KUBERNETES NETWORKING

KUBERNETES NETWORKING

- Networking is the backbone of modern technology.
- It is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work.
- The network model is implemented by the container runtime on each node.
- The most common container runtimes use **Container Network Interface (CNI)** plugins to manage their **network and security capabilities**.
- There are 4 distinct networking problems to address:
  - **Highly-coupled container-to-container communications.**
  - **Pod-to-Pod communications.**
  - **Pod-to-Service communications.**
  - **External-to-Service communications.**

CONTAINER TO CONTAINER COMMUNICATION:

- This is solved by Pods and localhost communications.
- In Kubernetes, a Pod is the basic unit of deployment. It's essentially a group of one or more containers that are deployed together in the same isolated space. These containers inside a Pod share the same network namespace.
- Imagine you have a Pod with two containers: a web server and a database server. The web server reserved port 80 for itself and the database server reserved port 3306.

POD TO POD COMMUNICATION:

- All Pods in a Kubernetes cluster reside in a single, flat, shared network-address space.
- A flat network-address space means, Every Pod gets its own IP address.
- Pods can communicate with all other pods in the cluster using pod IP addresses (without NAT)
- If some Pods live on one node, and some other Pods live different node, which are separated by additional, external networks.

SERVICE:

- Pod To Service Communications covered by Services.
- Kubernetes services provide a way of abstracting access to a group of pods as a network service. which can be reached by a single, fixed **DNS name** or **IP address**.
- The set of Pods targeted by a Service is usually determined by a **selector.**

Single Port Service:

Suppose you have a set of Pods where each listen on TCP port 9376 and contains a label app=MyApp:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Multi-Port Services:

For some Services, you need to expose more than one port. When using multiple ports for a Service, you must give all of your port's names.

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

DISCOVERING SERVICES:

- Kubernetes supports 2 primary modes of finding a Service.
  - Environment variables
  - DNS

Environment variables:

- When a Pod is run on a Node, the kubelet adds a set of environment variables for each active Service.
- It adds {SVCNAME}_SERVICE_HOST and {SVCNAME}_SERVICE_PORT variables, where the Service name is upper-cased and dashes are converted to underscores. It also supports variables.

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

DNS:

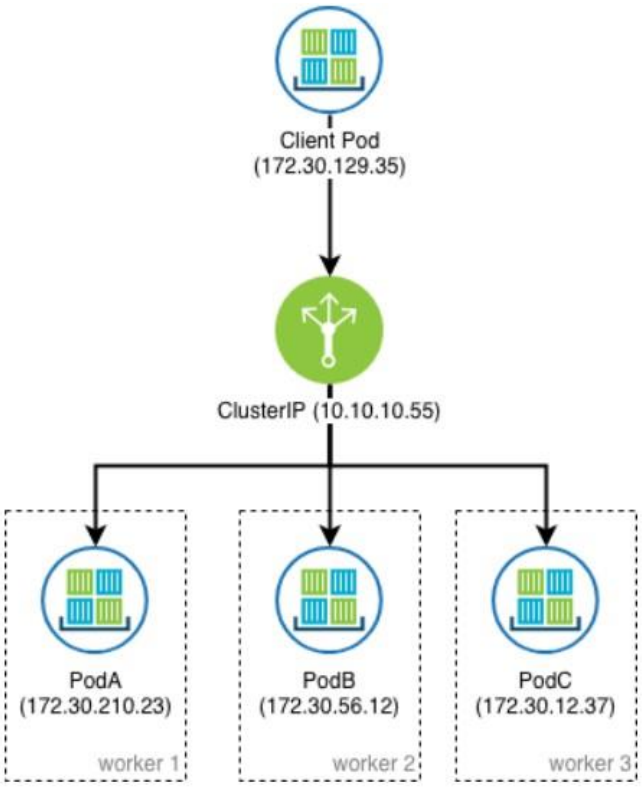
- A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one.
- If DNS has been enabled throughout your cluster, then all Pods should automatically be able to resolve Services by their DNS name.
- For example, if you have a Service called **my-service** in a Kubernetes namespace **my-ns,** the control plane and the DNS Service acting together create a DNS record for **my-service.my-ns.**
- Kubernetes also supports **DNS SRV (Service)** records for named ports.
- The Kubernetes DNS server is the only way to access **ExternalName** Services.

