Kubernetes on Triton

The Hard Way

Cloud Infrastructure Provisioning

The following guest post is based on Kelsey Hightower's well-regarded Kubernetes The Hard Way, published under the Apache 2.0 license in GitHub.

This document, like Hightower's original, is intended to instruct readers on the different components of Kubernetes, how they relate to each other, and how they relate to the host cloud infrastructure. In short, it's one of the best ways to learn how to deploy Kubernetes by hand, prior to engaging in an automation strategy.

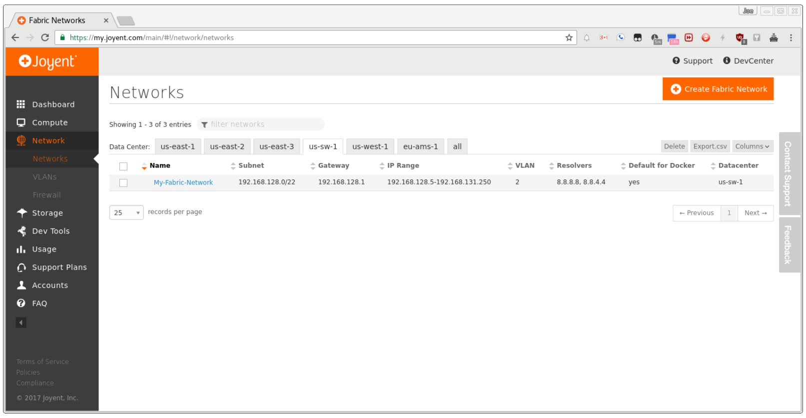
Read on to learn how to deploy a Kubernetes cluster on Triton in hardware virtual machines. This tutorial can be used for running an example HA Kubernetes cluster using the Triton Cloud service.

Begin by creating a My Joyent account, and installing and configuring the Triton CLI.

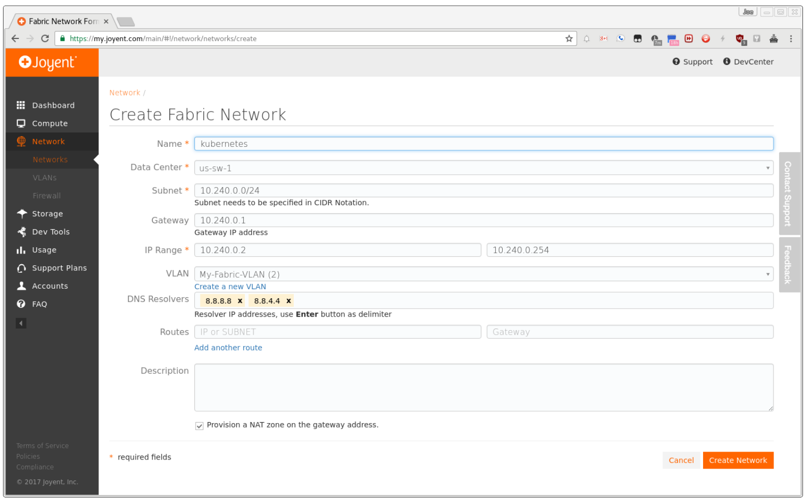
For this tutorial we will be using the us-sw-1 data center

Network

Create a private Joyent Fabric Network for your Kubernetes installation.



From the networks screen, select `Create Fabric Network`



Create a network with these attributes:

- Name: kubernetes

- Data Center: us-sw-1
- Subnet: 10.240.0.0/24
- Gateway: 10.240.0.1
- IP Range: 10.240.0.2 10.240.0.254
- VLAN: *Leave at default*
- DNS Resolvers: 8.8.8.8 8.8.4.4
- Provision a NAT zone on the gateway address: *Checked*

Provision Virtual Machines

Provision the compute instances required for running a H/A Kubernetes cluster. All the VMs in this lab will be provisioned using Ubuntu 16.04 mainly because it runs a newish Linux Kernel that has good support for Docker. A total of 7 virtual machines will be created.

