Networking in Kubernetes :

Kubernetes networking is an integral part of managing and making communication easier within a Kubernetes cluster. It manages a wide range of operations, such as:

- Handling internal container communication
- Exposing the containers to the internet

First, let's review the basic networking types in a Kubernetes cluster:

- Container-to-container networking
- Pod-to-pod networking
- Pod-to-service networking
- Internet-to-service networking

Container-to-container networking :

In the most basic configurations, container to container communication within a single pod takes place via the localhost and port numbers. This is possible because the containers in the pod are located in the same network namespace.

A network namespace is a logical networking stack that includes routes, firewall rules, and network devices which provides a complete network stack for processes within the namespace. There can be multiple namespaces on the same virtual machine.

In Kubernetes, there is a hidden container called a pause container that runs on all the pods in Kubernetes. This container keeps the namespace open even if all the other containers in the pod are nonfunctional.

A new networking namespace is assigned to each pod created. Containers within the pod can use this networking namespace via localhost. However, a user must be aware of the port conflicts: If the containers use the same port, networking issues will arise within the containers.

Pod-to-pod networking :

Each pod in a Kubernetes node gets assigned a dedicated IP address with a dedicated namespace. Due to an automatic assignment, users do not need to explicitly create links between pods. This allows us to tread pod networking operations such as port allocation, load balancing, and naming similar to a virtual machine or a physical host.

Kubernetes imposes three fundamental requirements on any network.

1) Pods on a node can communicate with all pods on all nodes without NAT.
2) Agents on a node (system daemons, kubelet) can communicate with all the pods on that specific node.
3) Pods in a host network of a node can communicate with all pods on all nodes without NAT (Only for platforms such as Linux, which supports pods running on the host network)

Communication between pods in the same node

Kubernetes creates a virtual ethernet adapter for each pod, and it is connected to the network adaptor of the node.

When a network request is made, the pod connects through the virtual ethernet device associated with the pod and tunnels the traffic to the ethernet device of the node.

In a Kubernetes node, there is a network bridge called cbro, which facilitates the communication between pods in a node. All the pods are a part of this network bridge. When a network request is made, the bridge checks for the correct destination (pod) and directs the traffic.

Communication between pods in different nodes

When the requested IP address cannot be found within the pod, the network bridge directs the traffic to the default gateway. This would then look for the IP within the cluster level.

Kubernetes keeps a record of all the IP ranges associated with each node. Then an IP address is assigned to the pods within the nodes from the range assigned to the node. When a request is made to an IP address in the cluster, it will:

1) Look for the IP range for the requested IP.
2) Then direct it to the specific node and then to the correct pod.

3) **Pod-to-service networking :**

A service is an abstraction that routes traffic to a set of pods. In other words, a service will map a single IP address to a set of pods.

When a network request is made, the service proxies the request to the necessary pod. This proxy service happens via a process called Kube-proxy that runs inside each node.

The service would get its IP within the cluster when a request first reaches the service IP before being forwarded to the actual IP of the pod. This is an important feature in Kubernetes—it decouples the dependency of networking directly to each pod. The pods can be created and destroyed without worrying about network connectivity. This is because the service will automatically update its endpoints when the pods change with the IP of the pod.

The service would get its IP within the cluster. Any request first reaches the service IP before being forwarded to the actual IP of the Pod. This is an important feature in Kubernetes as it decouples the dependency of networking directly to each Pod. Thus, pods can get created and destroyed without worrying about network connectivity as the service will automatically update its endpoints with the IP of the pod that it targets as the Pods changes.

A service knows which Pods to target using the label selector. The label selector is the core grouping primitive in Kubernetes; via a label selector, the client or user can identify a set of objects.

- The label selector will look for the specified labels in each Pod and match them with the service accordingly.
- Without the selector, properly configured services cannot keep track of the Pods and will lead to communication issues within the cluster.

DNS for internal routing in a Kubernetes cluster

Each cluster comes with inbuilt service for DNS resolution.

- A domain name is assigned to each service in the cluster
- A DNS name is assigned to the Pods.

(These can be manually configured via the YAML file using the hostname and subdomain fields.)

When a request is made via the domain name, the Kubernetes DNS service will resolve the request and point it to the necessary internal service from the service. The Kube-proxy process will point it to the endpoint pod.

Internet-to-service networking :

In the above sections, we have covered the internal networking basics of a Kubernetes cluster. The next step is to expose the containerized application to the internet.