PUBLISHING SERVICES (SERVICETYPES):

- Kubernetes ServiceTypes allow you to specify what kind of Service you want. Service Types are:
  - ClusterIP
  - NodePort
  - LoadBalancer
  - ExternalName

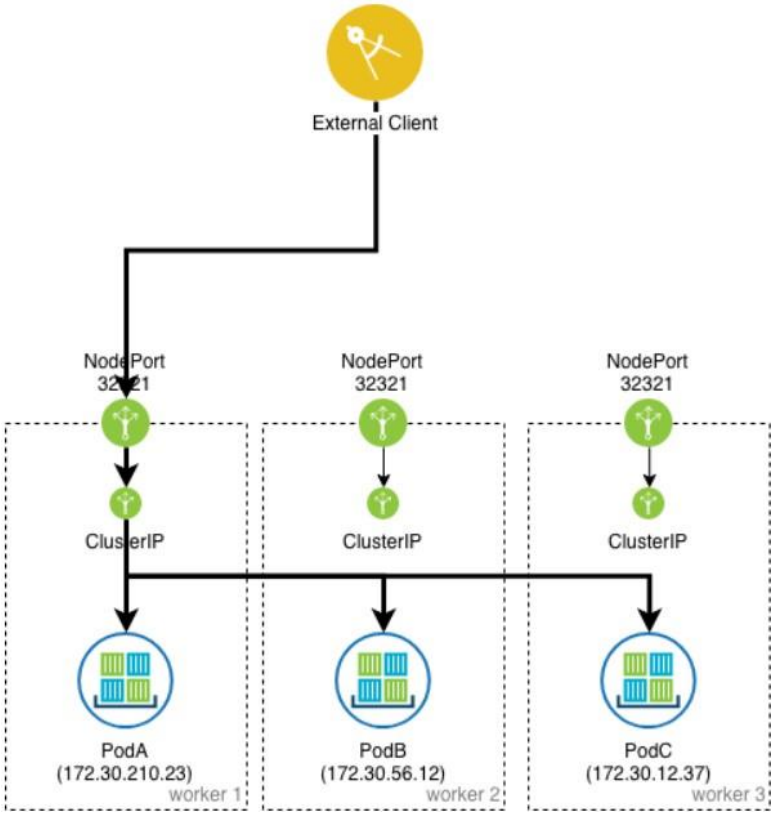
CLUSTERIP:

- Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is default ServiceType.
- Internally, Kubernetes resolves the label selector to a set of pods, and takes the ephemeral Pod IP addresses and generates Endpoints resources that the ClusterIP proxies traffic to.

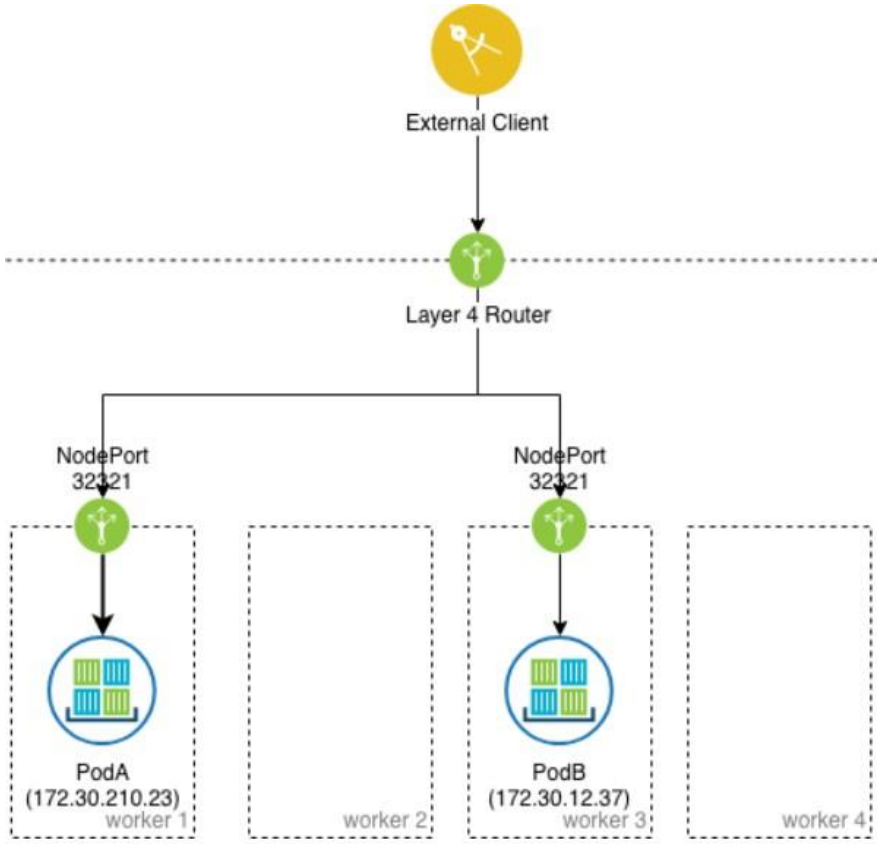


NODEPORT:

- Services of type NodePort build on top of ClusterIP type services by exposing the ClusterIP service outside of the cluster on high ports (default 30000-32767).
- If no port number is specified then Kubernetes automatically selects a free port.
- The local kube-proxy is responsible for listening to the port on the node and forwarding client traffic on the NodePort to the ClusterIP.
- This Service is visible as <NodeIP>:spec.ports[*].nodePort.



- NodePort can be useful when manually configuring external load balancers to forward layer 4 traffic from clients outside of the cluster to a particular set of pods that are running in the Kubernetes cluster.
- In such cases, the specific port number that is used for NodePort must be set ahead of time, and the external load balancer must be configured to forward traffic to the listening port on all worker nodes.

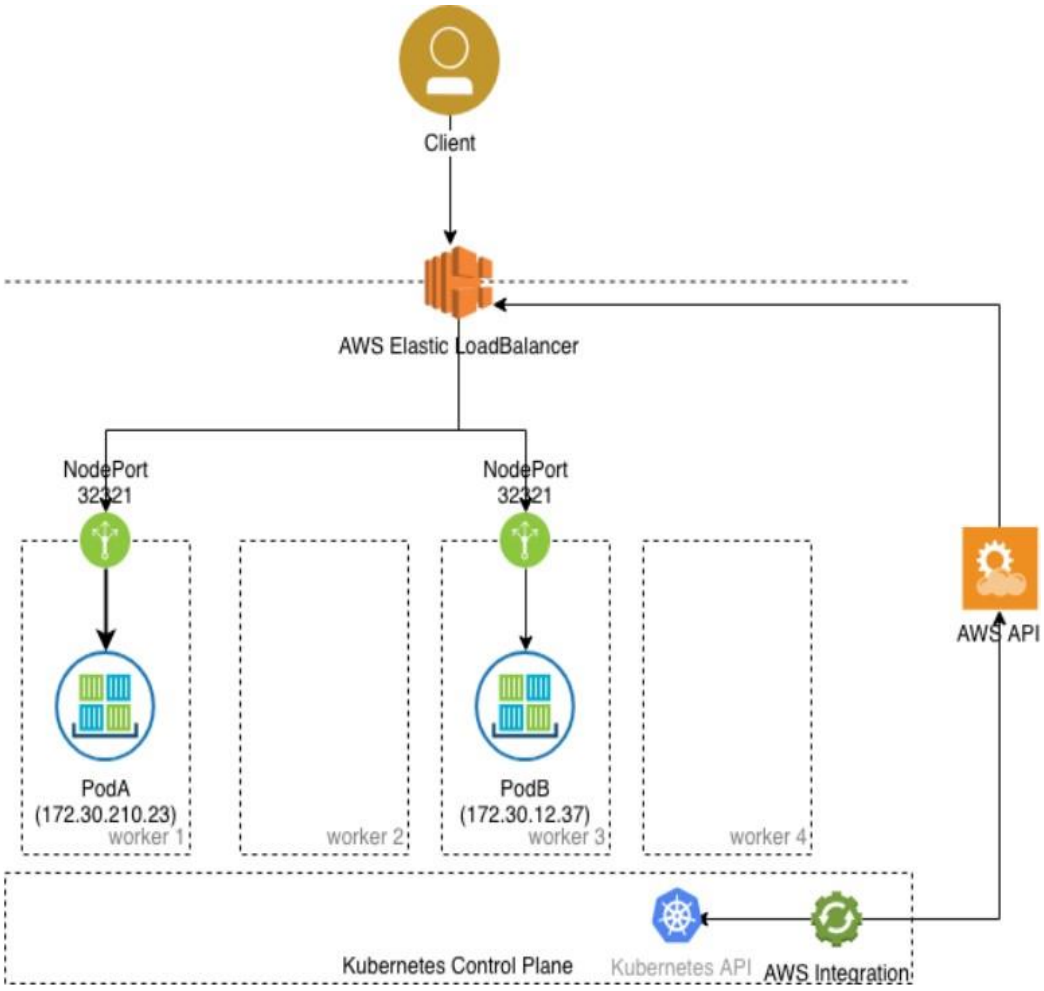


Example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app.kubernetes.io/name: MyApp
  ports:
      # By default and for convenience, the `targetPort` is set to the same value as the `port` field.
    - port: 80
      targetPort: 80
      # Optional field
      # By default and for convenience, the Kubernetes control plane will allocate a port from a range (default: 30000-32767)
      nodePort: 30007
```

LOADBALANCER:

- The LoadBalancer service type is built on top of NodePort service types by provisioning and configuring external load balancers from public and private cloud providers. It exposes services that are running in the cluster by forwarding **layer 4** traffic to worker nodes.
- This is a dynamic way of implementing a case that involves external load balancers and **NodePort** type services.



- The actual creation of the load balancer happens asynchronously, and information about the provisioned balancer is published in the Service's **.status.loadBalancer** field.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  clusterIP: 10.0.171.239
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 192.0.2.127
```

EXTERNALNAME:

- Services of type ExternalName map a Service to a DNS name, not to a typical selector such as my-service or cassandra . You specify these Services with the spec.externalName parameter.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

NOTE: ExternalName accepts an IPv4 address string, but as a DNS name comprised of digits, not as an IP address.

DNS FOR SERVICES AND PODS:

- Kubernetes creates DNS records for services and pods. You can contact services with consistent DNS names instead of IP addresses.
- Every Service defined in the cluster is assigned a DNS name.
- DNS Records are:
  - Services DNS Records
  - Pods DNS Records

SERVICES DNS RECORDS:

A/AAAA RECORDS:

- A/AAAA Record is the most basic type of a DNS record used to point a domain or subdomain to a certain IP address.
