REDES

Containers - Kubernetes

REDES

Containers

Containers

- Permiten que cada aplicación se ejecute en un entorno propio separado del resto de las aplicaciones que están en el SO

- Funciones del SO utilizadas por los containers:
  - Overlay FileSystem
  - CGroups
  - Namespaces
  - capabilities
  - seccomp_bpf
  - pivot_root

Containers - Problemas que resuelve

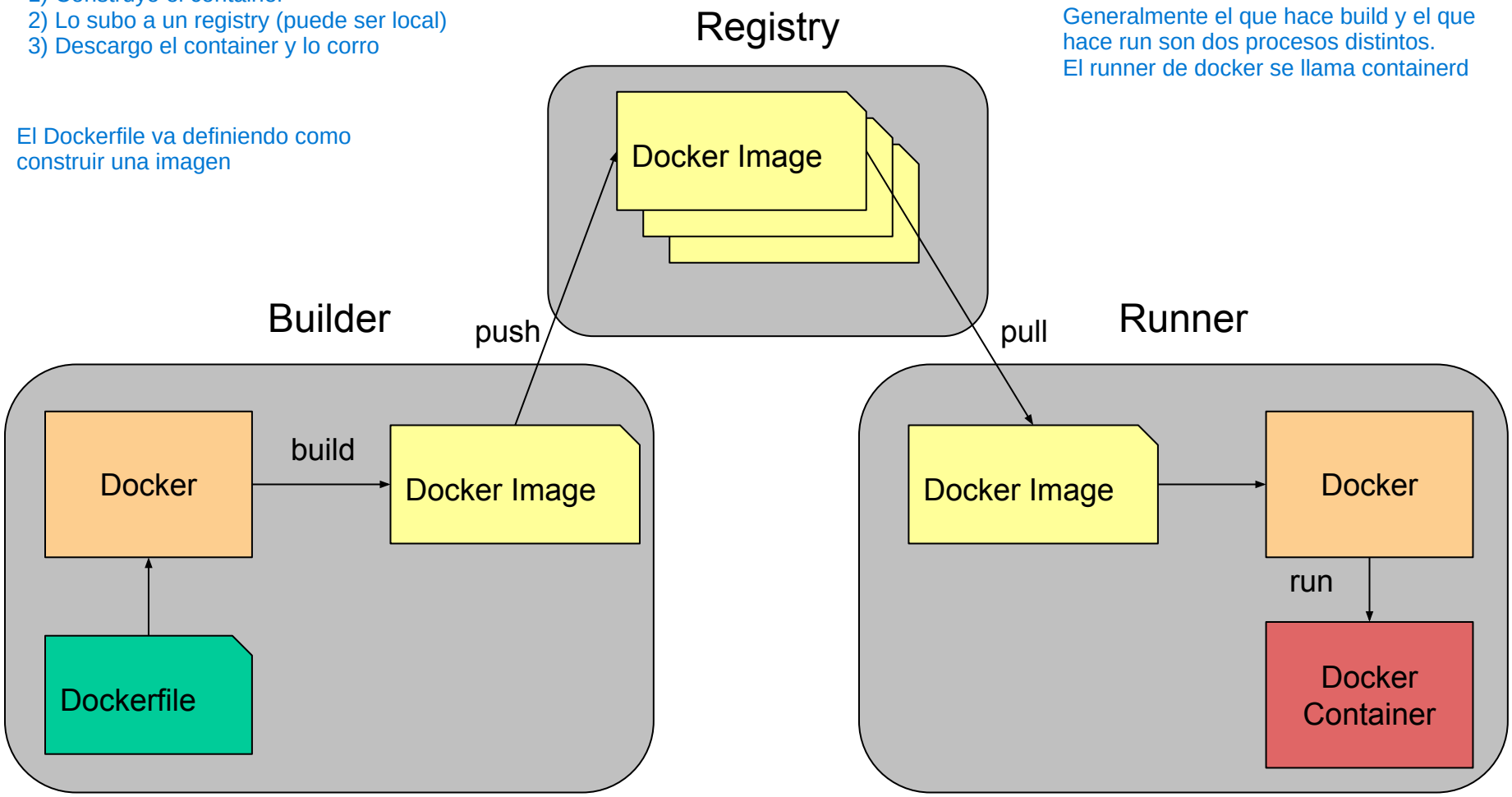
- Crea un entorno COMPLETO que permite la construcción de un paquete (build) para que luego se pueda ejecutar en otro ambiente (deploy & run).
- Resuelve el problema de hacer el **deploy & run** de una aplicación sin tener que preocuparse por todas las dependencias que pueda tener.
- Si bien hay opciones usualmente son Linux
- Containers más utilizados **Docker**
- Hay otras opciones: podman, lxd, containerd

Containers - Workflow

Proceso:
1) Construyo el container
2) Lo subo a un registry (puede ser local)
3) Descargo el container y lo corro

El Dockerfile va definiendo como construir una imagen

Generalmente el que hace build y el que hace run son dos procesos distintos.
El runner de docker se llama containerd



Containers - Nomenclatura

- Image: paquete que incluye todo lo necesario para ejecutar un proceso.
- Image Builder: el que genera la imagen
- Container Runtime: el que ejecuta el container.
- Image Registry: repositorio de imágenes.
- Container: proceso y ambiente que utilizan la imagen para ejecutar.

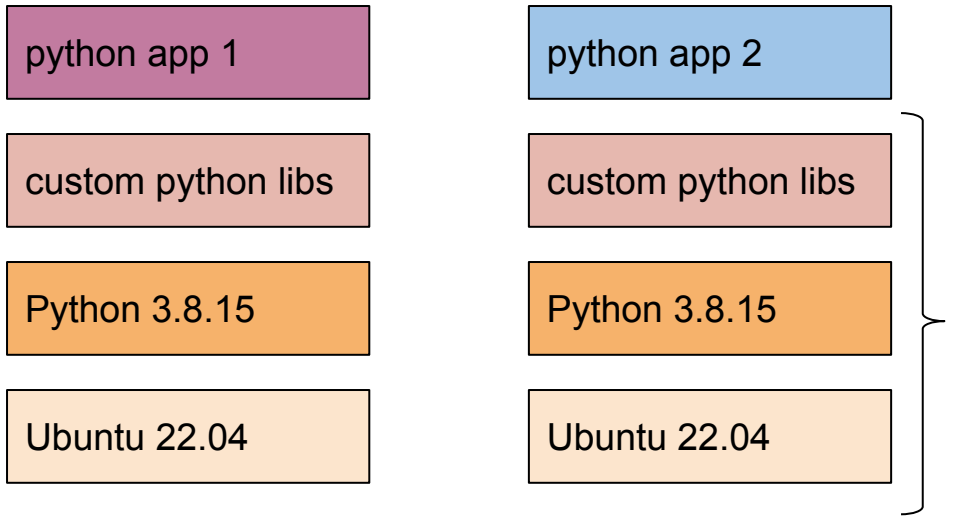
Containers - Image Layers

- Las imágenes se construyen por capas (Layers).
- Inicia por el SO, sus librerías de base, se le van incorporando archivos hasta completar TODO lo necesario para ejecutar el proceso principal.
- Cada capa tiene como dependencia su capa inferior
  - Si el mismo archivo se encuentra en más de una capa se usa la versión superior

    El overlay hace que por ejemplo si mi base layer tiene un archivo y yo por arriba hago una layer custom que tiene ese mismo archivo, se usa el del custom.
  - Todo lo que se escribe durante la ejecución va a una Layer efímera que se borra al terminar el container

    El Volume es un storage persistente que se monta en el overlay (pero no es una capa)
  - Para que no se pierda es necesario usar un **Volume**

Containers - Image Layers

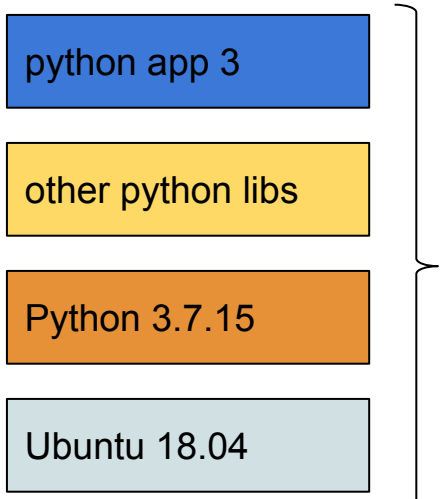


Si cambia únicamente la app se baja sólo esa capa

Compartido, se baja una sola vez

Ejemplo: yo hago un contianer con base Ubuntu 22.04. Sobre eso, le pongo un python. Sobre eso, le pongo librerias de python y sobre eso pongo mi app. La idea es que si tengo dos containers con todo igual menos la app, usan las mismas imagenes.

