Blockchain on vSphere

By VMware

Introduction

Blockchain is an emerging technology which has been gaining traction globally during the past few years. Industries like finance, logistics, IoT, are actively working on research and pilot projects using blockchain.

Despite the attention to blockchain, the installation and management of a blockchain service is very complicated and requires sophisticated domain knowledge. Blockchain on vSphere (BoV) is a tool for users to deploy Hyperledger Fabric v1.0 on vSphere platform. With only a few commands, a cluster of Fabric node is up and running on vSphere. The blockchain developers can quickly focus on implementation of the business logic.

Hyperledger is an open source project hosted by Linux Foundation. It was created to advance cross-industry blockchain technologies. It is by far the most mature open source project for enterprise to try and use distributed ledger technology(DLT).

Use Case

There are three personas of a blockchain service: Cloud Admin, Blockchain Admin, Blockchain Developer. They collaborate at different levels of the system. The Cloud Admin provisions and monitor the infrastructure such as vSphere. The Blockchain Admin manages the blockchain platform (Hyperledger Fabric). The Blockchain Developer focuses on application development by using the blockchain platform.

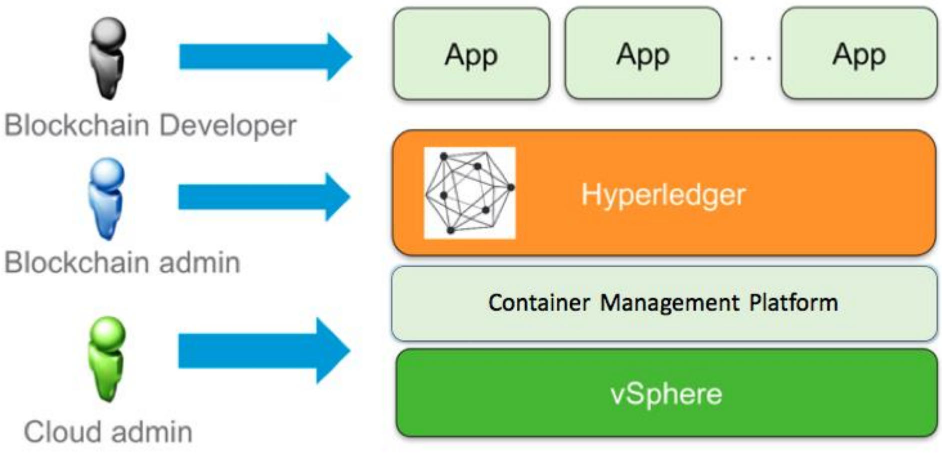


Figure 1

Architecture

Hyperledger Fabric is a distributed system implemented using containers. It can be deployed on the platform that support OCI container standard. Kubernetes will be used here to manage the Fabric containers. We have used the below architecture for Fabric v1.0:

- Use namespaces to maintain different Fabric organizations.
- Customizable number of peers in an organization.
- Isolation through Persistent Volume.

