ibm-mq-0 ClusterIP   10.0.98.242    <none>        1/TCP                           17m    <none>
ibmmq-scale-demo                           NodePort    10.0.159.3     <none>        1414:31250/TCP                  3m4s   mqscalinggroup=MQSCALEGROUP1
kubernetes                                 ClusterIP   10.0.0.1       <none>        443/TCP                         39d    <none>
mq-scale-1-ibm-mq                          NodePort    10.0.154.147   <none>        9443:31335/TCP,1414:32635/TCP   19m    app=ibm-mq,release=mq-scale-1
mq-scale-1-ibm-mq-metrics                  ClusterIP   10.0.77.73     <none>        9157/TCP                        19m    app=ibm-mq,release=mq-scale-1
mq-scale-2-ibm-mq                          NodePort    10.0.243.104   <none>        9443:31816/TCP,1414:32023/TCP   18m    app=ibm-mq,release=mq-scale-2
mq-scale-2-ibm-mq-metrics                  ClusterIP   10.0.18.85     <none>        9157/TCP                        18m    app=ibm-mq,release=mq-scale-2
mq-scale-3-ibm-mq                          NodePort    10.0.32.239    <none>        9443:31728/TCP,1414:31608/TCP   18m    app=ibm-mq,release=mq-scale-3
mq-scale-3-ibm-mq-metrics                  ClusterIP   10.0.121.23    <none>        9157/TCP                        18m    app=ibm-mq,release=mq-scale-3
$
```

From the picture above you can see numerous services in my cluster. The services that were created when we deployed the certified container will be called **mq-scale-1-ibm-mq**, **mq-scale-2-ibm-mq** and **mq-scale-3-ibm-mq**. In the output, look under the *ports* collumn to see what port has been exposed for the internal 1414 port. In my screenshot you can see i have the following ports:

- mq-scale-1-ibm-mq = 32635
- mq-scale-2-ibm-mq = 32023
- mq-scale-3-ibm-mq = 31608

We now have all the connection details we need in order to connect our producer and consumer applications.

Preparing our test application

Before we can connect the test applications we need to build them. In the gist you downloaded there is a dockerfile which is used to build a container image containing the test application. Run the following command to build it:

```
docker build -t mqtestapp .
```

The test application has two modes of operation:

- -put mode. In this mode it will constantly connect, place a message and disconnect until it is stopped.
- -get mode. In this mode it will constantly try and consume messages from a queue until it is stopped.

The IBM MQ Client best practices suggests that a client application should connect and then complete all of it's put or get operations before disconnect. The test application, in put mode, does not follow these best practices. Instead it will connect, complete one operation and then disconnect. The reason for this is because kubernetes services only route traffic on initial connection. To get the application to spread messages across all of the queue managers we must repeatadly connect and disconnect.

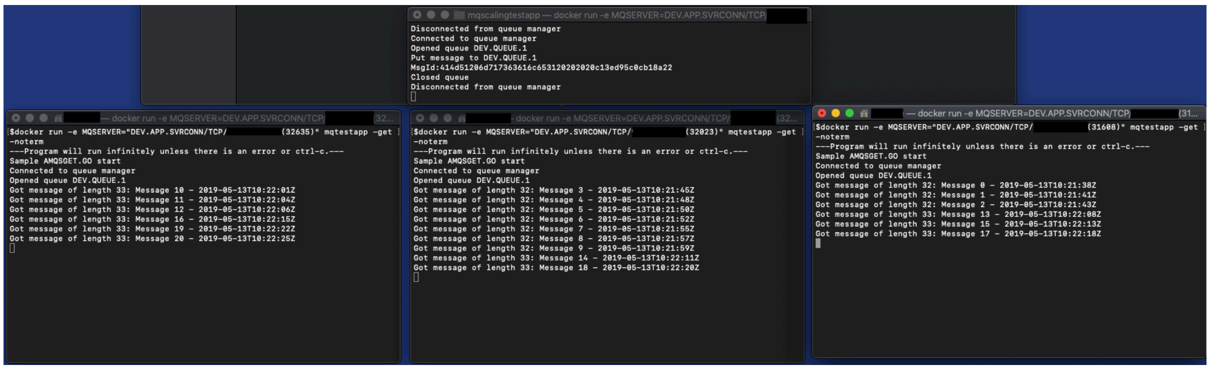
Running the test application

Run the following commands in 3 seperate terminal windows to connect the consumer applications to each queue manager. You will need to replace *-Port 1/2/3-* with your own port numbers for each queue manager and *-Cluster address-* with the IP address or hostname you use to connect to your ICP cluster:

```
docker run -e MQSERVER="DEV.APP.SVRCONN/TCP/-Cluster address-(-Port 1-)"
mqtestapp -get -noterm
docker run -e MQSERVER="DEV.APP.SVRCONN/TCP/-Cluster address-(-Port 2-)"
mqtestapp -get -noterm
docker run -e MQSERVER="DEV.APP.SVRCONN/TCP/-Cluster address-(-Port 3-)"
mqtestapp -get -noterm
```

The consumer applications will now connect and constantly looking for incoming messages. Next we have to connect our producer application. Run the following command to run the producer application using the port number we found in the “Creating your producer service” section.