La idea del Overlay filesystem, es que yo cuando accedo a un directorio, los veo a todos como el mismo filesystem. Pero en realidad, estoy compartiendo las imagenes.



Requiere un entorno totalmente distinto que App1 y App2 pero puede correr en la misma máquina

Containers - Registry

- Docker Hub es el más usado y el default
- Es posible levantar un registry privado
- Para usar otro registry es necesario poner el origen explícitamente
- Las imágenes se bajan a la máquina local para ejecutar

Containers - Ejemplo de Dockerfile

FROM centos:7.4.1708

Image ID: centos
tag: 7.4.1708

La base image

RUN yum -y install net-tools logrotate tcpdump less

Cosas que quiero ejecutar para ir generando mi capa

ENV INSTANCE_ID=AABBCC

Configuro una env var

WORKDIR /opt/hello/

ENTRYPOINT ["./helloworld.sh"]

El entrypoint es para definir lo que quiero que se ejecute cuando prendo el container. El CMD es para pasarle parametros al ENTRYPOINT.

CMD ["Hello", "World"]

Containers - CGroups

- CGroup: Control Group

  Son los que permiten asociar
  CPU y memoria a cada container

- Restringen el uso del CPU y memoria
- Se genera un CGroup de CPU y otro de memoria por container.
- Se le asigna una cuota a cada CGroup
- De esta manera el container puede usar como máximo esa cantidad de recursos.

Containers - Namespaces

- Generan un entorno acotado para los procesos que corren en él
- Siempre hay un namespace default.
- Distintos tipos de Namespaces:
  - **pid** - Se asignan pids distintos dentro del NS

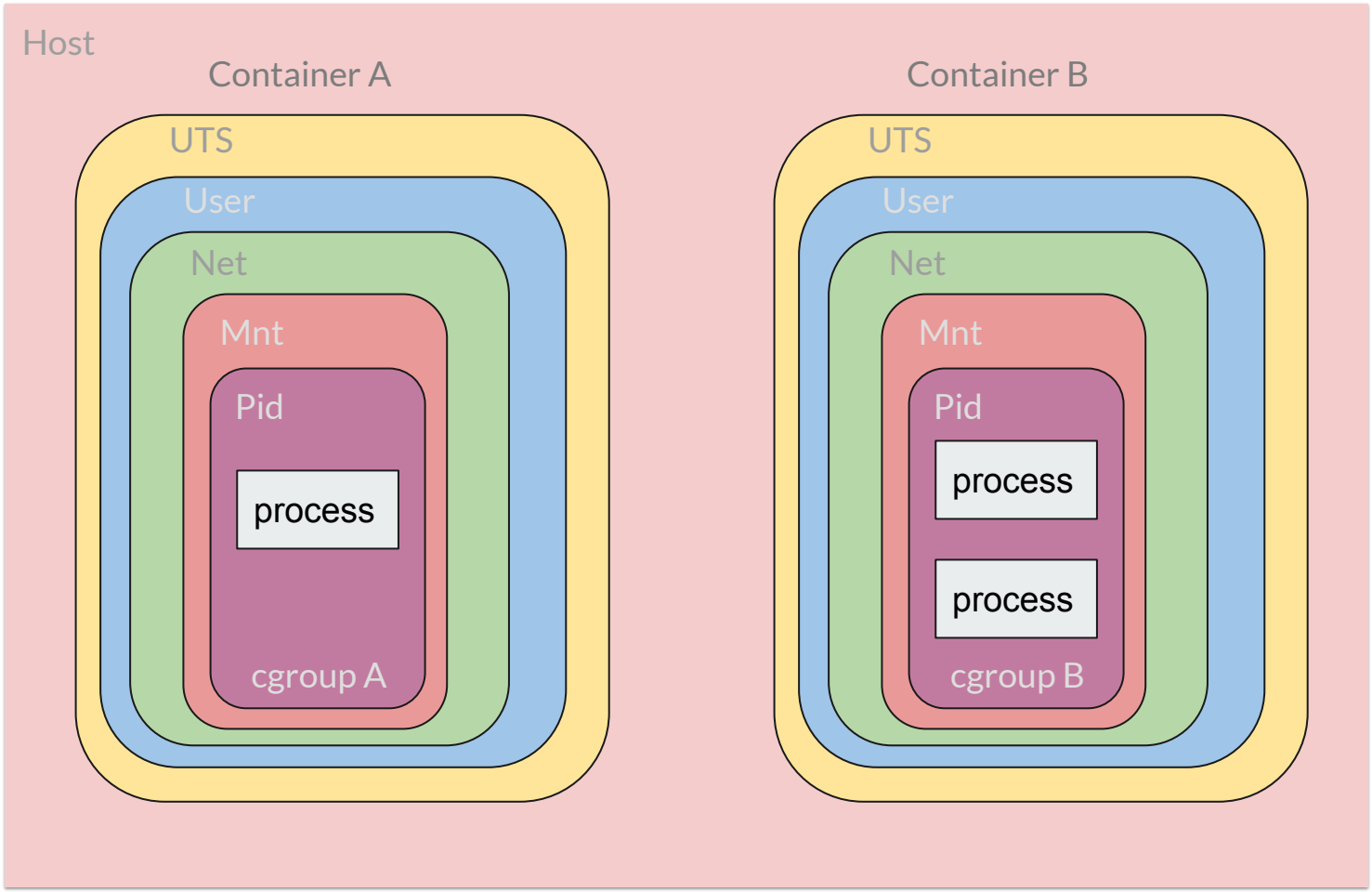
    Si un container ejecuta ps, entonces no puede ver todo el resto de los procesos que estan corriendo en la computadora. Esto es porque esta en un namespace de su propio pid.
  - **user** - usuarios propios del NS (root dentro del container != root dentro del host)
  - **uts** - hostname que ve el container
  - **ipc** - interprocess communication

    Para los semaforos y eso
  - **mnt** - ve su propio FS que es el overlay

    Solamente puede ver su propio filesystem
  - **net** - ve su propio stack de red e interfaces

erA

Containers - Namespaces



Containers - capabilities

- Capabilities son los permisos de bajo nivel que tiene cada usuario
- El usuario root también tiene capabilities como CAP_SYS_ADMIN o CAP_NET_ADMIN
- Algunas de estas capabilities se encuentran limitadas para usuarios dentro de los containers por motivos de seguridad

La idea es que el usuario dentro del container (por mas que yo sea root) tenga menos permisos. Entonces no puedo cambiar cosas como configuraciones de red.

