Table of Contents *generated with DocToc*

– Exercise

Minikube

Objective

Using Minikube to create a single node Kubernetes cluster on your laptop, and run an example application to validate your installation works.

Note, a lot of resources are blocked by GFW, e.g. minikube iso, gcr.io docker images. Simply following upstream guide is not enough to run a local kubernetes.

Prerequisites

Minikube Binary

For macos, install homebrew, then:

```
brew cask install minikube
```

For linux,

```
curl -Lo minikube http://ozqvc9zbu.bkt.clouddn.com/minikube && chmod +x minikube && sudo mv
```

For windows,

```
curl -LO http://ozqvc9zbu.bkt.clouddn.com/minikube-windows-amd64.exe
```

Virtualbox

Minikube is running in a virtual machine. For macos, you can choose xhyve driver, VirtualBox or VMware. For linux, the choices are VirtualBox and KVM. You can also use minikube in linux without using hypervisor. For simplicity, we use VirtualBox for both environments.

Kubectl Binary

For macos, run:

```
brew cask install kubectl
```

For linux, run

```
curl -Lo kubectl http://ozqvc9zbu.bkt.clouddn.com/kubectl && chmod +x kubectl && sudo mv kub
```

For windows,

```
curl -LO http://ozqvc9zbu.bkt.clouddn.com/kubectl.exe
```

Start minikube

Make sure minkube version is at least v0.22.0:

```
$ minikube version
minikube version: v0.22.0
```

If not, please run upgrade (re-install) it first.

Then download cache for minikube iso and docker images.

```
mkdir -p ~/.minikube/cache
wget http://7xli2p.dl1.z0.glb.clouddn.com/minikube-v0.22.0-cache.tar.gz -O - | tar zx -C ~/
```

Start minikube:

```
$ minikube start
Starting local Kubernetes v1.7.0 cluster...
Starting VM...
SSH-ing files into VM...
Setting up certs...
Starting cluster components...
Connecting to cluster...
Setting up kubeconfig...
Kubectl is now configured to use the cluster.
```

Verify installation

We can verify installation via:

```
$ kubectl get nodes
NAME       STATUS    AGE       VERSION
minikube   Ready     26d       v1.7.0
```

Manual

Objective

Install Kubernetes entirely manually, without the help of any other tools (like kubeadm, Ansible playbook, etc). Essentially, these tools just automate the manual process.

To do this, the instructions provided in the kubernetes the hard-way is very useful. That guide runs a Kubernetes cluster from scratch using GCE VMs, and

configures a highly available control plane with a public external loadbalancer. In this lab, we'll use bare-metal VMs. To simplify the process, we use a single master setup. The following guide is a modified version from Kelsey's guide.

Prerequisites

As mentioned in lab requirement doc, you'll need: * At least two VMs with at least 2C 4G running CentOS 7 * The VMs must have internet connection * The VMs can access each other, and firewall is disabled * The VMs can access each other using respective hostname; if VM1 hostname is `master` and VM2 hostname is `worker-1`, then VM1 should be able to ping VM2 using `ping worker-1`

Assumptions

As we've already mentioned (slide chapter 3), there's a couple of considerations before installing a Kubernetes cluster, below we briefly outline our choices: * The Lab is based on Kubernetes 1.8 * Cluster Pod CIDR is 10.244.0.0/16 * worker-1 CIDR: 10.254.1.0/24, worker-2 CIDR: 10.254.2.0/24, etc * Cluster Service CIDR is 10.250.0.0/24 * 'kubernetes' service will be running at 10.250.0.1 * 'dns' service will be running at 10.250.0.10 * Network plugin is canal

All the operations below need to be carried out on master, unless otherwise noted.

Install client tools

We'll need to install three client tools: * kubectl - the Kubernetes cli * cfs and cfssljson - provision a PKI Infrastructure and generate TLS certificates.

Install cfssl and cfssljson

```
wget --timestamping \
  http://ozqvc9zbu.bkt.clouddn.com/cfssl_linux-amd64 \
  http://ozqvc9zbu.bkt.clouddn.com/cfssljson_linux-amd64

chmod +x cfssl_linux-amd64 cfssljson_linux-amd64
sudo mv cfssl_linux-amd64 /usr/local/bin/cfssl
sudo mv cfssljson_linux-amd64 /usr/local/bin/cfssljson
```

Install kubectl (version 1.8)

```
wget --timestamping \
```

```
  http://ozqvc9zbu.bkt.clouddn.com/kubectl

chmod +x kubectl
sudo mv kubectl /usr/local/bin/
```

Provision Certificates

It is recommended to use HTTPS for inter-component communication in Kubernetes, as well as accessing Kubernetes API from outside world. Here we need to provision a CA and generate certificates for this purpose.

