Kubernetes Components

-- Control Plane Node: Just like normal networking, this is the aspect of kubernetes that does not process user traffic, it comprises of components several components which are explained below:

** API Server: All operations and management goes through it. It is connected to all the other components. It is called API server because irrespective of how you are communicating with a kubernetes cluster you are always interfacing through an API service. The kubectl client also connects to the API server to carry out various OAM, the kubectl takes input from user, transforms it to an API call and then sends it to the API server, also the kubectl receives the response from the API server and then converts to a human readable format.

** Kube-controller: This comprises of all the processes that are responsible for controlling the different processes. All actions are related to one controller or the other. Examples are the replication controller, node controller etc. The controller works by trying reconcile actual state of the kubernetes to a desired state, this is also called a control-loop.

** key/value store: Kubernetes by default is a stateless application, this means it does not store any configurations within itself, it relies on an another software to do this. A full fledged database is not suited for this purpose since most of the state are essentially key/value pairs. The major software that is used for is ETCD, other solutions can be used for this purpose but ETCD has become the defacto standard. ***Only the API server has direct access to the ETCD.***

** Kube-scheduler: This is the control plane component that is responsible for deciding on which node to place the POD (containers), it depends on the API server to know when to act.

-- Worker Node Components: You can compare this to the user-plane of a kubernetes cluster but officially it is called the worker-node components. These are the components that actually provides the processes that will deal with the actual workloads. They are:

** CRI: Container runtime interface. There are different container engines like docker, containerd, rkt etc that are responsible for running the containers (they can be run independently of kubernetes). But kubernetes needs a common interface for communicating with them. All of them are actually developed outside the kubernetes source code, this discouples kubernetes from trying to make every container runtime to fit in which will be a maintenance nightmare. A good example of a common interface is a routing protocol like BGP, those that maintain the BGP standard does not care how vendors create routing softwares, all they can assure is that if your router can adhere to the BGP protocol rules then it should be able to communicate with any other BGP router, the maintenance of the BGP software itself is left to the different vendors/developers. The same is true with kubernetes, it tries to decouple the code base from adhering to a particular container runtime, the kubernetes developers concentrates on creating CRI and then it is up to software vendors to make sure that they create container runtime that meets the CRI specifications.

** CNI: Container Network Interface. This is responsible for the networking aspects of kubernetes. Kubernetes leaves this work also to what is called CNI plugins that are written by other vendors.

The CNI plugins takes care of networking tasks like IP addressing for the PODs and security within the kubernetes cluster. Depending on your needs, the choice of CNI plugins will give different capabilities. Some gives the possibility of advertising kubernetes internal IP to routers that are running outside of the kubernetes cluster via routing protocols, others provide enhanced performance for latency sensitive workloads, some make use of overlays and so on and so forth.

** CSI: Container Storage Interface - Just like the CRI and CNI, this interface is dedicated to providing storage solution. This makes it possible to use different storage solutions depending on the platform you are using or specific requirements.