Containers - seccomp_bpf

- seccomp_bpf permite controlar las **system calls** que se pueden ejecutar.
- Docker bloquea varias system calls peligrosas (por ejemplo reboot)

Containers - pivot_root

- cambia el root filesystem a un subdirectorio del host para ese container (el subdirectorio en donde esta montado el overlay)
- similar a chroot pero más seguro dado que chroot tiene vulnerabilidades

Para que mi container vea como '/' lo que esta montado en '/var/docker/overlay2/...', que es el overlay filesystem

La idea de Kubernetes es que yo tengo un registry con imagenes, y una "configuracion deseada".
Es la configuracion que quiero que tenga un grupo de hosts que corran una serie de containers.
Kubernetes se encarga de que mi configuracion deseada se respete en cada container.
Kubernetes nos ayuda cuando se cae uno de esos container, o cuando yo quiero cambiar en tiempo real esa configuracion.

REDES

Kubernetes

Un ejemplo de rollout & rollback es que yo hago rollout de mi version 2 de mi servicio. Veo que anda mal, y vuelvo rapidamente a la V1.

Funcionalidades Principales

- **Rollout & Rollback**: permite el deployment y/o rollback automático y progresivo de cambios mientras son monitoreados.
- **Service Discovery**: permite configurar las relaciones entre servicios por nombre.

Los containers me van a exponer servicios de red. Yo voy a poder acceder a esos servicios (ya sea otros containers o un tercero) mediante el service.

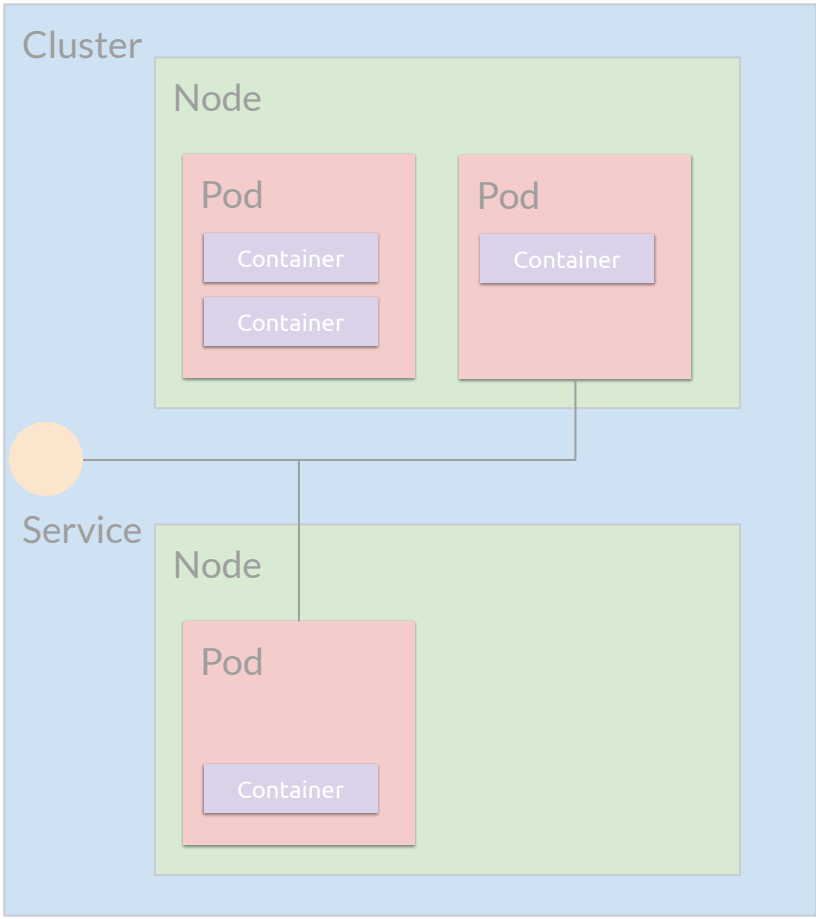
- **Service Load Balancing**: permite balancear los pedidos entre servicios.
- **Storage Orchestration**: montar volúmenes locales, cloud y otros.
- **Self Healing**: reinicia container caídos, mueves containers a otras VMs cuando alguna se cae, realiza health checks y quita de los servicios a los que no los cumplen.
- **Horizontal Scaling**: escala automática o manualmente en containers (pods) y/o VMs (nodos).
- **Manejo de Configuración y Secretos**: permite guardar de forma segura la configuración e información sensible

El autoscaling levanta nuevos pods o nodos si se van cayendo

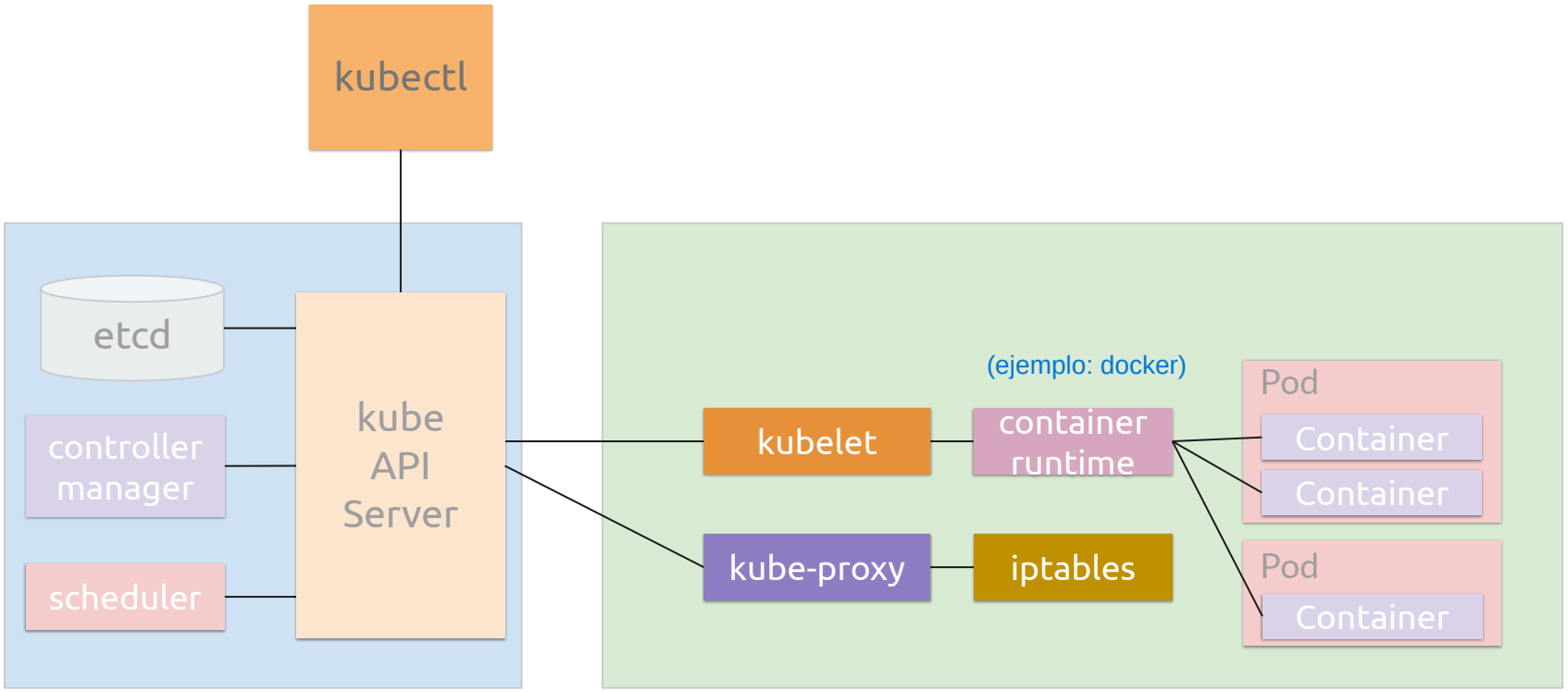
Conceptos Principales

- **Cluster**: deployment completo de Kubernetes (puede ser en multiples computadoras fisicas o virtuales)
- **Nodo**: VM o servidor físico en donde se ejecutarán los containers.

  Se garantiza que cada pod corre en un solo nodo
- **Pod**: La unidad básica de ejecución que maneja Kubernetes. Puede contener uno o más Containers.
- **Service**: La manera de acceder a los pods que implementan algún servicio en particular.



