Talos Linux

Provision nodes

After creating the nodes in Proxmox run the following commands. In this example, I have 3 API controllers and 3 worker nodes. Each controller will have an etcd database.

```
talosctl gen config koryscluster https://<haproxy IP>:6443
```

Edit the worker.yaml that gets created by the last command. We will need to add longhorn support.

```
    kubelet:
        extraMounts:
          - destination: /var/lib/longhorn
            type: bind
            source: /var/lib/longhorn
            options:
              - bind
              - rshared
              - rw
```

Apply the config to the controlplanes.

```
talosctl apply-config --insecure -n <IP ctl-1> --file controlplane.yaml
talosctl apply-config --insecure -n <IP ctl-2> --file controlplane.yaml
talosctl apply-config --insecure -n <IP ctl-3> --file controlplane.yaml
```

Bootstrap the cluster. This will initialize etcd.

```
talosctl bootstrap -n <IP ctl-1> -e <IP ctl-1> --talosconfig=./talosconfig
```

Generate the k8s config file

```
talosctl kubeconfig ./kube -n <IP ctl-1> -e <IP ctl-1> --talosconfig=./talosconfig
```

Either you can copy the config to ~/.kube/config or export as an environment variable. Lets confirm the cluster is alive and we can talk to the API servers via HA proxy.

```
kubectl get nodes
```

After this is successful, we can get the workers setup. These will hold our longhorn data. Let add more space to these and we will need to have talos upgrade the config to add iscsi support.

```
talosctl apply-config --insecure -n <IP worker-1> --file worker.yaml
talosctl apply-config --insecure -n <IP worker-2> --file worker.yaml
talosctl apply-config --insecure -n <IP worker-3> --file worker.yaml
```

Lets start the upgrade process. We can use image factory to setup a new config. The hash needed for longhorn support is already setup with the following. Note the version number must be greater than or equal to the current talos version installed on the node.

```
talosctl upgrade --talosconfig=./talosconfig --nodes <IP worker-1> -e <IP worker-1> --image
factory.talos.dev/installer/613e1592b2da41ae5e265e8789429f22e121aab91cb4deb6bc3c0b6262961245:v1.9.1
talosctl upgrade --talosconfig=./talosconfig --nodes <IP worker-2> -e <IP worker-2> --image
factory.talos.dev/installer/613e1592b2da41ae5e265e8789429f22e121aab91cb4deb6bc3c0b6262961245:v1.9.1
talosctl upgrade --talosconfig=./talosconfig --nodes <IP worker-3> -e <IP worker-3> --image
factory.talos.dev/installer/613e1592b2da41ae5e265e8789429f22e121aab91cb4deb6bc3c0b6262961245:v1.9.1
```

We now have a working cluster. Nothing besides the core k8s system is installed. We can refer to the Kubernetes section to continue setup.

How to scale down a Talos cluster

To remove nodes from a Talos Linux cluster:

```
talosctl -n <IP.of.node.to.remove> reset
```

```
kubectl delete node <nodename>
```

Node setup with Nvidia GPU

Introduction

Refer to the previous page for the general concept of Talos worker.yaml. This will build on the same concept but add required extensions and load modules needed for pods to see a GPU and use it for AI workloads. This guide will reference this documentation.

Image Factory

First, we need to create the new image that will be loaded on the node. We have the option to choose from open-source Nvidia drivers to the proprietary version. The following are the extensions you need to search for, depending on your choice. You will notice they have a lts and production version. I have been using LTS. The main item to note is that they should match. Meaning if you choose LTS chose this for both options.

