Kubernetes Networking

Services-

- A K8s service creates a single access point for a group of Pods.
- Networking in Kubernetes (it helps us to expose pods internally (so that another service/app can connect) as well externally)
- Selector and services name should be same.
- A service in Kubernetes is the entry for traffic into your application. It can be used for accessing an application just internally in the Kubernetes cluster or to expose the application.
- Basically, services are a type of resource that configures a proxy to forward the requests to a set of pods, which will receive traffic & is determined by the selector. Once the service is created it has an assigned IP address which will accept requests on the port.

Network plugins in Kubernetes:

- **CNI plugins**:

  The CNI networking plugin supports **pod's ingress and egress traffic**.
  The CNI plugin is selected by passing Kubelet the –network-plugin=cni

- **Kubernet plugin**:

  Kubenet is a very basic, simple network plugin, on Linux only. It does not, of itself, implement more advanced features like **cross-node networking** or network policy. It is typically used together with a cloud provider that sets up routing rules **for communication between nodes, or in single-node environments.**
- **Kubenet creates a Linux bridge named cbr0 and creates a veth pair for each pod with the host end of each pair connected to cbr0**
- The kubenet plugin is selected by passing Kubelet the –network –network-plugin=kubenet

Selector-

- Pods are associated to services via **key value pair** called **labels and selectors**, and in any pod the labels that matches the selector will automatically be discovered by the services.
- Labels do not provide uniqueness. In general, we can say many objects can carry the same labels (If u want to assign a service to a specific pod, we can use same label which we have provided during pod creation)
- based on selector we can connect with any pod.
- it should match with resource labels.

- The selector field defines how the Deployment finds which Pods to manage. In this case, you simply select a label that is defined in the Pod template (app: nginx)

How Does Kubernetes Networking Compared to Docker Networking?

- Kubernetes manages networking through CNI's on top of docker and just attaches devices to docker. While docker with docker swarm also has its own networking capabilities, such as overlay, bridging, etc, the CNI's provide similar types of functions.

There are 2 types of communication-

- **Intra-node Pod Network**
  Intra-node pod network is basically the communication between two different pods on the same node.
- **Inter-node pod network**
  Intra-node pod network is basically the communication between two different pods on the different node.

In Kubernetes Networking we have

- Container to container communication: two or more containers communication
- Pod to Pod communication: is the communication in between two different pods, having various images and replicas.
- Pod-to-Service communication: is how a service enables pod to communicate with any other pod
- External-to-service Communication: is how an external to service communicate which are from external network sources to cluster sources via ingress network.

In k8s we have 4 types of services/How do external services connect to these networks right?

1. **ClusterIP:**

   - It exposes resources at cluster level (all the nodes including master) internally so that other app/service can connect. Intranet.
   - A ClusterIP service is the default Kubernetes service. It gives you a service inside your cluster that other apps inside your cluster can access. There is no external access.

2. **NodePort:**

   - Exposes resources externally (means through internet). (end user can connect using Node Port).
   - NodePort **is a configuration setting you declare** in a service's YAML.
   - **Set the service spec's type to NodePort**. Then, Kubernetes will allocate a specific port on each Node to that service, and any request to your cluster on that port gets forwarded to the service.
   - Drawback- It has a range of **30000-32767** (we can expose only 2767 service externally)
   - You can only have once service per port.
   - **If your Node/VM IP address change, you need to deal with that**.
   - Internet

3. **LoadBalancer:**

   - Similar to NodePort (expose resources externally).
   - In the backend it creates NLB (Network load balancer)
   - A Load Balancer service is the standard way to expose a service to the internet. On GKE, this will spin up a Network Load Balancer that will give you a **single IP address** that will forward all traffic to your service.
   - You can set a service to be of type LoadBalancer the same way you'd set NodePort— specify the type property in the service's YAML.
   - **There needs to be some external load balancer functionality in the cluster, typically implemented by a cloud provider.**
   - **This is typically heavily dependent on the cloud provider—GKE creates a Network Load Balancer with an IP address that you can use to access your service.**
   - **Every time you want to expose a service to the outside world, you have to create a new LoadBalancer and get an IP address.**
   - **Drawback-Costly.**

4. **Ingress:** Refer ingress service section.

Syntax-Create services

```
vi syntax-service.yml#service (NodePort) manifest yaml file to expose pod

apiVersion: v1
kind: Service
metadata:
  name: <service name>                    # ex. mynginx-service/mytomcat-service/myapache-service
spec:
  selector:
ports:                                #port properties
   - port: <app default port>
  type: <services name>                   #NodePort/LoadBalancer/Ingress
```

Example-Services filefor (nginx/tomcat/apache)

```
vi nginx-service.yml#vi tomcat-service.yml/vi apache-service.yml

apiVersion: v1
kind: Service
metadata:
  name: nginx-service-prakash# tomcat-service-prakash/apache-service-prakash
spec:
  selector:
env: dev
    area: euro_region
authod: prakash
    app: abc
    tier: frontend
  ports:
   - port: 80#tomcat(8080)/apache(80)
  type: NodePort#We can provide any port name like-LoadBalancer/Ingress/CustomIP/Nodeport
```

To verify-

kubectl apply -f <service file name>

kubectl describe svc nginx-services-praksh…….to check to whom it is assigned.

Kubectl describe pods <pod name>

Copy Nodeport range and instance ip …chk in browser u will get the Message which we have added in the index .html file.

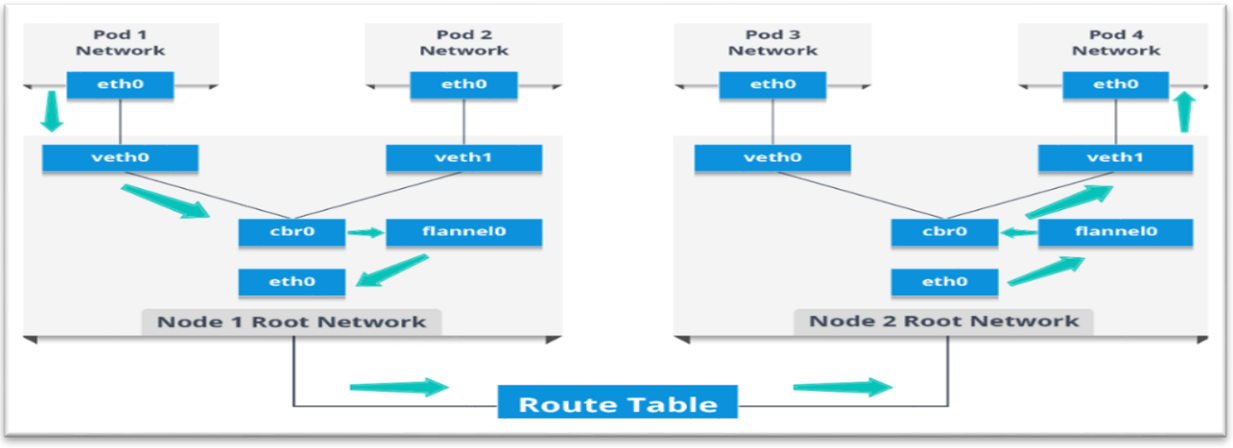
Ingress Service: -

https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html
https://kubernetes.io/docs/concepts/services-networking/ingress/
https://hackernoon.com/expose-your-kubernetes-service-from-your-own-custom-domains-cc8a1d965fc
https://www.magalix.com/blog/deploying-an-application-on-kubernetes-from-a-to-z

- Exposes resources externally.
- It is a collection of inbound and outbound rules.
- In the backend it uses NodePort and Load Balancer.
- In production we mainly Ingress service.
- It exposes services to outside the world **,it maps your URL to the services**.
- You configure access by creating a collection of rules that define which inbound connections reach which services.
- Ingress is actually NOT a type of service. Instead, it sits in front of multiple services and act as a "smart router" or entrypoint into your cluster.
- Ingress, on the other hand, is a completely independent resource to your service. **You declare, create and destroy it separately to your services.**
- This makes it decoupled and isolated from the services you want to expose. It also helps you to consolidate routing rules into one place.
- **The one downside is that you need to configure an Ingress Controller for your cluster. But that's pretty easy.**

- **Example-How external services are connected with the help of an ingress network.**

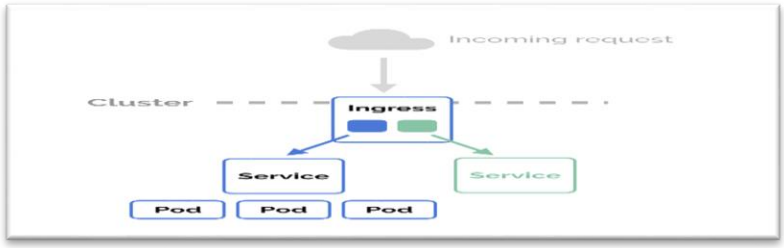
- We have 2 nodes, having the pod and root network namespaces with a Linux bridge. In addition to this, we also have a new virtual ethernet device called flannel0(network plugin) added to the root network.
- Now, we want the packet to flow from pod1 to pod 4.