Virtual Machines

Bastion

```
triton instance create \
 --wait \
 --name=kubernetes \
 -N Joyent-SDC-Public,kubernetes \
 -m user-data="hostname=kubernetes" \
 ubuntu-16.04 g4-highcpu-1G
```

Controllers

```
triton instance create \
 --wait \
 --name=controller0 \
 -N kubernetes \
 -m user-data="hostname=controller0" \
 ubuntu-16.04 g4-highcpu-1G
```

```
triton instance create \
 --wait \
 --name=controller1 \
 -N kubernetes \
 -m user-data="hostname=controller1" \
 ubuntu-16.04 g4-highcpu-1G
```

```
triton instance create \
 --wait \
 --name=controller2 \
 -N kubernetes \
 -m user-data="hostname=controller2" \
 ubuntu-16.04 g4-highcpu-1G
```

Worker Agents

```
triton instance create \
 --wait \
 --name=worker0 \
 -N kubernetes \
 -m user-data="hostname=worker0" \
 ubuntu-certified-16.04 k4-highcpu-kvm-1.75G
```

```
triton instance create \
 --wait \
 --name=worker1 \
 -N kubernetes \
 -m user-data="hostname=worker1" \
 ubuntu-certified-16.04 k4-highcpu-kvm-1.75G
```

```
triton instance create \
 --wait \
 --name=worker2 \
 -N kubernetes \
 -m user-data="hostname=worker2" \
 ubuntu-certified-16.04 k4-highcpu-kvm-1.75G
```

Allow root login to certified image

```
BASTION=$(triton ip kubernetes)
IP=$(triton ip worker0)
ssh -o proxycommand="ssh root@$BASTION -W %h:%p" ubuntu@$IP sudo cp
/home/ubuntu/.ssh/authorized_keys /root/.ssh/
IP=$(triton ip worker1)
ssh -o proxycommand="ssh root@$BASTION -W %h:%p" ubuntu@$IP sudo cp
/home/ubuntu/.ssh/authorized_keys /root/.ssh/
IP=$(triton ip worker2)
ssh -o proxycommand="ssh root@$BASTION -W %h:%p" ubuntu@$IP sudo cp
/home/ubuntu/.ssh/authorized_keys /root/.ssh/
```

You should now have the following compute instances.

```
triton instances
```

```
SHORTID   NAME         IMG                                STATE    FLAGS  AGE
e2a3b8a6  kubernetes   ubuntu-16.04@20161213              running  -      1d
b080b2c4  controller0  ubuntu-16.04@20161213              running  -      1d
846d3bb7  controller1  ubuntu-16.04@20161213              running  -      1d
e3226afb  controller2  ubuntu-16.04@20161213              running  -      1d
753c5f0c  worker0      ubuntu-certified-16.04@20161221  running  K      1d
7bb2a272  worker1      ubuntu-certified-16.04@20161221  running  K      1d
c34597cf  worker2      ubuntu-certified-16.04@20161221  running  K      1d
```

Setting up a Certificate Authority and TLS Cert Generation

Set up the necessary PKI infrastructure to secure the Kubernetes components. This leverages CloudFlare's PKI toolkit, cfssl, to bootstrap a Certificate Authority and generate TLS certificates.

You will generate a single set of TLS certificates that can be used to secure the following Kubernetes components:

- etcd
- Kubernetes API Server
- Kubernetes Kubelet

NOTE	*Best Practice* In production you should strongly consider generating individual TLS certificates for each component.

After completing this section you should have the following TLS keys and certificates:

```
ca-key.pem
ca.pem
kubernetes-key.pem
kubernetes.pem
```

Install CFSSL

This lab requires the `cfssl` and `cfssljson` binaries. Download them from the cfssl repository.

OS X

```
wget -O /usr/local/bin/cfssl https://pkg.cfssl.org/R1.2/cfssl_darwin-amd64
wget -O /usr/local/bin/cfssljson https://pkg.cfssl.org/R1.2/cfssljson_darwin-amd64
chmod +x /usr/local/bin/cfssl
chmod +x /usr/local/bin/cfssljson
```

Linux

```
wget -O /usr/local/bin/cfssl https://pkg.cfssl.org/R1.2/cfssl_linux-amd64
wget -O /usr/local/bin/cfssljson https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
chmod +x /usr/local/bin/cfssl
chmod +x /usr/local/bin/cfssljson
```