Certificate Authority

The certificate authority is used to generate additional TLS certificates. It will be trusted across all components in Kubernetes.

Create the CA configuration file:

```
cat > ca-config.json <<EOF
{
  "signing": {
    "default": {
      "expiry": "8760h"
    },
    "profiles": {
      "kubernetes": {
        "usages": ["signing", "key encipherment", "server auth", "client auth"],
        "expiry": "8760h"
      }
    }
  }
}
EOF
```

Create the CA certificate signing request:

```
cat > ca-csr.json <<EOF
{
  "CN": "Kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "China",
```

```
      "L": "Shanghai",
      "O": "Kubernetes",
      "OU": "Shanghai",
      "ST": "Shanghai"
    }
  ]
}
EOF
```

Now generates the CA certificate:

```
cfssl gencert -initca ca-csr.json | cfssljson -bare ca
```

Send to workers:

```
scp ca* root@${WORKER1_IP}:
```

The result is two files, one certificate and one private key, i.e. `ca-key.pem` and `ca.pem`.

Client and Server Certificates

Below we will generate client and server certificates for each Kubernetes component and a client certificate for the Kubernetes admin user.

Create admin user client certificate:

```
cat > admin-csr.json <<EOF
{
  "CN": "admin",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "China",
      "L": "Shanghai",
      "O": "system:masters",
      "OU": "Kubernetes",
      "ST": "Shanghai"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
```

```
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  admin-csr.json | cfssljson -bare admin
```

The result is three files, certificate signing request, admin private key and admin certificate, i.e. admin.csr, admin-key.pem and admin.pem.

Kubelet client certificates

As described in Kubernetes the hard-way, each kubelet is identified as a unique worker. It is authorized by the Node Authorizer - kubelet needs to identified itself in `system:nodes` group, with a username of `system:node:<nodeName>`.

Since each kubelet needs a certificate, we'll need to generate multiple certificates. Below we demonstrate the process for one worker node - others will be the same. In the guide, we use `${WORKER1_HOSTNAME}` and `${WORKER1_IP}` to denote the worker node's hostname and IP address. In the following code snippet, we create worker config, generate certificates and copy them to worker node:

```
cat > worker-1-csr.json <<EOF
{
  "CN": "system:node:${WORKER1_HOSTNAME}",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "China",
      "L": "Shanghai",
      "O": "system:nodes",
      "OU": "Kubernetes",
      "ST": "Shanghai"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -hostname=${WORKER1_HOSTNAME},${WORKER1_IP} \
  -profile=kubernetes \
  worker-1-csr.json | cfssljson -bare worker-1
```

Send to workers:

```
scp worker-1* root@${WORKER1_IP}:
```

Generate kube proxy client certificates

Below we generate client for kube-proxy. Unlike kubelet, all proxies use the same certificate.

```
cat > kube-proxy-csr.json <<EOF
{
  "CN": "system:kube-proxy",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "China",
      "L": "Shanghai",
      "O": "system:node-proxier",
      "OU": "Kubernetes",
      "ST": "Shanghai"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  kube-proxy-csr.json | cfssljson -bare kube-proxy
```

Send to workers:

```
scp kube-proxy* root@${WORKER1_IP}:
```

Generate API server certificates

Below we generate Kubernetes API server certificates, used for clients and various other components to connect to API server. We'll use `${MASTER_IP}` to denote master's IP address. Note that when generating the certificates, the hostname includes an IP address "10.250.0.1", which is the IP address for `kubernetes` service - the service is used for Pod to connect to Kubernetes API server.

```
cat > kubernetes-csr.json <<EOF
{
  "CN": "kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "China",
      "L": "Shanghai",
      "O": "Kubernetes",
      "OU": "Kubernetes",
      "ST": "Shanghai"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -hostname=10.250.0.1,${MASTER_IP},127.0.0.1,kubernetes.default \
  -profile=kubernetes \
  kubernetes-csr.json | cfssljson -bare kubernetes
```

Kubernetes Configuration Files

The Kubernetes configuration files are used for clients to connect to Kubernetes server. Kubernetes components, as well as kubectl, accept the configuration file; they will parse the information provided in the file to access Kubernetes cluster.