- "Normal" Services are assigned a DNS A or AAAA record, depending on the IP family of the service, for a name of the form **my-svc.my-namespace.svc.cluster-domain.example**. This resolves to the cluster IP of the Service.

SRV RECORDS:

- SRV records facilitate service discovery by describing the protocol/s and address of certain services.
- An SRV record usually defines a symbolic name and the transport protocol (e.g., TCP) used as part of the domain name and defines the priority, weight, port, and target for a given service (see the example below)

POD DNS RECORDS:

A/AAAA RECORDS:

- A/AAAA Record is the most basic type of a DNS record used to point a domain or subdomain to a certain IP address.

POD'S HOSTNAME AND SUBDOMAIN FIELDS:

- The default hostname for a pod is defined by a pod's metadata.name value. However, users can change the default hostname by specifying a new value in the optional hostname field. Users can also define a custom subdomain name in a subdomain field.

POD'S DNS CONFIG:

- Pod's DNS Config allows users more control on the DNS settings for a Pod.
- The dnsConfig field is optional and it can work with any dnsPolicy settings. However, when a Pod's dnsPolicy is set to " None ", the dnsConfig field has to be specified.
- The properties a user can specify in the **dnsConfig** field:
  - nameservers
  - searches
  - options

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster-domain.example
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

$kubectl create -f <file-name>

$kubectl get pods

$kubectl exec -it dns-example -- cat /etc/resolv.conf

CONNECTING APPLICATIONS WITH SERVICES:

- Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. Kubernetes gives every pod its own cluster-private IP address, so you do not need to explicitly create links between pods or map container ports to host ports.
- This means that containers within a Pod can all reach each other's ports on localhost, and all pods in a cluster can see each other without NAT.

CREATING A POD:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
```

$kubectl apply -f <file-name>

$kubectl get pods -l run=my-nginx -o wide

$kubectl get pods -l run=my-nginx -o yaml | grep podIP

CREATING A SERVICE:

- So, we have pods running nginx in a flat, cluster wide, address space.
- A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality.

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

$kubectl create -f <file-name>

$kubectl get services

$kubectl get svc my-nginx

ACCESSING THE SERVICE:

- Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS.

  Environment Variables:

  $kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep SERVICE

  DNS:

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

  $kubectl run curl --image=radial/busyboxplus:curl -i --tty

  [root@curl-131556218-9fnch:/]$ nslookup my-nginx

  NOTE: $curl https://<EXTERNAL-IP>:<NODE-PORT> -k