Setting up a Certificate Authority

Create the CA configuration file

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

Generate the CA certificate and private key

Create the CA CSR

```
echo '{
  "CN": "Kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "Kubernetes",
      "OU": "CA",
      "ST": "Oregon"
    }
  ]
}' > ca-csr.json
```

Generate the CA certificate and private key

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

Results

```
ca-key.pem
ca.csr
ca.pem
```

Verification

```
openssl x509 -in ca.pem -text -noout
```

Generate the single Kubernetes TLS Cert

Generate a TLS certificate that will be valid for all Kubernetes components. This is being done for ease of use. In production you should generate individual TLS certificates for each component type. All replicas of a given component type must share the same certificate.

Set the Kubernetes Public Address

```
IPS=$(triton instance ls -o ips | grep -v IPS | tr '\n' ',' | tr -d '[]')
```

Create the `kubernetes-csr.json` file:

```
cat > kubernetes-csr.json <<EOF
{
  "CN": "kubernetes",
  "hosts": [
    "kubernetes",
    ${IPS}
    "127.0.0.1"
  ],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "Kubernetes",
      "OU": "Cluster",
      "ST": "Oregon"
    }
  ]
}
EOF
```

Generate the Kubernetes certificate and private key

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

Results

```
kubernetes-key.pem
kubernetes.csr
kubernetes.pem
```

Verification

```
openssl x509 -in kubernetes.pem -text -noout
```

Copy TLS Certs

Copy the TLS certificates and keys to each Kubernetes host

```
IPS=$(triton instance ls -o ips | grep -v IPS | tr '\n' ',' | tr -d '[]')
BASTION=$(triton ip kubernetes)
tar cf - ca.pem kubernetes-key.pem kubernetes.pem |
  ssh root@$BASTION tar xf -
for ip in $(echo $IPS | tr ',"' ' '); do
  tar cf - ca.pem kubernetes-key.pem kubernetes.pem |
  ssh -o StrictHostKeyChecking=no \
  -o proxycommand="ssh root@$BASTION -W %h:%p"
  root@$ip tar xf -
done
```

Bootstrapping a H/A etcd cluster

Bootstrap a 3 node etcd cluster. The following virtual machines will be used:

- controller0
- controller1
- controller2

Why

All Kubernetes components are stateless which greatly simplifies managing a Kubernetes cluster. All state is stored in etcd, which is a database and must be treated specially. To limit the number of compute resource to complete this lab etcd is being installed on the Kubernetes controller nodes. In production environments etcd should be run on a dedicated set of machines for the following reasons:

- The etcd lifecycle is not tied to Kubernetes. We should be able to upgrade etcd independently of Kubernetes.
- Scaling out etcd is different than scaling out the Kubernetes Control Plane.
- Prevent other applications from taking up resources (CPU, Memory, I/O) required by etcd.

Provision the etcd Cluster

Run the following commands on `controller0`, `controller1`, and `controller2`:

TLS Certificates

The TLS certificates created in the Setting up a CA and TLS Cert Generation section will be used to secure communication between the Kubernetes API server and the etcd cluster. The TLS certificates will also be used to limit access to the etcd cluster using TLS client authentication. Only clients with a TLS certificate signed by a trusted CA will be able to access the etcd cluster.

Download and Install the etcd binaries

Download the official etcd release binaries from `coreos/etcd` *GitHub project*

```
wget https://github.com/coreos/etcd/releases/download/v3.0.10/etcd-v3.0.10-linux-
amd64.tar.gz
```

Extract and install the `etcd` *server binary and the* `etcdctl` *command line client*

```
tar -xvf etcd-v3.0.10-linux-amd64.tar.gz
mv etcd-v3.0.10-linux-amd64/etcd* /usr/bin/
```

All etcd data is stored under the etcd data directory.

NOTE	*Best Practice* In production the etcd data directory should be backed by a persistent disk.

Copy the TLS certificates to the etcd configuration directory

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

The internal IP address will be used by etcd to serve client requests and communicate with other etcd peers.