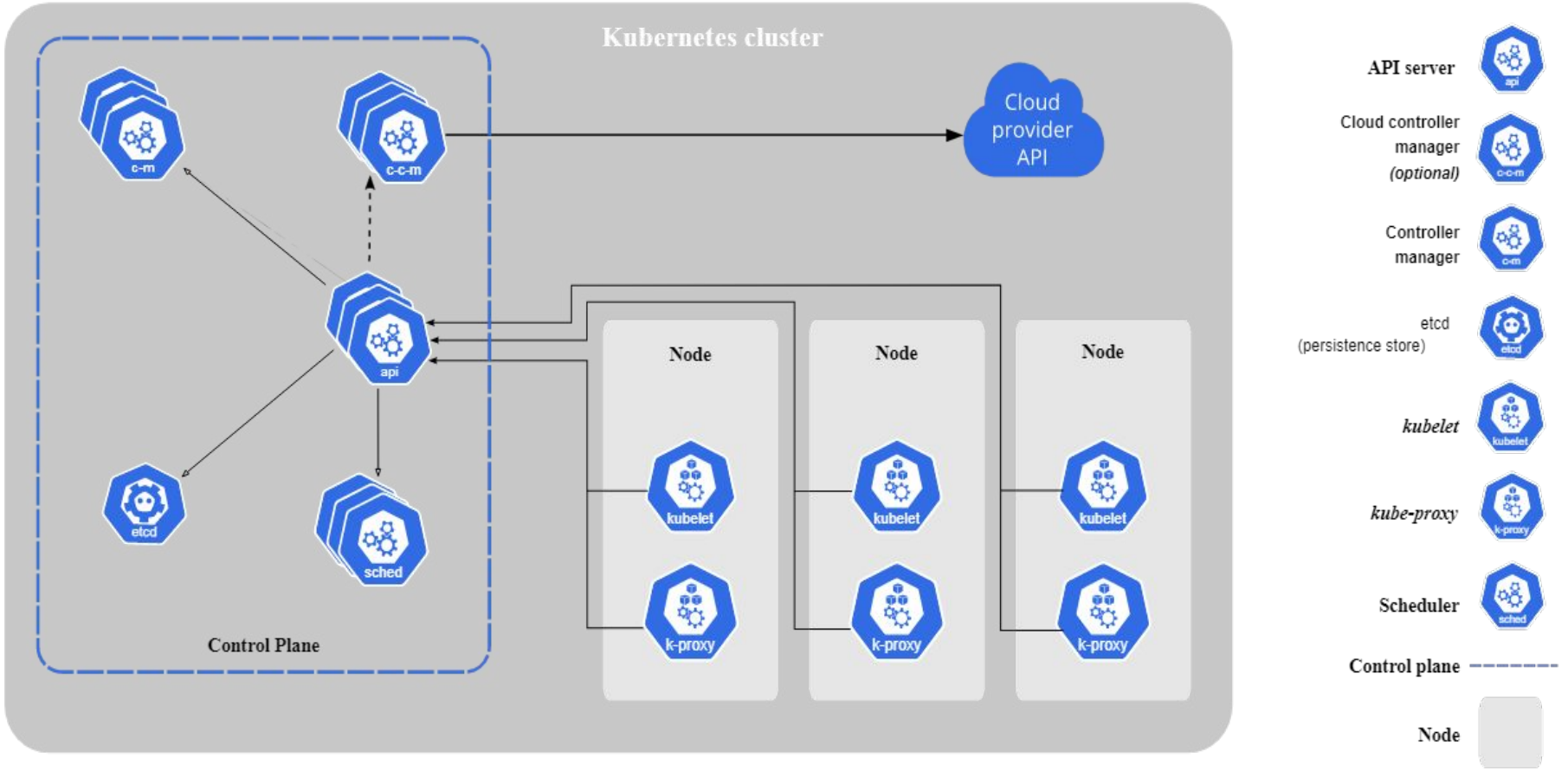
** Kubelet: This makes sure that every container is running inside a POD (this is the smallest unit scope of computing in kubernetes). Kubernetes does not deal directly in terms containers even though it is called a container orchestration software. Kubelet uses the CRI, CNI, CSI to wrap containers inside a POD. It also communicates to the API server to know when there is POD spec that needs to be deployed on the worker node it is running on. PODs can contain one or more containers.

** Kube-proxy: This deals with the issue of how kubernetes services can communicate internally among themselves and externally with user traffic. It makes use of IP tools like iptables and ipvs. Although in recent times, it can be completely replaced by some CNI plugins (e.g. Cilium, cailco-ebpf) that also have the feature in-built. The use of iptables or ipvs is determined on if the kubernetes cluster has several hundreds or thousands of worker nodes. If the number of worker nodes/user traffic is huge, ipvs is recommended.

-- Other important cluster components: These are essential components that provides services that are crucial to the smooth running of a kubernetes cluster but they are strictly managed outside the scope of kubernetes components:

** Cluster DNS: Workloads in kubernetes require some form of service discovery. One of the ways to solve this is by creating service names that can be mapped to POD IPs, so since this is identical to what a DNS software does hence the reason it is needed. Currently the de facto DNS solution that is used is coreDNS, it comes by default in almost all kubernetes distributions.

** Cloud controller: If deployed this will be part of the control plane nodes. It facilitates the communication between kubernetes and the underlying cloud platform. For instance if kubernetes requires a Load Balancer it will just send the request to the cloud controller, it does not need to understand the intricacies of how the cloud platform works. While it is not strictly important for a kubernetes cluster that is running on a cloud platform, it is strongly advised that you incorporate this in the control-plane if you need kubernetes to request some features like volumes, load balancing etc from the underlying cloud platform. Most popular cloud platforms like AWS, GCP, Azure, Linode, Openstack etc. have cloud controllers for their specific features. However this is one of the challenges of baremetal deployments (if cloud solutions like Openstack is not used) if features like load balancing is required, ways of how to resolve this will be the focus of this training.