Arquitectura de Kubernetes



Control Plane 1 - N

Node 1 - M

Hay minimo 3 control plane por redundancia.
Son VMs puntuales que controlan a todos los nodos que estan disponibles.
Kubectl usa el kube API server, que es una api rest que toma todas las acciones sobre el cluster.
Ejecuta contra una base de datos VALUE-PAIR (etcd) muy simple pero distribuida (que siempre converge).
Todo el resto de los procesos son stateless (todo lo van a buscar al API server).

El kubelet es el proceso de kubernetes que corre en el nodo. Es el encargado de administrar el pod de ese nodo. Sabe que pod tiene que correr porque se lo dice el API server. El controller manager crea los nodos logicamente y el scheduler le dice al kubelet que lo cree.
El kubelet levanta los pods a traves del container runtime.

https://www.youtube.com/shorts/aB0zE-gzgkY

El kube-proxy configura todo el networking de un nodo, generalmente tocando los ip-tables.
Les crea ips privadas a los nodos y permite conectarlos entre si.

Componentes

- **kube API Server**: Única interfaz por la que se ejecutan comandos sobre el cluster
- **kubectl**: Interfaz de configuración y administración de Kubernetes. Ejecuta los comandos haciendo llamadas a la API.
- **Controllers (controller manager, scheduler, kubelet)**: Ejecutan un loop donde:
  - chequean estado de la config
  - chequean estado real del sistema
  - ejecutan acciones para que converjan
- **etcd**: Base de datos de Key Value Pairs con configuración distribuida redundante
- **kubeproxy**: manejo de todo el networking del nodo en base a configuración de iptables, rutas, etc.

Funcionamiento General

- Todo se ejecuta o chequea usando la **Kube API**.
- La **Kube API** principalmente tiene como interfaz el **etcd** donde escribe y lee.
- Todos los procesos son **Stateless** y el estado se encuentra únicamente en el etcd.
- El **Scheduler** tiene por función asignar a un nodo los pods que no tienen uno asignado.
- El **kubelet** (que corre en todos los nodos) tiene por función crear y matar los pods dentro de un nodo (y configurar la red interna de los mismos).
- El **kubeproxy** (que corre en todos los nodos) tiene por función mantener la configuración de networking externa del nodo (services, etc.) usando reglas de iptables.
- El **Controller Manager** maneja casos específicos de pods como CronJobs, ReplicaSets, DaemonSets, etc.

Definición de Objetos en Kubernetes

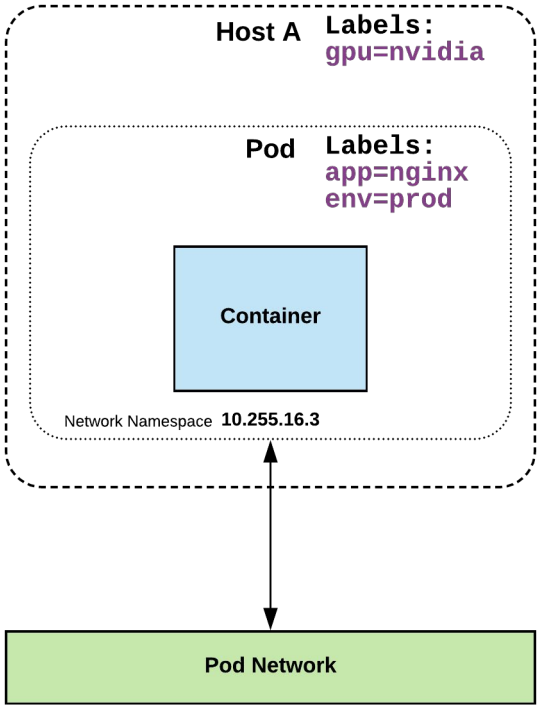
- Los objetos se definen en archivos Manifest.
- Estos archivos tienen formato YAML
- Deben tener los siguientes parámetros:
  - **apiVersion**: versión de la API de Kubernetes para el objeto
  - **kind**: tipo de objeto
  - **metadata.name**: nombre único del objeto

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
  namespace: default
  uid: f8798d82-1185-11e8-94ce-080027b3c7a6
```

Labels

- Key Value Pairs que identifican y describen y agrupan un grupo de objetos
- No definen unicidad, la idea es que apliquen a muchos objetos
- Se definen como parte de la metadata

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-label-example
  labels:
    app: nginx
    env: prod
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
```



Selectors

- Se usan los labels para especificar a qué objetos aplican los comandos u otros objetos

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-label-example
  labels:
    app: nginx
    env: prod
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
  nodeSelector:
    gpu: nvidia
```

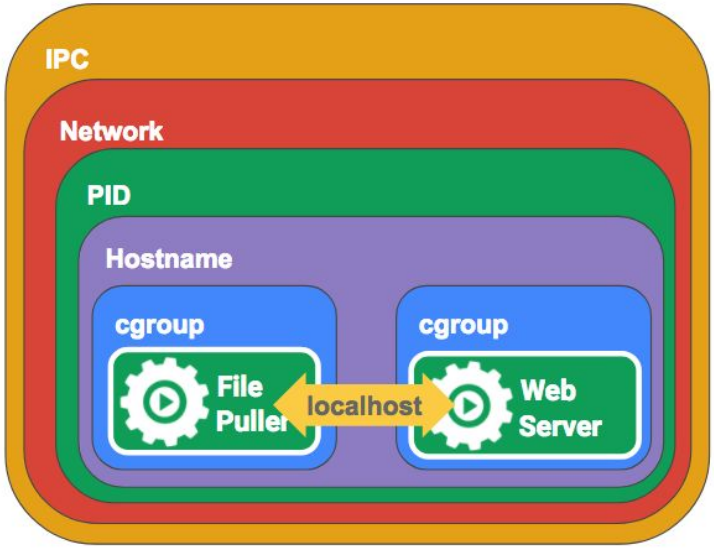
Aca le estoy diciendo que label quiero que tenga el nodo donde quiero correr determinado pod

Pods

- Unidad mínima de trabajo en Kubernetes
- Pueden estar compuestos por uno o más containers
- Son **efímeros**!! La idea es que se puedan matar en poco tiempo
- Los containers comparten todos los namespaces

(pero tienen un cgroup especifico cada uno)

En esencia es como si fueran un container
mas grande con varios procesos adentros



Ejemplos de Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
```

puerto que abre el pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-example
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
    ports:
    - containerPort: 80
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
  - name: content
    image: alpine:latest
    command: ["/bin/sh", "-c"]
    args:
      - while true; do
          date >> /html/index.html;
          sleep 5;
        done
    volumeMounts:
    - name: html
      mountPath: /html
  volumes:
  - name: html
    emptyDir: {}
```

Parámetros del Container de un Pod

- **name** - Nombre del container
- **image** - Imágen del container
- **ports** - arreglo de los puertos a exponer por fuera del pod
- **env** - arreglo de variables de environment
- **command** - arreglo de Entrypoint (equivalente al ENTRYPOINT de Docker)
- **args** - argumentos a pasarle al command (equivalente al CMD de Docker)

Container

```
name: nginx
image: nginx:stable-alpine
ports:
  - containerPort: 80
    name: http
    protocol: TCP
env:
  - name: MYVAR
    value: isAwesome
command: [“/bin/sh”, “-c”]
args: [“echo ${MYVAR}”]
```

Pod Template

- Define un Pod genéricamente
- No define el nombre ni versión
- Define sus Labels y su imágen
- Usado por otras entidades para crear Pods