Unlike the previous networking types, exposing the Kubernetes cluster to the internet depends on both the:

- Underlying cluster infrastructure
- Network configurations

The most common method used to handle traffic is by using a load balancer. In Kubernetes, we can configure load balancers. When a service is created, users can:

- Specify a load balancer that will expose the IP address of the load balancer.
- Direct traffic to the load balancer and communicate with the service through it.

Tools for Kubernetes networking :

There are plenty of tools that we can use to configure networking within Kubernetes while adhering to all the guidelines and requirements imposed by the cluster.

- **Cilium :**
  It is an open-source software for providing and securing network connectivity between application containers. Cilium is L7/HTTP aware and can enforce network policies from L3-L7 using identity-based security models.

- **Flannel :**
  It is a simple network overlay that meets Kubernetes requirements.

- **Kube-router :**
  It is a turnkey solution for Kubernetes networking. Kube-router provides a lean and powerful alternative to default Kubernetes network components from a single DaemonSet/Binary.

- **Antrea :**
  It is a Kubernetes-native open source networking solution that leverages Open vSwitch as the networking data plane.

And here are cloud vendor-specific container network interfaces:

AWS VPC CNI for Kubernetes is the Amazon Web Services specific CNI.
Azure CNI for Kubernetes is the Microsoft Azure specific CNI.

Creating a Manifest for creating Nginx Pod :

```
#vim nginx-pod.yml
apiVersion : v1
kind : Pod
metadata :
  name : nginx-pod
  labels :
    app : nginx
    tier : dev
spec :
  containers :
  - name : nginx-container
    image : nginx
:wq!
```

#kubectl create -f nginx-pod.yml (**Deploying POD using the manifest file**)

#kubectl get pods (**Lists all the pods in K8's Cluster**)

#kubectl get pod <Pod Name> (**Displays information of a specific Pod**)

#kubectl get pod -o wide (**To fetch on which worker node the pod is scheduled**)

#kubectl get pod nginx-pod -o yaml (**Printing pod configuration information in YML Format**)
#kubectl get pod nginx-pod -o json (**Printing pod configuration information in JSON Format**)

#kubectl describe pod nginx-pod (**Displaying the detailed information of a specific pod**)

#kubectl delete pod nginx-pod (**Deleting Pod**)

#kubectl get pod

ReplicationController :

If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods. Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated. For example, your pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, you should use a ReplicationController even if your application requires only a single pod. A ReplicationController is similar to a process supervisor, but instead of supervising individual processes on a single node, the ReplicationController supervises multiple pods across multiple nodes.

ReplicationController is often abbreviated to "rc" in discussion, and as a shortcut in kubectl commands.

A simple case is to create one ReplicationController object to reliably run one instance of a Pod indefinitely. A more complex use case is to run several identical replicas of a replicated service, such as web servers

- Replication controllers and pods are associated with "**LABELS**"

Creating Manifest for Replication Controller :

#vim nginx-rc.yml
apiVersion : v1
kind : ReplicationController

```
metadata :
  name : nginx-rc
spec :
  replicas : 3
  template :
    metadata :
      name : nginx-pod
      labels :
        app : nginx-app
    spec :
      containers :
      - name : nginx-container
        image : nginx
        ports :
        - containerPort : 80
  selector :
    app : nginx-app
:wq!
```

#kubectl get rc (**Lists all the available Replication Controllers on K8's Cluster**)

#kubectl get pods

#kubectl get po -l app=nginx-app (**Displaying pods having a spcific labels**)

#kubectl describe rc nginx-rc (**Displaying complete information of Replication Controller**)

#kubectl get po -o wide(**Displaying complete information of Replication Controller**)

NOTE : Manually turn of the worker node which has POD for Testing the availability

#kubectl get nodes

#kubectl get pods

#kubectl get po -o wide

Now we will scale up the nginx App :

#kubectl scale rc nginx-rc --replicas=6 (**Scaling up to total 6 Replicas**)

#kubectl get rc nginx-rc

#kubectl get pods

#kubectl get po -o wide

Now we will scale down the nginx App :

#kubectl scale rc nginx-rc --replicas=2 (**Scaling down to 2 Replicas**)

#kubectl get rc nginx-rc

#kubectl get pods

#kubectl get po -o wide

Deleting Replication Controller :

#kubectl delete -f nginx-rc.yml

#kubectl get rc

#kubectl get pods

ReplicaSet :

A ReplicaSet (RS) is one of the Kubernetes controllers that ensures a specified number of pod replicas are running at a given time. Without ReplicaSets, we would have to create multiple manifests for the number of pods needed which is very tedious.