Set The Internal IP Address

```
INTERNAL_IP=$(ip addr show eth0 | awk '/inet /{gsub(/\/[0-9][0-9]/,"");print $2}')
```

Each etcd member must have a unique name within an etcd cluster

```
ETCD_NAME=$(mdata-get user-data | sed 's/.*hostname=\([^ ]*\).*$/\1/')
```

Set the controller addresses

```
INITIAL_CLUSTER=$(triton instance ls -o name,ips | awk '/controller/{gsub(/[^0-
9.]/,"",$2);print $1 "=https://" $2":2380"}' | tr '\n' ',')
```

The etcd server will be started and managed by systemd.

Create the etcd systemd service file

```
echo "[Unit]
Description=etcd
Documentation=https://github.com/coreos

[Service]
ExecStart=/usr/bin/etcd --name $ETCD_NAME \
  --cert-file=/etc/etcd/kubernetes.pem \
  --key-file=/etc/etcd/kubernetes-key.pem \
  --peer-cert-file=/etc/etcd/kubernetes.pem \
  --peer-key-file=/etc/etcd/kubernetes-key.pem \
  --trusted-ca-file=/etc/etcd/ca.pem \
  --peer-trusted-ca-file=/etc/etcd/ca.pem \
  --initial-advertise-peer-urls https://$INTERNAL_IP:2380 \
  --listen-peer-urls https://$INTERNAL_IP:2380 \
  --listen-client-urls https://$INTERNAL_IP:2379,http://127.0.0.1:2379 \
  --advertise-client-urls https://$INTERNAL_IP:2379 \
  --initial-cluster-token etcd-cluster-0 \
  --initial-cluster $INITIAL_CLUSTER \
  --initial-cluster-state new \
  --data-dir=/var/lib/etcd
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
" > /etc/systemd/system/etcd.service
```

Enable and start etcd

```
systemctl daemon-reload
systemctl enable etcd
systemctl start etcd
```

Verification

```
systemctl status etcd --no-pager
```

NOTE Remember to run these steps on `controller0`, `controller1`, and `controller2`

Verification

Once all 3 etcd nodes have been bootstrapped verify the etcd cluster is healthy, on one of the controller nodes run:

Request etcd cluster health

```
etcdctl --ca-file=/etc/etcd/ca.pem cluster-health
```

```
member 3a57933972cb5131 is healthy: got healthy result from https://10.240.0.12:2379
member f98dc20bce6225a0 is healthy: got healthy result from https://10.240.0.10:2379
member ffed16798470cab5 is healthy: got healthy result from https://10.240.0.11:2379
cluster is healthy
```

Bootstrapping an H/A Kubernetes Control Plane

Bootstrap a 3 node Kubernetes controller cluster. The following virtual machines will be used:

- controller0
- controller1
- controller2

Why

The Kubernetes components that make up the control plane include the following components:

- Kubernetes API Server
- Kubernetes Scheduler
- Kubernetes Controller Manager

Each component is being run on the same machines for the following reasons:

- The Scheduler and Controller Manager are tightly coupled with the API Server
- Only one Scheduler and Controller Manager can be active at a given time, but it's ok to run multiple at the same time. Each component will elect a leader via the API Server.
- Running multiple copies of each component is required for H/A
- Running each component next to the API Server eases configuration.

Provision the Kubernetes Controller Cluster

Run the following commands on `controller0`, `controller1`, and `controller2`.

Download and install the Kubernetes controller binaries

```
wget -P /usr/local/bin/ https://storage.googleapis.com/kubernetes-
release/release/v1.5.2/bin/linux/amd64/{kube-apiserver,kube-controller-manager,kube-
scheduler,kubectl}
chmod +x /usr/local/bin/{kube-apiserver,kube-controller-manager,kube-
scheduler,kubectl}
```

TLS Certificates

The TLS certificates created in the Setting up a CA and TLS Cert Generation section will be used to secure communication between the Kubernetes API server and Kubernetes clients such as `kubectl` and the `kubelet` agent. The TLS certificates will also be used to authenticate the Kubernetes API server to etcd via TLC client auth.