```
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx
```

ReplicaSet

Conjunto de pods. Yo solo defino cuantas replicas a levantar

- Manejo de réplicas de los pods
- Incluye el manejo del scheduling, escalabilidad y borrado
- Su misión es asegurar que siempre exista el número requerido de pods de cada tipo
- Los parámetros principales son:
  - **replicas**: la cantidad de instancias deseadas del pod
  - **selector**: el tipo de pod a instanciar

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  template:
    metadata:
      labels:
        app: nginx
        env: prod
    spec:
      containers:
      - name: nginx
        image: nginx:stable-alpine
        ports:
        - containerPort: 80
```

Le pido que levante pod templates con estos labels

Deployment

- Construye sobre el ReplicaSet
- Permite manejar Updates y Rollbacks
- Crea identificadores automáticos con cada update para los replicaSets y sus Pods
- Permite definir la cantidad de versiones anteriores a mantener
- Permite definir la estrategia de actualización:
  - Recreate: baja todos los pods primero y luego sube la nueva versión No hay availability por un tiempo
  - RollingUpdate: va bajando algunos y creado otros de manera gradual Esto permite mantener la availability

Parámetros de un Deployment

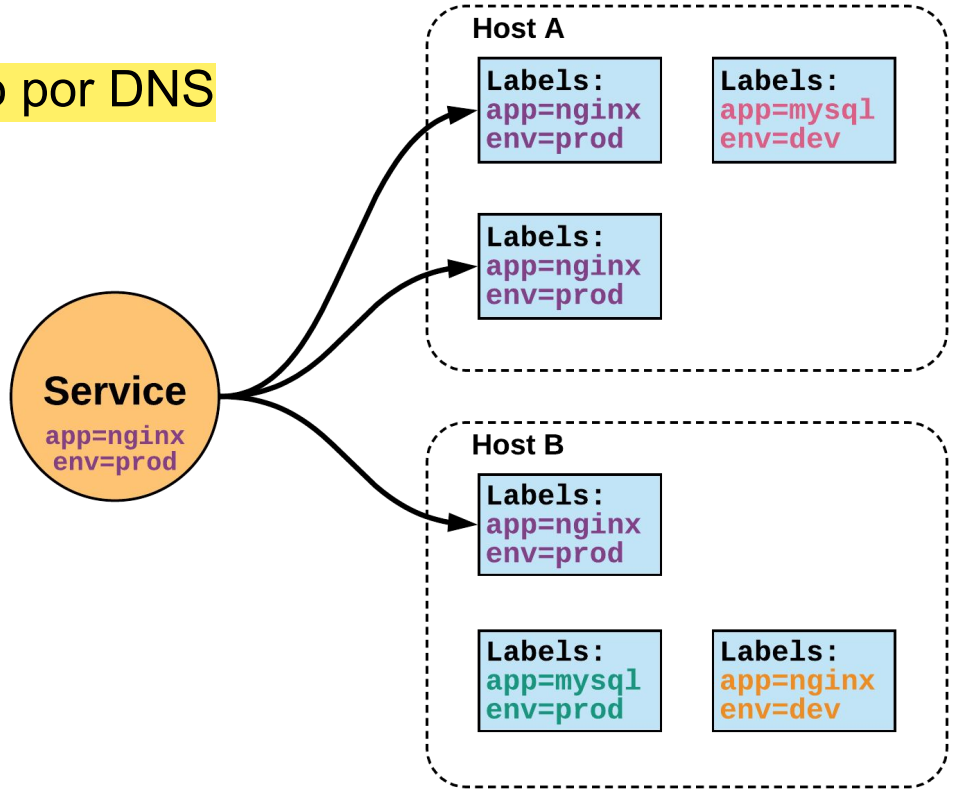
- **revisionHistoryLimit**: la cantidad de Deployments previos a guardar para un Rollback
- **strategy**: Recreate o RollingUpdate.
  - **Recreate**: se matan todos los pods y se vuelven a crear
  - **RollingUpdate**: itera por los pods bajando los viejos y subiendo los nuevos de acuerdo a los parámetros: **maxSurge** y **maxUnavailable**.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-example
spec:
  replicas: 3
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    <pod template>
```

Servicios

- Forma de acceder a los pods
- Recurso permanente - No efímero
- Balancea los pedidos entre los pods
- IPs estáticas del cluster
- Nombres de servicio resuelto por DNS

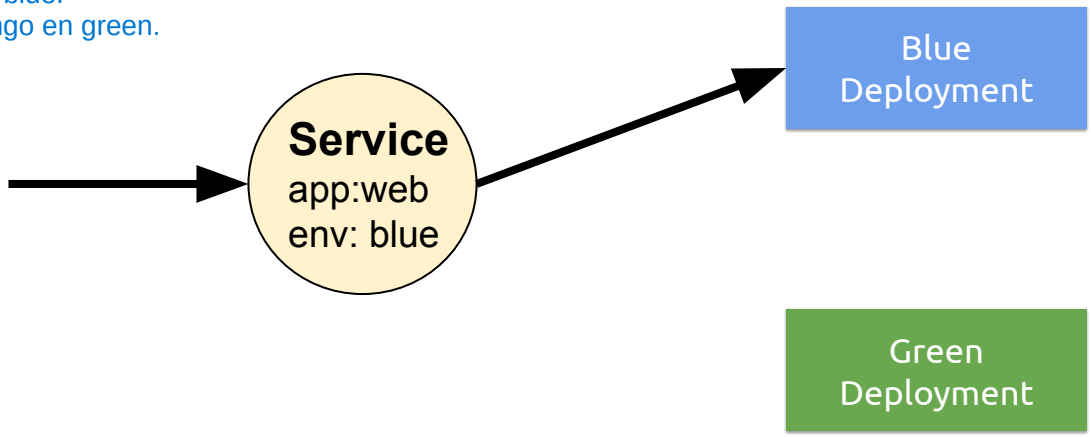
```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  selector:
    app: nginx
    env: prod
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```



Blue Green Deployments

- Tener en paralelo dos deployments activos
- Pueden ser versiones Actual y Nueva
- Pueden ser variantes que estoy probando}
- Defino los dos deployments con labels distintos
- Defino un servicio que apunte a uno u otro

Ninguno de los dos colores son el activo o no, es solo una manera de nombrarlos.
Por ejemplo:
Tengo mi V1 de mi app corriendo en blue y la V2 corriendo en green.
Si quiero pasar a V3, la pongo en blue.
Ahora si quiero pasar a V4, la pongo en green.



Networking General

- Todos los containers dentro de un Pod se pueden comunicar libremente
- Cualquier Pod puede comunicarse con cualquier otro sin NAT
- Todos los nodos pueden comunicarse con todos los pods sin NAT y viceversa
- La IP en que se ve un Pod es la misma que ven todos los otros Pods (porque no hay NAT).

Networking General - 2

- Todos los containers de un mismo Pod comparten un único namespace de networking y por lo tanto comparten la misma IP.
- Los containers dentro de un pod se comunican por localhost del namespace que comparten.
- Entre Pods se comunican desde la IP asignada interna al cluster que permanece fija mientra el pod se encuentre activo.
- Para un Servicio se asigna una IP fija del cluster independientemente del estado de los Pods

Ademas del IP de cada pod, cada servicio tiene su propia IP (haya o no haya pods asociados a ese servicio). Son IPs "virtuales" porque no estan asociadas a ningun proceso.