Kubernetes cluster
c-m
c-c-m
Cloud provider API
api
etcd
sched
Control Plane
Node
kubelet
k-proxy
Node
kubelet
k-proxy
Node
kubelet
k-proxy
API server
Cloud controller manager (optional)
Controller manager
etcd (persistence store)
kubelet
kube-proxy
Scheduler
Control plane
Node

PODs, ReplicaSets,Deployment,Statefulset,Daemonset,Job,Cronjob

POD: This is the smallest compute unit in kubernetes, it is important to note that every other thing resolves round pods in a kubernetes cluster. Without pods, the kubernetes cluster is not useful! A pod is what actually runs the containers. While you can create a pod directly, this should not be done unless you are carrying out some tests, Why is this? Once a pod dies (maybe deleted or due to some other circumstances), it will not be re-created until you manually create it again. This is why you should use other controllers to deploy pods, these controllers will then manage the life-cycle of pods. These controllers are discussed next.

ReplicaSet: This is used to make that the desired number of pods are running, the desired number is called REPLICA(s). For instance if the desired pod replica is 4 i.e. 4 pods of an application must always be running, if one of the pod is deleted which will reduce the number of pods to 3, the replicaset controller will detect that the actual state (i.e. 3 pods) does not match the desired state (i.e. 4 pods), it will then try to reconcile the actual state to the desired state by creating one more pod. The replicaset configuration can be changed at any time i.e. replicas can be increased or reduced and the number of running pods will be adjusted accordingly. While pods should be run via replicaset, replicaset may not fulfil some additional features like rolling back to an earlier configuration. Hence it is advised that replicaset should be used directly instead it should be used via deployment controller.

Deployment: This is the recommended controller that should be used especially when you want to run stateless applications (applications that are not tied to their running state like databases). So what you create essentially is a deployment which in turn will create a replicaset which will then create the required number of pods. You don’t need to create the replicaset or pods directly, the deployment controller does that for you automatically.

POD Template

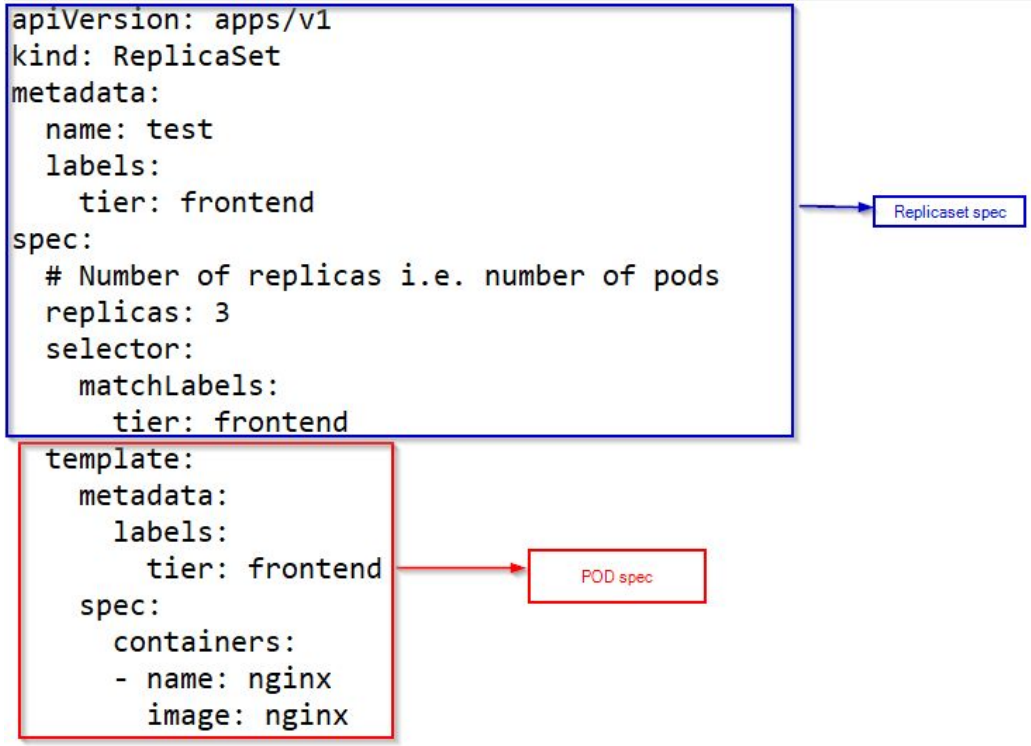
```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: test
  name: test
spec:
  containers:
  - image: nginx
    name: test
```

--- There are no replicas configured at all, this will just create a single instance of an nginx application called test.