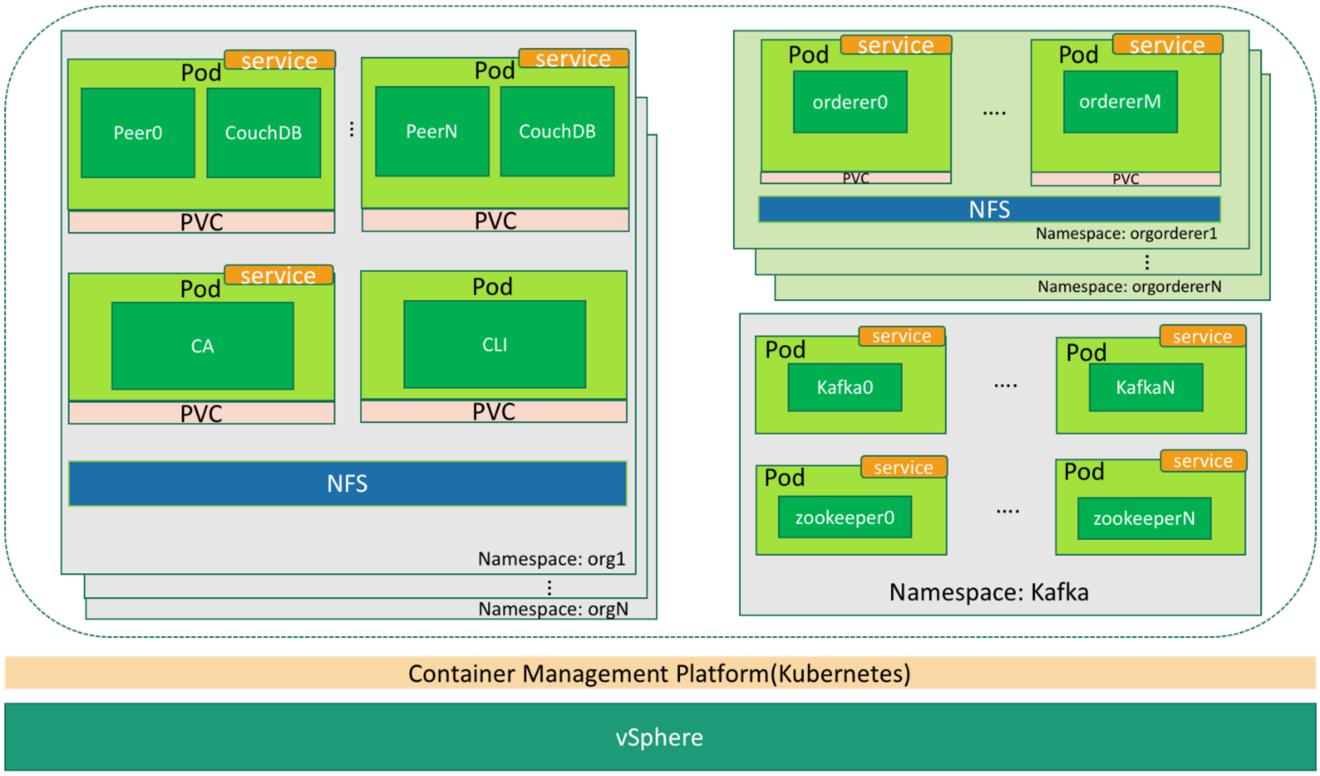


Figure 2

Deployment Instructions

Prerequisites:

1) vCenter 6.0+, with at least one ESXi host.

2) A NFS server for storing configuration files of the Fabric cluster.

3) Internet connection is required during installation.

4) A Linux host with Docker 1.11+ installed to run commands.

5) Python3.5 installed on the Linux host (with PyYAML module).

After you downloaded the package of BoV, you can follow the below steps to install Fabric 1.0 on vSphere. If you choose to use an existing Kubernetes cluster, you can start from Step 3.

1. Prepare vSphere environment

In vCenter, configure or create the below resources for Kubernetes:

- Create a resource pool for the Kubernetes, such as kub-pool;
- Select a datastore used by the Kubernetes, such as datastore1;
- Select a network for the Kubernetes nodes such as VM Network. The network must have DHCP service to provide dynamic IP address and can connect to internet.

2. Deployment Kubernetes

We will deploy Kubernetes by using open source project *Kubernetes Anywhere* (https://github.com/kubernetes/kubernetes-anywhere).

- Download this OVA template file and import it into vCenter. You will get a VM template named "KubernetesAnywhereTemplatePhotonOS". More information about importing the OVA can be found here: https://github.com/kubernetes/kubernetes-anywhere/blob/master/phase1/vsphere/README.md

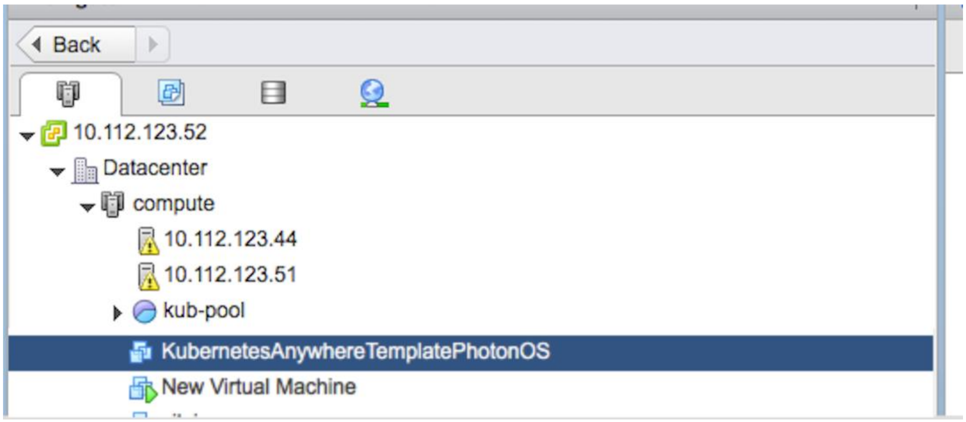


Figure 3

- On the Linux host, run the deployment container of *Kubernetes Anywhere* , you will stay inside the container's command prompt after the commands:

  $ docker pull cnastorage/kubernetes-anywhere

  $ docker run -it --rm --env="PS1=[container]:\w> " --net=host cnastorage/kubernetes-anywhere:v1 /bin/bash

- Inside the container start to deploy Kubernetes:
  - `[container]:/opt/kubernetes-anywhere> make deploy`
  - Input parameters according to the prompts. The following is an example:(to accept the default value, just press the enter key.)
    - number of nodes (phase1.num_nodes) [4] (NEW) **4**
    - cluster name (phase1.cluster_name) [kubernetes] (NEW) **kubernetes**
    - cloud provider: gce, azure or vsphere (phase1.cloud_provider) [gce] (NEW) **vsphere**
    - vCenter URL Ex: 10.192.10.30 or myvcenter.io (phase1.vSphere.url) [] (NEW) **10.112.123.52**
    - vCenter port (phase1.vSphere.port) [443] (NEW) **443**
    - vCenter username (phase1.vSphere.username) [] (NEW) **administrator@vsphere.local**
    - vCenter password (phase1.vSphere.password) [] (NEW) **MyPassword#3**
    - Does host use self-signed cert (phase1.vSphere.insecure) [Y/n/?] (NEW) Y

    Please note only input Y when your VC use a self-singed certificate(Please aware that use self-signed certificate are prone to the man in the middle attack).
    - Datacenter (phase1.vSphere.datacenter) [datacenter] (NEW) **Datacenter**
    - Datastore (phase1.vSphere.datastore) [datastore] (NEW) **datastore1**

- Deploy Kubernetes Cluster on 'host' or 'cluster' (phase1.vSphere.placement) [cluster] (NEW) **cluster**
- vspherecluster (phase1.vSphere.cluster) [] (NEW) **compute**
- Do you want to use the resource pool created on the host or cluster? [yes, no] (phase1.vSphere.useresourcepool) [no] (NEW) **yes**
- Name of the Resource Pool. If Resource pool is enclosed within another Resource pool, specify pool hierarchy as ParentResourcePool/ChildResourcePool (phase1.vSphere.resourcepool) (NEW) **kub-pool**
- Number of vCPUs for each VM (phase1.vSphere.vcpu) [1] (NEW) **4**
- Memory for each VM (phase1.vSphere.memory) [2048] (NEW) **4096**
- Network for each VM (phase1.vSphere.network) [VM Network] (NEW)**VM Network**
- Name of the template VM imported from OVA. If Template file is not available at the destination location specify vm path (phase1.vSphere.template) [KubernetesAnywhereTemplatePhotonOS.ova] (NEW) **KubernetesAnywhereTemplatePhotonOS**
- Flannel Network (phase1.vSphere.flannel_net) [172.1.0.0/16] (NEW)
- ** Phase 2: Node Bootstrapping*
- installer container (phase2.installer_container) [docker.io/cnastorage/k8s-ignition:v2] (NEW)
- docker registry (phase2.docker_registry) [gcr.io/google-containers] (NEW)
- kubernetes version (phase2.kubernetes_version) [v1.6.5] (NEW)
- bootstrap provider (phase2.provider) [ignition] (NEW)
- ** Phase 3: Deploying Addons.*
- Run the addon manager? (phase3.run_addons) [Y/n/?] (NEW)
- Run kube-proxy? (phase3.kube_proxy) [Y/n/?] (NEW)
- Run the dashboard? (phase3.dashboard) [Y/n/?] (NEW)
- Run heapster? (phase3.heapster) [Y/n/?] (NEW)
- Run kube-dns? (phase3.kube_dns) [Y/n/?] (NEW)
- Run weave-net? (phase3.weave_net) [N/y/?] (NEW) **N**