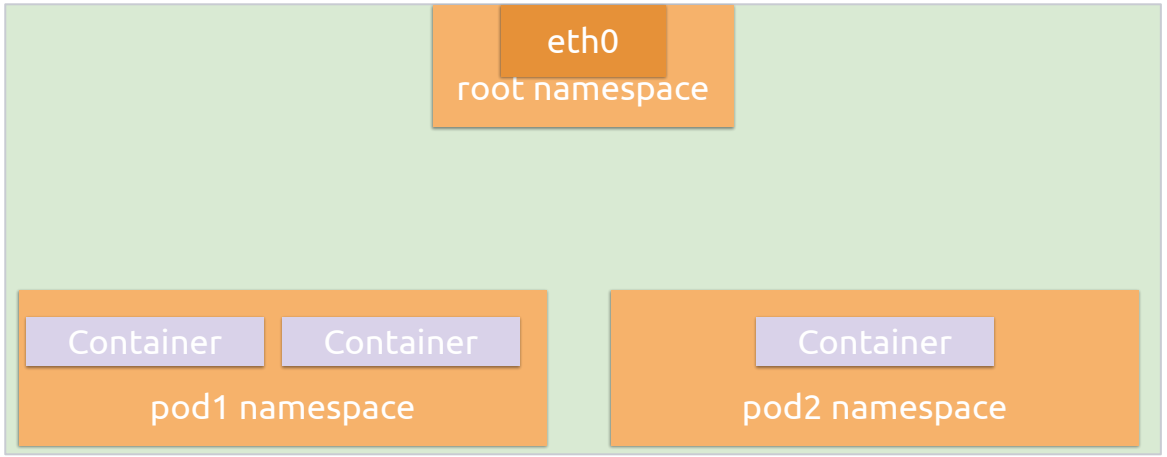
- La comunicación externa al cluster se maneja de distintas maneras con un LoabBalancer externo por ejemplo

Comunicación entre Containers - Mismo Pod

networking namespace

- Provee su propio stack de networking
- Provee sus:
  - rutas
  - interfaces
  - reglas de firewall
- Los containers de un mismo POD se comunican por localhost
- El localhost de los containers del pod no ve los otros containers en el mismo Nodo.

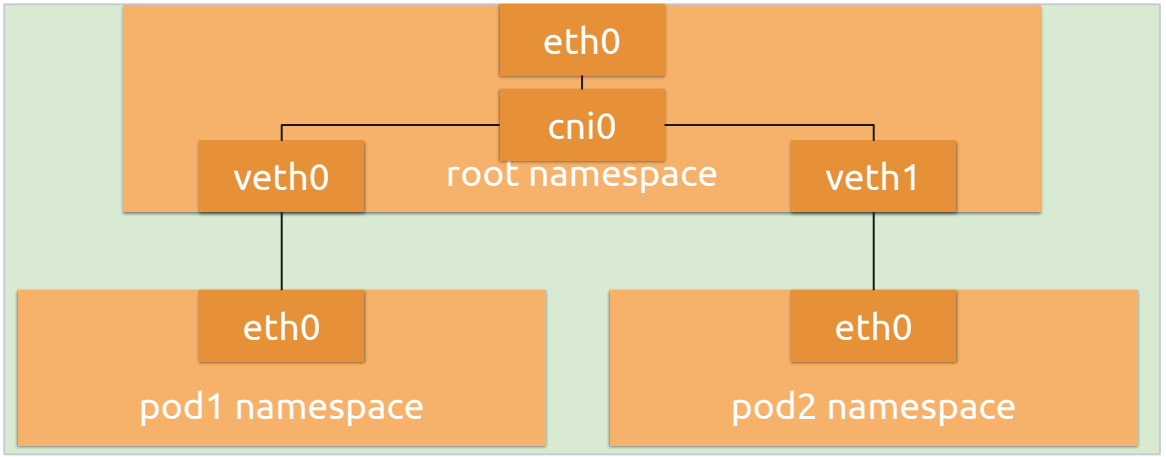
El localhost del pod1 es distinto al del pod2



Comunicación entre Pods - Mismo Nodo

Existe un bridge virtual que se genera en el nodo. Une a las interfaces virtuales con el eth0 root, para poder salir del nodo.
Para comunicarse entre pods, el camino es eth0 <-> veth0 <-> cni0 <-> veth1 <-> eth0

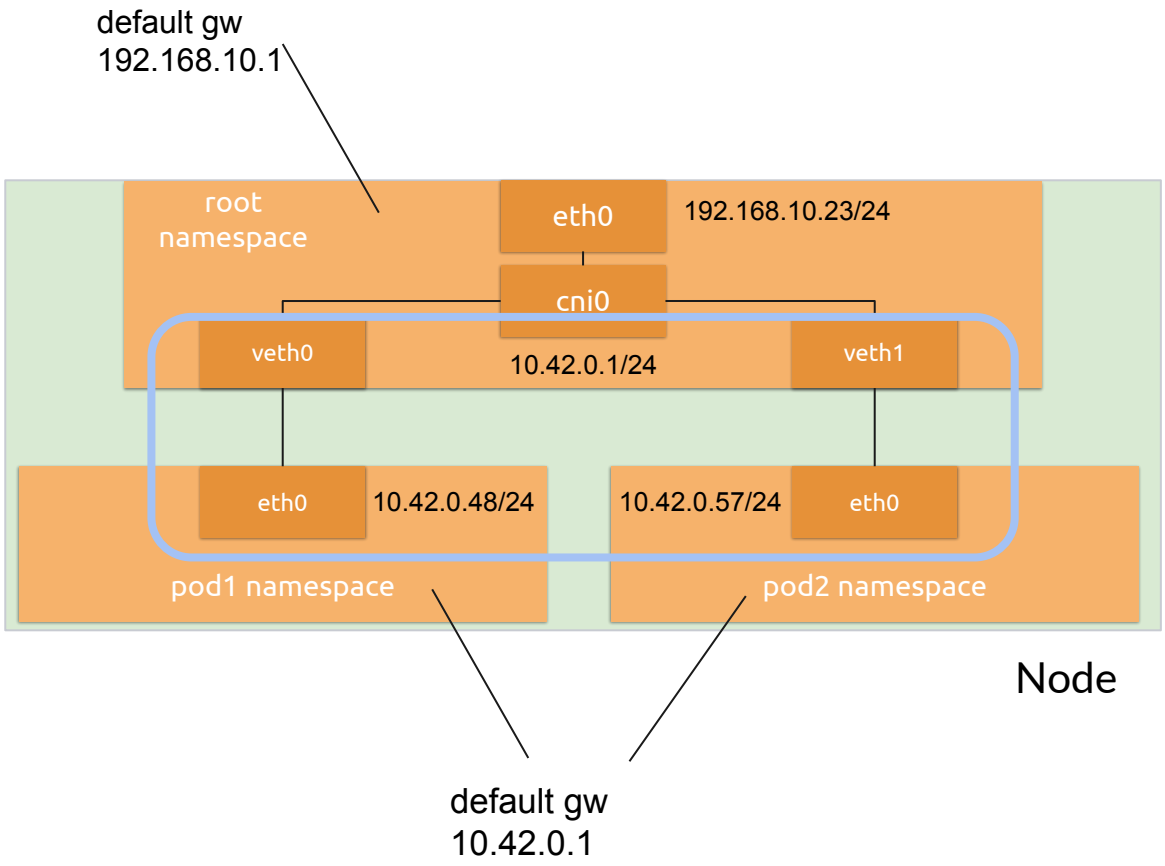
- Cada Pod tiene su eth0 con su IP en la red de los Pods
- Estos eth0 se encuentran conectados al root namespace con una veth0 en ese namespace
- Existe un virtual bridge dentro del root namespace que conecta todos los vethx dentro del nodo y el eth0 del propio nodo
- De esta manera un pod se conecta con otro a través de la IP asociada al Pod.



