Kubernetes Networking: Core Concepts and Communication

PART-1

1. Core Concepts of Kubernetes Networking

Kubernetes networking is built around a few core principles. These principles form the backbone of how services, pods, and external traffic interact within the cluster. Kubernetes abstracts many networking complexities through CNIs (Container Network Interfaces), flat networks, and its service model, making Kubernetes networking scalable and adaptable to various cloud and on-premises environments.

Kubernetes assumes a few fundamental principles:

1. **Every pod gets its own IP address**, and this address is routable across the entire cluster.
2. **Pods communicate with each other using their IP addresses**, regardless of which node they're on.
3. **No Network Address Translation (NAT)** is required for pod-to-pod communication within the cluster.
4. **Network traffic flows freely** across the entire cluster unless explicitly restricted by network policies.

1.1. Kubernetes Networking Model

Kubernetes' IP-per-pod model is one of its defining features, contrasting sharply with container management platforms that rely on port mapping. Each pod gets its own IP, which simplifies container-to-container communication by allowing containers in different pods to communicate as if they were on the same network.

Kubernetes implements this via a network overlay, managed by a CNI plugin. Some popular CNIs include:

- **Flannel**: Simplifies network configuration by providing an overlay network using VXLAN.
- **Calico**: Combines routing and network policies with performance optimization through BGP (Border Gateway Protocol).
- **Weave**: Offers simplicity with encrypted communications and a self-managing network mesh.

In the Kubernetes networking model, traffic between pods is assumed to flow freely, but restrictions can be applied via **network policies** to control ingress and egress traffic at the pod level.

Container Network Interface (CNI) Overview

A **Container Network Interface (CNI)** is the layer that abstracts the complexity of how pods communicate inside the cluster. The CNI is a specification for container runtime plugins to set up networking. Popular CNIs like Calico, Flannel, and Weave take care of IP address assignment, routing, and traffic policies within the Kubernetes cluster.

Each CNI plugin implements the Kubernetes networking model differently. For example:

- **Calico** uses BGP routing, making it more suitable for large-scale environments.
- **Flannel** uses a simpler VXLAN approach that encapsulates network traffic.
- **Weave** provides an encrypted mesh network for pod communication.

When deploying a Kubernetes cluster, the choice of CNI affects how the cluster handles networking, load balancing, security policies, and scalability.

---

2. Pod-to-Pod Communication

Pod-to-pod communication is one of the fundamental networking behaviors in Kubernetes. It allows microservices-based applications, composed of multiple pods, to interact seamlessly within a cluster. Communication between pods happens either within the same node or across nodes.

2.1. Intra-Node Pod Communication

Pods within the same node communicate through the node's internal networking. When two or more pods reside on the same node, they can communicate using their unique IP addresses. These IP addresses are assigned by the CNI plugin and managed by Kubernetes.

Intra-Node Communication Example:

1. **Two Pods on the Same Node**: Pod A and Pod B reside on Node 1. The CNI plugin assigns them IP addresses 10.244.1.2 and 10.244.1.3, respectively.
2. **Pod Communication**: If Pod A wants to send a request to Pod B, it can use Pod B's IP address to establish a connection.

This communication does not require any additional network hops or encapsulation and happens directly within the node.

```
# Get the pod's internal IP address

kubectl get pods -o wide
```

This command provides the IP addresses of all the pods. Pods within the same node can then communicate using these IPs directly, using HTTP requests, for example.

2.2. Inter-Node Pod Communication

When pods reside on different nodes, Kubernetes uses the CNI plugin to manage the communication between them. The communication between different nodes typically involves routing traffic through an overlay network or via direct routes depending on the CNI plugin being used.

How Inter-Node Communication Works

1. **Node 1 and Node 2**: Let's assume Pod A is on Node 1 and Pod B is on Node 2. Each pod has its own unique IP assigned by the CNI.
2. **CNI Plugin**: The CNI plugin, such as Calico or Flannel, ensures that the traffic between nodes is properly routed, either through an overlay network (such as VXLAN) or directly using BGP (in the case of Calico).
3. **Traffic Encapsulation**: If using an overlay network (like with Flannel), the traffic is encapsulated and tunneled from one node to another to maintain the illusion of a flat network.

