Handwritten Kubernetes Services Guide

Easy Notes on Service, kube-proxy & CNI (Easy Kubernetes Guide)

What is a service ?

A **Service** in Kubernetes helps expose an application running in your cluster by providing a single, stable network endpoint. This ensures that even if your application runs in multiple Pods (which may change dynamically), clients can always access it reliably.

Why Do We Need a Service?

1. **Pods Are Temporary (Ephemeral)** – Kubernetes creates and destroys Pods as needed, so their IP addresses change dynamically.
2. **Consistent Access** – If an application has multiple backend Pods, a Service provides a fixed way to reach them.
3. **Load Balancing** – Services distribute traffic among multiple Pods automatically.

How a Service Works?

- A **Service** groups a set of Pods using **labels and selectors**.
- It gives them a **stable network identity (ClusterIP) instead of IP**.
- Clients can use this identity to interact with the backend Pods.

For example, assume we have a **backend application** running in 3 Pods. Instead of connecting to individual Pods, we create a Service that directs traffic to all healthy backend Pods.

Types of Services

1. ClusterIP (Default)

- Exposes the Service only within the cluster.
- Not accessible from outside.
- Example: Internal communication between microservices.

  Example YAML:

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: my-service
spec:
  selector:
    app: my-backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Here, all Pods with label **app=my-backend** will be reachable at port **80** via the Service.

---

2. NodePort

- Exposes the Service on a specific port of each Node.
- Accessible from outside the cluster using **NodeIP:NodePort**.
- Useful for debugging or basic external access.

  Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: my-backend
  ports:
    - port: 80
      targetPort: 9376
      nodePort: 30007  # This port is accessible on every node
```

If the cluster node has IP **192.168.1.10**, then you can access the service at:

http://192.168.1.10:30007

---

3. LoadBalancer

- Used when running Kubernetes on cloud providers (AWS, GCP, Azure).
- It provisions an **external load balancer** to distribute traffic.
- Suitable for public-facing applications.

  Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: LoadBalancer
  selector:
    app: my-backend
  ports:
```

```
  - port: 80
    targetPort: 9376
```

- If the cloud provider assigns **192.0.2.127** as the external IP, users can access it via:

http://192.0.2.127

---

4. ExternalName

- Maps a Service to an external DNS name.
- No traffic is routed inside the cluster; instead, it resolves to an external resource.

  Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-database
spec:
  type: ExternalName
  externalName: db.example.com
```

When applications inside the cluster query **my-database**, they get redirected to **db.example.com**.

Other Important Concepts

1. Multi-Port Services

- Services can expose multiple ports for different functionalities.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-backend
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
    - name: https
      protocol: TCP
      port: 443
      targetPort: 9377
```

This allows HTTP traffic on **port 80** and HTTPS traffic on **port 443**.

---

2. Headless Services

- A Service without a ClusterIP (clusterIP: None).
- Used for direct Pod-to-Pod communication (e.g., databases).

```
apiVersion: v1
kind: Service
metadata:
  name: my-headless-service
spec:
  clusterIP: None
  selector:
    app: my-backend
  ports:
    - port: 80
      targetPort: 9376
```

Applications query DNS and get direct Pod IPs instead of a single service IP.

---

3. Service Discovery

There are **two ways** clients inside the cluster can discover Services:

1. **Environment Variables:** Kubernetes automatically sets environment variables for each Service.
2. **DNS Resolution:** Services are registered in the cluster's DNS and can be accessed using my-service.default.svc.cluster.local.

Now In the Background

Some of the most important things you need to know:

Understanding Kubernetes Networking: kube-proxy and CNI Plugin

In Kubernetes, a **Service** allows applications to communicate with each other reliably, even when the underlying Pods are created and destroyed dynamically. But how does this communication actually happen? This is where two important networking components come into play:

1. **kube-proxy** – It ensures that traffic from a **Service** reaches the correct **Pod**.
2. **CNI Plugin (Container Network Interface)** – It enables internal networking between **Pods**, even when they are on different nodes.

Without these two networking components, a **Service** would not be able to route traffic to application Pods, and Pods wouldn't be able to communicate across the cluster. Let's break them down in simple terms.

---

1. What is kube-proxy?

kube-proxy is a network service that makes sure traffic from a **Service** reaches the right **Pod**. It acts like a traffic controller inside the Kubernetes cluster.

How kube-proxy Works?

- Each **Service** in Kubernetes gets a stable IP address.
- When a request comes to a Service, **kube-proxy** finds a healthy **Pod** behind that Service.

- It forwards the request to the selected Pod.
- If a Pod goes down, **kube-proxy** automatically redirects traffic to another healthy Pod.