Node

IPs y Redes en el Nodo

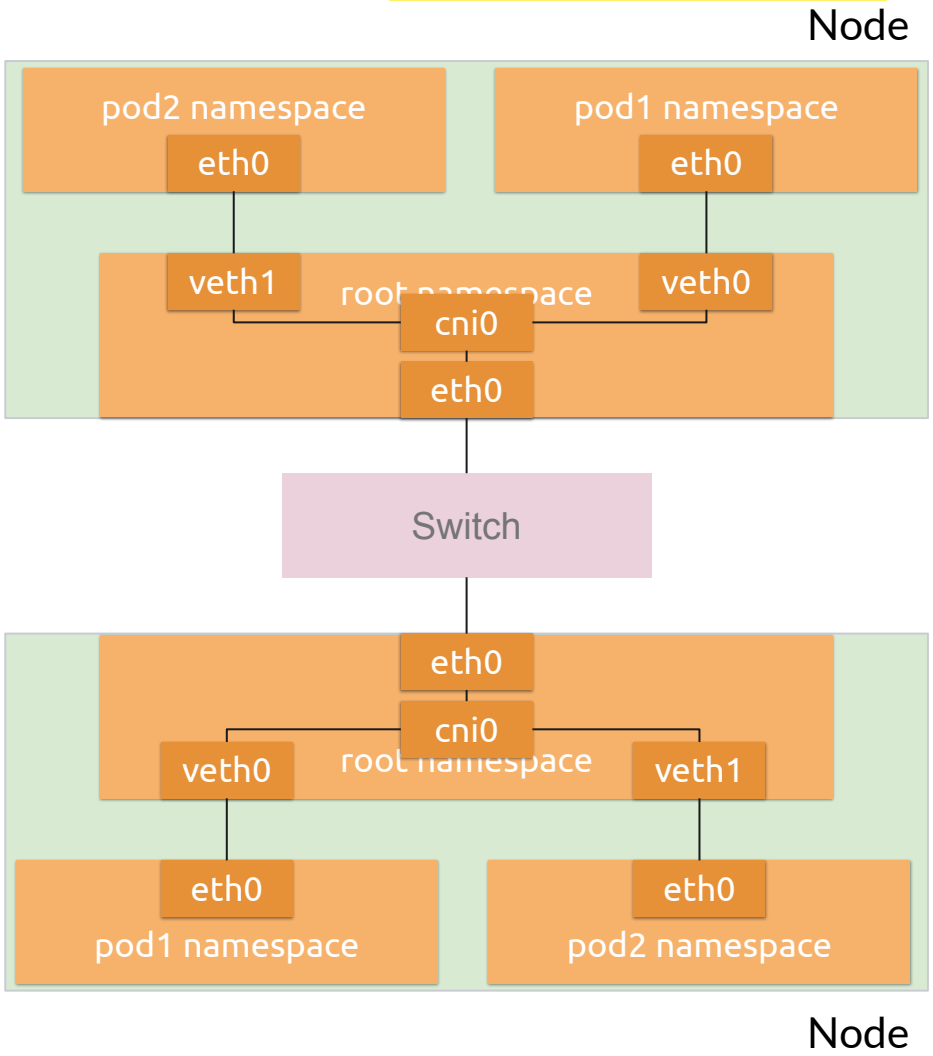
- Cada Nodo tiene una red privada /24 propia
- Cada Pod tiene en su eth0 una IP en esa red
- El bridge (cni0) tiene IP y es el GW de esa red
- En las rutas del namespace de cada pod el default GW es la IP del bridge
- En las rutas del nodo (root namespace) el default GW es el gw de la red y el device es el eth0 de la máquina
- Todo el deployment tiene una red /16 donde se encuentran todos los pods de todos los nodos



(Contiene todas las /24)

Comunicación entre Pods - distinto Nodo

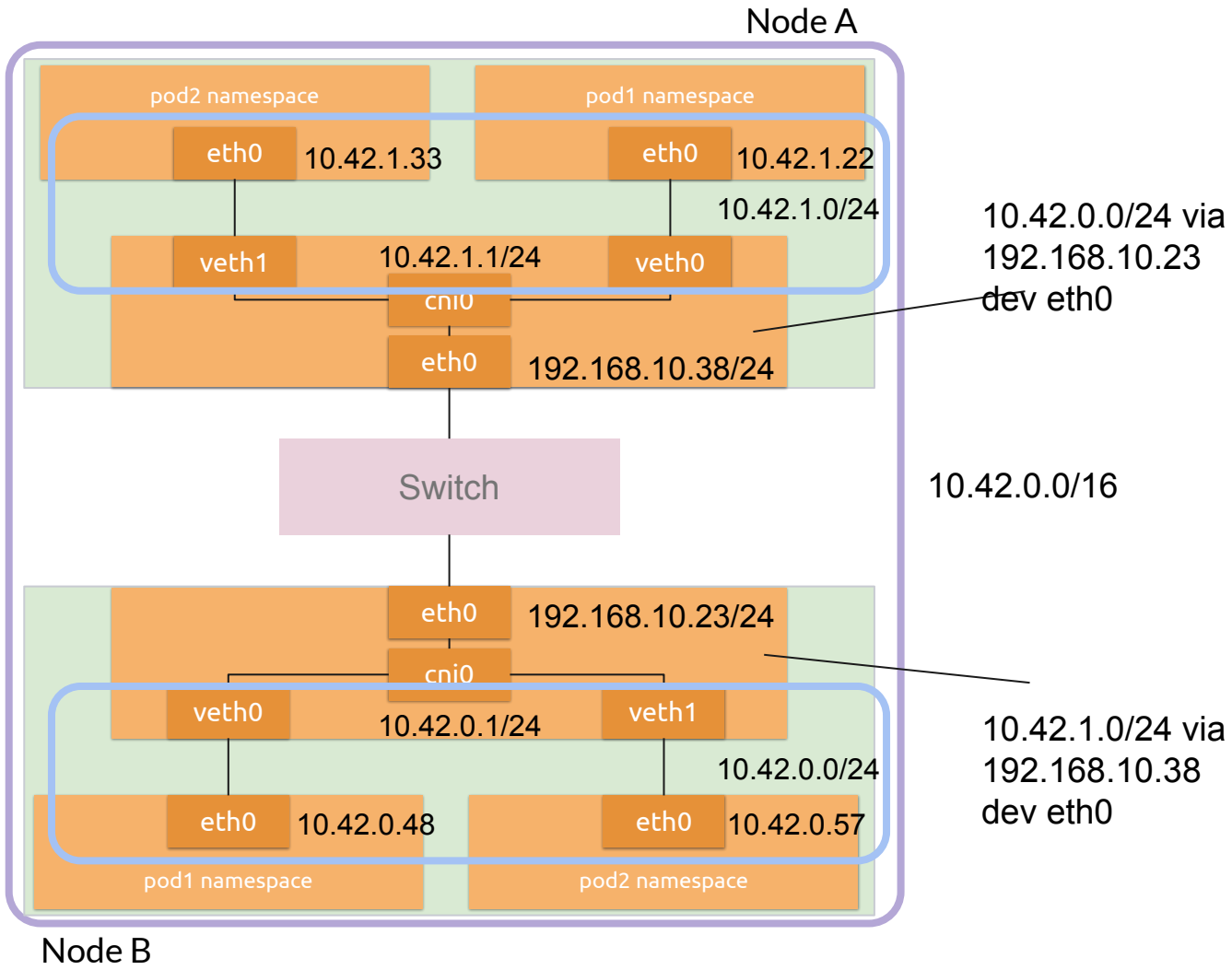
- Cada Pod sale con su IP y se comunica con cualquier otro Pod usando la IP de ese pod.
- Los paquetes salen con esa IP y se enruta al nodo correcto.
- Esto depende de cada implementación. Puede ser por reglas de enrutamiento de cada nodo o por overlay network



Si es por reglas de enrutamiento, cada nodo tiene que conocer las tablas de ruteo del resto de los nodos