Copy the TLS certificates to the Kubernetes configuration directory

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

Setup Authentication and Authorization

Authentication

Token based authentication will be used to limit access to the Kubernetes API. The authentication token is used by the following components:

- kubelet (client)
- Kubernetes API Server (server)

The other components, mainly the `scheduler` and `controller manager`, access the Kubernetes API server locally over the insecure API port which does not require authentication. The insecure port is only enabled for local access.

Create a token file

```
COMMON_TOKEN=$(head /dev/urandom | base32 | head -c 8)
echo "${COMMON_TOKEN},admin,admin
${COMMON_TOKEN},scheduler,scheduler
${COMMON_TOKEN},kubelet,kubelet
" > /var/lib/kubernetes/token.csv
echo "Your common token is: ${COMMON_TOKEN}"
```

Make a note of your common token for later.

Authorization

Attribute-Based Access Control (ABAC) will be used to authorize access to the Kubernetes API. In this lab ABAC will be setup using the Kubernetes policy file backend as documented in the Kubernetes authorization guide.

Create the authorization policy file:

```
cat << EOF > /var/lib/kubernetes/authorization-policy.jsonl
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec":
{"user":"*", "nonResourcePath": "*", "readonly": true}}
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec":
{"user":"admin", "namespace": "*", "resource": "*", "apiGroup": "*"}}
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec":
{"user":"scheduler", "namespace": "*", "resource": "*", "apiGroup": "*"}}
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec":
{"user":"kubelet", "namespace": "*", "resource": "*", "apiGroup": "*"}}
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec":
{"group":"system:serviceaccounts", "namespace": "*", "resource": "*", "apiGroup": "*",
"nonResourcePath": "*"}}
EOF
```

Kubernetes API Server

Create the systemd service file

```
INTERNAL_IP=$(ip addr show eth0 | awk '/inet /{gsub(/\/[0-9][0-9]/,"");print $2}')
ETCD_SERVERS=$(triton instance ls -o name,ips | awk '/controller/{gsub(/[^0-
9.]/,"",$2);print "https://" $2":2380"}' | tr '\n' ',')
echo "[Unit]
Description=Kubernetes API Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/local/bin/kube-apiserver \
  --admission
-control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuo
ta \
  --advertise-address=${INTERNAL_IP} \
  --allow-privileged=true \
  --apiserver-count=3 \
  --authorization-mode=ABAC \
  --authorization-policy-file=/var/lib/kubernetes/authorization-policy.jsonl \
  --bind-address=0.0.0.0 \
  --enable-swagger-ui=true \
  --etcd-cafile=/var/lib/kubernetes/ca.pem \
  --insecure-bind-address=0.0.0.0 \
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \
  --etcd-servers=${ETCD_SERVERS} \
  --service-account-key-file=/var/lib/kubernetes/kubernetes-key.pem \
  --service-cluster-ip-range=10.32.0.0/24 \
  --service-node-port-range=30000-32767 \
  --tls-cert-file=/var/lib/kubernetes/kubernetes.pem \
  --tls-private-key-file=/var/lib/kubernetes/kubernetes-key.pem \
  --token-auth-file=/var/lib/kubernetes/token.csv \
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
" > /etc/systemd/system/kube-apiserver.service
```

Enable and start api server

```
systemctl daemon-reload
systemctl enable kube-apiserver
systemctl start kube-apiserver
```

Verification

```
systemctl status kube-apiserver --no-pager
```

Kubernetes Controller Manager

```
echo "[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/local/bin/kube-controller-manager \
  --allocate-node-cidrs=true \
  --cluster-cidr=10.200.0.0/16 \
  --cluster-name=kubernetes \
  --leader-elect=true \
  --master=http://$INTERNAL_IP:8080 \
  --root-ca-file=/var/lib/kubernetes/ca.pem \
  --service-account-private-key-file=/var/lib/kubernetes/kubernetes-key.pem \
  --service-cluster-ip-range=10.32.0.0/24 \
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
" > /etc/systemd/system/kube-controller-manager.service
```

Enable and start controller manager

```
systemctl daemon-reload
systemctl enable kube-controller-manager
systemctl start kube-controller-manager
```

Verification

```
systemctl status kube-controller-manager --no-pager
```

Kubernetes Scheduler

```
echo "[Unit]
Description=Kubernetes Scheduler
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/local/bin/kube-scheduler \
  --leader-elect=true \
  --master=http://$INTERNAL_IP:8080 \
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
" > /etc/systemd/system/kube-scheduler.service
```

