Kubernetes Networking

An In-Depth Look

Lisa Zangrando

Kubernetes Networking

Overview

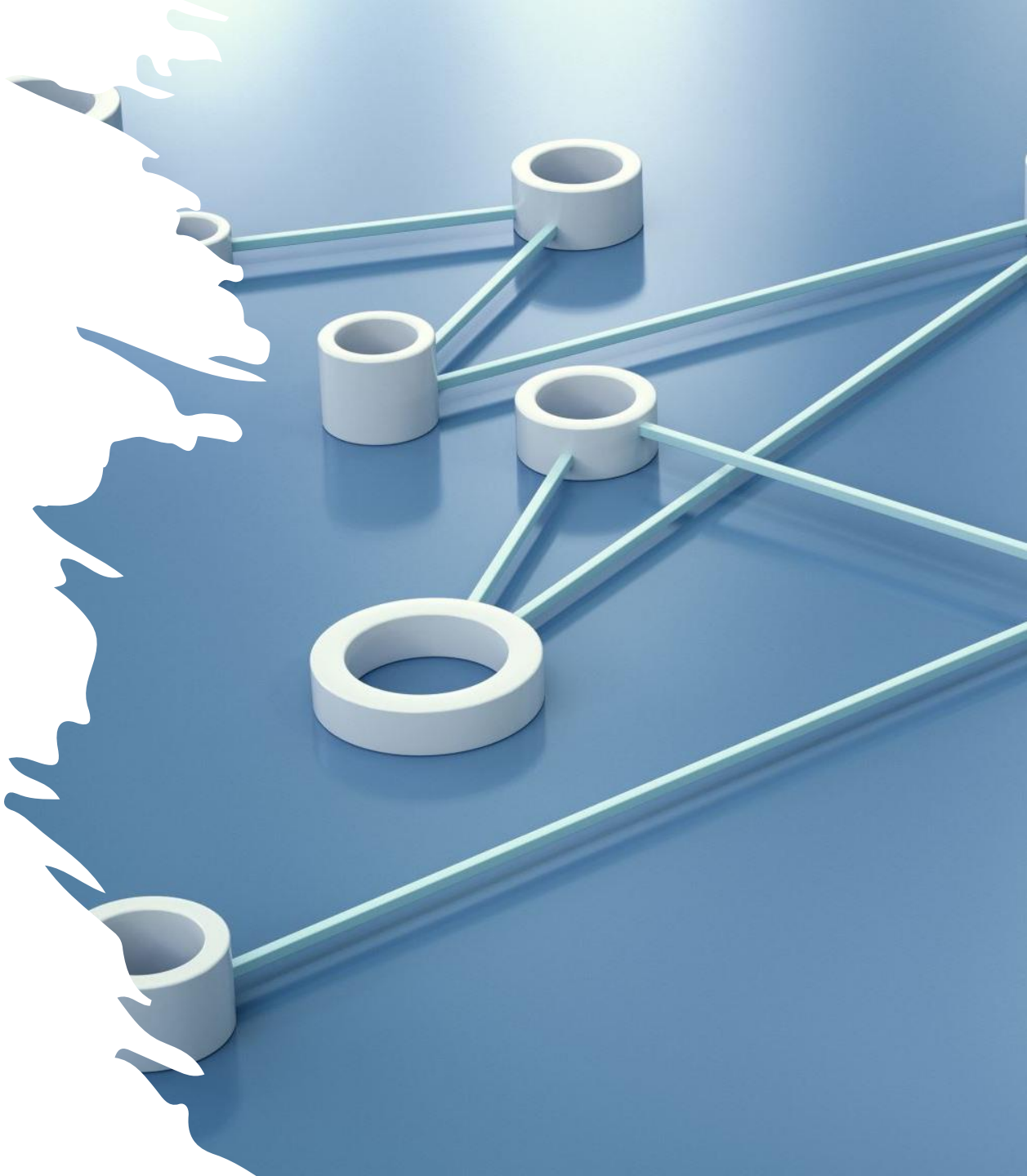
The Kubernetes networking model allows the different parts of a Kubernetes cluster, such as Nodes, Pods, Services, and outside traffic, to communicate with each other.

Why understanding it matters

- Properly configure your environment.
- Enable **complex networking scenarios**.

Key concepts covered

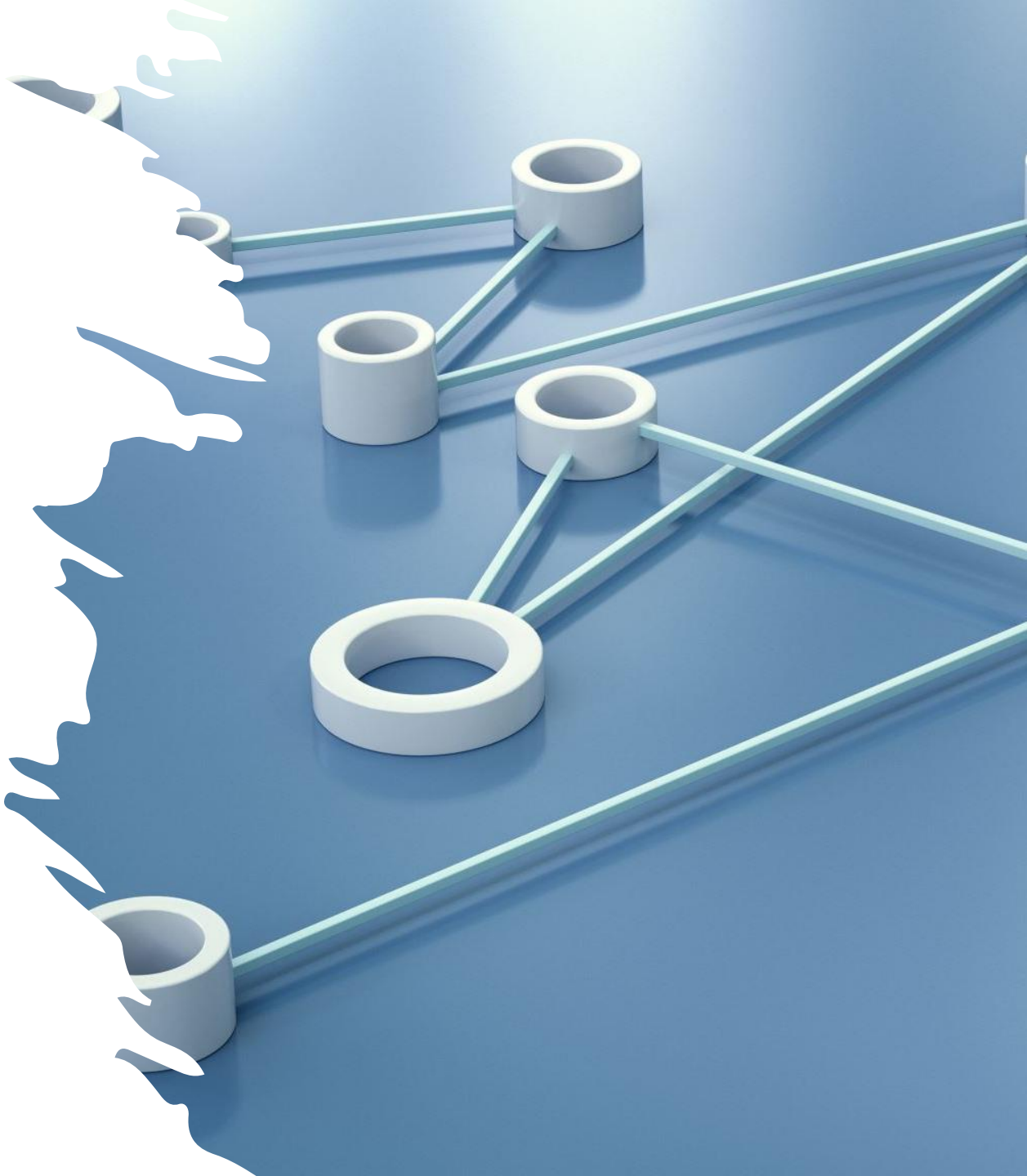
- **Networking Model**
- **Cluster communication types**:
  - Container-to-Container
  - Pod-to-Pod
  - Pod-to-Service
  - Internet-to-Service