- So, the packet leaves pod1's network at eth0 and enters the root network at veth0.

- Then it is passed on to cbr0, which makes the ARP (Address resolution protocol , it's a network layer protocol) request to find the destination and it thereafter finds out that nobody on this node has the destination IP address.
- So, the bridge sends the packet to flannel0 as the node's route table is configured with flannel0.
- Now, the flannel daemon talks to the API server of Kubernetes to know all the pod IPs and their respective nodes to create mappings for pods IPs to node IPs.
- The network plugin wraps this packet in a UDP packet with extra headers changing the source and destination IP's to their respective nodes and sends this packet out via eth0.
- Now, since the route table already knows how to route traffic between nodes, it sends the packet to the destination node2.
- The packet arrives at eth0 of node2 and goes back to flannel0 to de-capsulate and emits it back in the root network namespace.
- Again, the packet is forwarded to the Linux bridge to make an ARP request to find out the IP that belongs to veth1.
- The packet finally crosses the root network and reaches the destination Pod4.

How to write manifest file for Ingress-



1. Choose your ingress controller from below link (https://kubernetes.github.io/ingress-nginx/deploy/)

examples:

- Nginx Load Balancer:
- GCP https Load Balancer - GCE
- AWS ALB Ingress Controller
- AKS Application Gateway Ingress Controller
- HAProxy
- Istio

You can choose based on provider/feature/encryption method/cost

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-0.32.0/deploy/static/provider/baremetal/deploy.yaml

2. check if you have set it up correctly.

Kubectl get svc -n ingress-nginx
kubectl get pods --all-namespaces -l app=ingress-nginx

This kubectl get pods -n ingress-nginxhas set up the Nginx Ingress Controller. Now, we can create Ingress resources in our Kubernetes cluster and route external requests to our services.

Manifest file for 2 services-

First, let's create two services to demonstrate how the Ingress routes our request. We'll run two web applications that output a slightly different response.

For 1st service

```
kind: Pod
apiVersion: v1
metadata:
  name: apple-app
  labels:
    app: apple
spec:
  containers:
    - name: apple-app
      image: hashicorp/http-echo
args:
       - "-text=apple"

---
kind: Service
apiVersion: v1
metadata:
  name: apple-service
spec:
  selector:
    app: apple
  ports:
    - port: 5678 # Default port for image
```

For 2nd service

```
kind: Pod
apiVersion: v1
metadata:
  name: banana-app
  labels:
    app: banana
spec:
  containers:
    - name: banana-app
      image: hashicorp/http-echo
args:
       - "-text=banana"
---
```

```
kind: Service
apiVersion: v1
metadata:
  name: banana-service
spec:
  selector:
    app: banana
  ports:
    - port: 5678 # Default port for image
kind: Pod
apiVersion: v1
metadata:
  name: banana-app
  labels:
    app: banana
spec:
  containers:
    - name: banana-app
      image: hashicorp/http-echo
args:
       - "-text=banana"
---
apiVersion: v1
kind: Service

metadata:
  name: banana-service
spec:
  selector:
    app: banana
  ports:
    - port: 5678 # Default port for image
```

Create the resources

```
$ kubectl apply -f apple.yaml
$ kubectl apply -f banana.yaml
```

Now, declare an Ingress to route requests to `/apple` to the first service, and requests to `/banana` to second service. Check out the Ingress' `rules` field that declares how requests are passed along.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    ingress.kubernetes.io/rewrite-target: /
```

```
spec:
  rules:
  - http:
      paths:
        - path: /apple
          backend:
serviceName: apple-service
servicePort: 5678
        - path: /banana
          backend:
serviceName: banana-service
servicePort: 5678
```

Create the Ingress in the cluster-`kubectl create -f ingress.yaml`

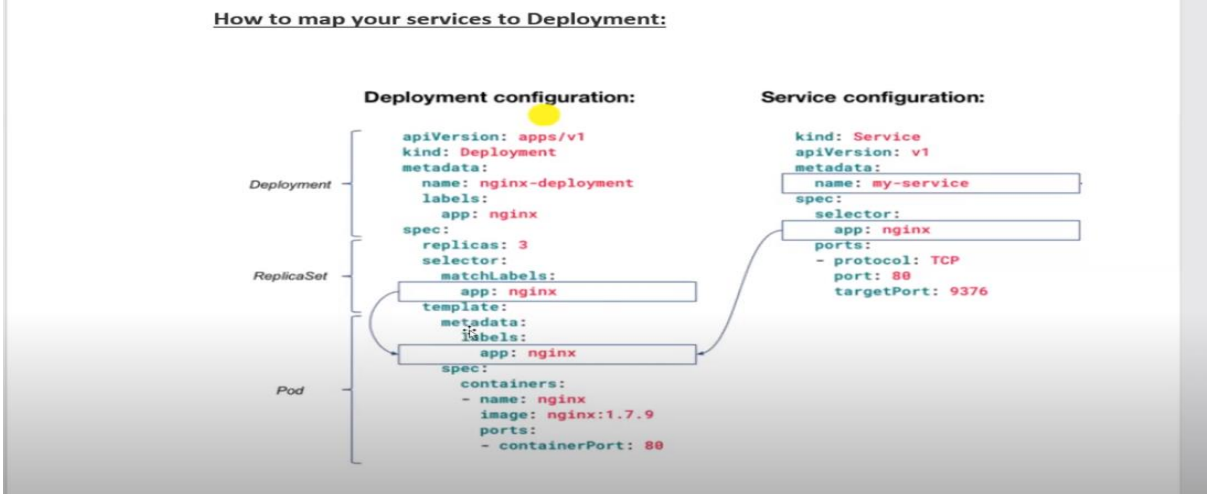
To verify-

```
$ curl -kL http://localhost/apple
apple
$ curl -kL http://localhost/banana
banana
$ curl -kL http://localhost/notfound
default backend - 404
```

==========

How to assign service to a specific pod/How to map your services to Deployment:

Using selector and label concept



vi myapp-service-deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeployment-nginx
  labels:                          #labels for your kind, kubectl describe deployment mydeployment-nginx
```

```
    app: myapp
    cloud: aws
spec:
  replicas: 2
  selector:
matchLabels:
     country: india
  template:
    metadata:
     labels:
       country: india
    spec:
     containers:
       - name: mycontainer
         image: nginx
         ports:
           - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: mydeployment-nginx-service
spec:
  selector:
    country: india
  ports:
    - port: 80
  type: NodePort
```

To verify-
kubectl apply -f <file name>
kubectl get deployment
kubectl delete pod <pod name for depployment> #k8s will maintain the desired state
kubectl get pods
kubectl get svc

How to access from brwoser: <pub ip of any node>:<NodePort>
==========26-9-2020

Q. How to deploy pods on any specific node?
Nodeselector

- **Means we have to label each node and we need to define it in the dockerfile**.——
- **syntax:kubeclt label node <node name><key of node selector>=<Value of node selector>**
  ex: kubectl label node ip-172-31-34-184 mynode-dev=dev-vm-01
- **kubectl get node --show-labels**……to get all labled node info