In previous versions of Kubernetes, this was called Replication Controller. The main difference between the two is that ReplicaSets allow us to us something called Label Selector. Labels are key value pairs used to specify attributes of objects that are meaningful and useful to users, so keep in mind that It doesn't change the way the core system works. Label Selectors is used to identify a set of objects in Kubernetes.

Creating Manifest for replica Set :

```
#vim nginx-rs.yml
apiVersion : apps/v1
kind : ReplicaSet
metadata :
  name : nginx-rs
spec :
  replicas : 3
  template :
    metadata :
      name : nginx-pod
      labels :
        app : nginx-app
        tier : frontend
    spec :
      containers :
      - name : nginx-container
        image : nginx
        ports :
        - containerPort : 80
  selector :
    matchExpressions :
    - {key: tier,app, operator: In, values: [frontend, nginx-app]}
:wq!
```

#kubectl create -f nginx-rs.yml (**Deploying the Manifest file for creating Replica Set**)
#kubectl get pods -o wide
#kubectl get pods -l app=nginx-app
#kubectl get rs
#kubectl get rs nginx-rs
#kubectl get rs nginx-rs -o wide
#kubectl describe rs nginx-rs

NOTE :

To perform how Replicaset is providing the availability to the pods, Do manually power of your worker node and test Whether Replica Set is recreating the new POD as per the desired number mentioned in the manifest file.

Scale up :

#kubectl scale rs nginx-rs --replicas=7
#kubectl get pods -o wide
#kubectl get rs nginx-rs -o wide

Scale Down :

#kubectl scale rs nginx-rs --replicas=3
#kubectl get pods -o wide
#kubectl get rs nginx-rs -o wide

Cleanup :

#kubectl delete -f nginx-rs.yml
#kubectl get rs
#kubectl get pods -l app=nginx-app

Kubernetes DaemonSet

The DaemonSet feature is used to ensure that some or all of your pods are scheduled and running on every single available node. This essentially runs a copy of the desired pod across all nodes.

When a new node is added to a Kubernetes cluster, a new pod will be added to that newly attached node.

When a node is removed, the DaemonSet controller ensures that the pod associated with that node is garbage collected. Deleting a DaemonSet will clean up all the pods that DaemonSet has created.

DaemonSets are an integral part of the Kubernetes cluster facilitating administrators to easily configure services (pods) across all or a subset of nodes.

DaemonSet use cases :

DaemonSets can improve the performance of a Kubernetes cluster by distributing maintenance tasks and support services via deploying Pods across all nodes. They are well suited for long-running services like monitoring or log collection. Following are some example use cases of DaemonSets:

- To run a daemon for cluster storage on each node, such as glusterd and ceph.
- To run a daemon for logs collection on each node, such as Fluentd and logstash.
- To run a daemon for node monitoring on every note, such as Prometheus Node Exporter, collectd, or Datadog agent.

Depending on the requirement, you can set up multiple DaemonSets for a single type of daemon, with different flags, memory, CPU, etc. that supports multiple configurations and hardware types.

Scheduling DaemonSet pods :

By default, the node that a pod runs on is decided by the Kubernetes scheduler. However, DaemonSet pods are created and scheduled by the DaemonSet controller. Using the DaemonSet controller can lead to Inconsistent Pod behavior and issues in Pod priority preemption.

To mitigate these issues, Kubernetes (ScheduleDaemonSetPods) allows users to schedule DaemonSets using the default scheduler instead of the DaemonSet controller. This is done by adding the NodeAffinity term to the DaemonSet pods instead of the .spec.nodeName term. The default scheduler is then used to bind the Pod to the target host.

Daemon Set Lab Demo 1 (This Example is going to deploy One Pod Per Worker Node) :

```
#vim fluentds.yml
apiVersion : apps/v1
kind : DaemonSet
metadata :
  name : fluentd-ds
spec :
```

```
  template :
    metadata :
      labels :
        name : fluentd
    spec :
      containers :
      - name : fluentd
        image : gcr.io/google-containers/fluentd-elasticsearch:1.20
  selector :
    matchLabels :
      name : fluentd
:wq!
```

#kubectl get nodes
#kubectl create -f fluentds.yml
#kubectl get ds
#kubectl get pods
#kubectl get pods -o wide
#kubectl describe ds fluentd-ds

Cleanup :
#kubectl delete ds fluentd-ds
#kubectl get ds
#kubectl get pods
Lab Demo 2 : (In this example lets deploy pod with Nginx container on selected worker nodes)

```
#vim nginx-ds-subsetnodes.yml
apiVersion : apps/v1
kind : DaemonSet
metadata :
  name : nginx-ds
spec :
  template :
    metadata :
      labels :
        name : nginx
    spec :
      containers :
```

```
    - name : nginx-container
      image : nginx
    nodeSelector :
      node : server
  selector :
    matchLabels :
      name : nginx
```

:wq!