Why is kube-proxy Important?

- Ensures that traffic from Services reaches the correct Pod.
- Handles **load balancing** between multiple Pods.
- Works on **every node** to manage network traffic.

Example: Imagine a website where users visit www.myapp.com. This request reaches a **Service** in Kubernetes. The **kube-proxy** decides which backend **Pod** should handle the request and forwards the traffic accordingly.

---

2. What is a CNI Plugin?

The **CNI Plugin (Container Network Interface)** is responsible for providing **network communication between Pods**.

How CNI Plugin Works?

- When a **Pod** is created, the CNI plugin assigns it a unique **IP address**.
- It ensures that all **Pods** can communicate with each other, even if they are running on different nodes.
- It makes sure that all **same-label** Pods (like replicas of a Deployment) can talk to each other easily.

Why is CNI Plugin Important?

- Ensures **all Pods can communicate** inside the cluster.
- Provides networking across **different nodes**.
- Helps maintain internal **localhost-style** communication between similar Pods.

Example: If you have multiple replicas of a backend application (app: backend), the CNI plugin makes sure they can talk to each other smoothly.

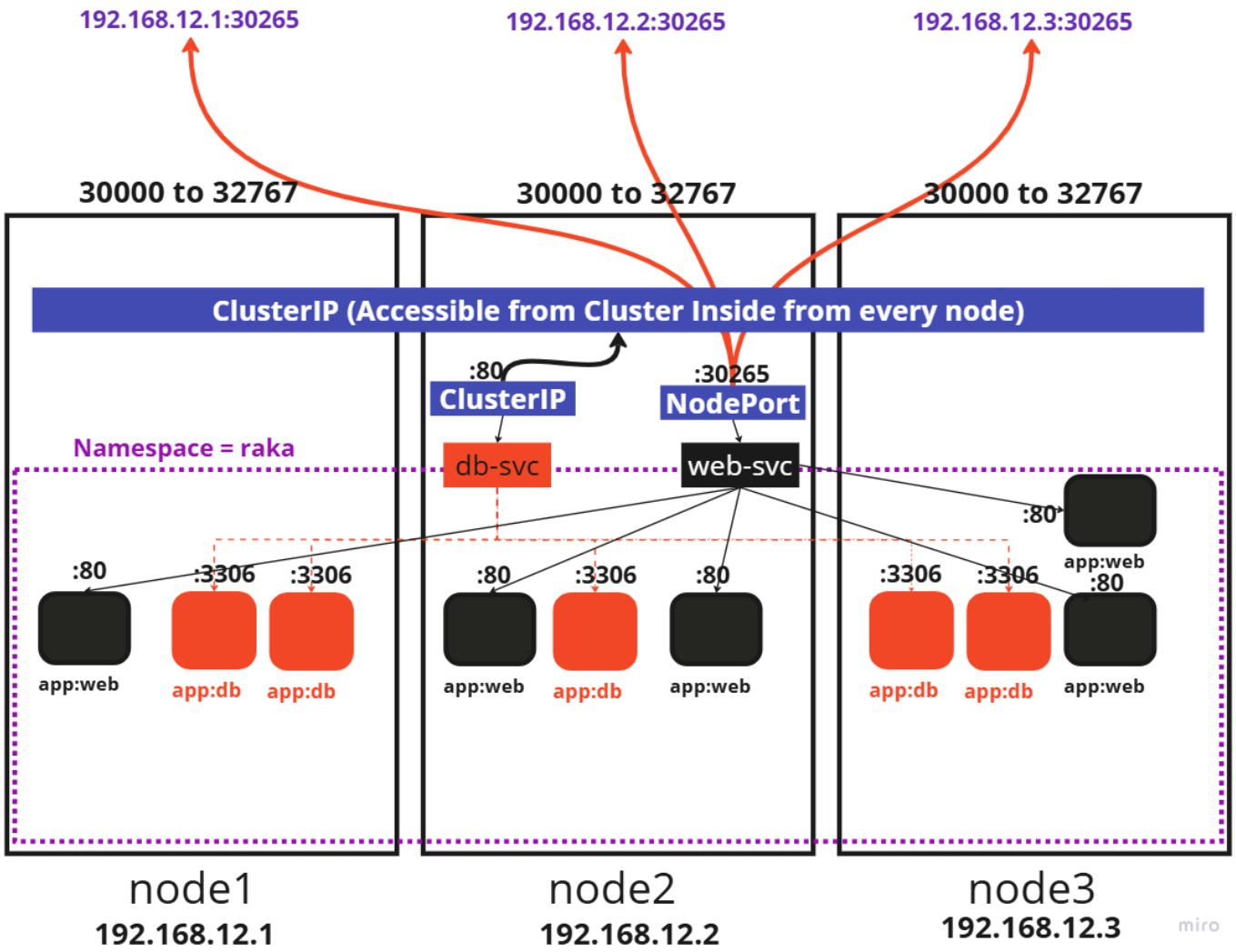
Difference Between kube-proxy and CNI Plugin