➢ Wait for the Kubernetes cluster to be created. Use the following two commands to check Kubernetes cluster status:

$ export KUBECONFIG=phase1/vsphere/.tmp/kubeconfig.json
$ kubectl cluster-info

It will display the cluster information similar to the below:

```
Kubernetes master is running at https://10.112.122.30
Heapster is running at https://10.112.122.30/api/v1/proxy/namespaces/kube-system/services/heapster
KubeDNS is running at https://10.112.122.30/api/v1/proxy/namespaces/kube-system/services/kube-dns

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Figure 4

➢ Copy the content of the file **phase1/vsphere/.tmp/kubeconfig.json** to your **Linux host** and save the content to the ~**/.kube/config .** You may need to create **.kube** directory if it does not exist on the Linux host:

$ scp phase1/vsphere/.tmp/kubeconfig.json username@LinuxHost:~/.kube/config

➢ Exit the container and return to the Linux host.

➢ On your Linux host run the following command:

$ kubectl cluster-info

You will get the same result as Figure 4.

❖ Configure the DNS used by Docker on all Kubernetes worker nodes.

The Fabric creates a Docker container to run chaincode which is out of the Kubernetes' control. Therefore, Docker daemon needs to include the correct DNS information of the Kubernetes network. It should include both the Kubernetes' DNS and the worker's DNS that used to reach the internet.

Make the follow change on all your Kubernetes worker nodes. (In this example, it includes node1, node2, node3 and node4)

1) Edit the /etc/default/docker file and change its content similar to the following:

DOCKER_OPTS=" --bip=172.1.42.1/24 --ip-masq=false --mtu=1472 --dns=10.0.0.10 --dns=10.117.0.1 --dns-search default.svc.cluster.local --dns-search svc.cluster.local \
    --dns-opt ndots:2 --dns-opt timeout:2 --dns-opt attempts:2 "

NOTE: In the above example, Kubernetes Anywhere set **10.0.0.10** as the DNS server of Kubernetes network. **10.117.0.1** is the DNS server of the worker node's network. Please replace the value based on your network configuration.

If you have a proxy server for Docker to pull images, please also add them in the /etc/default/docker file, e.g. :

HTTP_PROXY=http://yourproxyserver:3128
HTTPS_PROXY=https://yourproxyserver:3128

2) Restart the Docker service to allow the changes to take effect:

$ systemctl dockerd-daemon restart

$ systemctl dockerd restart

Now the Kubernetes installation has been completed.

- Update software in Kubernetes worker nodes.

As the kubernetes worker VMs are base on Photon ova which is created on Mar/2017 your are suggested to run "tdnf makecache && tdnf distro-sync" to keep the software keep to latested.

The default login info for the worker node is: root/kubernetes and your are strongly recommend to change those default password to a more secure one.

3. Deploy blockchain platform (Fabric)

- You need to setup a NFS service and export a shared directory (e.g. /opt/share). You can check the setting on your NFS server (10.112.122.9 in this example)

    root@ubuntu:~# showmount -e 10.112.122.9

    Export list for 10.112.122.9:

*/opt/share **

The NFS client needs to have read/write access to the /opt/share folder. If there is no authentication required by NFS (i.e. anonymous access), the folder's owner and group needs to be changed to **nobody:nogroup.** Otherwise the Kubernetes pods will encounter permission error. You can simply run the below command on the NFS server:

$ chmod -R nobody:nogroup /opt/share

- Mount the /opt/share to your Linux host:

  $ mount 10.112.122.9:/opt/share /opt/share

- Download the BoV package file Baas.tar and extract the files.

- Change the current directory to ./baas and run the command to download tools required by Fabric. Two tools, cryptogen and configtxgen, will be saved in ./bin directory.:

  $ cd ./baas
  $ curl
  https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/fabric/hyperledger-fabric/linux-amd64-1.0.0/hyperledger-fabric-linux-amd64-1.0.0.tar.gz | tar xz

  $ chmod +x ./bin/*

- In the setupCluster/templates/ directory, update two template files with your NFS server's IP address.

  1) fabric_1_0_template_pod_cli.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
    name: $artifactsName
spec:
    capacity:
       storage: 500Mi
    accessModes:
       - ReadWriteMany
    nfs:
      path: /opt/share/channel-artifacts
      server: 10.112.122.9 # change to you
---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    namespace: $namespace
    name: $artifactsName
spec:
   accessModes:
     - ReadWriteMany
   resources:
      requests:
        storage: 10Mi
```