First we need to label the Workernodes inside the cluster :
#kubectl get nodes
#kubectl label nodes wn1 wn2 node=server (**Creating labels to worker nodes**)
#kubectl get nodes --show-labels (**Lists all the nodes with the labels created**)

#kubectl create -f nginx-ds-subsetnodes.yml
#kubectl get pods
#kubectl get pods -o wide
#kubectl get ds
#kubectl describe ds nginx-ds

Cleanup :
#kubectl delete ds nginx-ds
#kubectl get ds
#kubectl get pods

What is a Kubernetes DaemonSet?

The DaemonSet feature is used to ensure that some or all of your pods are scheduled and running on every single available node. This essentially runs a copy of the desired pod across all nodes.

When a new node is added to a Kubernetes cluster, a new pod will be added to that newly attached node.

When a node is removed, the DaemonSet controller ensures that the pod associated with that node is garbage collected. Deleting a DaemonSet will clean up all the pods that DaemonSet has created.

DaemonSets are an integral part of the Kubernetes cluster facilitating administrators to easily configure services (pods) across all or a subset of nodes.

DaemonSet use cases :

DaemonSets can improve the performance of a Kubernetes cluster by distributing maintenance tasks and support services via deploying Pods across all nodes. They are well suited for long-running services like monitoring or log collection. Following are some example use cases of DaemonSets:

- To run a daemon for cluster storage on each node, such as glusterd and ceph.
- To run a daemon for logs collection on each node, such as Fluentd and logstash.
- To run a daemon for node monitoring on every note, such as Prometheus Node Exporter, collectd, or Datadog agent.

Depending on the requirement, you can set up multiple DaemonSets for a single type of daemon, with different flags, memory, CPU, etc. that supports multiple configurations and hardware types.

Scheduling DaemonSet pods :

By default, the node that a pod runs on is decided by the Kubernetes scheduler. However, DaemonSet pods are created and scheduled by the DaemonSet controller. Using the DaemonSet controller can lead to Inconsistent Pod behavior and issues in Pod priority preemption.

To mitigate these issues, Kubernetes (ScheduleDaemonSetPods) allows users to schedule DaemonSets using the default scheduler instead of the DaemonSet controller. This is done by adding the NodeAffinity term to the DaemonSet pods instead of the .spec.nodeName term. The default scheduler is then used to bind the Pod to the target host.

Daemon Set Lab Demo 1 (This Example is going to deploy One Pod Per Worker Node) :

#vim fluentds.yml
apiVersion : apps/v1
kind : DaemonSet

```
metadata :
  name : fluentd-ds
spec :
  template :
    metadata :
      labels :
        name : fluentd
    spec :
      containers :
      - name : fluentd
        image : gcr.io/google-containers/fluentd-elasticsearch:1.20
  selector :
    matchLabels :
      name : fluentd
:wq!
```

```
#kubectl get nodes
#kubectl create -f fluentds.yml
#kubectl get ds
#kubectl get pods
#kubectl get pods -o wide
#kubectl describe ds fluentd-ds
```

Cleanup :

```
#kubectl delete ds fluentd-ds
#kubectl get ds
#kubectl get pods
```

Lab Demo 2 : (In this example lets deploy pod with Nginx container on selected worker nodes)

```
#vim nginx-ds-subsetnodes.yml
apiVersion : apps/v1
kind : DaemonSet
metadata :
  name : nginx-ds
spec :
  template :
    metadata :
```

```
      labels :
        name : nginx
    spec :
      containers :
      - name : nginx-container
        image : nginx
      nodeSelector :
        node : server
  selector :
    matchLabels :
      name : nginx
```

:wq!

First we need to label the Workernodes inside the cluster :
#kubectl get nodes
#kubectl label nodes wn1 wn2 node=server (**Creating labels to worker nodes**)
#kubectl get nodes --show-labels (**Lists all the nodes with the labels created**)

#kubectl create -f nginx-ds-subsetnodes.yml
#kubectl get pods
#kubectl get pods -o wide
#kubectl get ds
#kubectl describe ds nginx-ds

Cleanup :
#kubectl delete ds nginx-ds
#kubectl get ds
#kubectl get pods