The Kubelet Configuration File

As above, we'll only do this for one worker; please repeat the process for other nodes. Make sure to send the config to workers at last (the commands are running on master).

```
kubectl config set-cluster kubernetes-training \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=https://${MASTER_IP}:6443 \
  --kubeconfig=worker-1.kubeconfig
```

```
kubectl config set-credentials system:node:worker-1 \
  --client-certificate=worker-1.pem \
  --client-key=worker-1-key.pem \
  --embed-certs=true \
  --kubeconfig=worker-1.kubeconfig

kubectl config set-context default \
  --cluster=kubernetes-training \
  --user=system:node:worker-1 \
  --kubeconfig=worker-1.kubeconfig

kubectl config use-context default --kubeconfig=worker-1.kubeconfig
```

Send to workers:

```
scp worker-1.kubeconfig root@${WORKER1_IP}:
```

The Kube-proxy Configuration File

Generating Kube-proxy configuration file is similar to Kubelet:

```
kubectl config set-cluster kubernetes-training \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=https://${MASTER_IP}:6443 \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config set-credentials kube-proxy \
  --client-certificate=kube-proxy.pem \
  --client-key=kube-proxy-key.pem \
  --embed-certs=true \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config set-context default \
  --cluster=kubernetes-training \
  --user=kube-proxy \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config use-context default --kubeconfig=kube-proxy.kubeconfig
```

Send to workers:

```
scp kube-proxy.kubeconfig root@${WORKER1_IP}:
```

Data Encryption Key

As mentioned in the slides, Kubernetes stores all metadata in etcd, a key/value store. In Kubernetes 1.7, a new feature is introduced which supports encrypting API resource before persistence. More information about the topic can be found here.

```
ENCRYPTION_KEY=$(head -c 32 /dev/urandom | base64)

cat > encryption-config.yaml <<EOF
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
      - secrets
    providers:
      - aescbc:
          keys:
            - name: key1
              secret: ${ENCRYPTION_KEY}
      - identity: {}
EOF
```

Bootstrapping the etcd instance

Here we will run a single etcd instance at master node. In production deployment, you can choose to run an etcd cluster (usually three etcd instances), and you can choose to run etcd either on master or on separate dedicated machines.

Download and Install etcd

You only need to install etcd on master node.

```
wget --timestamping \
  http://ozqvc9zbu.bkt.clouddn.com/etcd-v3.2.8-linux-amd64.tar.gz

tar -xvf etcd-v3.2.8-linux-amd64.tar.gz
sudo mv etcd-v3.2.8-linux-amd64/etcd* /usr/local/bin/
rm -rf etcd-v3.2.8-linux-amd64 etcd-v3.2.8-linux-amd64.tar.gz
```

Configure and Start etcd Server

Once etcd is installed, we configure systemd unit and start etcd server.

```
sudo mkdir -p /etc/etcd /var/lib/etcd
sudo cp ca.pem kubernetes-key.pem kubernetes.pem /etc/etcd/

cat > etcd.service <<EOF
[Unit]
Description=etcd
Documentation=https://github.com/coreos
[Service]
ExecStart=/usr/local/bin/etcd \\
  --name master \\
  --cert-file=/etc/etcd/kubernetes.pem \\
  --key-file=/etc/etcd/kubernetes-key.pem \\
  --peer-cert-file=/etc/etcd/kubernetes.pem \\
  --peer-key-file=/etc/etcd/kubernetes-key.pem \\
  --trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://${MASTER_IP}:2380 \\
  --listen-peer-urls https://${MASTER_IP}:2380 \\
  --listen-client-urls https://${MASTER_IP}:2379,http://127.0.0.1:2379 \\
  --advertise-client-urls https://${MASTER_IP}:2379,http://127.0.0.1:2379 \\
  --data-dir=/var/lib/etcd
Restart=on-failure
RestartSec=5
[Install]
WantedBy=multi-user.target
EOF

sudo mv etcd.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable etcd
sudo systemctl start etcd
```

Verify etcd Server

To validate etcd is running, run:

```
ETCDCTL_API=3 etcdctl member list
```

Bootstrapping the Kubernetes Master Nodes

Below we'll run Kubernetes control plane on master node. Recall that Kubernetes control plane contains Kubernetes API server, scheduler and controller manager. Note that we are bringing up Kubernetes cluster using systemd with component

specific binaries, another common practice nowadays is to run these components using Pod, managed by Kubernetes itself, aka, self-hosting.