Enable and start scheduler

```
systemctl daemon-reload
systemctl enable kube-scheduler
systemctl start kube-scheduler
```

Verification

```
systemctl status kube-scheduler --no-pager
```

Verification

```
kubectl get componentstatuses
```

```
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-1               Healthy   {"health": "true"}
etcd-0               Healthy   {"health": "true"}
etcd-2               Healthy   {"health": "true"}
```

NOTE	Remember to run these steps on `controller0`, `controller1`, and `controller2`

Setup Kubernetes API Server Frontend Load Balancer

~~~~~~~~~~~~~~~~~~~~~~~~

Setting up a load balancer is out of the scope of this tutorial. A frontend load balancer, like haproxy,
~~~~~~~~~~~~~~~~~~~~~~~~

will need to be set up on bastion in order to access any of your apis or services externally. If you don't want to set up haproxy yet, simply complete all the remaining tasks from the bastion.

Kubernetes Workers

Bootstrap 3 Kubernetes worker nodes. The following virtual machines will be used:

- worker0
- worker1
- worker2

Why

Kubernetes worker nodes are responsible for running your containers. All Kubernetes clusters need one or more worker nodes. We are running the worker nodes on dedicated machines for the following reasons:

- Ease of deployment and configuration
- Avoid mixing arbitrary workloads with critical cluster components. We are building machine with just enough resources so we don't have to worry about wasting resources.

Some people would like to run workers and cluster services anywhere in the cluster. This is totally possible, and you'll have to decide what's best for your environment.

Run the following commands on `worker0`, `worker1`, `worker2`:

Move the TLS certificates in place

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

Docker

Kubernetes should be compatible with the Docker 1.9.x - 1.12.x:

Download and install docker

```
wget https://get.docker.com/builds/Linux/x86_64/docker-1.12.1.tgz
tar -xvf docker-1.12.1.tgz
cp docker/docker* /usr/bin/
```

Create the Docker systemd service file:

```
sh -c 'echo "[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.io

[Service]
ExecStart=/usr/bin/docker daemon \
  --iptables=false \
  --ip-masq=false \
  --host=unix:///var/run/docker.sock \
  --log-level=error \
  --storage-driver=overlay
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target" > /etc/systemd/system/docker.service'
```

Enable and start docker

```
systemctl daemon-reload
systemctl enable docker
systemctl start docker
```

Verification

```
docker version
```

kubelet

Download and install CNI plugins

```
mkdir -p /opt/cni
wget https://storage.googleapis.com/kubernetes-release/network-plugins/cni-
07a8a28637e97b22eb8dfe710eeae1344f69d16e.tar.gz
tar -xvf cni-07a8a28637e97b22eb8dfe710eeae1344f69d16e.tar.gz -C /opt/cni
```

Download and install the Kubernetes worker binaries:

```
wget -P /usr/bin https://storage.googleapis.com/kubernetes-
release/release/v1.5.2/bin/linux/amd64/{kubectl,kube-proxy,kubelet}
chmod +x /usr/bin/{kubectl,kube-proxy,kubelet}
```

Set `COMMON_TOKEN` to the value you saved in Authentication

Configure kubelet:

```
CONTROLLER0=$(triton instance ls -o name,ips | awk '/controller0/{gsub(/[^0-
9.]/,"",$2);print "https://" $2 ":6443"}')
API_SERVERS=$(triton instance ls -o name,ips | awk '/controller/{gsub(/[^0-
9.]/,"",$2);print "https://" $2":6443"}' | tr '\n' ',')

mkdir -p /var/lib/kubelet/
echo "apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority: /var/lib/kubernetes/ca.pem
    server: ${CONTROLLER0}
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubelet
  name: kubelet
current-context: kubelet
users:
- name: kubelet
  user:
    token: $COMMON_TOKEN" > /var/lib/kubelet/kubeconfig
```