Open source extensions

- `nvidia-open-gpu-kernel-modules`
- `nvidia-container-toolkit`

Proprietary extensions

- `nonfree-kmod-nvidia`
- `nvidia-container-toolkit`

Provision node

Now that we have the image file, we can add this to our worker.yaml, as seen on the previous page. This should look similar to the following under install.

```
machine:
  install:
    disk: /dev/nvme0n1 # The disk used for installations.
    image:
factory.talos.dev/installer/53b5a3efb4cca0300d7947f45577df156effe1be2f373daf3ffd8d7ba08ea899:v1.9.5
    wipe: false
```

Depending on the node type (virtual/physical), we need to boot the new node from the iso or image that came out of our image factory process. After the node is booted and ready to accept configuration, let's tell it to install these extensions and join our cluster.

```
talosctl apply-config --insecure -n <new node IP> --file worker.yaml
```

After the node reboots, we can verify the extensions are installed by running the following

```
talosctl get extensions
```

The output should look similar to the following

```
NODE            NAMESPACE   TYPE              ID          VERSION   NAME                          VERSION
192.168.249.11   runtime     ExtensionStatus   0           1         iscsi-tools                   v0.1.6
192.168.249.11   runtime     ExtensionStatus   1           1         nonfree-kmod-nvidia-lts        535.216.03-v1.9.1
192.168.249.11   runtime     ExtensionStatus   2           1         nvidia-container-toolkit-lts   535.216.03-v1.17.2
192.168.249.11   runtime     ExtensionStatus   3           1         schematic
4ba64c429e0aa252d716a668cf66b056b6ee3805f0ee0d7258a3a71e81df8e50
192.168.249.11   runtime     ExtensionStatus   modules.dep   1        modules.dep                   6.12.6-talos
```

Patch node

Now we need to patch the node to load the nvidia modules. Create a patch-gpu.yaml.

```
machine:
  kernel:
    modules:
      - name: nvidia
      - name: nvidia_uvm
      - name: nvidia_drm
      - name: nvidia_modeset
  sysctls:
    net.core.bpf_jit_harden: 1
```

Apply the patch with the following.

```
talosctl patch mc --patch @patch-gpu.yaml
```

Confirm the patch with the following commands.

```
talosctl read /proc/driver/nvidia/version


NVRM version: NVIDIA UNIX x86_64 Kernel Module  535.216.03  Fri Oct 25 22:43:06 UTC 2024

GCC version:  gcc version 14.2.0 (GCC)
```

```
talosctl read /proc/modules


nvidia_uvm 1908736 0 - Live 0xffffffffc4175000 (PO)

nvidia_drm 94208 0 - Live 0xffffffffc0575000 (PO)

nvidia_modeset 1531904 2 nvidia_drm, Live 0xffffffffc4356000 (PO)

nvidia 62771200 19 nvidia_uvm,nvidia_modeset, Live 0xffffffffc0596000 (PO)
```

Patch runtime

We need to set Nvidia as the default runtime. This can be done with a YAML inside of k8s, but I found the best way is to apply this to the node. I don't know why this is required, but pods would not see the GPU or even run nvidia-smi without this

create a file runtime-patch.yaml

```
- op: add
  path: /machine/files
  value:
    - content: |
        [plugins]
          [plugins."io.containerd.cri.v1.runtime"]
            [plugins."io.containerd.cri.v1.runtime".containerd]
              default_runtime_name = "nvidia"
      path: /etc/cri/conf.d/20-customization.part
      op: create
```

```
talosctl patch mc --patch @runtime-patch.yaml
```

Nvidia device plugin

We can use Helm to install the device plugin. This runs a daemonset looking for nodes with a GPU. It will only run on those nodes.

Time slicing is not enabled by default. This means one GPU can only be used by one pod.The following configuration will enable one GPU to share time with many pods at once.

Create the config file time-slicing-config.yaml

```
version: v1
flags:
  migStrategy: none
sharing:
  timeSlicing:
    resources:
    - name: nvidia.com/gpu
      replicas: 5
```

Now, we can pass the config file to helm and install the device plugin.

```
helm repo add nvdp https://nvidia.github.io/k8s-device-plugin
helm repo update
helm install nvidia-device-plugin nvdp/nvidia-device-plugin --version=0.13.0 --set=runtimeClassName=nvidia \
--set-file config.map.config=./time-slicing-config.yaml \
--namespace nvidia-system
```

Testing

Let us run this pod to test if the GPU is picked up correctly.