Calico's Approach:

- Calico uses BGP to directly exchange routing information between nodes. This avoids the overhead of tunneling traffic and allows for high-performance pod-to-pod communication across nodes. Calico can also enforce network policies to restrict traffic between specific pods.

Example of Inter-Node Pod Communication:

If Pod A (IP: 10.244.1.3) on Node 1 wants to communicate with Pod B (IP: 10.244.2.4) on Node 2, the CNI plugin ensures that the packet is routed correctly between nodes, even though the pods are on different physical hosts.

Network Plugin Configuration for Pod-to-Pod Communication

For example, Flannel can be installed using the following YAML:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

With Flannel configured, Kubernetes will manage the networking across nodes, ensuring seamless pod-to-pod communication.

---

3. Pod-to-Service Communication

Pods need to communicate not just with each other but also with Kubernetes services that expose workloads and provide stable endpoints. Services abstract the underlying pods and provide a consistent interface for clients (whether internal or external to the cluster).

3.1. The ClusterIP Service

A **ClusterIP** service is the default service type in Kubernetes. It exposes an internal IP address that other pods inside the cluster can use to communicate with the service. A ClusterIP service allows for dynamic scaling of backend pods without affecting the client-side configuration, as the service IP remains constant.

- **Service Discovery**: Kubernetes provides internal DNS-based service discovery, allowing pods to communicate using service names instead of hardcoded IP addresses.
- **Load Balancing**: The service acts as a load balancer, distributing traffic across multiple pod replicas.

ClusterIP Service Example

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: ClusterIP
```

In this example:

- The service backend-service will forward traffic on port 80 to the backend pods' port 8080.
- Pods in the cluster can access the service using http://backend-service.

DNS-Based Pod-to-Service Communication

Kubernetes automatically creates a DNS entry for each service, making it easy for pods to communicate with services using service names. For example, if the service backend-service is in the default namespace, it will be available at http://backend-service.default.svc.cluster.local.

3.2. Headless Services for Stateful Applications

Sometimes, it's necessary to bypass the load-balancing behavior of services and connect directly to the individual pods behind a service. This is where **headless services** come into play. A headless service does not provide a ClusterIP or load balancing. Instead, it directly exposes the IP addresses of the individual pods.

Headless Service Example

```
apiVersion: v1
kind: Service
metadata:
  name: stateful-app
spec:
  clusterIP: None
  selector:
    app: myapp
  ports:
    - port: 80
```

In this example:

- clusterIP: None creates a headless service, meaning clients will get the pod IP addresses directly when querying the service.
- This is useful for stateful applications like databases, where each pod may need to be addressed individually.

4. External Access via NodePort and LoadBalancer

In addition to internal pod-to-service communication, Kubernetes provides mechanisms for exposing services to the external world. Two primary methods for external access are **NodePort** and **LoadBalancer** services.

4.1. NodePort Service

A **NodePort** service allows you to expose a service on a specific port on each node in the cluster. This port is within the range 30000-32767. External clients can then access the service by sending traffic to any node's IP address on the specified NodePort.

How NodePort Works

When you create a NodePort service, Kubernetes allocates a port from the NodePort range. This port is then opened on every node in the cluster, and any traffic sent to that port is forwarded to the service.

NodePort Service Example

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  type: NodePort
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 30080
```

In this example:

- Kubernetes will expose the service on port 30080 across all nodes in the cluster.
- External clients can access the service using http://<node-ip>:30080.

4.2. LoadBalancer Service

In cloud environments (AWS, GCP, Azure), Kubernetes supports **LoadBalancer** services, which automatically provision external load balancers from the cloud provider's infrastructure. This service type simplifies the process of exposing services directly to the internet and includes built-in load balancing across nodes and pod replicas.

How LoadBalancer Works

1. When a LoadBalancer service is created, Kubernetes provisions an external load balancer using the cloud provider's APIs.
2. The load balancer is configured to route traffic to the NodePorts of the service, which in turn forward traffic to the backend pods.