Download and install binaries

```
wget --timestamping \
  "http://ozqvc9zbu.bkt.clouddn.com/kube-apiserver" \
  "http://ozqvc9zbu.bkt.clouddn.com/kube-scheduler" \
  "http://ozqvc9zbu.bkt.clouddn.com/kube-controller-manager"

chmod +x kube-apiserver kube-controller-manager kube-scheduler
sudo mv kube-apiserver kube-controller-manager kube-scheduler /usr/local/bin/
```

Configure API Server

Below we start Kubernetes API server using systemd. Refer to official documentation for command line arguments. Please pay attention to following options:

- admission-control
- service-cluster-ip-range
- service-node-port-range

```
sudo mkdir -p /var/lib/kubernetes/
sudo cp ca.pem ca-key.pem kubernetes-key.pem kubernetes.pem encryption-config.yaml \
  /var/lib/kubernetes/

cat > kube-apiserver.service <<EOF
[Unit]
Description=Kubernetes API Server
Documentation=https://github.com/kubernetes/kubernetes
[Service]
ExecStart=/usr/local/bin/kube-apiserver \\
  --admission-control=Initializers,NamespaceLifecycle,NodeRestriction,LimitRanger,ServiceAcc
  --advertise-address=${MASTER_IP} \\
  --allow-privileged=true \\
  --audit-log-maxage=30 \\
  --audit-log-maxbackup=3 \\
  --audit-log-maxsize=100 \\
  --audit-log-path=/var/log/audit.log \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
```

```
  --etcd-certfile=/var/lib/kubernetes/kubernetes.pem \\
  --etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \\
  --etcd-servers=http://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --experimental-encryption-provider-config=/var/lib/kubernetes/encryption-config.yaml \\
  --insecure-bind-address=127.0.0.1 \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-certificate=/var/lib/kubernetes/kubernetes.pem \\
  --kubelet-client-key=/var/lib/kubernetes/kubernetes-key.pem \\
  --kubelet-https=true \\
  --runtime-config=api/all \\
  --service-account-key-file=/var/lib/kubernetes/ca-key.pem \\
  --service-cluster-ip-range=10.250.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --tls-ca-file=/var/lib/kubernetes/ca.pem \\
  --tls-cert-file=/var/lib/kubernetes/kubernetes.pem \\
  --tls-private-key-file=/var/lib/kubernetes/kubernetes-key.pem \\
  --v=2
Restart=on-failure
RestartSec=5
[Install]
WantedBy=multi-user.target
EOF

sudo mv kube-apiserver.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable kube-apiserver
sudo systemctl start kube-apiserver
```

To verify, run "kubectl get componentstatuses".

Configure Controller Manager

Refer to official documentation for command line options.

```
cat > kube-controller-manager.service <<EOF
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/kubernetes/kubernetes
[Service]
ExecStart=/usr/local/bin/kube-controller-manager \\
  --address=0.0.0.0 \\
  --allocate-node-cidrs=true \\
  --cluster-cidr=10.244.0.0/16 \\
  --cluster-name=kubernetes \\
  --cluster-signing-cert-file=/var/lib/kubernetes/ca.pem \\
```

```
  --cluster-signing-key-file=/var/lib/kubernetes/ca-key.pem \\
  --leader-elect=true \\
  --master=http://127.0.0.1:8080 \\
  --root-ca-file=/var/lib/kubernetes/ca.pem \\
  --service-account-private-key-file=/var/lib/kubernetes/ca-key.pem \\
  --service-cluster-ip-range=10.250.0.0/24 \\
  --v=2
Restart=on-failure
RestartSec=5
[Install]
WantedBy=multi-user.target
EOF

sudo mv kube-controller-manager.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable kube-controller-manager
sudo systemctl start kube-controller-manager
```

To verify, run "kubectl get componentstatuses".