Figure 5

  2) fabric_1_0_template_pod_namespace.yaml

```yaml
apiVersion: v1
kind: Namespace
metadata:
    name: $org

---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: $pvName
spec:
  capacity:
    storage: 500Mi
  accessModes:
    - ReadWriteMany
  nfs:
    path: $path
    server: 10.112.122.9  #ch
```

Figure 6

- The file setupCluster/cluster-config.yaml contains the definition of the blockchain service topology. You can modify it based on your requirement. Below is an example:

```
Tenant: f-1

OrdererOrgs:
  - Name: Orderer
    Domain: orgorderer
    Template:
      Count: 1               # 1 orderer,  change this number to create more orderers

PeerOrgs:
  - Name: Org1
    Domain: org1             # the domain name of Org1 is org1
    Template:
      Count: 2               # Org1 will have 2 peer nodes.
    Users:
      Count: 1
  - Name: Org2
    Domain: org2
    Template:
      Count: 2               # Org2 has 2 peer nodes
    Users:
      Count: 1
  - Name: Org3
    Domain: org3
    Template:
      Count: 3               # Org3 has 3 peer nodes
    Users:
      Count: 1
```

- Change directory to baas/setupCluster and run *generateAll.sh* to create all the Fabric configuration files and Kubernetes pods definition files for the blockchain service.

  $ cd baas/SetupCluster

  $ sudo bash generateAll.sh

  You can view the generated configuration files at the /opt/share directory.

- Change directory to the baas/setupCluster/transform and run the following command to deploy Fabric as pods on Kubernetes:

  $ python3.5 run.py

- Check the created blockchain pods:

  $ kubectl get pods --all-namespaces

```
root@master [ ~ ]# kubectl get pods --all-namespaces -o wide
NAMESPACE      NAME                                    READY   STATUS    RESTARTS   AGE   IP              NODE
kube-system    etcd-server-master                      1/1     Running   1          24d   10.112.122.30   master
kube-system    heapster-v1.2.0-867844254-wghgj         2/2     Running   14         24d   172.1.86.3      node3
kube-system    kube-apiserver-master                   1/1     Running   1          24d   10.112.122.30   master
kube-system    kube-controller-manager-master          1/1     Running   1          24d   10.112.122.30   master
kube-system    kube-dns-v19-l0bdz                      3/3     Running   15         18d   172.1.86.2      node3
kube-system    kube-dns-v19-ng1hc                      3/3     Running   21         24d   172.1.12.2      node2
kube-system    kube-proxy-044r8                        1/1     Running   7          24d   10.112.122.32   node2
kube-system    kube-proxy-0786x                        1/1     Running   2          17d   10.112.122.31   node4
kube-system    kube-proxy-dvnt2                        1/1     Running   8          24d   10.112.122.48   node3
kube-system    kube-proxy-n6rvx                        1/1     Running   1          24d   10.112.122.30   master
kube-system    kube-proxy-xrm5q                        1/1     Running   9          24d   10.112.122.40   node1
kube-system    kube-scheduler-master                   1/1     Running   1          24d   10.112.122.30   master
kube-system    kubernetes-dashboard-1019458639-1cx1v   1/1     Running   1          16d   172.1.42.2      node4
org1-f-1       ca-3347986348-9jvht                     1/1     Running   0          1h    172.1.73.5      node1
org1-f-1       cli-1569835662-01c5z                    1/1     Running   0          1h    172.1.12.5      node2
org1-f-1       peer0-org1-f-1-1343141255-h8kgk         2/2     Running   0          1h    172.1.86.6      node3
org1-f-1       peer1-org1-f-1-2603922830-35q6c         2/2     Running   0          1h    172.1.12.6      node2
org2-f-1       ca-2708682628-qpz64                     1/1     Running   0          1h    172.1.73.3      node1
org2-f-1       cli-2586364563-vclmr                    1/1     Running   0          1h    172.1.42.4      node4
org2-f-1       peer0-org2-f-1-3143546256-9prph         2/2     Running   0          1h    172.1.73.4      node1
org2-f-1       peer1-org2-f-1-110343575-06pvc          2/2     Running   0          1h    172.1.42.5      node4
org3-f-1       ca-349255610-628k1                      1/1     Running   0          1h    172.1.12.3      node2
org3-f-1       cli-3602893464-7f6g1                    1/1     Running   0          1h    172.1.73.2      node1
org3-f-1       peer0-org3-f-1-649967001-0v813          2/2     Running   0          1h    172.1.12.4      node2
org3-f-1       peer1-org3-f-1-1910748576-1j1bv         2/2     Running   0          1h    172.1.86.5      node3
org3-f-1       peer2-org3-f-1-3171530151-6whd0         2/2     Running   0          1h    172.1.42.3      node4
orgorderer-f-1 orderer0-orgorderer-f-1-73543963-plclz  1/1     Running   0          1h    172.1.86.4      node3
```