Kubernetes Networking model

- The Kubernetes networking model is designed around the following key principles:
  - Every pod gets its own IP address
  - Containers within a pod share the pod IP address and can communicate freely with each other
  - Pods can communicate with all other pods in the cluster using pod IP addresses (without NAT)
  - Isolation (restricting what each pod can communicate with) is defined using network policies
  - Plugin-based flexibility and customization.

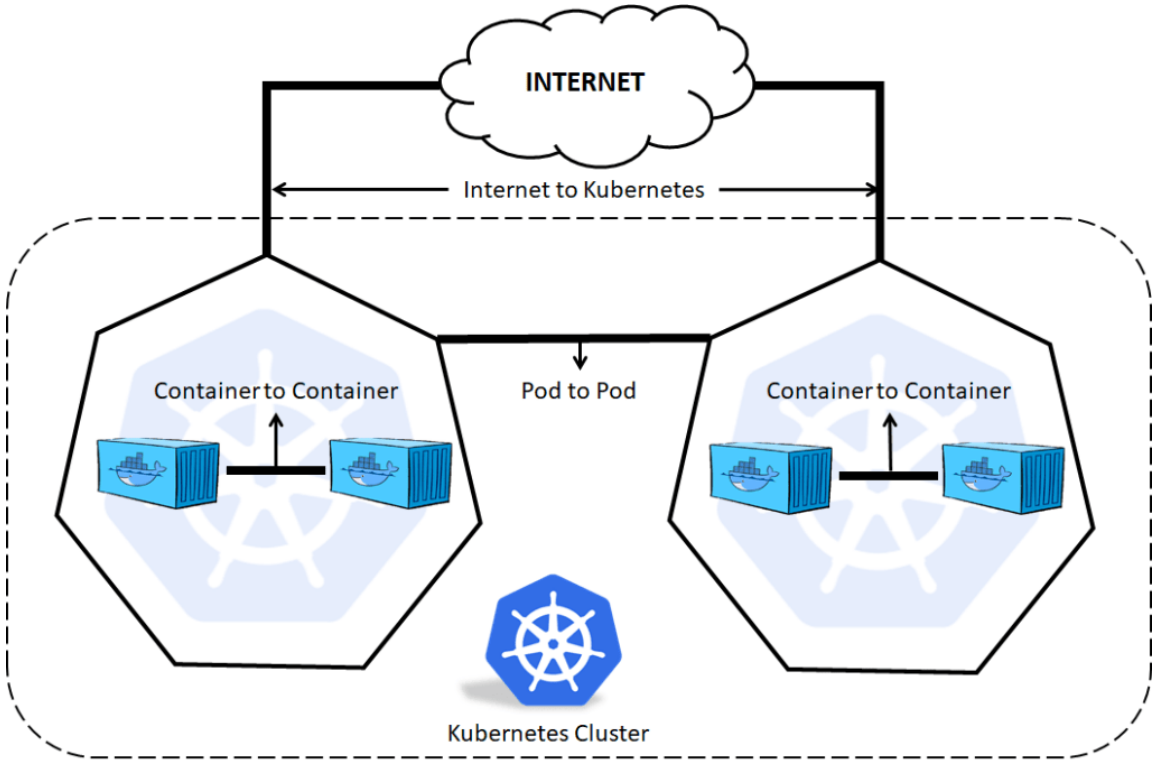
- This style of network is referred to as a “**flat network**”
  - From a pod's view, the cluster is a single network plane