Configure Scheduler

Refer to official documentation for command line options.

```
cat > kube-scheduler.service <<EOF
[Unit]
Description=Kubernetes Scheduler
Documentation=https://github.com/kubernetes/kubernetes
[Service]
ExecStart=/usr/local/bin/kube-scheduler \\
  --leader-elect=true \\
  --master=http://127.0.0.1:8080 \\
  --v=2
Restart=on-failure
RestartSec=5
[Install]
WantedBy=multi-user.target
EOF

sudo mv kube-scheduler.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable kube-scheduler
sudo systemctl start kube-scheduler
```

To verify, run "kubectl get componentstatuses".

Bootstrapping the Kubernetes Worker Nodes

Note all the following commands need to be ran in worker nodes.

Download and Install Binaries

The following commands will install docker. It is worth note that cri-containerd, a project under incubation in Kubernetes organization, is in the process of replacing docker; but for now, a lot of production deployment still use docker.

```
sudo yum install -y yum-utils device-mapper-persistent-data lvm2

sudo yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-c

sudo yum makecache fast
sudo yum -y install docker-ce

sudo systemctl enable docker
sudo systemctl start docker
```

The following commands will install kubelet, kube-proxy.

```
sudo yum install socat

wget --timestamping \
  http://ozqvc9zbu.bkt.clouddn.com/kube-proxy \
  http://ozqvc9zbu.bkt.clouddn.com/kubectl \
  http://ozqvc9zbu.bkt.clouddn.com/kubelet

sudo mkdir -p \
  /etc/cni/net.d \
  /opt/cni/bin \
  /var/lib/kubelet \
  /var/lib/kube-proxy \
  /var/lib/kubernetes \
  /var/run/kubernetes

chmod +x kubectl kube-proxy kubelet
sudo mv kubectl kube-proxy kubelet /usr/local/bin/
```

Configure the Kubelet

Kubelet is the central unit for managing container workloads. Refer to the official documentation for command line options.

```
sudo cp worker-1-key.pem worker-1.pem /var/lib/kubelet/
```

```
sudo cp worker-1.kubeconfig /var/lib/kubelet/kubeconfig
sudo cp ca.pem /var/lib/kubernetes/

cat > kubelet.service <<EOF
[Unit]
Description=Kubernetes Kubelet
Documentation=https://github.com/kubernetes/kubernetes
After=docker.service
Requires=docker.service
[Service]
ExecStart=/usr/local/bin/kubelet \\
  --allow-privileged=true \\
  --anonymous-auth=false \\
  --authorization-mode=Webhook \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --cluster-dns=10.250.0.10 \\
  --cluster-domain=cluster.local \\
  --image-pull-progress-deadline=2m \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --register-node=true \\
  --require-kubeconfig \\
  --runtime-request-timeout=15m \\
  --pod-infra-container-image=cargo.caicloud.io/caicloud/pause-amd64:3.0 \\
  --tls-cert-file=/var/lib/kubelet/worker-1.pem \\
  --tls-private-key-file=/var/lib/kubelet/worker-1-key.pem \\
  --v=2
Restart=on-failure
RestartSec=5
[Install]
WantedBy=multi-user.target
EOF

# Turnning off swap is required by kubelet.
swapoff -a

sudo mv kubelet.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable kubelet
sudo systemctl start kubelet
```

Configure the Kube-proxy

Refer to the official documentation for command line options.

```
sudo cp kube-proxy.kubeconfig /var/lib/kube-proxy/kubeconfig
sudo cp worker-1-key.pem worker-1.pem /var/lib/kubelet/

cat > kube-proxy.service <<EOF
[Unit]
Description=Kubernetes Kube Proxy
Documentation=https://github.com/kubernetes/kubernetes
[Service]
ExecStart=/usr/local/bin/kube-proxy \\
  --cluster-cidr=10.244.0.0/16 \\
  --kubeconfig=/var/lib/kube-proxy/kubeconfig \\
  --proxy-mode=iptables \\
  --v=2
Restart=on-failure
RestartSec=5
[Install]
WantedBy=multi-user.target
EOF

sudo mv kube-proxy.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable kube-proxy
sudo systemctl start kube-proxy
```

RBAC for Kubelet Authorization

Kubernetes API server will connect to kubelet to retrieve logs, metrics, etc. By default, kubelet doesn't allow such access, we need to authorize the action.