Figure 7

You can also check it through the Kubernetes dashboard UI (see Figure 8). If you don't know how to access the UI, just follow these steps:

Get NodePort mapping
kubectl describe service kubernetes-dashboard --namespace=kube-system| grep -i NodePort

Output:
Type: NodePort
NodePort: <unset> 31165/TCP

Get node it is running on
kubectl get pods --namespace=kube-system| grep -i dashboard

Output:
kubernetes-dashboard-1019458639-1cx1v 1/1 Running 1

kubectl describe pod kubernetes-dashboard-1763797262-fzla9 --namespace=kube-system| grep Node

Output:
Node: node4/10.112.122.31
Node-Selectors: <none>

Select the public IP for the node via or use govc or vCenter UI
kubectl describe node node1| grep Address

Open the <IP Addr>:<NodePort> in a browser

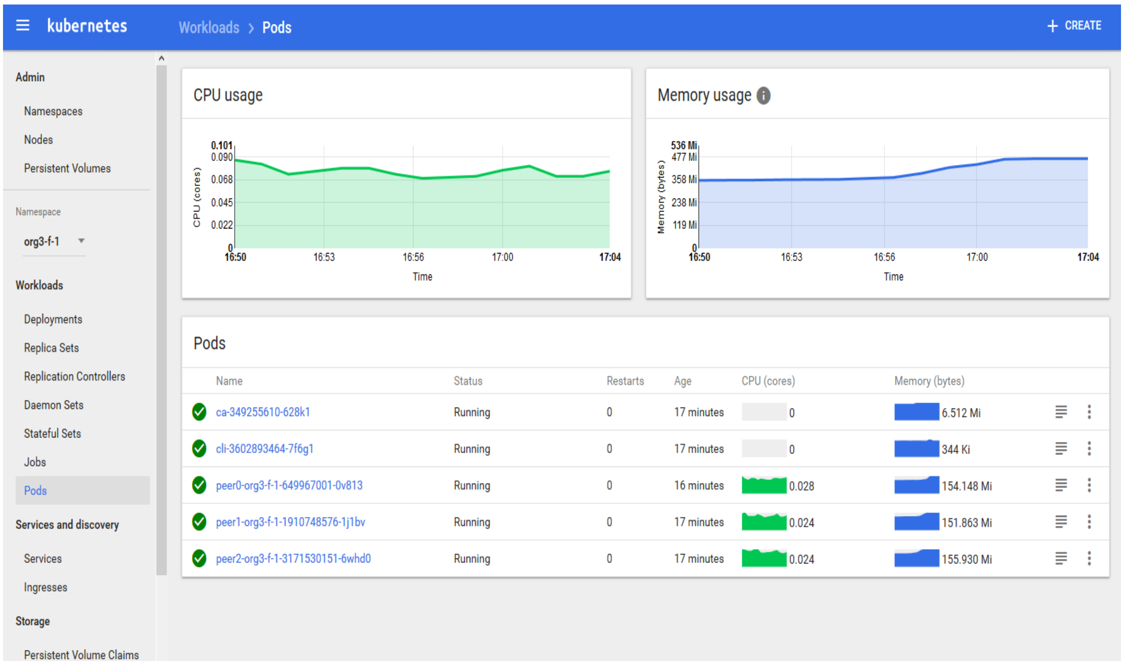


Figure 8

4. Verify the environment by running a sample chaincode

Please follows next few steps to create channel artifacts that are shared by cli all pods (two organizations in the example):

- Change working path to baas/setupCluster/ :