Kubernetes Networking model

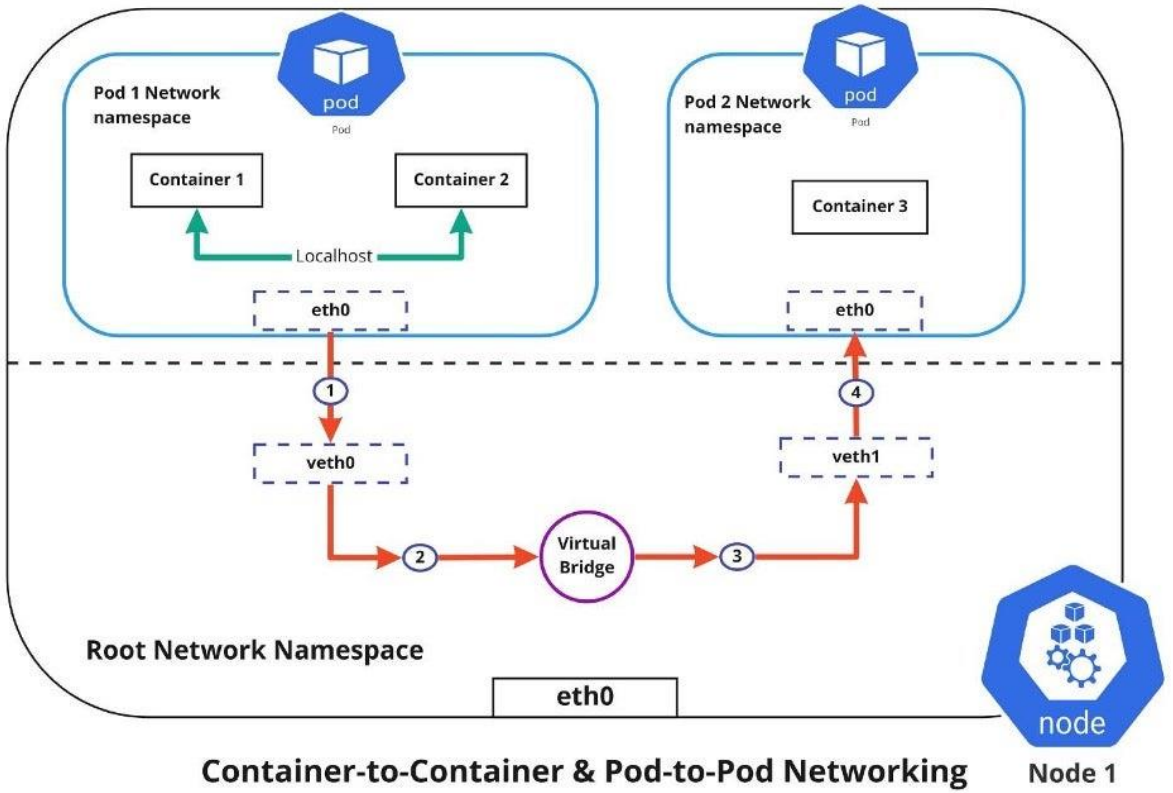
- Given these constraints, Kubernetes networking can be broken into four distinct problems to solve:
  - **Container-to-Container Networking**: how containers within the same Pod communicate.
  - **Pod-to-Pod Networking**: how Pods communicate with each other across nodes.
  - **Pod-to-Service Networking**: how Pods interact with Services, including load balancing and discovery.
  - **Internet-to-Service Networking**: how external traffic reaches cluster Services.

- And to solve them, Kubernetes employs several key networking components and resources:
    - Network namespaces, iptables, CNI plugins, Services…



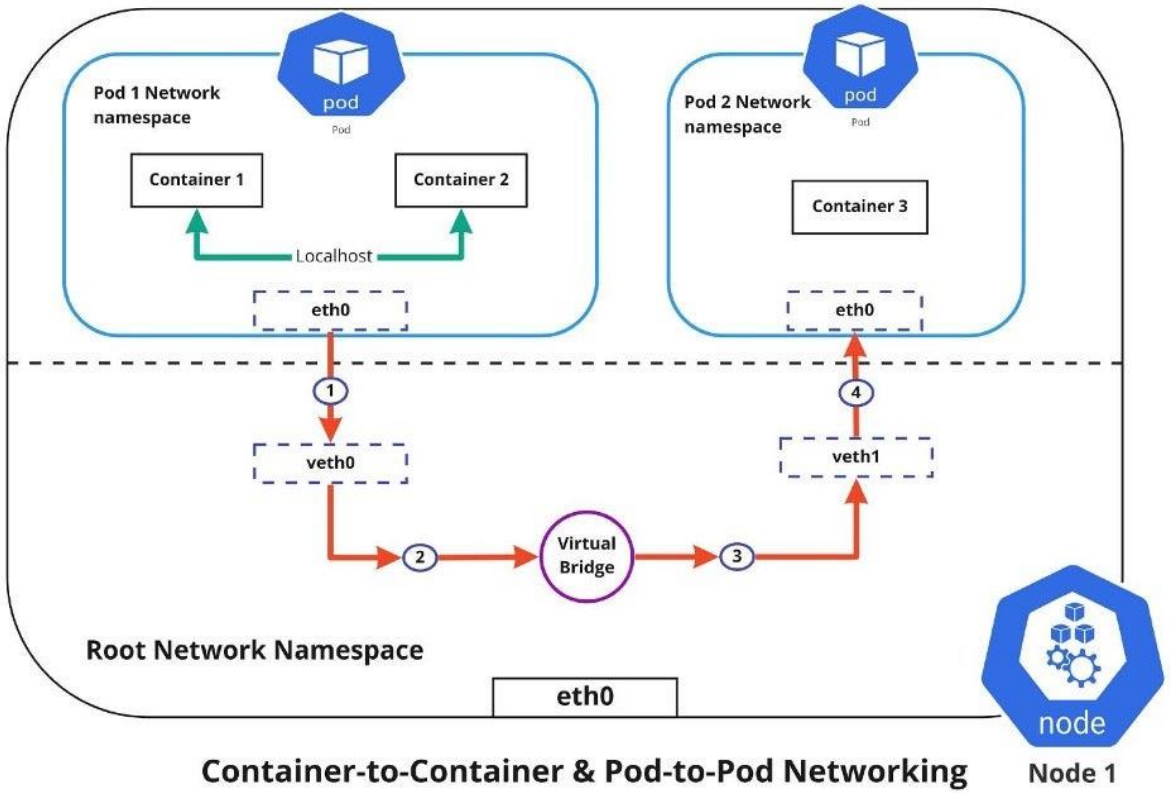
Container-to-Container networking

- Pod is modelled as a group of containers
- How containers within the same Pod communicate?
- Occurs through the **Pod (Linux) Network Namespace**
  - logical networking stack with its own logical router, firewall, and other network devices.
  - It allows for separate network interfaces and routing tables isolated from the rest of the system.
  - Container within the Pods will communicate with each other via **localhost** within the same Pod Network namespace
  - each Pods will communicate with each other with **Root Network Namespace** created within the Node (*eth0)*