Manifest file for nodeSelector field-nginx/apache/tomcat

```
apiVersion: v1
kind: Pod
metadata:
  name: mynginx#apache/tomcat
labels:
    env: dev
    author: prakash
    app: abc
    tier: frontend
spec:
  containers:
    - name: mynginxcont#myapachecont/mytomcatcont
      image: pkw0301/maynginxdemo:01
imagePullPolicy: IfNotpresent
      ports:
        - containerPort: 80#apache-80/tomcat-8080
nodeSelector:
mynode-dev: dev-vm-01
```

==========

HPA (Horizontal Pod Autoscaler)

- With Horizontal Pod Autoscaling, Kubernetes automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU utilization
  (we also can set some other metrics such as memory)
- The resource determines the behaviour of the controller. The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user. (with a default value of 30 seconds)

Scaling policies:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
scaleTargetRef:

apiVersion: apps/v1
 kind: Deployment
    name: mydeployment-nginx                    #<Deployment Name>
minReplicas: 1
maxReplicas: 10
```

targetCPUUtilizationPercentage: 50

To verify-
kubectl get hpa
kubectl get hpa --watch
kubectl get pod –watch .....Watch is for current status.
kubectl get svc
curl localhost

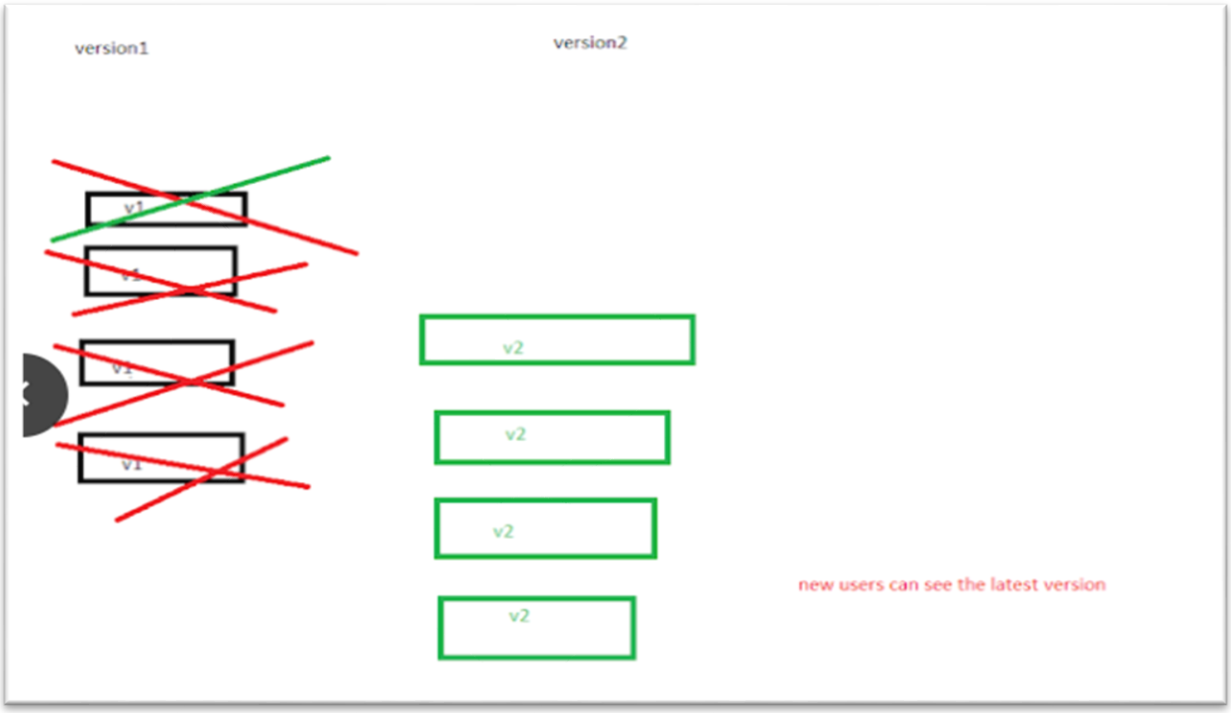
```
root@ip-172-31-22-213:~# kubectl get svc
NAME                       TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes                 ClusterIP   10.96.0.1       <none>        443/TCP        47m
mydeployment-nginx-service NodePort    10.105.37.249   <none>        80:31797/TCP   52s
root@ip-172-31-22-213:~# curl localhost:31797
<!DOCTYPE html>
```

==========

Rolling out new feature

- Without effecting end user in production environment by using this feature we can roll out the new feature.
- A feature rollout is the software development process of introducing a new feature to a set of users.
- A rolling update is the process of updating an application — whether it is a new version or just updated configuration
- To update a service without an outage, kubectl supports what is called rolling update, which updates one pod at a time, rather than taking down the entire service at the same time.
- Once the rollout is complete, the old controller is deleted
- Maintain min number of pod 75% , 25% capacity can be scaled/rollout/deleted.
- **Max Unavailable (means how many would be available)**
  - The maximum number of Pods that can be unavailable during the update process (The default value is 25 % for unavailable, then 75% would be available)
  - Ex: lets imagine we have 4 pods, here once we rollout new feature then 25% would be unavailable (means 3 pods will not get updated, one 1 gets updates.
  - Once 1st pod get updated then 2nd pod would get new updates and so on)
- **Max Surge**
  - The maximum number of Pods that can be created over the desired number of Pods (The default value is 25 %.)
  - (let's imagine we have 4 pods, then k8s will creates 1 pods in 1st rollout attempt and so on)

- Ideally, pods unavailability should be equal to max surge, so that k8s can handle load request pleasantly)



version 1 developments:

vi Dockerfile ---------version1

```
FROM nginx
COPY index.html /usr/share/nginx/html
EXPOSE 80
```

vi index.html ----------version1

```
Hi, Version1 deployed :)
```

docker build -t pkw0301/nginx-may01:01 .
docker images
docker push pkw0301/nginx-may01:01

vi k8s-may01.yml----------version1

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-may01
  labels:
    app: myapp
    cloud: aws
```

```
spec:
  replicas: 2
  selector:
matchLabels:
     country: india
  strategy:                          # Rolling out strategy:-
rollingUpdate:
maxSurge: 25%#the max no of pods that can be created
over the desired number, this is default value
maxUnavailable: 25%                #the max no if pods that can be unavailable/deleted
during the update process, if the default value
   type: RollingUpdate#is set to 25% for unavailable, then 75% would be available
template:
   metadata:
     labels:
       country: india
   spec:
     containers:
       - name: mycontainer
         image: pkw0301/nginx-may01:01
imagePullPolicy: Always  #Pull policy is set to always.
so during rollout new feature. it will always pull
        ports:                     # new image from container registry. consider latest version of docker
image
          - containerPort: 80

---
apiVersion: v1
kind: Service
metadata:
  name: nginx-may01-service
spec:
  selector:
    country: india
  ports:
    - port: 80
  type: NodePort
```

Let's update the code Version2

vi index.html ----------version2

Hi, Version2 deployed :)

Update docker tag

docker build -t pkw0301/nginx-may01:02 . ----------Version2

Upload new docker image

docker push pkw0301/nginx-may01:02 .

Commands To verify-

Rollout new feature:

syntax:

kubectl set image deployment <deployment name><container name>=<new docker image> --record

Example-

kubectl set image deployment nginx-may01 mycontainer=pkw0301/nginx-may01:02 --record

Then Reload the URL , New version will be deployed successfully without any downtime.