  $ cd baas/setupCluster/

- Create mychannel by using configtxgen:

  $ ../bin/configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/channel.tx -channelID mychannel

- Create configuration transaction for updating peer0.org1 to be the anchor peer of mychannel:

  $ ../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx -channelID mychannel -asOrg Org1MSP

- Create configuration transaction for updating peer0.org2 to be anchor peer of mychannel:

  $ ../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org2MSPanchors.tx -channelID mychannel -asOrg Org2MSP

- Copy *./channel-aritfacts* directory into */opt/share/* so that all cli pods can see it:

  $ cp -r ./channel-artifacts /opt/share

Download the chaincode example from Hyperledger [Fabric project](). Then copy the *chaincode_example02* directory into */opt/share/channel-aritfacts/* .

In the below commands, it first creates a channel, then it joins peer0 of Org1 into the channel. Next, it installs and instantiates the chaincode with two key/value pairs: a=*100* and b=*200*. After that, it adds peer0 of Org2 to the channel and installs the chaincode. Next it creates a transaction to transfer value 10 from *a* to *b*. It then queries the value of *a* which should be displayed as 90.

❖ Command for Org1:

1. Find out the cli pod instance of Org1:

   $ kubectl get pods –namespace=org1-f-1

```
root@master [ ~ ]# kubectl get pods --namespace org1-f-1
NAME                              READY     STATUS    RESTARTS   AGE
ca-3347986348-9jvht               1/1       Running   0          2h
cli-1569835662-01c5z              1/1       Running   0          2h
peer0-org1-f-1-1343141255-h8kgk   2/2       Running   0          2h
peer1-org1-f-1-2603922830-35q6c   2/2       Running   0          2h
root@master [ ~ ]#
```

Figure 9

   As Figure 9 shows, cli of org1 was named *cli-1569835662-01c5z*

2. Enter the cli pod of org1 for a Fabric command prompt:

   $ kubectl exec -it cli-1569835662-01c5z bash --namespace=org1-f-1

3. In the cli pod, create a channel named *mychannel*:

   $ peer channel create -o orderer0.orgorderer-f-1:7050 -c mychannel -f ./channel-artifacts/channel.tx

4. After the channel is created, a file *mychannel.block* from orderer is returned. It will be used by other peers to join the channel later. Figure 10 shows the output of channel creation command:

```
2017-08-07 08:52:11.803 UTC [main] main -> INFO 024 Exiting.....
root@cli-1569835662-01c5z:/opt/gopath/src/github.com/hyperledger/fabric/peer# ls
channel-artifacts  mychannel.block
root@cli-1569835662-01c5z:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

Figure 10

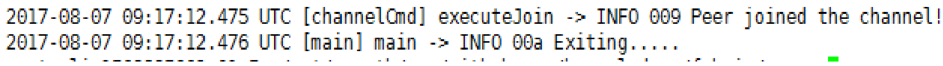
5. copy *mychannel.block* to NFS shared folder so that other cli can use it:

   $ cp mychannel.block ./channel-artifacts

6. Peer0 of Org1 joins the *mychannel*:

$ peer channel join -b ./channel-artifacts/mychannel.block

If everything worked fine, the output should look like Figure 11:



```
2017-08-07 09:17:12.475 UTC [channelCmd] executeJoin -> INFO 009 Peer joined the channel!
2017-08-07 09:17:12.476 UTC [main] main -> INFO 00a Exiting.....
```

Figure 11

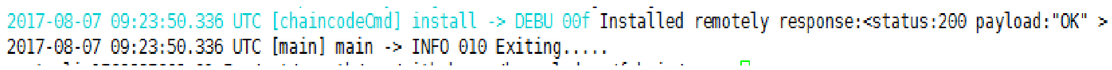
7. Update peer0 of Org1 as the anchor peer of *mychannel*:

$ peer channel update -o orderer0.orgorderer-f-1:7050 -c mychannel -f ./channel-artifacts/Org1MSPanchors.tx

8. Install chaincode *mycc* on peer0:

$ peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/peer/channel-artifacts/chaincode_example02

This output indicates the chaincode was installed successfully:



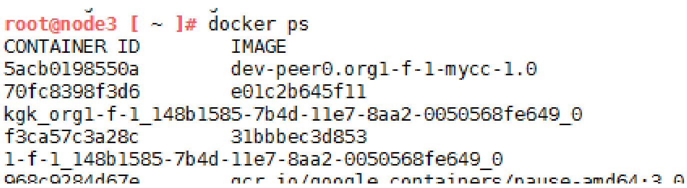
```
2017-08-07 09:23:50.336 UTC [chaincodeCmd] install -> DEBU 00f Installed remotely response:<status:200 payload:"OK" >
2017-08-07 09:23:50.336 UTC [main] main -> INFO 010 Exiting.....
```

Figure 12

9. Instantiate the chaincode *mycc*:

$ peer chaincode instantiate -o orderer0.orgorderer-f-1:7050 -C mychannel -n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"

A chaincode container was created. However, the container is not managed by Kubenetes due to the design of Fabric. The command above use pee0.org1(which is scheduled to node3) to instantiate chaincode. The chaincode container can be found by running "docker ps" at node3 like Figure 13:



```
root@node3 [ ~ ]# docker ps
CONTAINER ID        IMAGE
5acb0198550a        dev-peer0.org1-f-1-mycc-1.0
70fc8398f3d6        e01c2b645f11
kgk_org1-f-1_148b1585-7b4d-11e7-8aa2-0050568fe649_0
f3ca57c3a28c        31bbbec3d853
1-f-1_148b1585-7b4d-11e7-8aa2-0050568fe649_0
968c9284d67e        gcr.io/google_containers/pause-amd64:3.0
```

Figure 13

10. Query the chaincode:

$ peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'

Check the chaincode state, the value of "a" should be 100 as the instantiation defined:

```
Query Result: 100
2017-08-07 10:02:03.710 UTC [main] main -> INFO 008 Exiting.....
```

Figure 14

❖ Similar commands can be applied in Org2:

1. Find out the cli pod instance of Org2:

$ kubectl get pods –namespace=org2-f-1

```
root@master [ ~ ]# kubectl get pods --namespace org2-f-1
NAME                              READY     STATUS    RESTARTS   AGE
ca-2708682628-qpz64               1/1       Running   0          2h
cli-2586364563-vclmr              1/1       Running   0          2h
peer0-org2-f-1-3143546256-9prph   2/2       Running   0          2h
peer1-org2-f-1-110343575-06pvc    2/2       Running   0          2h
```

Figure 15

As Figure 15 showed, cli of org2 was named cli-2586364563-vclmr

2. Enter the cli pod of org2 for a Fabric command prompt:

$ kubectl exec -it cli-2586364563-vclmr --namespace=org2-f-1   bash

3. In cli pod of org2, Peer0 of org2 joins the channel:

$ peer channel join -b ./channel-artifacts/mychannel.block

```
2017-08-07 09:17:12.475 UTC [channelCmd] executeJoin -> INFO 009 Peer joined the channel!
2017-08-07 09:17:12.476 UTC [main] main -> INFO 00a Exiting.....
```

Figure 16

4. Update Peer0 of org2 as an anchor peer:

$ peer channel update -o orderer0.orgorderer-f-1:7050 -c mychannel -f ./channel-artifacts/Org2MSPanchors.tx

5. Install chaincode:

$ peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/peer/channel-artifacts/chaincode_example02

```
2017-08-07 09:23:50.336 UTC [chaincodeCmd] install -> DEBU 00f Installed remotely response:<status:200 payload:"OK" >
2017-08-07 09:23:50.336 UTC [main] main -> INFO 010 Exiting.....
```

Figure 17

6. Query the chaincode:

$ peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'

Since chaincode *mycc* was instantiated, this command returns the same "a" value as instantiation defined:

```
Query Result: 100
2017-08-07 10:02:03.710 UTC [main] main -> INFO 008 Exiting.....
```

Figure 18

7. Invoke the chaincode to update the ledger of the channel. This creates a transaction to transfer value 10 from *a* to *b.*:

$ peer chaincode invoke -o orderer0.orgorderer-f-1:7050 -C mychannel -n mycc -c '{"Args":["invoke","a","b","10"]}'

8. Query "a" again, it should be 90 now:

$ peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}

```
Query Result: 90
2017-08-07 10:13:01.232 UTC [main] main -> INFO 008 Exiting.....
```

Figure 19

Now you have created a fully functional blockchain service and run a chaincode example on the service.