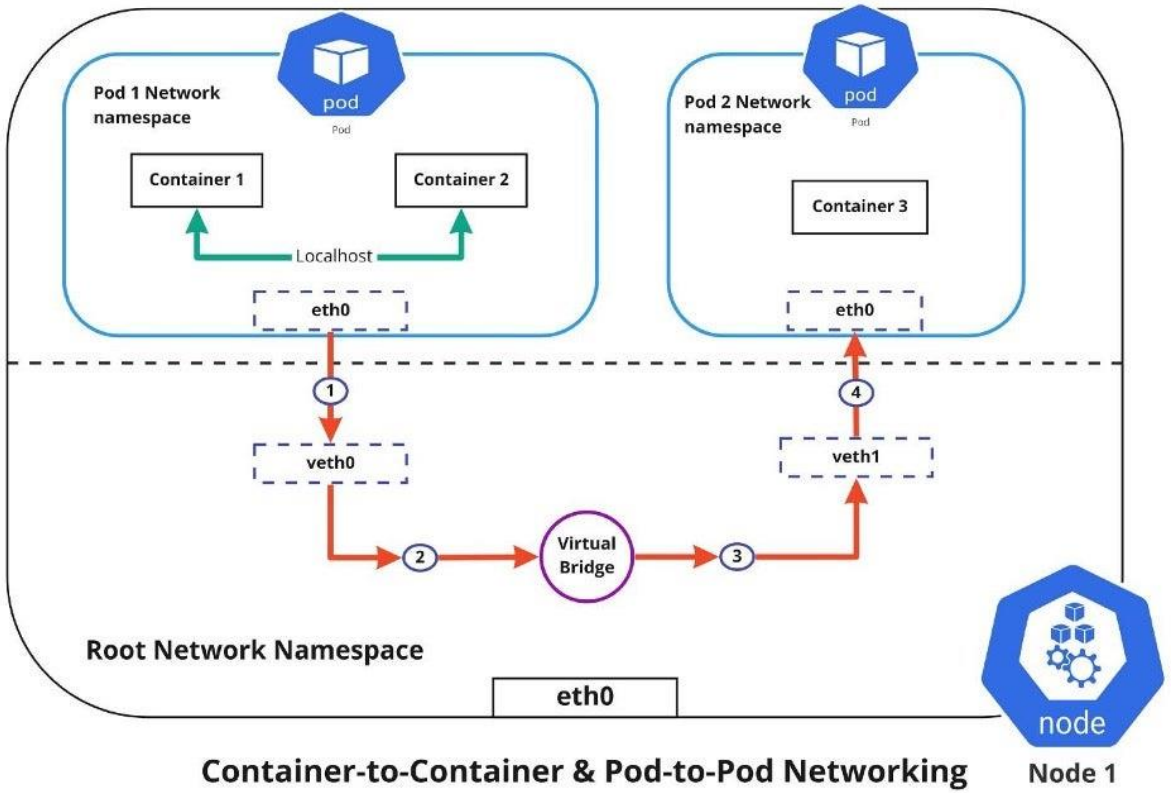
Pod-to-Pod networking
(same node)

- Pods network namespaces are connected via **virtual ethernet devices** (veth pairs) to root network namespace
- A **virtual network bridge** allows traffic between these interfaces, with communication using **ARP** (Address Resolution Protocol)
- Operates at **Layer 2 (Data Link)** using **MAC addresses** for packet forwarding.
- When a packet arrives:
    1. The bridge checks the destination **MAC address**.
    2. If the destination is local (on the same node), it forwards the packet to the appropriate veth interface.
    3. If the destination is not local, it sends the packet to the **default route** (gateway).



Pod-to-Pod networking
(same node)

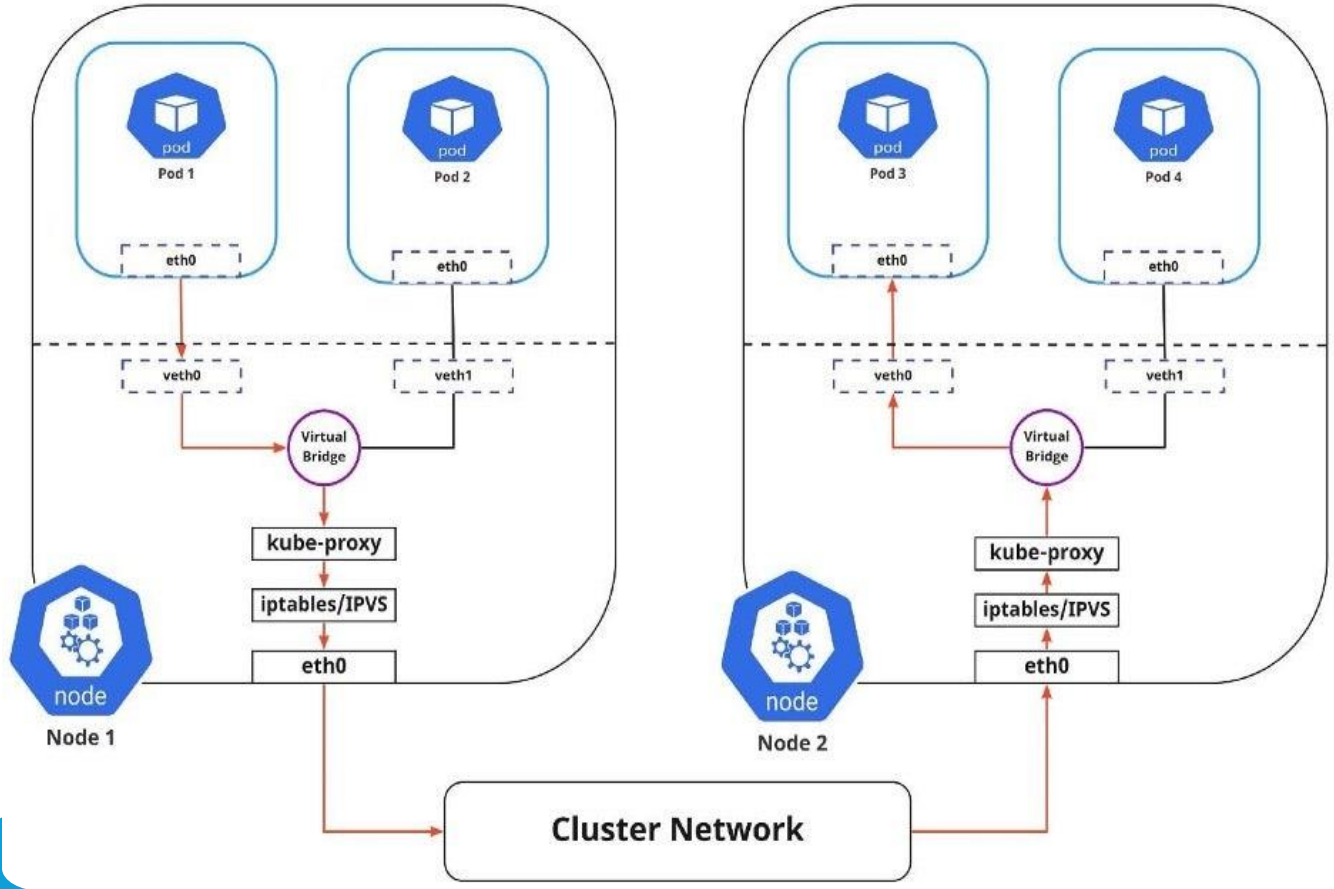
- If data is sent from Pod 1 to Pod 2, the flow of events would like this (refer to diagram)
    1. Pod 1 traffic flows through eth0 to the root network namespaces virtual interface veth0.
    2. Then traffic goes via veth0 to the virtual bridge which is connected to veth1.
    3. Traffic goes via the virtual bridge to veth1.
    4. Finally, traffic reaches eth0 interface of Pod 2 via veth1.