This tutorial sets the Kubelet `--authorization-mode` flag to Webhook. Webhook mode uses the `SubjectAccessReview` API to determine authorization. When client (here API server) connects to Kubelet, it will post an object similar to the following:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "kube-system",
      "verb": "get",
      "group": "*",
      "resource": "pods"
    },
    "user": "jane",
    "group": [
```

```
      "group1",
      "group2"
    ]
  }
}
```

This verifies if user `jane`, group `group1` and `group2` are allowed to get pods in `kube-system` namespace.

In our setup, we create the `system:kube-apiserver-to-kubelet` ClusterRole with permissions to access the Kubelet API and perform most common tasks associated with managing pods:

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:kube-apiserver-to-kubelet
rules:
  - apiGroups:
      - ""
    resources:
      - nodes/proxy
      - nodes/stats
      - nodes/log
      - nodes/spec
      - nodes/metrics
    verbs:
      - "*"
EOF
```

The Kubernetes API Server authenticates to the Kubelet as the `kubernetes` user using the client certificate as defined by the `--kubelet-client-certificate` flag.

Bind the `system:kube-apiserver-to-kubelet` ClusterRole to the `kubernetes` user:

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: system:kube-apiserver
  namespace: ""
roleRef:
```

```
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:kube-apiserver-to-kubelet
subjects:
  - apiGroup: rbac.authorization.k8s.io
    kind: User
    name: kubernetes
EOF
```

Provision Pod Network

As of now, Node is not ready since we haven't provisioned Pod Network. We haven't talked about setting up Pod network, so for now, just apply the following yaml files from canal.

```
kubectl apply -f http://ozqvc9zbu.bkt.clouddn.com/canal-rbrc.yaml
kubectl apply -f http://ozqvc9zbu.bkt.clouddn.com/canal.yaml
```

To verify that pod network works properly, run `kubectl get pods -n kube-system`, you should see following results:

```
[root@master ~]# kubectl get pods -n kube-system
NAME          READY     STATUS    RESTARTS   AGE
canal-mgzj9   3/3       Running   7          12h
```

Deploying the DNS Add-on

DNS add-on is used to provide DNS-based service discovery in a Kubernetes cluster. Deploying DNS add-on is as simple as creating a set of Pods.

```
kubectl apply -f http://ozqvc9zbu.bkt.clouddn.com/kube-dns.yaml
```

To verify DNS is working properly, we can run a busybox Pod and inspect the result of nslookup.

```
kubectl run busybox --image=cargo.caicloud.io/caicloud/busybox:1.24 --command -- sleep 3600

POD_NAME=$(kubectl get pods -l run=busybox -o jsonpath="{.items[0].metadata.name}")

kubectl exec -ti $POD_NAME -- nslookup kubernetes
```

You should see the following result:

```
Server:    10.250.0.10
Address 1: 10.250.0.10 kube-dns.kube-system.svc.cluster.local

Name:      kubernetes
Address 1: 10.250.0.1 kubernetes.default.svc.cluster.local
```

Smoke Test

Now that we have a Kubernetes cluster running, let's do a quick smoke test to make sure it works properly.

Deployments

Deploy Pods using Deployment

```
kubectl run nginx --image=cargo.caicloud.io/caicloud/nginx:1.13
```

```
kubectl get pods -l run=nginx
```

Verify that port forward is working

```
POD_NAME=$(kubectl get pods -l run=nginx -o jsonpath="{.items[0].metadata.name}")
```

```
kubectl port-forward $POD_NAME 9090:80
```

```
curl --head http://127.0.0.1:9090
```

Log and Exec

```
kubectl logs $POD_NAME
```

```
kubectl exec -it $POD_NAME sh
```

Services

Expose the deployment via Service

```
kubectl expose deployment nginx --port 80 --type NodePort
```

```
NODE_PORT=$(kubectl get svc nginx  --output=jsonpath='{range .spec.ports[0]}{.nodePort}')
```

```
curl -I http://${NODE_IP}:${NODE_PORT}
```

Exercise

Please deploy kubelet and kube-proxy on master node and register it as a worker node

- Hint: install docker and Kubernetes binaries
- Hint: generate certificates and kubeconfig for kubelet
- Hint: kube-proxy uses same certificate and kubeconfig as other nodes
- Hint: make sure you don't make mistake about hostname, IP address, etc