```
docker run -e MQSERVER="DEV.APP.SVRCONN/TCP/-Cluster address-(-Port-)"
mqtestapp -put -noterm
```



```
mqscalingtestapp — docker run -e MQSERVER=DEV.APP.SVRCONN/TCP/[redacted]
Disconnected from queue manager
Connected to queue manager
Opened queue DEV.QUEUE.1
Put message to DEV.QUEUE.1
MsgId:414d51206d717363616c653120202020c13ed95c0cb18a22
Closed queue
Disconnected from queue manager
```

```
[redacted] — docker run -e MQSERVER=DEV.APP.SVRCONN/TCP/[redacted] (32...
$docker run -e MQSERVER="DEV.APP.SVRCONN/TCP/[redacted](32635)" mqtestapp -get
-noterm
---Program will run infinitely unless there is an error or ctrl-c.---
Sample AMQSGET.GO start
Connected to queue manager
Opened queue DEV.QUEUE.1
Got message of length 33: Message 10 - 2019-05-13T10:22:01Z
Got message of length 33: Message 11 - 2019-05-13T10:22:04Z
Got message of length 33: Message 12 - 2019-05-13T10:22:06Z
Got message of length 33: Message 16 - 2019-05-13T10:22:15Z
Got message of length 33: Message 19 - 2019-05-13T10:22:22Z
Got message of length 33: Message 20 - 2019-05-13T10:22:25Z
```

```
[redacted] - docker run -e MQSERVER=DEV.APP.SVRCONN/TCP/[redacted] (32...
$docker run -e MQSERVER="DEV.APP.SVRCONN/TCP/[redacted](32023)" mqtestapp -get
-noterm
---Program will run infinitely unless there is an error or ctrl-c.---
Sample AMQSGET.GO start
Connected to queue manager
Opened queue DEV.QUEUE.1
Got message of length 32: Message 3 - 2019-05-13T10:21:45Z
Got message of length 32: Message 4 - 2019-05-13T10:21:48Z
Got message of length 32: Message 5 - 2019-05-13T10:21:50Z
Got message of length 32: Message 6 - 2019-05-13T10:21:52Z
Got message of length 32: Message 7 - 2019-05-13T10:21:55Z
Got message of length 32: Message 8 - 2019-05-13T10:21:57Z
Got message of length 32: Message 9 - 2019-05-13T10:21:59Z
Got message of length 33: Message 14 - 2019-05-13T10:22:11Z
Got message of length 33: Message 18 - 2019-05-13T10:22:20Z
```

```
[redacted] — docker run -e MQSERVER=DEV.APP.SVRCONN/TCP/[redacted](31...
$docker run -e MQSERVER="DEV.APP.SVRCONN/TCP/[redacted](31608)" mqtestapp -get
-noterm
---Program will run infinitely unless there is an error or ctrl-c.---
Sample AMQSGET.GO start
Connected to queue manager
Opened queue DEV.QUEUE.1
Got message of length 32: Message 0 - 2019-05-13T10:21:38Z
Got message of length 32: Message 1 - 2019-05-13T10:21:41Z
Got message of length 32: Message 2 - 2019-05-13T10:21:43Z
Got message of length 33: Message 13 - 2019-05-13T10:22:08Z
Got message of length 33: Message 15 - 2019-05-13T10:22:13Z
Got message of length 33: Message 17 - 2019-05-13T10:22:18Z
```

You will notice that once the producer application starts pushing messages, the consumer applications will start receiving the messages. Although the messages may not be evenly spread across all 3 queue managers, you should see messages appearing on all 3 consumers.

Considerations with this design

Although this design allows for basic workload balancing with the out-of-the-box certified container there are some areas to consider:

- This method of workload balancing works best when there are many producer applications. The reason is this will maximise the likelihood of an even application distribution. If this does not match your application setup, then you may need to investigate cross-queue manager workload balancing using MQ Clustering, however this is a topic for another blog post.
- Before you reach the need to horizontally scale your queue managers, you will have needed to scale the number of consumer applications. The great thing with MQ is that you can scale applications and queue managers independantly. However, you do need to ensure that each of queues has at least one consuming application servicing it to prevent messages going unprocessed. We’ve done that manually here but you could use the new MQ capability to help simplify it that is detailed in this blog post and it is possible to take advantage of it in kubernetes too.
- We’ve configured the system to maximise workload balancing of MQ connections, which works well for applications such as these. However, there are times when MQ requires multiple MQ connections to be directed to the same queue manager, for example when using JMS. This can be overcome with session affinity but this will result in a single put application always going to the same queue manager

Conclusions

In conclusion this blog post shows how you can create a fixed group of IBM MQ CloudPak queue managers. Unfortuntely, it does not detail how you can change that number once they are deployed. In the next part of this blog post series we will show you how to scale up or down the number of queue managers in this scaling group.