Feature	kube-proxy	CNI Plugin
Purpose	Connects Services to Pods	Connects Pods to each other
Works at	**Node level**	**Cluster level**
Manages	Service traffic routing	Pod-to-Pod networking

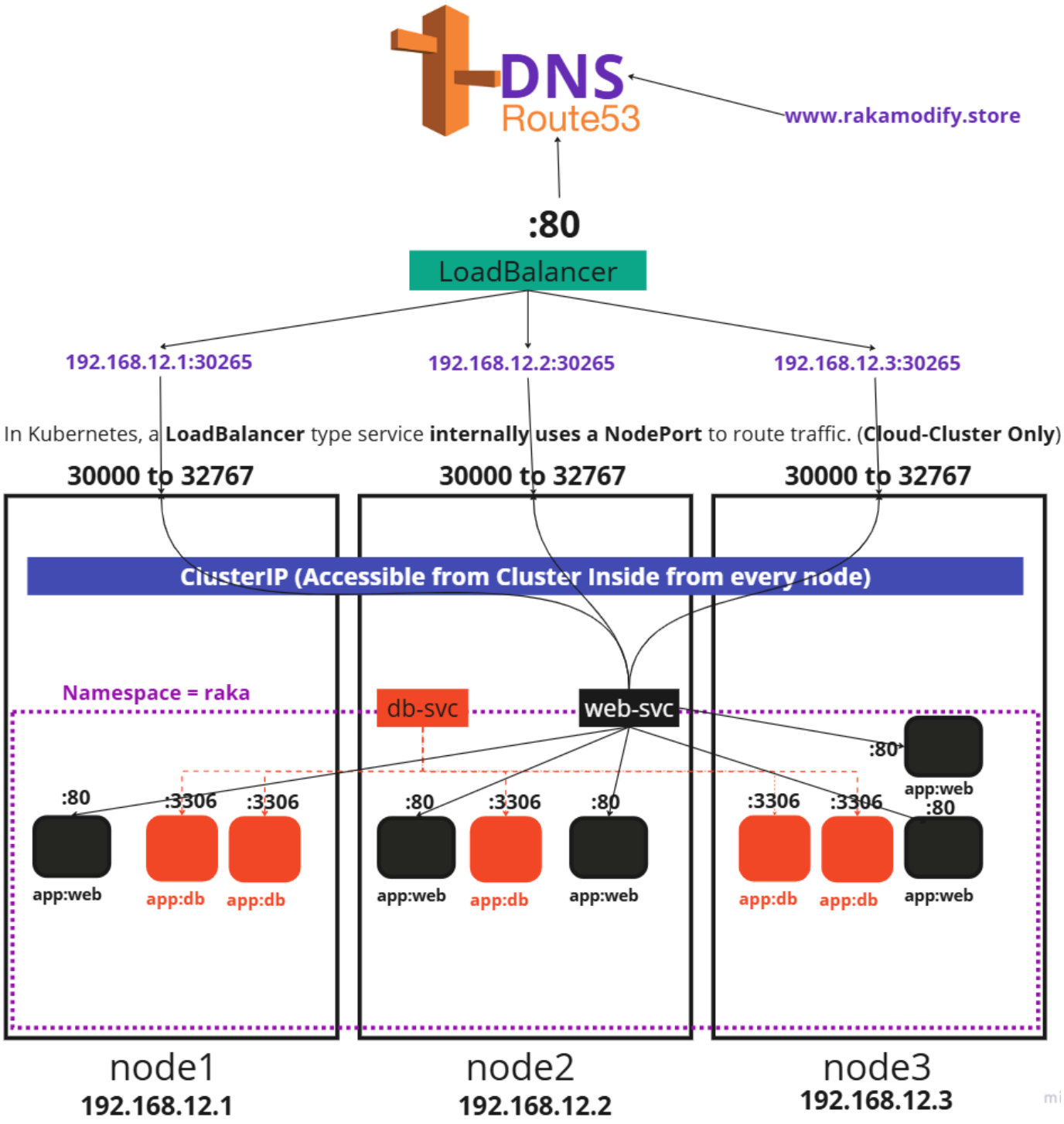
Feature	kube-proxy	CNI Plugin
Technology	Uses iptables or IPVS	Uses plugins like Flannel, Calico, etc.

