

# Kubernetes Networking Overview

Paul-Adrien Cordonnier - Adaltas Summit 2022





# Goal of the talk


- Give an overview of how the Kubernetes networking works
- Understand what each parts are good for
- It's a live demo lab where I'll show you live what I'm talking about

# Topics covered

1. Pods and containers networking
  - a. Namespaces and CNI
2. Services
  - a. ClusterIP, NodePort, LoadBalancer
3. DNS
  - a. CoreDNS
4. Outside access to the cluster
  - a. LoadBalancer
  - b. Ingress
  - c. Gateway
5. Service Mesh

# Part 1: Containers and Pods

Basic networking inside a Kubernetes cluster

The background of the slide is a dark navy blue. On the right side, there is an abstract geometric design consisting of several dark gray, three-dimensional rectangular blocks or planes that are stacked and offset from each other, creating a sense of depth. A light green parallelogram is positioned on one of the upper planes, and a blue parallelogram is on a lower plane, both adding a pop of color to the monochromatic scheme.

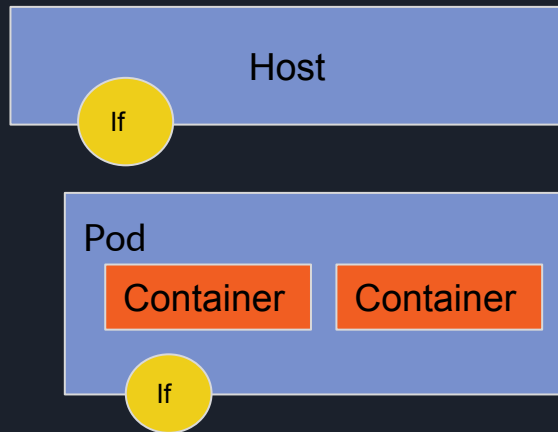


# Containers

- A container is not a actual thing, it is just a process isolated from the other using Linux Namespaces
  - pid namespace
  - user namespace
  - net (network) namespace
- A process isolated in a network namespace have its own IP stack, independent from the host
  - Interfaces (loopback, eth0)
  - Routes
  - IP Tables rules

# Pods

- Pods are the most basic unit of deployment inside Kubernetes
- A pod contains one or more containers
- Pods are atomic and containers in it work as a unit
- **Containers inside a Pod shares the same network namespace**
  - Very cheap for two containers inside a Pod to talk to each other
  - There are no abstraction layer needed for intra pod communications





# CNI

- Pods should be allowed to talk to each other
- Pods should have unique IP addresses
- Kubernetes relies on several plugins for core functionalities
  - CRI - Container Runtime Interface
  - CSI - Container Storage Interface
  - CNI - Container Networking Interface
- CNI allows users to use Software Defined Networking (SDN) in their Kubernetes environment
- The CNI is in charge of:
  - Pod network communications
  - Pod IP addresses
  - ...



# Hands on Lab Part 1

- Install Kubernetes
- Install a CNI
- Have a look at the IP stack in Pods

All our pods can now communicate ! In a way we have everything we need. But it is not very practical.

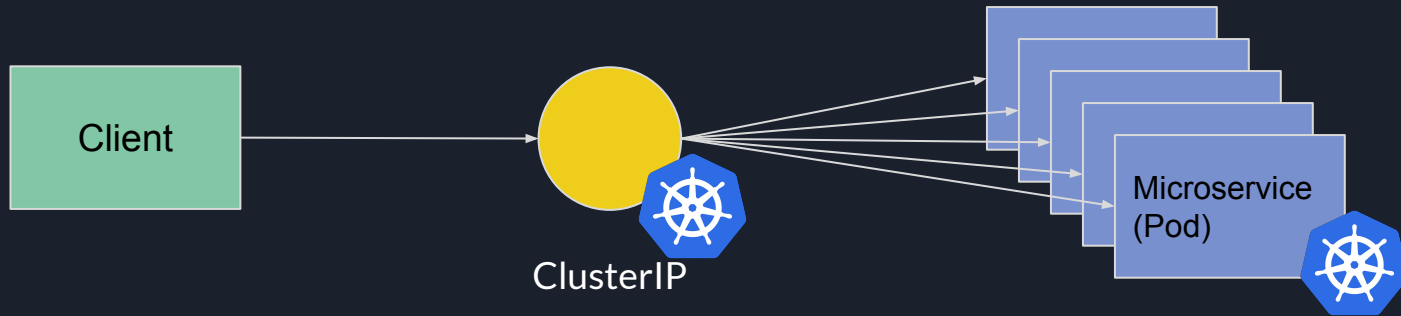
## Part 2: Services

Expose your microservices under a common IP

The background of the slide features a series of dark gray, three-dimensional rectangular blocks arranged in a perspective view, receding towards the top right. A light green parallelogram is positioned on one of the upper blocks, and a blue parallelogram is on a lower block, both oriented diagonally to match the perspective of the architecture.



# ClusterIP: expose your microservices under a common IP





# ClusterIP implementation

- kube-proxy
  - Use IPTables
  - Can use other modes
- Some CNI can implements ClusterIP



# IP Address ranges

- Hosts IP : Address given by your provider
- Pod IP : Unique IP address for a pod. The range is defined by the CNI
- Cluster IP : Unique IP address for a Service. The range is defined in K8s config

Each of these ranges must provide unique non-overlapping addresses !