Check the status for your deployment:

syntax:

kubectl rollout status <Deployment Type>/<Deployment NAme>

Example-

kubectl rollout status deployment/nginx-may01

How to check the History of new Rollout:-

syntax:

kubectl rollout history <Deployment Type>/<Deployment NAme>

Example-

kubectl rollout history deployment/nginx-may01

How to Roll-Back to the previous version:

syntax:

kubectl rollout undo <Deployment Type>/<Deployment NAme> --to-revision=1

Example-

kubectl rollout undo deployment/nginx-may01 --to-revision=1

==========

27-9-2020

Through AWS (Service is costly as compared to GKE)-

Search for EKS

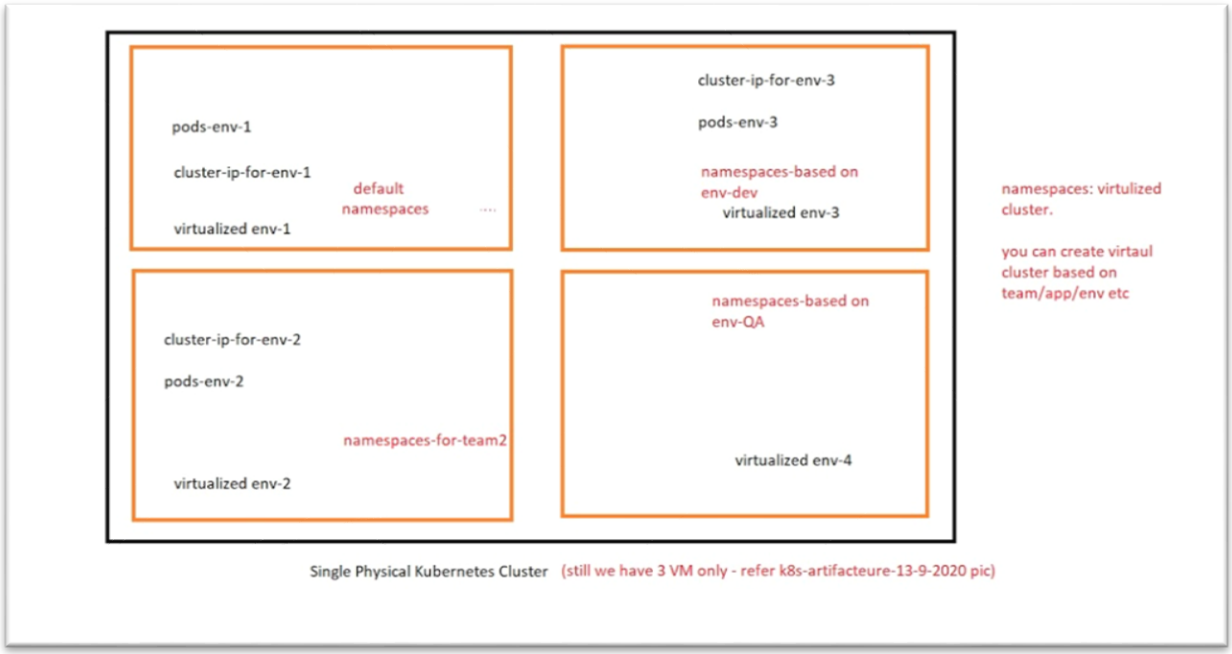
Through GKE Integration(Refer class document)-

1. Login to GCP (https://console.cloud.google.com/)-In Navigation Option- Kubernetes engine-cluster-Give cluster name-click on create.
2. Create k8s cluster-Home-cluster
3. You can create 1 or 2 VM as part of cluster:
4. Once cluster is ready then click on connect and run Gcloud Shell
5. Once your shell is ready then type below commands:
   kubectl run --image=nginx:latestmyweb
   kubectl get deployments
   kubectl get pods
   kubectl expose deployment <deployment name> --port=80 --target-port=80 --type=LoadBalancer
   kubectl get services
6. then copy load balancer's IP and paste in Google Chrome Browser:
   In backend GCP will create a Network Load Balancer: you can also check in console

==========

Namespace:

- If someone is asking to create 2 K8s clusters separately we can create but the problem is cost increase. So solution to this problem is **Namespaces**.
- Outer boundary is Physical cluster inside we have virtualized environment.
- It provides capability to virtualize k8s cluster (virtual cluster in Kubernetes)
- We can create namespaces for your environments example test, staging, production,
- By default, namespace is set as default.



- Namespace as a virtual cluster inside your Kubernetes cluster. You can have multiple namespaces inside a single Kubernetes cluster, and they are all **logically isolated from each other.**
- Namespace provides an additional qualification to a resource name. This is helpful when multiple teams are using the same cluster and there is a potential of name collision.
- It can be as a virtual wall between multiple clusters.
- **Inside the same namespace you can discover the other applications by service name.**
- The isolation namespaces provide allow you to reuse the same service name in different namespaces, resolving to the applications running in those namespaces. This allows you to create your different "environments" in the same

cluster if you wish to do so. For development, test, acceptance and production you would create 4 separate namespaces

- Kubectl get namespace

- **All objects such as pods, services, volumes, etc… are part of a namespace. If you do not specify a namespace when creating or viewing your objects, they will be created in the "default" namespace. When you want to interact with objects in a different namespace than "default", you must pass the -n flag to kubectl**

- kubectl get ns
  or
- kubectl get namespace

In k8s there are 4 namespaces

- **Default Namespace:** - by default k8s deploy resources in default namespaces. if you are not defining any namespaces in yamal file then k8s creates resources in default ns.

- **kube-system:**

  the resources which are created/managed k8s, we can't deploy resources in kube-system ns.
  kubectl get pods -n kube-system……Name space name
  Here u can see ebcd,scheduler and all.

- **kube-node-lease:** k8s uses kube-node-lease for node management.

- **kube-public:** data can be readable by non-authenticated/authenticated users. this ns is reserved for cluster usage. if some resources need to be shared with unauthenticated users.

Manifest file for Namespaces-

vi namcespace-deployment-demo.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-may01
  namespace: my-ns-poc#namespace field to deploy resources in custom or user managednamespaces.
labels:
```

```
    app: myapp
    cloud: aws
spec:
  replicas: 2……2 # tells deployment to run 2 pods matching the template
  selector:
matchLabels:
     country: india
  strategy:                          # Rolling out strategy:-
rollingUpdate:
maxSurge: 25%#the max no of pods that can be created over the desired number(default value)
maxUnavailable: 25% #the max no of pods that can be unavailable/deleted during the update.
    type: RollingUpdate#is set to 25% for unavailable, then 75% would be available