POD Template

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: test
  name: test
spec:
  containers:
  - image: nginx
    name: test
```

--- There are no replicas configured at all, this will just create a single instance of an nginx application called test.


apiVersion: apps/v1
kind: ReplicaSet
metadata:
name: test
labels:
tier: frontend
spec:
Number of replicas i.e. number of pods
replicas: 3
selector:
matchLabels:
tier: frontend
template:
metadata:
labels:
tier: frontend
spec:
containers:
- name: nginx
image: nginx
Replicaset spec
POD spec

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
```

Deployment spec

```yaml
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

Replicaset spec

```yaml
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

POD spec

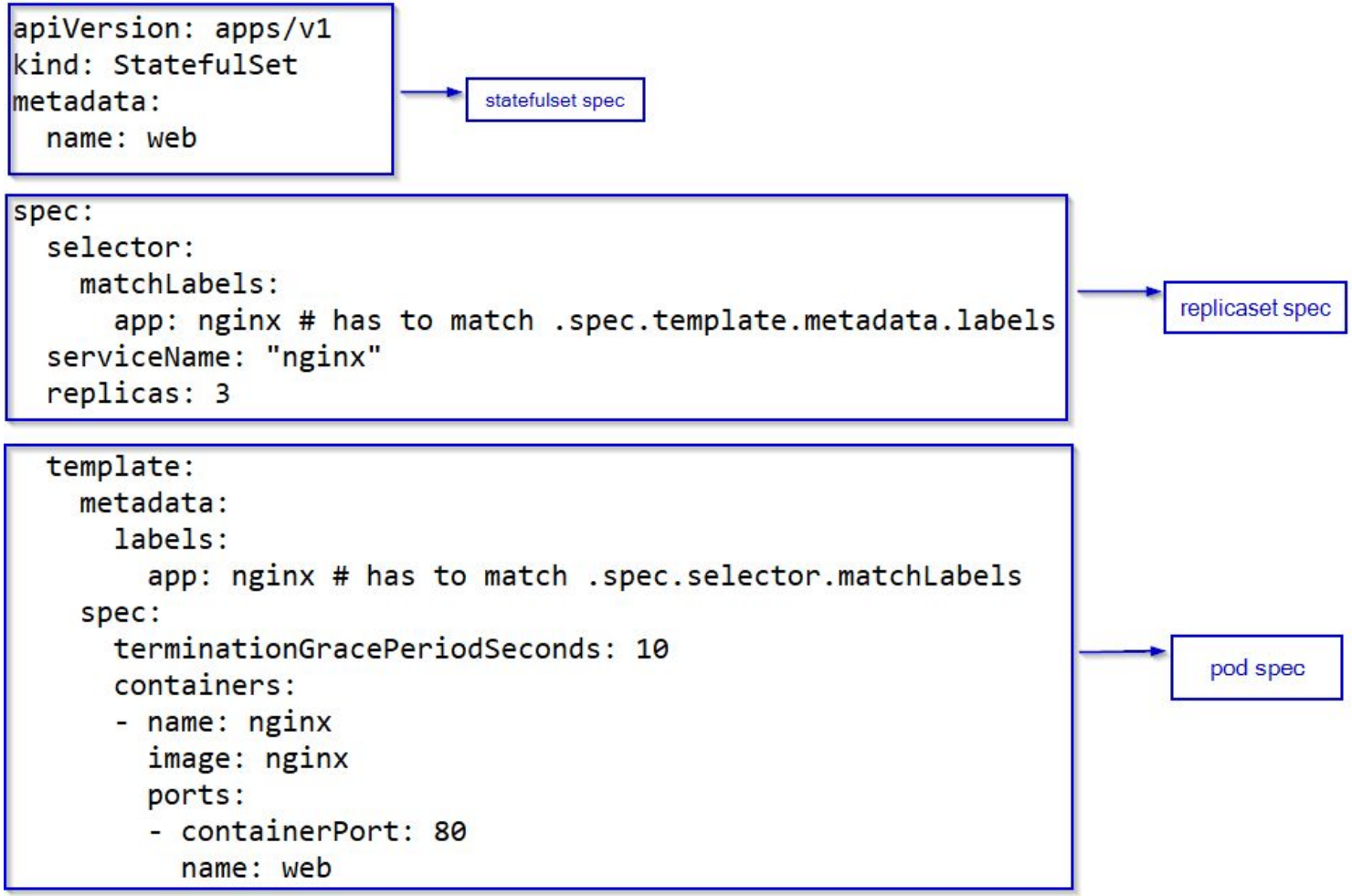
PODs, ReplicaSets,Deployment,Statefulset,Daemonset,Job,Cronjob