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  restartPolicy: Never
  containers:
    - name: cuda-container
      image: nvcr.io/nvidia/k8s/cuda-sample:devicequery-cuda11.7.1-ubuntu20.04
      resources:
        limits:
          nvidia.com/gpu: 1 # requesting 1 GPU
  tolerations:
```

```
- key: nvidia.com/gpu
  operator: Exists
  effect: NoSchedule
```

Switch CNI

Flannel to Calico

Introduction

When I first provided my Talos cluster, I didn't realize it comes with Flannel as the default CNI. At first, this wasn't a problem, but as I mature using Kubernetes, I quickly found I need more advanced network policies. These are the steps I took to convert a live cluster from Flannel to Calico with no downtime.

Preperation

Since this is a big move and a lot can go wrong, I went through several steps to ensure uptime.

1. Set up a test cluster to mirror your current setup. All the actions we are going to take can first be done there to ensure nothing pops up during conversion.
2. Snapshot all controllers (if possible) or do a backup of etcd

Convert the cluster

The steps are straightforward, and thankfully, Kubernetes is very good at extensibility, so adding dual CNI providers does not break networking.

First, let's create our namespace and set the proper permissions needed by Calico

```
kubectl create namespace tigera-operator
kubectl label namespace tigera-operator pod-security.kubernetes.io/enforce=privileged
```

Next, we will use Helm to install the provider. This can be done with regular manifests, but it will be easier to maintain and upgrade in the future with Helm. You may need to update the version.

```
helm install calico projectcalico/tigera-operator --version v3.29.3 --namespace tigera-operator
```

After some time, we can check to make sure that all pods are running and ready! This took a bit to finish so make sure its done before moving on.

```
kubectl get pods -n calico-system
```

Once that is complete, we can edit the default IP pool used by Calico. This is optional, just note the default is 192.168.0.0/16, which was not ideal for my network. Change the cidr field to what you require.

```
apiVersion: crd.projectcalico.org/v1
kind: IPPool
metadata:
  name: default-ipv4-ippool
spec:
  allowedUses:
  - Workload
  - Tunnel
  blockSize: 26
  cidr: 10.225.0.0/16
  ipipMode: Always
  natOutgoing: true
  nodeSelector: all()
  vxlanMode: Never
```

```
kubectl delete ippools.crd.projectcalico.org default-ipv4-ippool
kubectl apply -f ippool.yaml
```

The cluster should be in a stable state at this point. You will begin to see new pods with the updated IP range. The nodes may also have a secondary IP. This is ok. Now we can patch the controller nodes to remove the Flannel manifest.

```
cluster:
  network:
    cni:
      name: none
  proxy:
    disabled: true
```

```
talosctl patch mc --patch @cnipatch.yaml
```

Perform this step on all controllers. Then we can verify it worked by querying Talos controllers for their built-in manifest files. We do not want to see Flannel list anywhere.

```
talosctl get manifests -n <controller-ip>
```

Sample output:

```
NODE          NAMESPACE     TYPE      ID                          VERSION
192.168.249.4   controlplane   Manifest   00-kubelet-bootstrapping-token   1
192.168.249.4   controlplane   Manifest   01-csr-approver-role-binding     1
192.168.249.4   controlplane   Manifest   01-csr-node-bootstrap            1
192.168.249.4   controlplane   Manifest   01-csr-renewal-role-binding      1
192.168.249.4   controlplane   Manifest   11-core-dns                      1
192.168.249.4   controlplane   Manifest   11-core-dns-svc                  1
192.168.249.4   controlplane   Manifest   11-kube-config-in-cluster        1
```

The final step is to remove the flannel daemonset and configmap

```
kubectl delete daemonset -n kube-system kube-flannel
kubectl delete cm kube-flannel-cfg -n kube-system
```

Conclusion

You may notice that some pods still have the old cidr range. This is mostly okay. I found that networking for the most part still works. Its best to go through the namespaces and do a rollout restart on the various resources and they will pick up the new networking rules.