template:
    metadata:
      labels:
        country: india
    spec:
      containers:
        - name: mycontainer
          image: pkw0301/nginx-may01:01
imagePullPolicy: Always   #Pull policy is set to always.
so,during rollout new feature.it will always pull
          ports:                          #new image from container registry.
consider latest version of docker image
           - containerPort: 80
```

<u>you can custom namespace:based on team/app/env</u>
kubectl create ns <namespace name>
kubectl create ns my-ns-poc

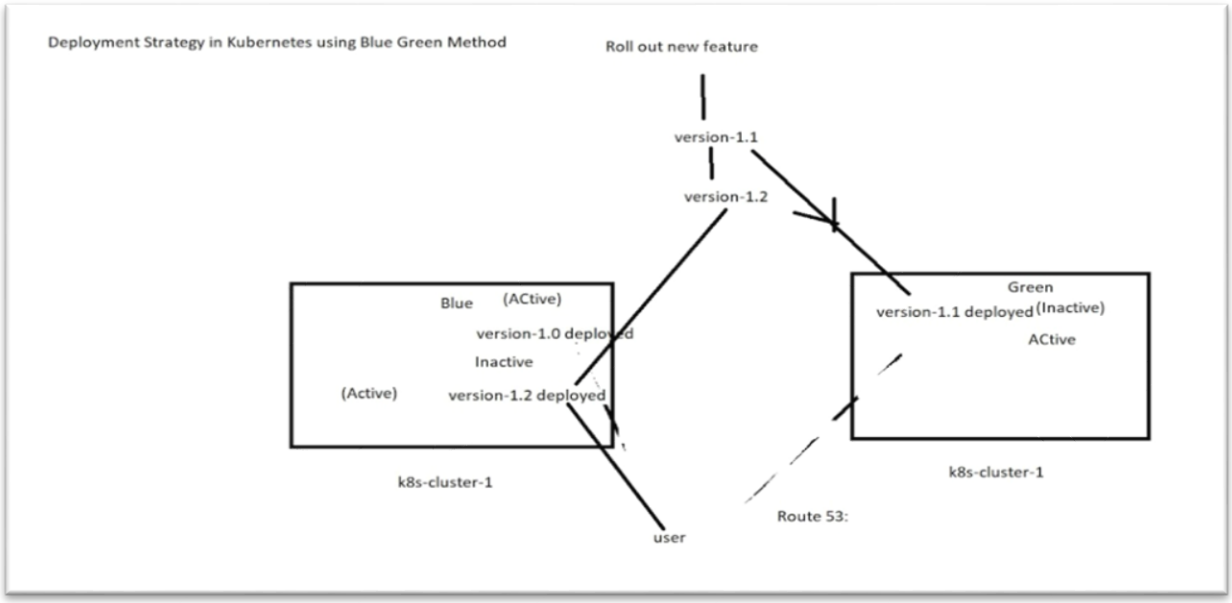
How to get deployed resources for any custom namepsace?????
kubectl get pods -n <namespace name>
ex: kubectl get pods -n my-ns-poc

How to get all the resources (resources deployed in any namespace)???
kubectl get pods --all-namespace

How to delete namespace????
kubectl delete ns <ns name>

==========
==========

Blue -Green deployment with Kubernetes



- It is also called as **Zero downtime deployment process** because in this type of process K8S is not going to delete or replacing exiting pod instead it creates new pod in new environment along with existing deployment.
- In this deployment method, two identical production environments work in parallel. One is the currently running production environment. It receives all user traffic (**depicted as Blue-Active**).
- The other environment is a clone of it, but idle (**Green-Inactive**). Both use the app configuration.
- When we want to deploy the new version of the application 1st it is deployed in the green environment and then testing and performance is monitored. Once the testing results are successful, application traffic is routed from blue to green.
- Green then becomes the new production. That is here green is active and blue is inactive now.
- Based on the selector in the production service, traffic will be routed either to deployment Blue or deployment Green.

LAB-

Practical demo: https://github.com/prakashk0301/blue-green-deployment-kubetnetes

1. Create app.py file
2. Create requirements.txt file
3. Create and push docker image

docker build -t pkw0301/app:1.0 .

docker login
docker push pkw0301/app:1.0

4. Create manifest file for blue environments
vi blue-1.yml

5. Create service manifest file for blue deployment
vi service-1.yaml

6. Deploy blue environment manifest file
kubectl apply -f blue-1.yml
kubectl get deployments
kubectl get pods

7. Create services for blue enviroments
kubectl apply -f service-1.yaml
kubectl get svc

access from browser: http://<K8s Master IP>:<NodePort Port>/ping

8. Now Let's change in the code (consider new feature)
in our app.py file, change the response from: Hi, This code is develop by Prakash Kumar: 1
to Hi, This code is develop by Prakash Kumar: 2

9. Now it's time to build docker image with new tag
docker build -t pkw0301/app:2.0 .
docker login
docker push pkw0301/app:2.0

10. Create manifest file for Green environments
vi green-2.yml

11. Deploy Green environment manifest file
kubectl apply -f green-2.yml --force
kubectl get deployments
kubectl get pods

12. In order to point to Green Environment update service manifest file and update version from 1.0 to 2.0
kubectl apply -f service-1.yaml

13. Reload application URL, now it should point to Green environment.
14. If there is a new change you can repeat the process.
15.Done.
==========

Interview question-On which google package or which Kubernetes you are working on??
Google engine OR GKE

How to restrict pod deployment on master node?

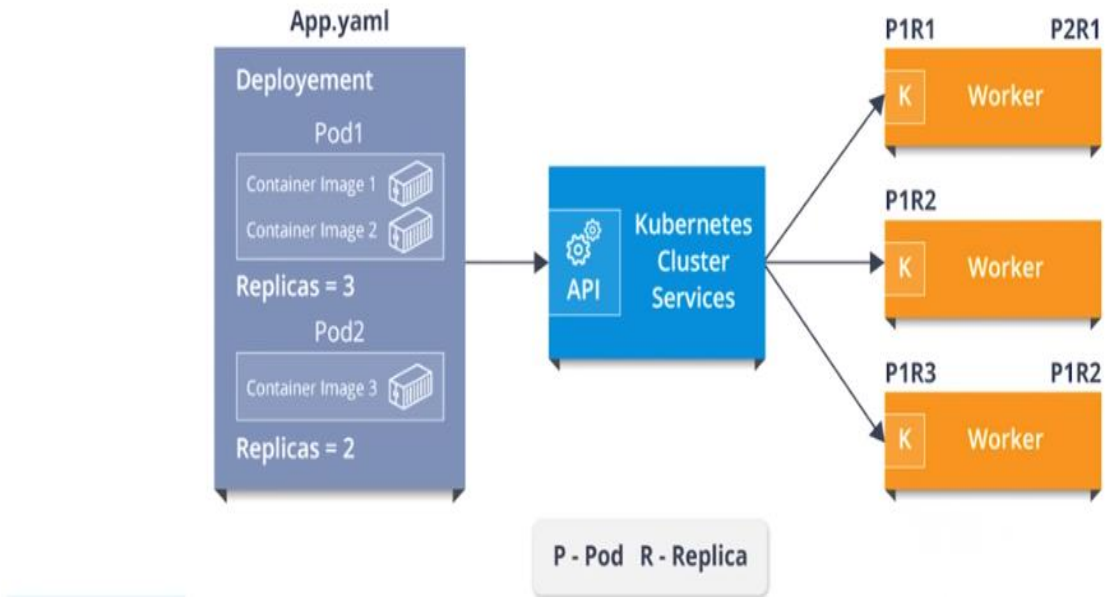
How to take backup of etcd??
Example-AWS S3

stateful or stateless application?????
Stateful- gke/ake.
Stateless-kuberdm

Kubernetes Cluster: Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit. The abstractions in Kubernetes allow you to deploy containerized applications to a cluster without tying them specifically to individual machines.

Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way.
A Kubernetes cluster consists of two types of resources:

- The Master coordinates the cluster
- Nodes are the worker that run applications



Kubernetes Namespaces:

Namespace as a **virtual cluster** inside your Kubernetes cluster. You can have multiple namespaces inside a single Kubernetes cluster, and they are all **logically isolated from each other**. Namespace provides an additional qualification to a resource name. This is helpful when multiple teams are using the same cluster and there is a potential of name collision. It can be as a virtual wall between multiple clusters.

Inside the same namespace **you can discover the other applications by service name**. The isolation namespaces provide allow you to **reuse the same service name in different namespaces**, resolving to the applications running in those namespaces. **This allows you to create your different "environments" in the same cluster** if you wish to do so. For development, test, acceptance and production you would create 4 separate namespaces

```
Kubectl get namespace
```

All objects such as pods, services, volumes, etc… are part of a namespace. If you do not specify a namespace when creating or viewing your objects, they will be created in the "default" namespace. When you want to interact with objects in a different namespace than "default", you must pass the -n flag to kubectl

```
kubectl get pods -n kube-system
kubectl get pod --all-namespaces
```

There are 3 Namespaces in k8s.

Default: The default namespace for objects with no other namespace

Kube-system: The namespace for objects created by the Kubernetes system

Kube-public: This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster.

How to create custom namespace:

```
kubectl create namespace test
kubectl delete ns test
```

vi deployment-namspace.yml

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: prod
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.9.1
        ports:
        - containerPort: 80
```

kubectl get deployment --all-namespaces

Similarly you create namespaces and assign to your **deployment, services, pod, replicas, replication controller** etc.

In **Kubernetes Networking** we have

- **Container to container communication**: two or more containers communication

- **Pod to Pod communication**: is the communication in between two different pods, having various images and replicas.

- **Pod-to-Service communication**: is how a service enables pod to communicate with any other pod

- **External-to-service Communication**: is how an external to service communicate which are from external network sources to cluster sources via ingress network.

How Does Kubernetes Networking Compared to Docker Networking?
Kubernetes manages networking through CNI's on top of docker and just attaches devices to docker. While docker with docker swarm also has its own networking capabilities, such as overlay, bridging, etc, the CNI's provide similar types of functions.

Network plugins in Kubernetes:

CNI plugins: The CNI networking plugin supports pod's ingress and egress traffic. The CNI plugin is selected by passing Kubelet the --network-plugin=cni

Kubenet plugin: Kubenet is a very basic, simple network plugin, on Linux only. It does not, of itself, implement more advanced features like cross-node networking or network policy. It is typically used together with a cloud provider that sets up routing rules for communication between nodes, or in single-node environments.
Kubenet creates a Linux bridge named cbr0 and creates a veth pair for each pod with the host end of each pair connected to cbr0
The kubenet plugin is selected by passing Kubelet the --network-plugin=kubenet

Services

- A service in Kubernetes is the entry for traffic into your application. It can be used for accessing an application just internally in the Kubernetes cluster or to expose the application.