Container-to-Container & Pod-to-Pod Networking

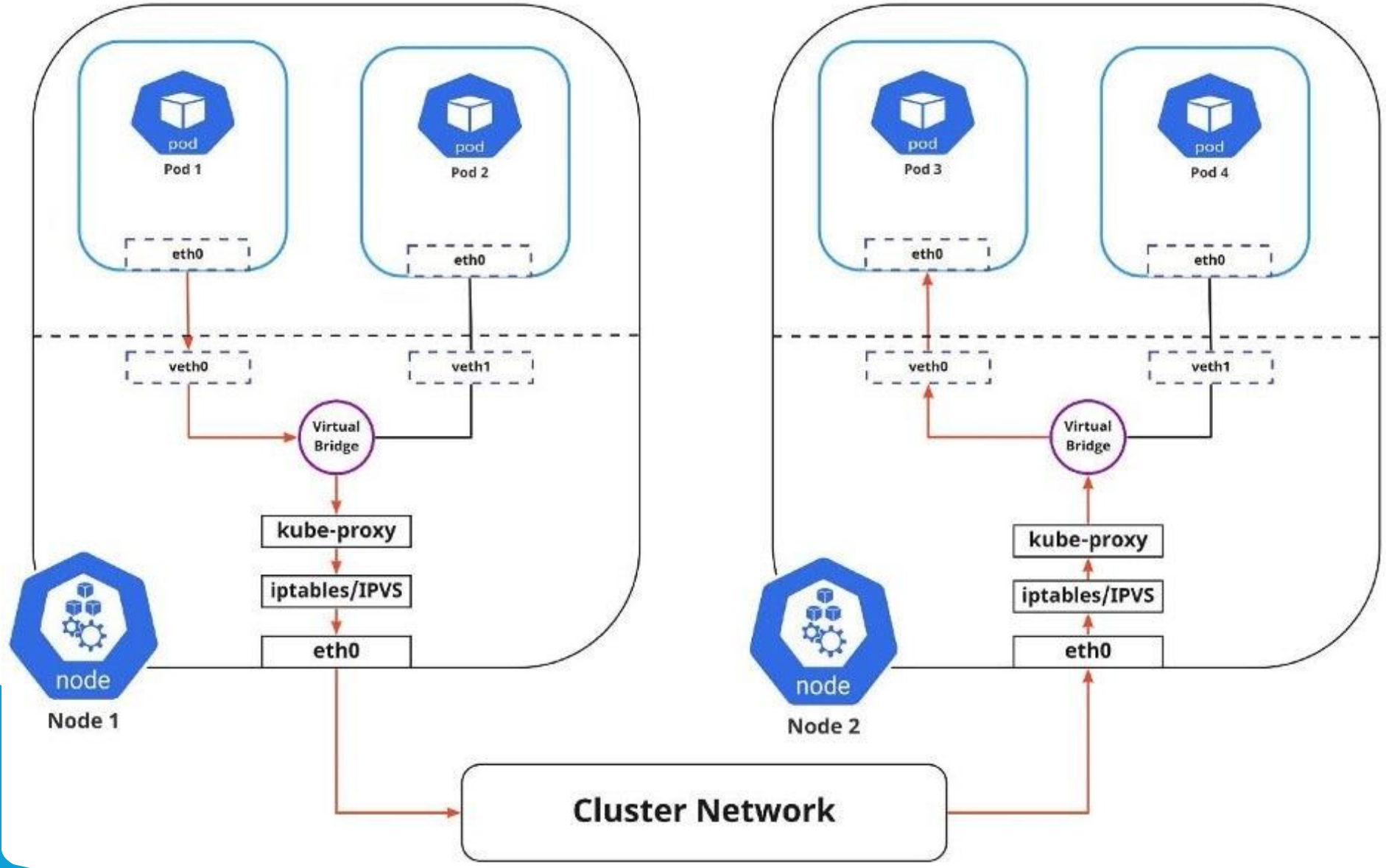
Node 1

Pod-to-Pod networking
(different nodes)



- How do Pods communicate across Nodes?
  - The CNI plugin assigns each Pod a unique IP within the cluster.
  - When a Pod sends traffic to another Pod on a different node:
    - The traffic exits the Pod through its veth interface.
    - The virtual bridge forwards the traffic to the default route if the destination is not local.
    - The default route sends the packet to the overlay/underlay network.
- Role of Overlay/Underlay networks:
  - **Overlay networks** (e.g., VXLAN) encapsulate traffic and handle IP-based routing between nodes.
  - **Underlay networks** rely on physical infrastructure (switches, routers) to forward packets based on Pod IPs (unencapsulated network as Border Gateway Protocol – BGP)
- On the destination node:
  - The packet enters the node's root network namespace.
  - It is forwarded to the destination Pod via the virtual bridge and veth interface.

Pod-to-Pod networking
(different nodes)



Container Network Interface
(CNI plugin)

- The **Container Network Interface (CNI)** is a specification by the **Cloud Native Computing Foundation (CNCF)** that standardizes the configuration of network interfaces for Linux containers.

- A **CNI plugin** is a software component based on the CNI specification. It:
  - Configures network interfaces for Linux containers.
  - Allocates networking resources like IPs.
  - Implements routing and enforces network policies.