LoadBalancer Service Example

```
apiVersion: v1
kind: Service
metadata:
  name: loadbalancer-service
spec:
  type: LoadBalancer
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

In this example:

- Kubernetes will create an external load balancer for the service.
- Clients can access the service using the load balancer’s external IP address.

Cloud-Specific Behavior

Each cloud provider integrates differently:

- **AWS**: Creates an Elastic Load Balancer (ELB).
- **GCP**: Creates a GCP Load Balancer.
- **Azure**: Provisions an Azure Load Balancer.

Once the load balancer is created, external traffic is forwarded to the nodes, which in turn route it to the pods behind the service.

---

5. Ingress: Managing External HTTP(S) Traffic

For more advanced traffic routing, especially for HTTP and HTTPS traffic, Kubernetes provides the **Ingress** API. Ingress allows you to define rules for how external traffic is routed to services within the cluster. Unlike NodePort and LoadBalancer services, which are tied to individual services, Ingress provides a central point to manage multiple services and paths.

5.1. What is Ingress?

Ingress is an API object that manages external access to services, usually over HTTP and HTTPS. It allows for:

- **Path-based routing**: Route traffic based on URL paths (e.g., /app1 to one service, /app2 to another).
- **Host-based routing**: Route traffic based on domain names (e.g., app1.mycompany.com to one service, app2.mycompany.com to another).
- **SSL/TLS termination**: Ingress can handle SSL/TLS termination at the edge, providing secure access to services.

5.2. Ingress Controller

Ingress requires an **Ingress Controller** to function. The Ingress Controller is responsible for processing Ingress rules and configuring a reverse proxy to route the traffic accordingly. Popular ingress controllers include:

- **NGINX Ingress Controller**
- **HAProxy Ingress Controller**
- **Traefik**

Without an ingress controller running in the cluster, the Ingress resources will have no effect.

5.3. Ingress Resource Example

Let’s say you have two services, app1-service and app2-service, and you want to route traffic based on the URL path.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  rules:
  - host: mycompany.com
    http:
      paths:
      - path: /app1
        pathType: Prefix
        backend:
          service:
            name: app1-service
            port:
              number: 80
      - path: /app2
        pathType: Prefix
        backend:
          service:
            name: app2-service
            port:
              number: 80
```

In this example:

- Traffic sent to mycompany.com/app1 will be routed to the app1-service.
- Traffic sent to mycompany.com/app2 will be routed to the app2-service.

5.4. TLS Termination with Ingress

Ingress also allows for **SSL/TLS termination**, which means that the ingress controller will handle the SSL certificates, providing HTTPS access to the services.

TLS Ingress Example

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  tls:
  - hosts:
    - mycompany.com
    secretName: tls-secret
  rules:
  - host: mycompany.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: app-service
            port:
              number: 80
```

In this example:

- TLS termination is handled by the ingress controller using the certificate stored in the Kubernetes secret tls-secret.
- The traffic is encrypted using HTTPS, and the ingress controller will decrypt the traffic and forward it to the app-service on port 80.

Advanced Features of Ingress

Ingress provides much more advanced traffic management features, including:

- **Rate limiting**: Limit the number of requests per second for certain paths or hosts.
- **Redirects**: Redirect traffic from HTTP to HTTPS.
- **Backend affinity**: Sticky sessions to ensure that traffic from the same client always hits the same backend pod.

Conclusion

Kubernetes networking is a vast and complex area, with many moving parts that enable seamless communication between pods, services, and external clients. From pod-to-pod communication using the CNI plugin, to exposing services externally using NodePort and LoadBalancer, to managing HTTP(S) traffic using Ingress, Kubernetes provides a flexible and scalable networking model.

In this detailed guide, we covered:

- **Core networking concepts** like the IP-per-pod model and flat network topology.
- **Pod-to-pod communication** both within and across nodes.

- **Services**, including ClusterIP, headless services, and service discovery via DNS.
- **External access** via NodePort and LoadBalancer services.
- **Ingress**, which provides advanced HTTP routing, SSL termination, and path-based routing.