Create the kubelet systemd service file

```
sh -c 'echo "[Unit]
Description=Kubernetes Kubelet
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=docker.service
Requires=docker.service

[Service]
EnvironmentFile=/etc/dockerenv.conf
ExecStart=/usr/bin/kubelet \
  --allow-privileged=true \
  --api-servers=${API_SERVERS} \
  --cloud-provider= \
  --cluster-dns=10.32.0.10 \
  --cluster-domain=cluster.local \
  --container-runtime=docker \
  --network-plugin=kubenet \
  --kubeconfig=/var/lib/kubelet/kubeconfig \
  --serialize-image-pulls=false \
  --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" > /etc/systemd/system/kubelet.service'
```

Enable and start kubelet

```
systemctl daemon-reload
systemctl enable kubelet
systemctl start kubelet
```

Verification

```
systemctl status kubelet --no-pager
```

kube-proxy

Create the kube-proxy service file

```
echo "[Unit]
Description=Kubernetes Kube Proxy
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/bin/kube-proxy \
  --master=${CONTROLLER0} \
  --kubeconfig=/var/lib/kubelet/kubeconfig \
  --proxy-mode=iptables \
  --v=2

Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target" > /etc/systemd/system/kube-proxy.service
```

Enable and start kube-proxy

```
systemctl daemon-reload
systemctl enable kube-proxy
systemctl start kube-proxy
```

Verification

```
systemctl status kube-proxy --no-pager
```

NOTE	Remember to run these steps on all workers.

Configuring the Kubernetes Client - Remote Access

Download and Install kubectl

OS X

```
wget -P /usr/local/bin/ https://storage.googleapis.com/kubernetes-
release/release/v1.5.2/bin/darwin/amd64/kubectl
chmod +x /usr/local/bin/kubectl
```

Linux

```
wget -P /usr/local/bin/ https://storage.googleapis.com/kubernetes-
release/release/v1.5.2/bin/linux/amd64/kubectl
chmod +x /usr/local/bin/kubectl
```

Configure Kubectl

Configure the kubectl client to point to the

Load Balancer (if configured) or do this from the bastion and configure it to point to a controller. Set KUBERNETES_PUBLIC_ADDRESS to the IP address of the desired endpoint.

Recall the common token we set up.

Also be sure to locate the CA certificate from Setting up a CA and TLS Cert Generation. Since we are using self-signed TLS certs we need to trust the CA certificate so we can verify the remote API Servers.

Build up the kubeconfig entry

```
kubectl config set-cluster kubernetes-the-hard-way \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=https://${KUBERNETES_PUBLIC_ADDRESS}:6443

kubectl config set-credentials admin --token ${COMMON_TOKEN}

kubectl config set-context default-context \
  --cluster=kubernetes-the-hard-way \
  --user=admin

kubectl config use-context default-context
```

At this point you should be able to connect securly to the remote API server.

```
kubectl get componentstatuses
```

```
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-2               Healthy   {"health": "true"}
etcd-0               Healthy   {"health": "true"}
etcd-1               Healthy   {"health": "true"}
```

```
kubectl get nodes
```

```
NAME                                   STATUS    AGE
fb011ce0-be49-4f5d-8d13-86153cdf42f7   Ready     7m
6bc8cb95-c927-4346-8ec8-e4e821dc4b22   Ready     5m
e8e8402e-3326-4d66-b3db-d62e588ac347   Ready     2m
```

Managing the Container Network Routes and Overlay Network

Now that each worker node is online we need to add an overlay network and routes to make sure that Pods running on different machines can talk to each other.

These instructions require kubectl on the workers. Please follow these instructions to install kubectl on each worker.

Container Subnets

The IP addresses for each pod will be allocated from the `podCIDR` range assigned to each Kubernetes worker through the node registration process.

The `podCIDR` will be allocated from the cluster cidr range as configured on the Kubernetes Controller Manager with the following flag:

```
--cluster-cidr=10.200.0.0/16
```

Based on the above configuration each node will receive a `/24` subnet. For example:

```
10.200.0.0/24
10.200.1.0/24
10.200.2.0/24
...
```

Populate the Routing Table

Populate the routing table with the l3 routes over our overlay network.