# Service Definition

- A service is a K8S resource
- Needs a unique name in a given namespace
- Requires:
  - Port to listen to
  - Port to send traffic to
- Should use a selector to target appropriate Pods

```
apiVersion: v1
kind: Service
metadata:
  name: website
spec:
  ports:
    - port: 80
      name: http
  selector:
    app: website
```



# Endpoints

- Services only declares the state we want.
- Endpoints actually represents the routing
- Endpoints are created automatically according to the selector in the Service Definition
- Can be created manually (i.e. when destination is not a Pod)



# Different types of Services

- ClusterIP
  - Intra cluster communications. Forward traffic and provides name resolution
- Headless
  - Provides name resolution but does not provide traffic forwarding
- NodePort
  - For external services access. Opens port on our machines that forwards to a Pod
- LoadBalancer
  - Equivalent to NodePort but ports are open on an external LoadBalancer
- ExternalName
  - Exposes an external service



# Hands on Lab Part 2

- Deploy an App
- Expose it with a ClusterIP
- Checks what's happening

## Part 3: DNS







# DNS in Kubernetes

- IP address are not made to be remembered
- DNS provide a name to your application

The CRI (Docker) provides containers with custom files:

- `/etc/hostname` - Give the container its name
- `/etc/hosts` - Give the container its IP address
- `/etc/resolv.conf` - Give the container its DNS

The containers asks CoreDNS for resolution:

- Installed as ReplicaSet Deployment
- Uses a ClusterIP (10.96.0.10)



# Service Name

- A ClusterIP service create a DNS record
- `myservice.mynamespace.svc.cluster.local`
  - `myservice` – the service name
  - `mynamespace` – the namespace
  - `svc` – the directory for services
  - `cluster.local` – the cluster suffix set when installing the cluster
- `mysql.prodns.svc.k8sprod.adaltas.cloud`
- Resolves to a clusterIP



# Headless Service

- A headless is a special type of ClusterIP that does not resolve to a single IP
- Useful when typical LoadBalancing does not make sense
- Resolving a Headless Service will responds with all the endpoints available instead of just one
- When using a StatefulSet Deployment, each Pod will have its own non-random name
  - Headless Service will made Pod Name resolvable with their IP
    - kafka-1.kafka.mynamespace.svc.cluster.local
    - kafka-2.kafka.mynamespace.svc.cluster.local
    - kafka-3.kafka.mynamespace.svc.cluster.local

```
apiVersion: v1
kind: Service
metadata:
  name: headlesswebsite
spec:
  ports:
    - port: 80
      name: http
  selector:
    app: website
  clusterIP: None
```



# Hands on Lab Part 3

- Use DNS records to access our application
- Create a Headless Service
- Checks DNS Records

## Part 4 : External Access to the cluster





# Reach our Services from the outside

Multiples ways from the outside to the cluster:

- HostPort - the Pod opens a port on its host that forward to a port in the pod's interface
- NodePort - It's a service, all the hosts in the cluster forwards to one ClusterIP
- LoadBalancer - Service that relies on a plugin to create an access from an external LB
  - Actually a NodePort
- Ingress - Kubernetes framework for HTTP/HTTPS proxying
- Gateway - Not (yet) stricly defined. Like an Ingress but with more options

HostPort and NodePort handle all the work, it's just hidden behind everything else



# HostPort

- Defined by a Pod
- Maps a port on the Host to a Pod in the Pod
- Good when used when DaemonSets (one Pod per Host), for admin purposes
- Sucks for everything else
  - Will stay in pending state if port on host is closed, which cluster user does not control
  - you need to know where the pod lands to access it

```
apiVersion: v1
kind: Pod
metadata:
  name: myhostportpod
spec:
  containers:
    - name: hp
      image: httpd
      ports:
        - containerPort: 80
          hostPort: 8080
```



# NodePort

- Similar to HostPort in that it opens a port on the hosts
- Opens a port on every host on the cluster, even if the Pod is not scheduled on the host
- The port range allowed is defined by the admin
- Is actually a ClusterIP behind the scenes

```
apiVersion: v1
kind: Service
metadata:
  name: website
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 31100
      name: http
  selector:
    app: website
```





# LoadBalancer

- Provision a LoadBalancer to an external provided load balancer using a cluster plugin:
  - Cloud Provider Kubernetes environments (Google GKE, Amazon EKS...) provides their LB solutions
  - On prem solutions exists: MetalLB, Netris
- Is actually a NodePort behind the scenes
  - which is a ClusterIP...

```
apiVersion: v1
kind: Service
metadata:
  name: website
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 80
      name: http
  selector:
    app: website
```



# Ingress

- Kubernetes Framework
- Kinda limited but portable
- Ingress controller needs to be installed
- HTTP/HTTPS Only

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-ingress
spec:
  ingressClassName: ambassador
  rules:
  - http:
      paths:
      - path: /engine
        pathType: Prefix
        backend:
          service:
            name: engine
            port:
              number: 80
```



# Gateway

- Not (yet !) in Kubernetes API
- Works like an Ingress, but much more powerful
- Support for virtually all protocols
  - gRPC
  - UDP
  - TCP
  - Apache Thrift
- Advanced security and Auth
- Tightly coupled to the 'gateway controller', not portable like ingress
  - Use CRDs