StatefulSet: This is used to manage stateful applications (like databases) where deployment controller will not be suitable. It also manages pods just like deployment but it guarantees the ordering and uniqueness of pods which is essential to stateful applications.

DaemonSet: This is used when it is desired to create pods across all the worker-nodes so as to provide essential services that are used at the node level like logging (since the container application logs are stored locally on each node). CNI plugins are usually deployed as daemonsets.

Job: This is simply used if you want to run just one task to completion. It creates one or more Pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created.

Cronjob: This simply creates job in a specified schedule, like if a job needs to be run at specific times. It follows the same principle as the normal crontab in linux.


apiVersion: apps/v1
kind: StatefulSet
metadata:
name: web
statefulset spec
spec:
selector:
matchLabels:
app: nginx # has to match .spec.template.metadata.labels
serviceName: "nginx"
replicas: 3
replicaset spec
template:
metadata:
labels:
app: nginx # has to match .spec.selector.matchLabels
spec:
terminationGracePeriodSeconds: 10
containers:
- name: nginx
image: nginx
ports:
- containerPort: 80
name: web
pod spec

Services, Endpoints, Ingress

Service: This is used to expose the applications that are running inside a pod within the cluster or externally. Below are some points about services:

** It uses labels to select the pod that it will serve.

** The service name can be resolved via DNS (cluster coreDNS server).

** Services in turn creates endpoints which are actually mapped to the pods that are related to the service parameters.

Service Types:

** ClusterIP - This type of service is meant to be referenced within the cluster itself because the IP address that the service name will be resolved to is only available within the kubernetes cluster, pods and other resources will be able to access it internally. However there is an exception to this, some CNI plugins can advertise clusterIPs directly into external networks.

** NodePort - This type of service uses the worker node ports to expose services that are within the kubernetes cluster. It should be noted this automatically creates a clusterIP also. So service requests will first of all hit a port in the worker node before been directed to the clusterIP which will in turn direct the traffic to the related pods.

Services, Endpoints, Ingress

Service Types:

** ***LoadBalancer*** - This exposes services using a loadbalancer but this is tightly coupled to the capability of the underlying platform, most public cloud provides this functionality via the cloud-controller feature that was discussed earlier. A loadbalancer service also creates a nodePort service first and the nodePort service will then be tied a clusterIP. So 2 additional things are been created when this kind of service is configured. If you are using a baremetal solution, it is still possible to have this feature by using other softwares that can provide this functionality or use CNI plugins that can advertise clusterIPs externally (thereby negating the need for a loadbalancer service type).

ExternalName: This maps a service name to another DNS name (CNAME) depending on the requirement.

Headless Service: This is a service type that is specifically configured without a clusterIP (this is set to none). This means that the service name will be resolved directly to the pod IPs and the logic of how to select which pod IP is to be used is left to the application that is calling the service.

Ingress: This is also used to expose services externally in a kubernetes cluster. It can be used to expose multiple services by providing a single point. It operates in most cases by using the concept of reverse proxy.

Services, Endpoints, Ingress

→ Ingress:

In addition it requires a separate software called ingress controller which will also be installed within the cluster. It can operate at layer 4 or 7 depending on the ingress controller software.

It should be noted that since the ingress controller is deployed as application within kubernetes then it means it also makes use of a service too. If the service type that is used is a loadbalancer then it can save some money especially on public clouds by using one loadbalancer for multiple services, the services can be distinguished by using domain names and paths. NodePort service type can also used. It can further provide more features like TLS termination, rate-limiting etc.