Use `kubectl` to print the `InternalIP` and `podCIDR` for each worker node.

```
kubectl get nodes \
  --output=jsonpath='{range
.items[*]}{.status.addresses[?(@.type=="InternalIP")].address} {.spec.podCIDR}
{"\n"}{end}'
```

Output:

```
10.240.0.20 10.200.0.0/24
10.240.0.21 10.200.1.0/24
10.240.0.22 10.200.2.0/24
```

Create an overlay network

Fabric Networks do not forward multicasting so we need to creae a unicast vxlan overlay. We will create it in such a way that it logically matches the podCIDR network.

NOTE	Do this on each worker:

```
INTERNAL_IP=$(ip addr show net0 | awk '/inet /{gsub(/\/[0-9][0-9]/,"");print $2}')
VXLAN_IP=$(kubectl get nodes \
  --output=jsonpath='{range .items[*]}
{.status.addresses[?(@.type=="InternalIP")].address} {.spec.podCIDR} {"\n"}{end}' |
  awk "/$INTERNAL_IP/{print \$2}" |
  sed -e 's/10\.200\./172.16./' -e 's@0/@1/@')
ip link add vxlan0 type vxlan id 1 dstport 0
eval $(kubectl get nodes \
  --output=jsonpath='{range
.items[*]}{.status.addresses[?(@.type=="InternalIP")].address}{"\n"}{end}' |
  grep -v $INTERNAL_IP |
  awk '{ print "bridge fdb append to 00:00:00:00:00:00 dst " $1 " via net0" }')
ip addr add $VXLAN_IP dev vxlan0
ip link set up vxlan0
```

Create Routes

NOTE	Do this on each worker:

```
eval $(kubectl get nodes \
  --output=jsonpath='{range
.items[*]}{.status.addresses[?(@.type=="InternalIP")].address} {.spec.podCIDR}
{"\n"}{end}' |
  grep -v $INTERNAL_IP |
  awk '{GW=$2;gsub("10.200","172.16",GW); gsub("0/24","1",GW); print "ip route add "
$2 " via " GW }')
```

Deploying the Cluster DNS Add-on

Deploy the DNS add-on which is required for every Kubernetes cluster. Without the DNS add-on the following things will not work:

- DNS based service discovery
- DNS lookups from containers running in pods

Cluster DNS Add-on

Create the `kubedns` service:

```
kubectl create -f https://raw.githubusercontent.com/kelseyhightower/kubernetes-the-
hard-way/master/services/kubedns.yaml
```

Verification

```
kubectl --namespace=kube-system get svc
```

```
NAME       CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kube-dns   10.32.0.10   <none>        53/UDP,53/TCP   5s
```

Create the `kubedns` deployment:

```
kubectl create -f https://raw.githubusercontent.com/kelseyhightower/kubernetes-the-
hard-way/master/deployments/kubedns.yaml
```

Verification

```
kubectl --namespace=kube-system get pods
```

```
NAME                           READY     STATUS    RESTARTS   AGE
kube-dns-v19-965658604-c8g5d   3/3       Running   0          49s
kube-dns-v19-965658604-zwl3g   3/3       Running   0          49s
```

Smoke Test

Perform a quick smoke test to demonstrate that this cluster is working

Test

```
kubectl run nginx --image=nginx --port=80 --replicas=3
```

```
deployment "nginx" created
```

```
kubectl get pods -o wide
```

```
NAME                     READY     STATUS    RESTARTS   AGE       IP           NODE
nginx-2032906785-ms8hw   1/1       Running   0          21s       10.200.2.2   worker2
nginx-2032906785-sokxz   1/1       Running   0          21s       10.200.1.2   worker1
nginx-2032906785-u8rzc   1/1       Running   0          21s       10.200.0.2   worker0
```

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

```
service "nginx" exposed
```

Grab the `NodePort` that was setup for the nginx service.

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

Grab the EXTERNAL_IP for one of the worker nodes

```
NODE_PUBLIC_IP=$(aws ec2 describe-instances \
  --filters "Name=tag:Name,Values=worker0" | \
  jq -j '.Reservations[].Instances[].PublicIpAddress')
```

Test the nginx service using cURL

```
curl http://${NODE_PUBLIC_IP}:${NODE_PORT}
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Congratulations. You now have a working cluster built the hard way on Triton.