- Basically, services are a type of resource that configures a proxy to forward the requests to a set of pods, which will receive traffic & is determined by the selector. Once the service is created it has an assigned IP address which will accept requests on the port.
- Now, there are various service types that give you the option for exposing a service outside of your cluster IP address.

Types of Services

There are mainly 3 types of services.

ClusterIP*:* This is the default service type which exposes the service on a cluster-internal IP by making the service only reachable within the cluster.

NodePort: This exposes the service on each Node's IP at a static port. Since, a **ClusterIP** service, to which the NodePort service will route, is automatically

created. We can contact the NodePort (range 30000–32767) service outside the cluster.

LoadBalancer: This is the service type which exposes the service externally using a cloud provider's load balancer. So, the NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

How do external services connect to these networks right?

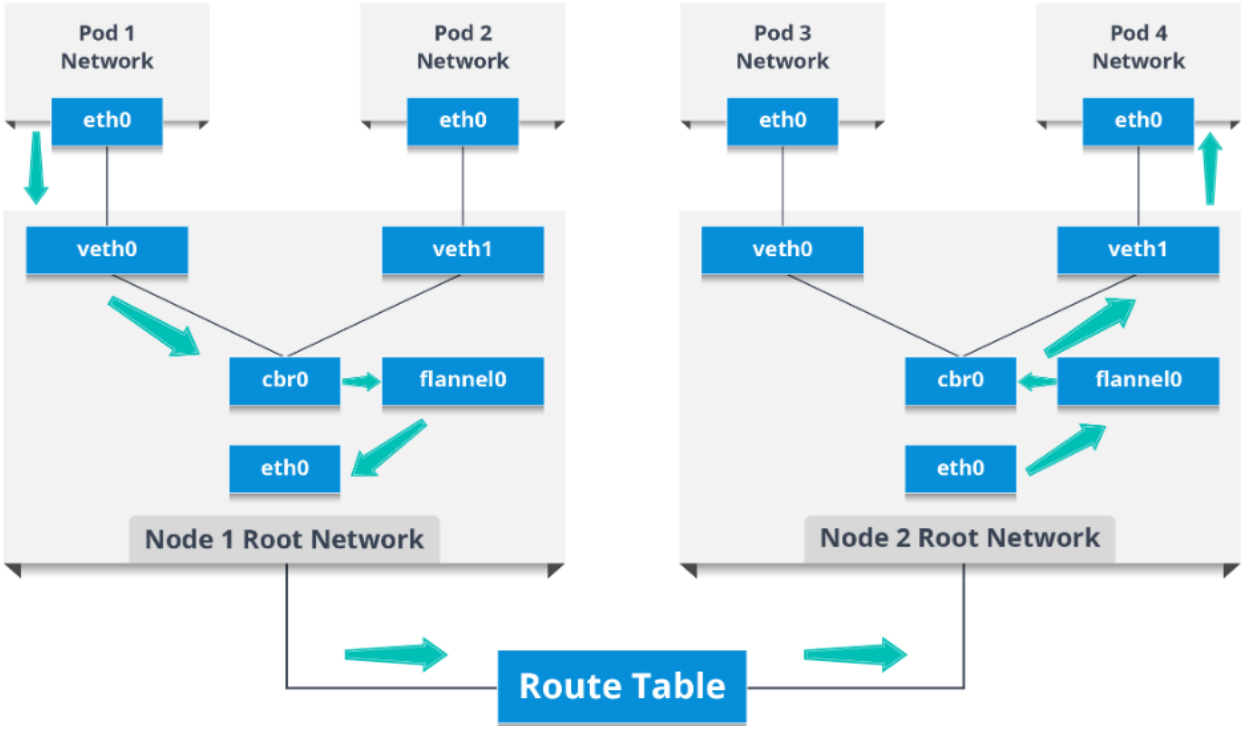
NodePort, LoadBalancer Ingress Network

Ingress Network

Ingress network is also an option for exposing services as it is a collection of rules that allow inbound connections that can be configured to give services externally through reachable URLs. **It supports URL based routing**. So, it basically acts as an entry point to the Kubernetes cluster that manages external access to the services in a cluster. **CNI** as an interface between network providers and **Kubernetes** networking

We have 2 nodes, having the pod and root network namespaces with a Linux bridge. In addition to this, we also have a new virtual ethernet device called flannel0(network plugin) added to the root network.

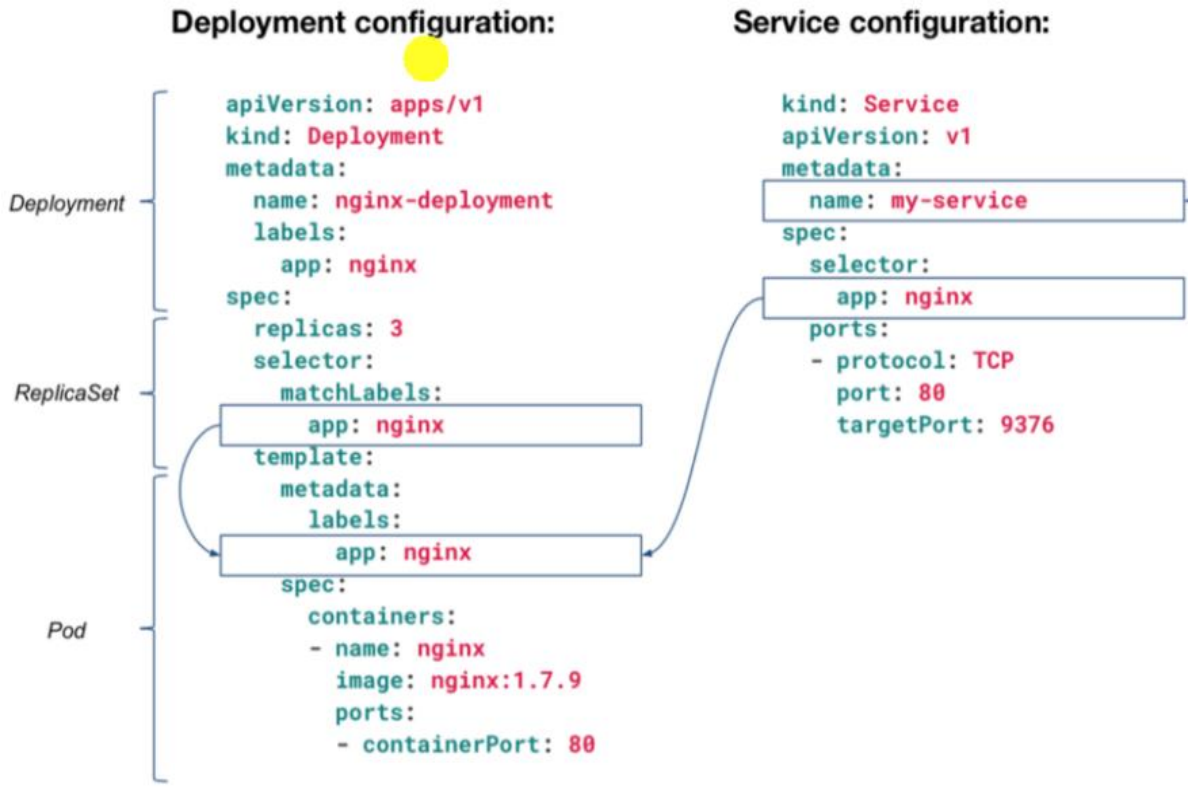
Now, we want the packet to flow from pod1 to pod 4.



- So, the packet leaves pod1's network at eth0 and enters the root network at veth0.
- Then it is passed on to cbr0, which makes the ARP (Address resolution protocol , it's a network layer protocol) request to find the destination and it thereafter finds out that nobody on this node has the destination IP address.
- So, the bridge sends the packet to flannel0 as the node's route table is configured with flannel0.
- Now, the flannel daemon talks to the API server of Kubernetes to know all the pod IPs and their respective nodes to create mappings for pods IPs to node IPs.
- The network plugin wraps this packet in a UDP packet with extra headers changing the source and destination IP's to their respective nodes and sends this packet out via eth0.
- Now, since the route table already knows how to route traffic between nodes, it sends the packet to the destination node2.
- The packet arrives at eth0 of node2 and goes back to flannel0 to de-capsulate and emits it back in the root network namespace.
- Again, the packet is forwarded to the Linux bridge to make an ARP request to find out the IP that belongs to veth1.
- The packet finally crosses the root network and reaches the destination Pod4.