Configmaps, Secrets, PVC, Namespace, kubectl, kubeconfig

Configmaps: These are used to store configuration parameters that can be used by pods. They can be used via environment variables or by mounting them as volumes in pods. Also note that configmaps are mounted as readonly, this follows the guideline that any changes that needs to be done should be from within the configmap not directly inside the pod (this is especially true if you are running multiple pods).

Secrets: These are used to store sensitive information by using base64 encoding. It does not provide encryption! It can also be referenced as environment variable or mounted as a volume.

PVC: Persistent volume claim. This takes care of providing volume storage to pods, it can request for a block volume directly from the underlying cloud platform or via storage solutions that caters for on-prem environment.

Namespace: This gives the capability of splitting a kubernetes into virtual clusters, this is synonymous to a VLAN. Resources names can be the same provided they are created in different namespaces. It also comes handy when you want separation of concerns when you are deploying an application that has several components, e.g you might want to deploy the logging and monitoring in a separate namespace while you use another namespace for other components like frontend/backend. It can also help to create virtual clusters for different departments/teams, so each team/department can be restricted to use the namespace that has been assigned to them. This is one of the crucial aspects of kubernetes that should be understood. It also affects the way service names are resolved via DNS, applications within the same namespace does not need to append the name of the namespace to the DNS query but when an application in another namespace needs to be accessed then the namespace must be appended to the service name.