By understanding these two, you can better manage Kubernetes networking and troubleshoot connectivity issues easily.

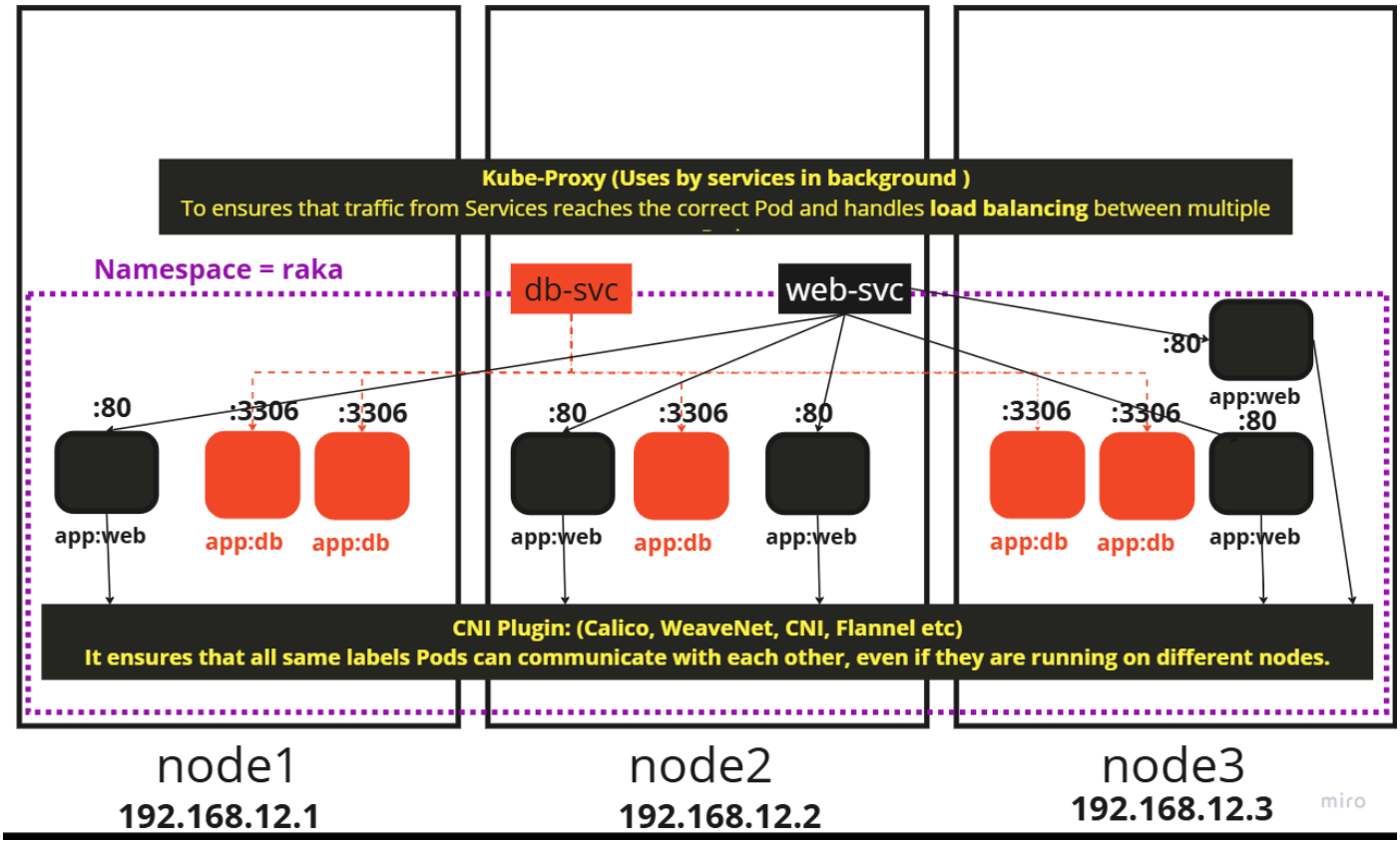
1. Diagram of ClusterIP and NodePort Service



2. Diagram of ClusterIP and NodePort Service



3. Diagram of Kube-Proxy and CNI plugin



My Final Thoughts

- Services ensure stable and reliable communication between Pods.
- Different Service types (ClusterIP, NodePort, LoadBalancer, ExternalName) cater to different needs.
- Load balancing, service discovery, and networking policies enhance Kubernetes' power.
- **kube-proxy** is like a **traffic manager** that connects Services to the right Pods.
- **CNI Plugin** ensures that all Pods can communicate internally, just like how computers in a network talk to each other.

Follow this channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**