That's how external services are connected with the help of an ingress network.

How to map your services to Deployment:



Lab 1:

Scenario: How can we assign a service to running deployment?

Step 1: Create a folder in your directory and change the working directory path to that folder

mkdir service-assignment
cd service-assignment

Step2: Now create deployment YAML files, for the web application

vi webapp.yml

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: webapp1
  labels:
    app: webapp-sql
    tier: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webapp-sql
      tier: frontend
  template:
    metadata:
      labels:
        app: webapp-sql
        tier: frontend
    spec:
      containers:
      - name: webapp1
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Step3: Once you create the deployment files, deploy the applications.

```
kubectl apply -f webapp.yml
kubectl get deployment
```

Step 4: Now, you have to create services (NodePort) for the applications.

vi webservice.yml

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-sql
spec:
  selector:
    app: webapp-sql
    tier: frontend
  ports:
```

```
  - port: 80
  type: NodePort
```

```
kubectl apply -f webservice.yml
kubectl get service
```

Step 5: Now, check the configuration of running pods.

```
kubectl get pods
```

Lab 2: Load Balancer

vi deployment-for-load-balancer.yml

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx-deployment
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx-deployment
    spec:
      containers:
      - name: nginx-deployment
        image: nginx:1.9.1
        ports:
        - containerPort: 80
```

vi service-for-loadbalancer.yml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-deployment
spec:
  selector:
    app: nginx-deployment
```

```
ports:
- port: 80
type: LoadBalancer
```

<VM IP>:80 , would work within cluster.

For external, map svc port and ip with AWS ELB (Though we installed **kubernetes using kubeadmin** and haven't mapped with AWS/or any cloud so can't get load balancer IP, we will get pending status for **EXTERNAL-IP**)

```
kubectl get svc nginx-deployment
```

Service discovery

Service discovery is the process of figuring out how to connect to a service. While there is a service discovery option based on environment variables available (disadvantage of doing this is that the scheduler might put both Pods on the same machine, which will take your entire Service down if it dies), the DNS-based (you can talk to the Service from any pod in your cluster,) service discovery is preferable.

```
kubectl get services kube-dns --namespace=kube-system
```

Endpoints in Services:

- Endpoints track the IP Addresses of the objects the service send traffic to.
- When a service selector matches a pod label, that IP Address is added to your endpoints

```
kubectl get svc
kubectl describe svc <service name>
kubectl get pods
kubeclt describe pod <pod name>
```

Kubernetes Installation:

To create a Kubernetes Cluster, we need minimum 2 EC2 instances.

Create first VM as AWS **ubuntu** instance (**t2 medium** for master)

Second VM as AWS ubuntu instance (**t2micro** for node)

Step 1. Run following command on Master and Node VM (VM1 & VM2), We need to install kubeadm,kubectl,kubelet and cni

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

Update both the Instances and install docker

```
apt-get update
apt-get install -y docker.io
sudo apt-get install -y iptables arptables ebtables
systemctl enable docker
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
```

(Some important process in kubernetes.)

kubeadm: command to bootstrap the cluster.

kubelet: component that runs on all of the machines in your cluster and does things like starting pods and containers.

kubectl: command line utility to communicate within your cluster.

kubernetes-cni: used for kubernetes networking)

Step 2. Run following command on **Master** VM (VM1) only

```
kubeadm init
```

Copy kubeadm join output command in notepad

(It will look like... kubeadm join --token ef6acb.11b6129ab3a2fbe0 172.31.22.102:6443 --discovery-token-ca-cert-hash sha256:e83e2872f599eea6b13d42a7b4a01a55958a78c6439192c8d2bb4578797de686)

Step 3. Run following command on Master VM (VM1)

```
sudo cp /etc/kubernetes/admin.conf $HOME/
sudo chown $(id -u):$(id -g) $HOME/admin.conf
export KUBECONFIG=$HOME/admin.conf
kubectl apply -f https://git.io/weave-kube
OR
kubectl apply -f https://cloud.weave.works/k8s/net?k8s-version=1.10
```

Step 4. To Docker cgroup driver matches the kubelet config (VM1)

```
docker info | grep -i cgroup
cat /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
```

Step 6. To restarting kubelet run following command (VM1)

```
systemctl daemon-reload
systemctl restart kubelet
```

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
kubectl taint nodes --all node-role.kubernetes.io/master-
kubectl apply -f https://git.io/weave-kube-1.6
```

Step 7. Run kubeadm join output command on node VM (VM2) (copy output command from step 2).

Once you run kubeadm join command (Token) on Node VM, your VM will become part of Kubernetes cluster.

`kubectl get node` ----> run on Master VM

wait for some time status would get changed from Not Ready to Ready

Step 8. To change label of your Node, Run following command on master VM

Syntax: `kubectl label nodes <node-name> <label-key>=<label-value>` ---->Node label syntax

Ex: `kubectl label nodes ip-172-31-21-230 node-role.kubernetes.io/ethans-prakash-node1=ip-172-31-21-230` ----> update your node name

kubectl label nodes ip-172-31-22-231 node-role.kubernetes.io/ethans-prakash-node2=ip-172-31-22-231

kubectl get nodes ----> to check nodes

kubectl get pods ----> to check running pods

Kubernetes Deployment

Deployment controller provides declarative updates for pods.

- Create our deployment to rollout replica sets, the replication creates pods in the background, check the status of rollout if it succeeds or not.
- Declares the state of the pods by updating the pod template spec of their deployment, a new replica that was created and the deployment managers moving the pods from the old replica sets to the new one at a controlled rate (all are pods are not getting updated in a single shot). Each new replica sets updates the revision of the deployment.
- Roll back to an earlier deployment version if the current state of the deployment is not stable. Each rollback updates the revision of that deployment.
- Scale up the deployment to facilitate to more load.
- Pause the deployment, this is to apply multiple fixes to its pod templates, and then resume it to start a new rollout.
- Use the status of that deployment as an indicator that the rollout has stuck.
- Clean up old replica sets. If they don't need any more

Perform Rolling Update

A rolling update is the process of updating an application — whether it is a new version or just updated configuration

To update a service without an outage, kubectl supports what is called rolling update, which updates one pod at a time, rather than taking down the entire service at the same time.

Once the rollout is complete, the old controller is deleted

Assume that we have a current replication controller named foo and it is running image image:v1

Syntax: kubectl rolling-update NAME NEW_NAME --image=IMAGE:TAG

kubectl rolling-update foo [foo-v2] --image=myimage:v2

Recovery

If a rollout fails or is terminated in the middle, it is important that the user be able to resume the roll out

Recovery is achieved by issuing the same command again:

kubectl rolling-update foo [foo-v2] --image=myimage:v2

Aborting a rollout

Abort is assumed to want to reverse a rollout in progress.

kubectl rolling-update foo [foo-v2] --rollback

mkdir && cd /root/ngninx1.9-lab

vi index,htm >>> hi, this is nginx 1.9.1

Lab 1: vi Dockerfile

```
FROM nginx:1.9.1
COPY index.html /usr/share/nginx/html
```

docker build -t nginx:1.9.1 .

s

Please put different content for both the index.html

Mkdir && cd /root/ngninx1.7-lab

vi index.htm >>> hi, this is nginx 1.7

vi Dockerfile

```
FROM nginx:1.7.9
COPY index.html /usr/share/nginx/html
```

docker build -t nginx:1.7.9 .

docker images

vi replication-nginx-1.7.9.yaml

We can create k8s manifest file :-

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: pkw0301/nginx:1.7.9
        ports:
        - containerPort: 80
```

```
kubectl create -f replication-nginx-1.7.9.yaml
kubectl get ReplicationController
or
kubectl get rc
kubectl describe ReplicationController <name of your ReplicationController >

kubectl expose rc <rc name> --port=80 –target-port=80 –type=NodePort

kubectl get svc

<VM IP>:<NodePort>
```

To update to version 1.9.1, (imperative method)