Configmaps, Secrets, PVC, Namespace, kubectl, kubeconfig

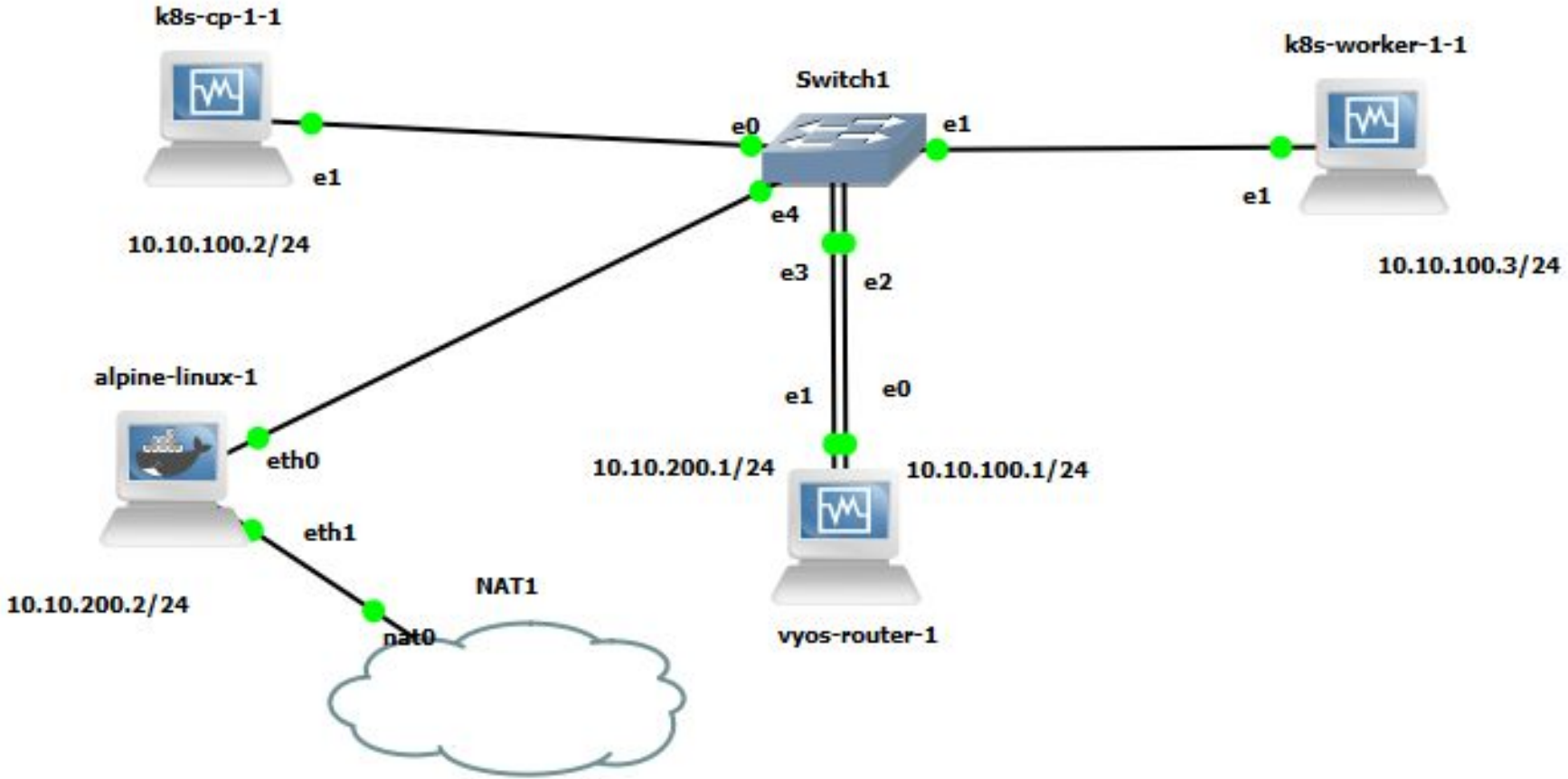
→ Namespace

This full DNS service name is of the form *<service-name>.<namespace-name>.svc.cluster.local*, but *<service-name>.<namespace-name>* is sufficient in most cases across the kubernetes cluster.

Kubeconfig: This is the file that contains specific information about how to connect to a kubernetes cluster, it has informations like the URL or IP of the API server (usually this might be a load balancer in case of HA/multiple control-plane nodes) and authentication parameters that will be used to communicate with the kube-api server component.

Kubectl: This is the client that is used to interact with the kubernetes cluster, it communicates with the API server by using the kubeconfig file. It converts user commands to the API calls before sending it to the API server, it will also convert API response back to a user friendly format. It is just one of the client software solutions, users can also create clients or communicate with the kubernetes cluster using programming languages.

Kubeadm Installation



Kubeadm Installation

sudo hostnamectl set-hostname k8s-cp-1

sudo hostnamectl set-hostname k8s-worker-1

Configure static IP on second network interface

sudo nano /etc/netplan/00-installer-config.yaml

```
# This is the network config written by 'subiquity'
network:
  ethernets:
    enp0s3:
      dhcp4: true
    enp0s8:
      addresses:
      - 10.10.100.2/24
  version: 2
```

sudo netplan apply

Kubeadm Installation

Map hostname to enp0s8 IP

sudo nano /etc/hosts

Disable swap
sudo swapon --show
sudo nano /etc/fstab (you can also use "sudo swapoff -a" but this will not be persistent upon reboot)
sudo reboot

Install Docker

curl -sL https://get.docker.com | sudo sh

sudo systemctl status docker

Verify kernel modules
lsmod | grep br_netfilter

sudo sysctl -a | grep net.bridge.bridge-nf-call

Kubeadm Installation

Install kubeadm binaries

```
sudo apt-get update && sudo apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl

sudo apt-mark hold kubelet kubeadm kubectl
```

Initialize kubeadm control-plane node

```
sudo kubeadm init --apiserver-advertise-address=10.10.100.2 --pod-network-cidr=192.168.0.0/16


sudo kubeadm join 10.10.100.2:6443 --token g65avn.98s7xvwigo6xckfz \
    --discovery-token-ca-cert-hash
sha256:aa675261e97d6b82609de4ec81608f365b8e6103e776c71d89e9e1e2919bd536
```

Kubeadm Installation

Install CNI Plugin

kubectl create -f https://docs.projectcalico.org/manifests/tigera-operator.yaml

kubectl create -f https://docs.projectcalico.org/manifests/custom-resources.yaml

Vyos Configuration

set interfaces ethernet eth0 address '10.10.100.1/24'
set interfaces ethernet eth1 address '10.10.200.1/24'
set protocols static route 192.168.230.0/26 next-hop 10.10.100.3

Kubeadm Installation

Alpine Docker Network Configuration

```
# Static config for eth0
auto eth0
iface eth0 inet static
        address 10.10.200.2
        netmask 255.255.255.0
# DHCP config for eth0
# auto eth0
# iface eth0 inet dhcp
auto eth1
iface eth1 inet dhcp
```

Kubeadm Installation

References:

https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/

https://docs.projectcalico.org/getting-started/kubernetes/quickstart

https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-binary-with-curl-on-linux