- **Role in Kubernetes**
  - It allows the **Pod networking** by working with the container runtime (e.g., containerd).
  - Ensures reliable communication between: Pods, Nodes, External network components

Container Network Interface
(CNI plugin)

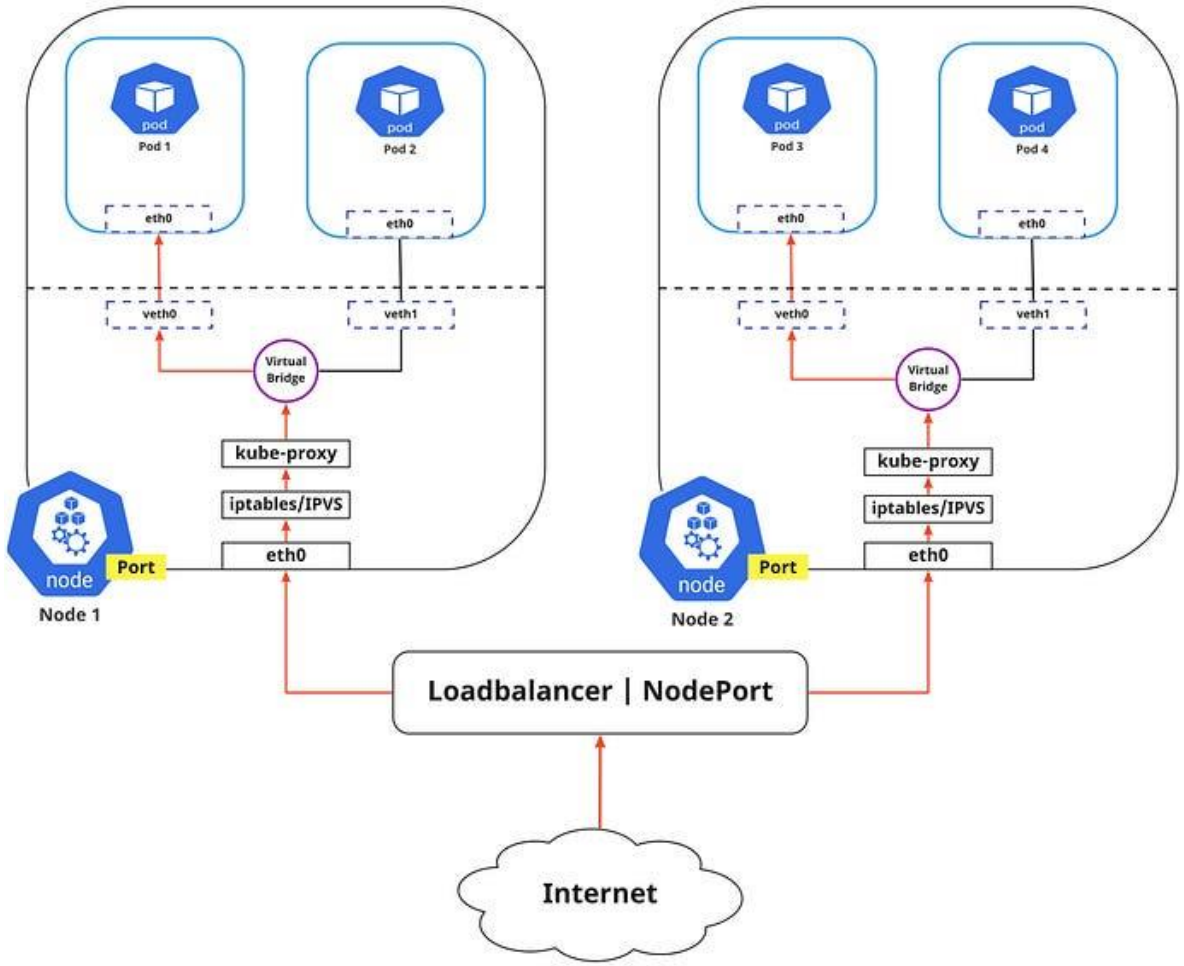
- **Common CNI Plugins**
  - **Calico:** focuses on security and network policies using BGP for routing.
  - **Flannel:** simplifies networking by creating an overlay network using VXLAN.
  - **Weave Net:** provides a simple and fast overlay network for Kubernetes.
  - **Cilium:** advanced networking with eBPF-based security policies and observability.
  - **Canal:** combines Flannel for networking and Calico for network policies.
  - **Kube-Router:** integrated networking, firewall, and routing for Kubernetes clusters.
  - **Multus:** allows Pods to attach to multiple network interfaces.
  - **Amazon VPC CNI:** optimized for AWS, enabling Pods to use VPC-native networking.
  - **Azure CNI:** integrates with Azure virtual networks for Kubernetes workloads.
  - **Google Cloud CNI:** provides seamless networking for Pods in GKE.
  - **Antrea:** implements Open vSwitch for Kubernetes networking.

Pod-to-Service networking

- **Pods are Dynamic!**
  - **Scale up or down** in response to changes in demand.
  - **Recreated** automatically after a crash or node failure.
  - **IP addresses change** with these events, which can complicate networking.
- Kubernetes solution: the **Service** abstraction:
  - Provides **stable network access** to a set of Pods, shielding clients from the dynamic changes of Pods.
  - Assigns a **long-term virtual IP** to the frontend, ensuring reliable communication with backend Pods.
  - **Load-balances traffic** directed to the virtual IP, distributing it evenly among the backend Pods.
  - Clients connect with the static virtual IP of the Service.

Pod-to-Service networking

- Kubernetes supports four primary Service types, each serving a distinct purpose:
  - **ClusterIP (Default)**
    - Exposes the Service on a cluster-internal IP, making it accessible only within the cluster.
    - internal communication only.
  - **NodePort**
    - Makes the Service accessible externally by exposing it on each Node's IP address and a static port (NodePort).
    - Can be accessed externally via <NodeIP>:<NodePort>.
  - **LoadBalancer**
    - Provides external access to the Service using a cloud provider's load balancer.
    - For environments requiring a single external endpoint.
  - **ExternalName**
    - Maps the Service to a CNAME record specified in the externalName field (e.g., foo.bar.example.com).
    - For directing traffic to external services outside the cluster.