```
kubectl set image deployment.v1.apps/my-nginx nginx=pkw0301/nginx:1.9.1

kubectl get svc

<VM IP>:<NodePort>
```

A rolling update works by:

1. Creating a new replication controller with the updated configuration.
2. Increasing/decreasing the replica count on the new and old controllers until the correct number of replicas is reached.
3. Deleting the original replication controller.

If you encounter a problem, you can stop the rolling update **midway and revert** to the previous version using `--rollback`:

```
kubectl rolling-update my-nginx --rollback
```

expose your rc

```
kubectl expose rc my-nginx --port=80 --target-port=80 --type=NodePort
Now if you want to rollback once rollout completed
kubectl rolling-update my-nginx --image=nginx:1.7.9
```

Rolling Updates with a Deployment

vi deployment-rollingupdate.yml

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
```

```
    - containerPort: 80
```

```
Kubectl create -f deployment-rollingupdate.yml
Kubectl get deployment
Kubectl describe deployment <deployment name>
```

To make change in our application lets update nginx image version from 1.7.9 to 1.9.8

Once you change in your yaml

```
kubectl apply –f <ymal file name>

kubectl get deployment
kubectl describe deployment <deployment name>
kubectl rollout status deployment/nginx-deployment
```

Rollback

```
kubectl rollout undo deployment/nginx-deployment

Once you rollback verify your configuration

kubeclt describe deployment <deployment name>
```

Note: During this new feature rollout

Max Unavailable (means how many would be available)
The maximum number of Pods that can be unavailable during the update process (The default value is 25 % for unavailable, then 75% would be available)

Ex: lets imagine we have 4 pods, here once we rollout new feature then 25% would be unavailable (means 3 pods will not get updated, one 1 gets updates.

Once 1st pod get updated then 2nd pod would get new updates and so on)

Max Surge

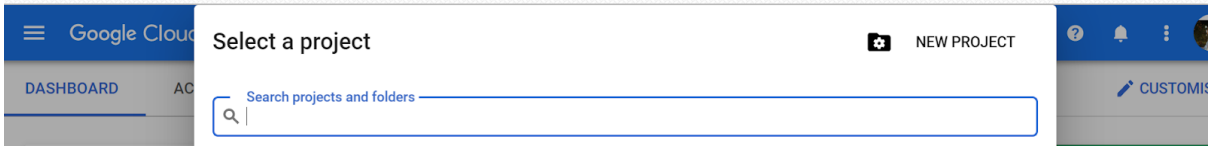
The maximum number of Pods that can be created over the desired number of Pods (The default value is 25 %.)

(let's imagine we have 4 pods, then k8s will creates 1 pods in 1st rollout attempt and so on)

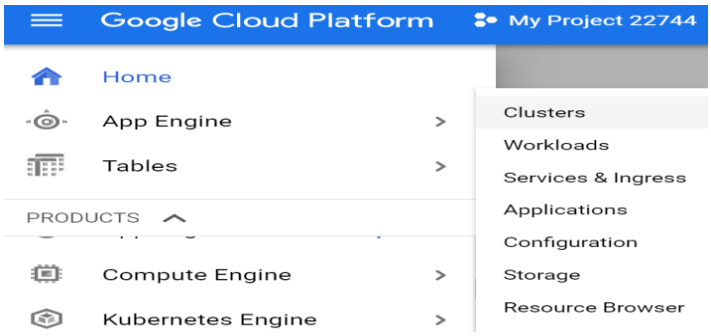
(Ideally, pods unavailability should be equal to max surge, so that k8s can handle load request pleasantly)

==========
==========GCP K8s Lab:

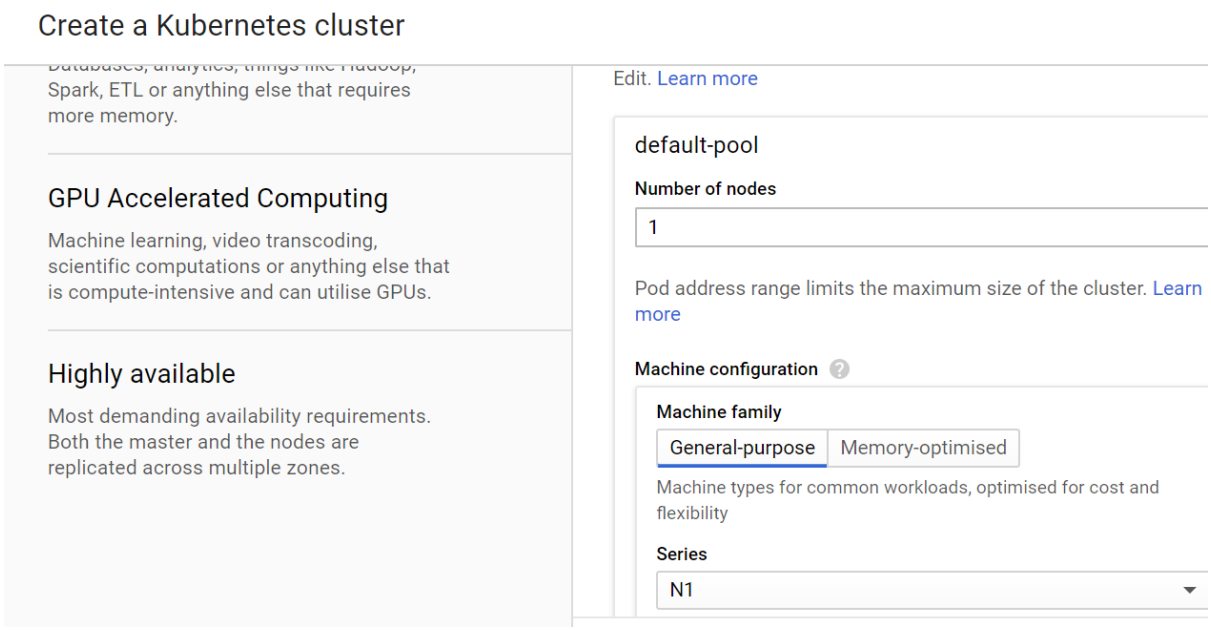
1. Login to GCP: https://console.cloud.google.com/
2. Create a project:



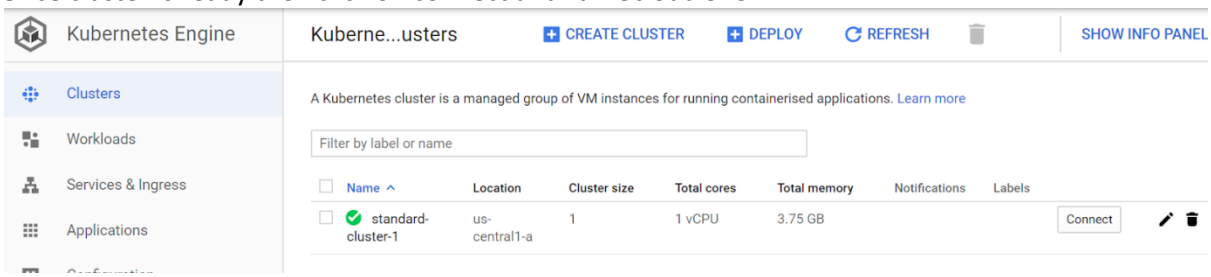
3. Create k8s cluster:



4. You can create 1 or 2 VM as part of cluster:

5. 

6. Once cluster is ready then click on connect and run Gcloud Shell



7. Once your shell is ready then type below commands:

kubectl run --image=nginx:latest myweb
kubectl get deployments
kubectl get pods

How to expose your deployments to outside the word using LoadBalancer or NodePort
syntax:
kubectl expose deployment <deployment name> --port=80 --target-port=80 --type=LoadBalancer

ex:
kubectl expose deployment myweb --port=80 --target-port=80 --type=LoadBalancer

kubectl get services
then copy load balancer's IP and paste in Google Chrome Browser:

In backend GCP will create a Network Load Balancer: you can also check in console

points to remember:
below commands are deprecated

kubectl run --image=nginx:latest myweb
kubectl run --generator=deployment/apps.v1 my-web2 --image=nginx

K8s recommends to use create command by having manifest file:-
kubectl create -f <manifest file name>