Defining a Service
(ClusterIP example)

- This example illustrates how to set up a Service to route traffic to two NGINX Pods, using a **ClusterIP** Service for exposure.
- The Pods are accessible only within the cluster

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

```
$ kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE   IP            NODE
nginx1     1/1     Running   0          65m   10.244.1.3    k8s-node
nginx2     1/1     Running   0          65m   10.244.1.4    k8s-node

$ kubectl get svc -o wide
NAME             TYPE      CLUSTER-IP     EXTERNAL-IP PORT(S) AGE SELECTOR
nginx-clusterip ClusterIP 10.103.197.222 <none>      80/TCP  46m app=nginx

$ kubectl describe svc nginx-clusterip
...
IPs:                      10.103.197.222
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
Endpoints:                10.244.1.3:80,10.244.1.4:80

#Access the Service from within the cluster (e.g., using another Pod):

kubectl exec -it dnsutils – sh
# curl http://10.103.197.222
<html>
<body>
  <h1>Nginx 1</h1>
</body>
</html>

# curl http://10.103.197.222
<html>
<body>
  <h1>Nginx 2</h1>
</body>
</html>
```

Defining a Service
(NodePort example)

- Kubernetes services can also define how a service is accessed from outside of the cluster, using one of the following:
  - A node port, where the service can be accessed via a specific port on every node
  - A load balancer, where a network load balancer provides a virtual IP address that the service can be accessed via from outside the cluster
- Same example as before but using **NodePort** type to expose our nginx

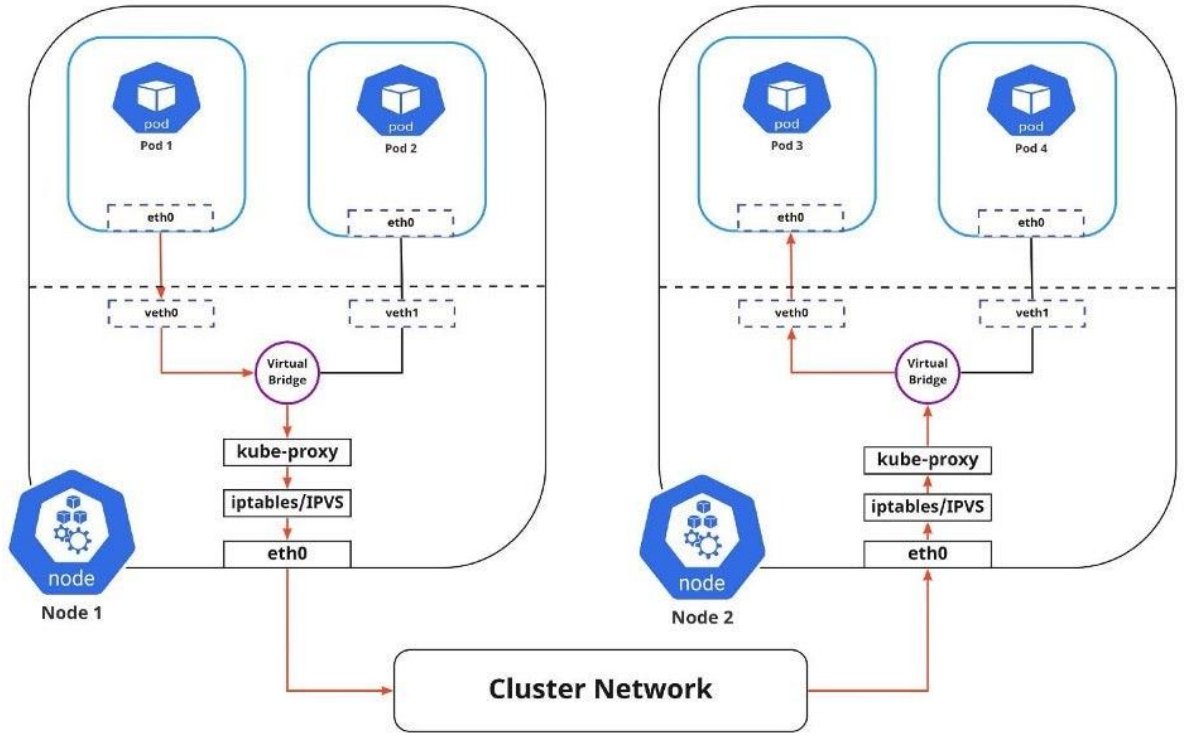
```
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

```
$ kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE   IP            NODE
nginx1     1/1     Running   0          65m   10.244.1.3    k8s-node
nginx2     1/1     Running   0          65m   10.244.1.4    k8s-node

$ kubectl get svc -o wide
NAME             TYPE       CLUSTER-IP      EXTERNAL-IP PORT(S)       SELECTOR
nginx-nodeport   NodePort   10.105.120.195  <none>      80:30155/TCP  app=nginx


$ kubectl describe svc nginx-nodeport
...
IPs:                      10.105.120.195
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30155/TCP
Endpoints:                10.244.1.3:80,10.244.1.4:80


#Access the Service from outside the cluster using the IP of the node:

# curl http://localhost:30155
<html>
<body>
  <h1>Nginx 1</h1>
</body>
</html>

# curl http://192.168.81.87:30155
<html>
<body>
  <h1>Nginx 2</h1>
</body>
</html>
```

Kube-proxy

- Each node runs a **kube-proxy,** the K8s process implementing Services
- kube-proxy creates and manages **iptables rules** to route incoming traffic destined for the Service IP to one or more backend pods.
- These iptables rules are maintained in the **NAT table** and are dynamically updated as pods are added or removed.

- `iptables -t nat -A PREROUTING -p tcp -d <Service-IP> --dport <Service-Port> -j DNAT --to-destination <Pod-IP>:<Pod-Port>`
- `iptables -t nat -A POSTROUTING -p tcp -d <Pod-IP> --dport <Pod-Port> -j SNAT --to-source <Node-IP>`
- In this example, PREROUTING rules modify incoming packets before they get routed, and POSTROUTING rules modify packets as they are about to leave the node, ensuring that packets reach the correct backend pods.



Kubernetes DNS

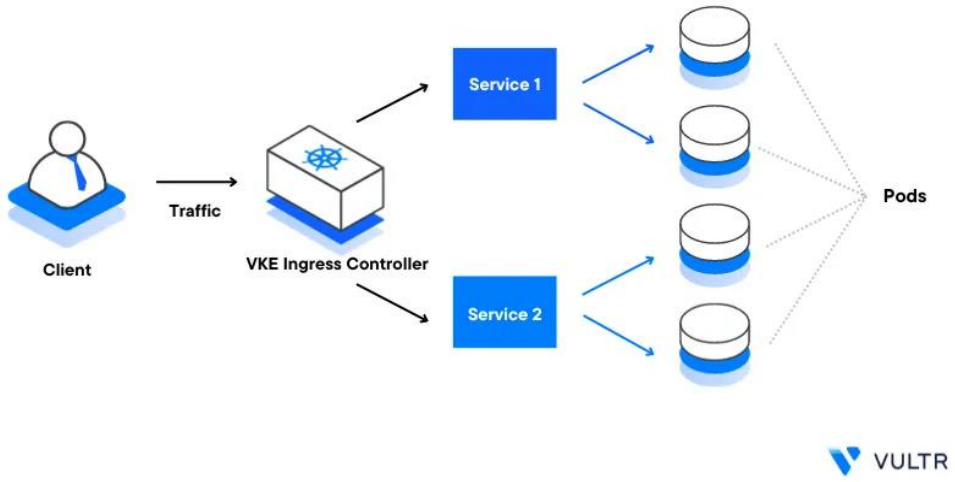
- Kubernetes DNS is a built-in service that enables **name resolution** within a Kubernetes cluster. It facilitates communication between pods and services using **human-readable names** instead of IP addresses.
- **DNS records** are automatically configured for all services and pods, streamlining service discovery.
- **Service Discovery:** automatically resolves service names to their Cluster IPs, enabling seamless communication.
- **Support for Namespaces:** uses Fully Qualified Domain Names (FQDNs) to uniquely identify services across namespaces.
    - service-name.namespace.svc.cluster.local
- **CoreDNS Integration**: Kubernetes uses CoreDNS as its default DNS server for efficient and scalable name resolution.

Kubernetes DNS
(example)

- Assume our nginx-service is running in the default namespace, its FQAN is: **nginx-service.default.svc.cluster.local.**
- Inside a pod, use the following command to resolve the service name: **nslookup nginx-service.default.svc.cluster.local.**

```
$ kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE   IP            NODE
nginx1     1/1     Running   0          65m   10.244.1.3    k8s-node
nginx2     1/1     Running   0          65m   10.244.1.4    k8s-node

$ kubectl get svc -o wide
NAME             TYPE       CLUSTER-IP      EXTERNAL-IP PORT(S) AGE SELECTOR
nginx-clusterip ClusterIP 10.103.197.222 <none>      80/TCP  46m app=nginx

$ kubectl describe svc nginx-clusterip
...
IPs:                      10.103.197.222
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
Endpoints:                10.244.1.3:80,10.244.1.4:80

#Access the Service from within the cluster (e.g., using another Pod):

kubectl exec -it dnsutils – sh
# nslookup  10.103.197.222
222.197.103.10.in-addr.arpa  name = nginx-service.default.svc.cluster.local.

# curl nginx-service.default.svc.cluster.local.
<html>
<body>
  <h1>Nginx 2</h1>
</body>
</html>
```

Internet-to-Service
(external traffic)



- Kubernetes services are isolated by default, making external access **complex**.
- **NodePort and LoadBalancer limitations**:
  - **NodePort**: manual port management, limited scalability.
  - **LoadBalancer**: high costs for multiple services.
- **Security Risks**: exposing services individually creates vulnerabilities.
- **Lack of centralized control**: difficult to manage traffic policies and routing.

- Advanced solutions: **Ingress** and **Gateways**
  - Simplifies **external access management** by abstracting networking complexities.
  - Keeps services **decoupled** from external dependencies.
- **Example**
  - Direct traffic to a backend application based on URL paths (e.g., /api → Service A, /web → Service B).

Ingress

- **Overview**:
  - Acts as a **central entry point** for HTTP/HTTPS traffic.
  - Defines **routing rules** based on paths or hostnames.
  - Handles **TLS termination** and other Layer 7 functions of OSI model.
  - An **Ingress Controller** (e.g. nginx, haproxy) applies these rules to route traffic effectively.
- **Advantages**:
  - **Centralized Management**: One entry point for multiple services.
  - **Cost-Efficiency**: Reduces reliance on individual load balancers.
  - **Extensibility**: Supports traffic shaping and basic policies.
- **Limitations**:
  - Limited to **Layer 7 (HTTP/HTTPS)** protocols.
  - Fragmentation due to **controller-specific customizations**.
  - Advanced features (e.g., rate limiting, canary releases) require external tools.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-myservicea
spec:
  ingressClassName: nginx
  rules:
  - host: myservice-a.foo.org
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: myservice-a
            port:
              number: 80
```

This configuration allows external HTTP requests to http://myservice-a.foo.org/ to be directed to the appropriate service within the cluster.

You need to configure a DNS **A record** to point the hostname (myservice-a.foo.org) to the **external IP address** of your Kubernetes ingress controller.

Gateway API
(new approach)

- **What is Gateway?**
  - A **next-generation API** standardizing how traffic is managed.
  - Expands beyond HTTP/HTTPS to support **L4-L7 protocols** (TCP/UDP, gRPC)
- **Advantages over Ingress**:
  - **Protocol agnostic**: handles traffic for diverse protocols.
  - **Improved extensibility**: native support for advanced policies like authentication, rate limiting, and A/B testing.
  - **Standardization**: provides consistent behavior across different implementations.
- **Key Components**:
  - **Gateway**: represents the physical or logical entry point (e.g., load balancer or proxy).
  - **HTTPRoute/TCPRoute**: specifies routing rules for different protocols.
  - **GatewayClass**: defines the capabilities of the underlying implementation (e.g., NGINX, Istio).

	Ingress	Gateway API
Protocol Support	HTTP/HTTPS only	L4 & L7 support
Traffic Management	Limited, vendor extensions required	Built-in advanced support
Portability	Vendor specific definitions	Standardized across implementations
Resource Objects	Ingress resource only	GatewayClass, Gateway, HTTPRoute, etc.
Routing Rules	Host/path-based only	Header-based also supported
Extending Capabilities	Custom annotations needed	Built-in advanced functionality

Thanks!

References

- https://kubernetes.io/docs/concepts/cluster-administration/networking/
- https://kubernetes.io/docs/concepts/services-networking/
- https://kubernetes.io/docs/concepts/services-networking/service/
- https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/
- https://kubernetes.io/docs/concepts/services-networking/gateway/
- https://medium.com/@extio/understanding-kubernetes-node-to-node-communication-a-deep-dive-e1d6a5ff87f3
- https://support.tools/post/kubernetes-networking-deep-dive/
- https://docs.cilium.io/en/stable/network/concepts/routing/