IPs y Redes entre Nodos

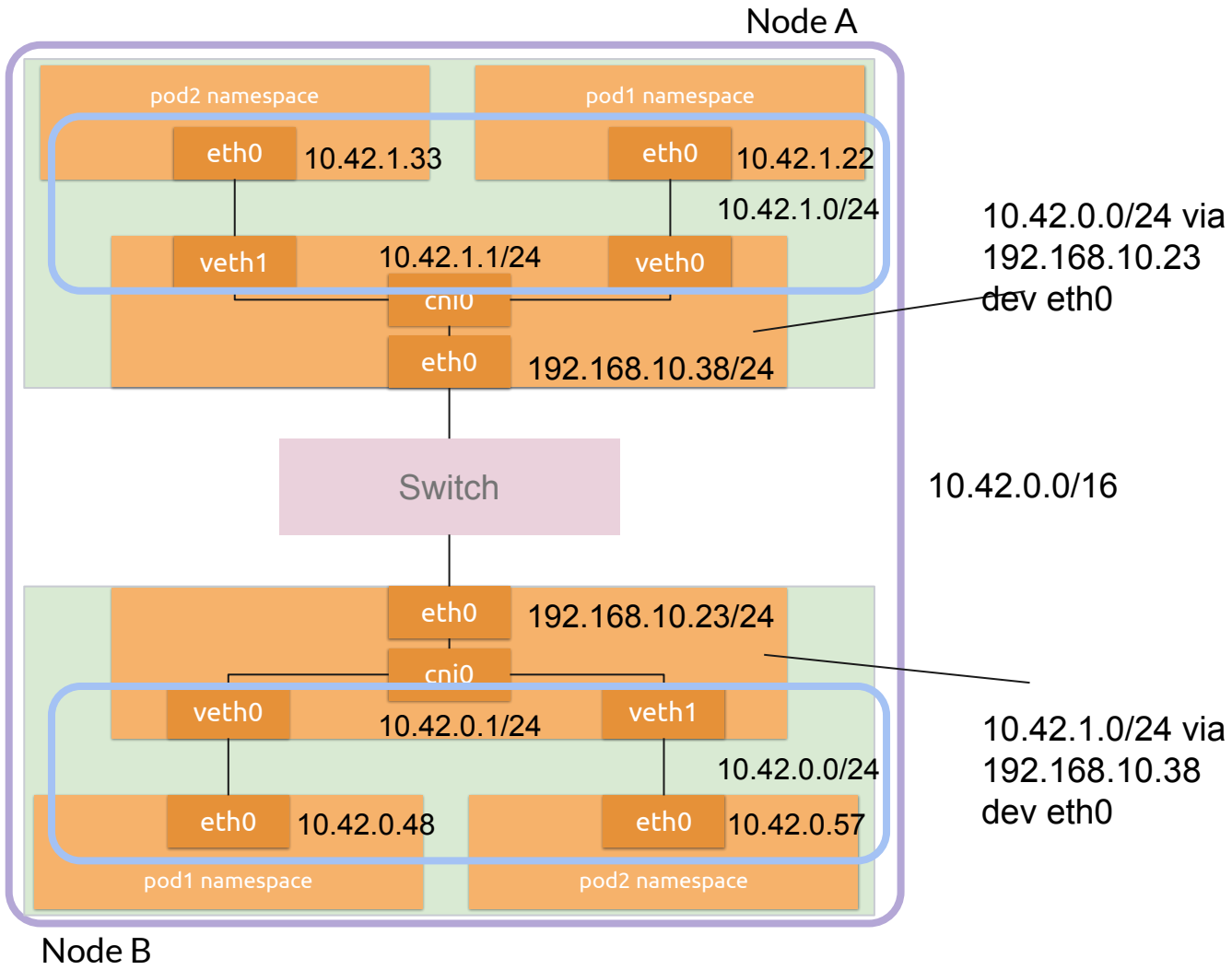
- Existe una red /16 que contiene todas las subredes de los nodos
- Los paquetes salen con esa IP y se enruta al nodo correcto por reglas de enrutamiento de cada nodo o por overlay network



(slide repetida =))

IPs y Redes entre Nodos

- Existe una red /16 que contiene todas las subredes de los nodos
- Los paquetes salen con esa IP y se enruta al nodo correcto por reglas de enrutamiento de cada nodo o por overlay network



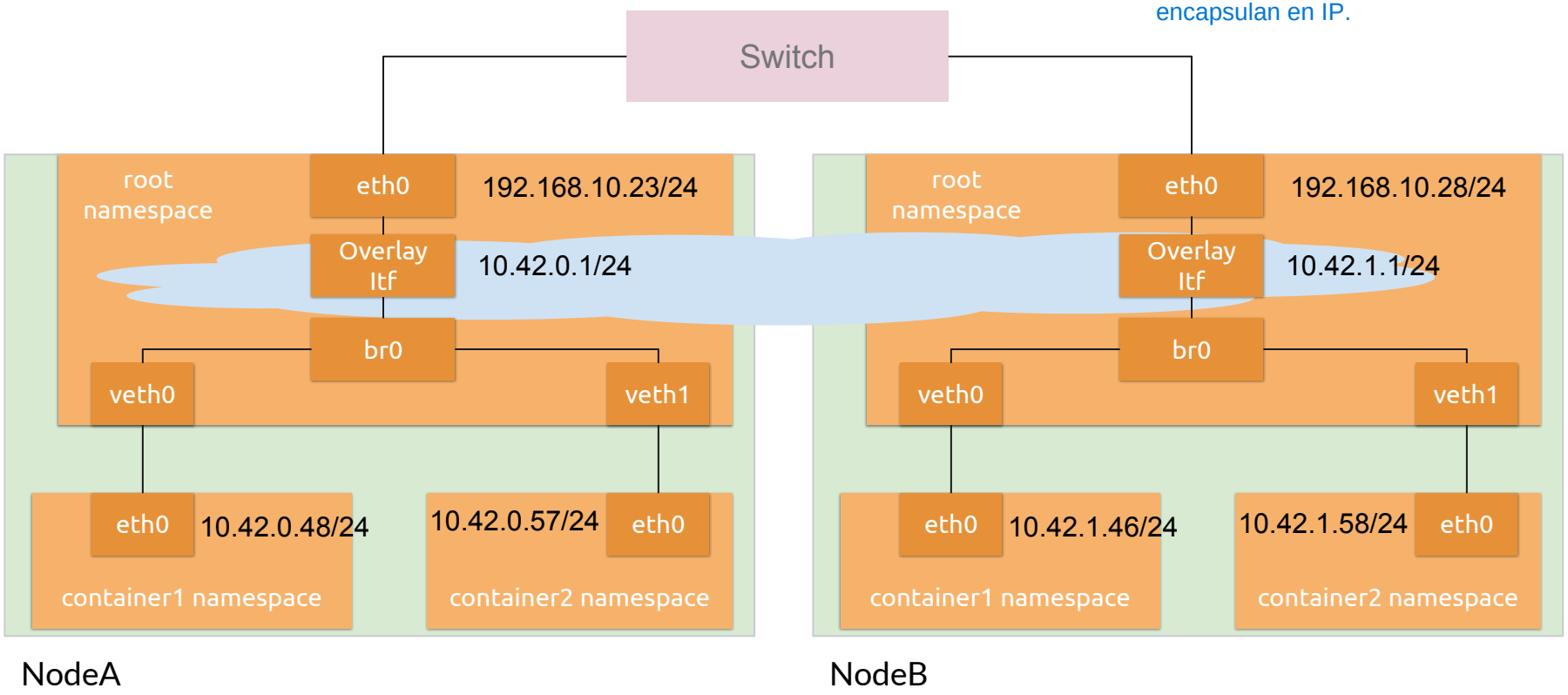
Comunicación entre Pods - distinto Nodo

Overlay Networking

Parecido a VPN pero en capa 2.
Se agrega una interfaz que me va a encapsular todo lo que quiera salir de los nodos. Se va a encargar de hacer el ruteo (usando una base de datos externa que le mantenga que IPs tiene cada nodo y cual es la IP de cada nodo).
Usando este metodo puedo tener algunos nodos on premise y otros en AWS, dado que los paquetes se encapsulan en IP.

Overlay Network

- Crea una red entre containers de distintos Hosts
- Los paquetes son encapsulados y desencapsulados

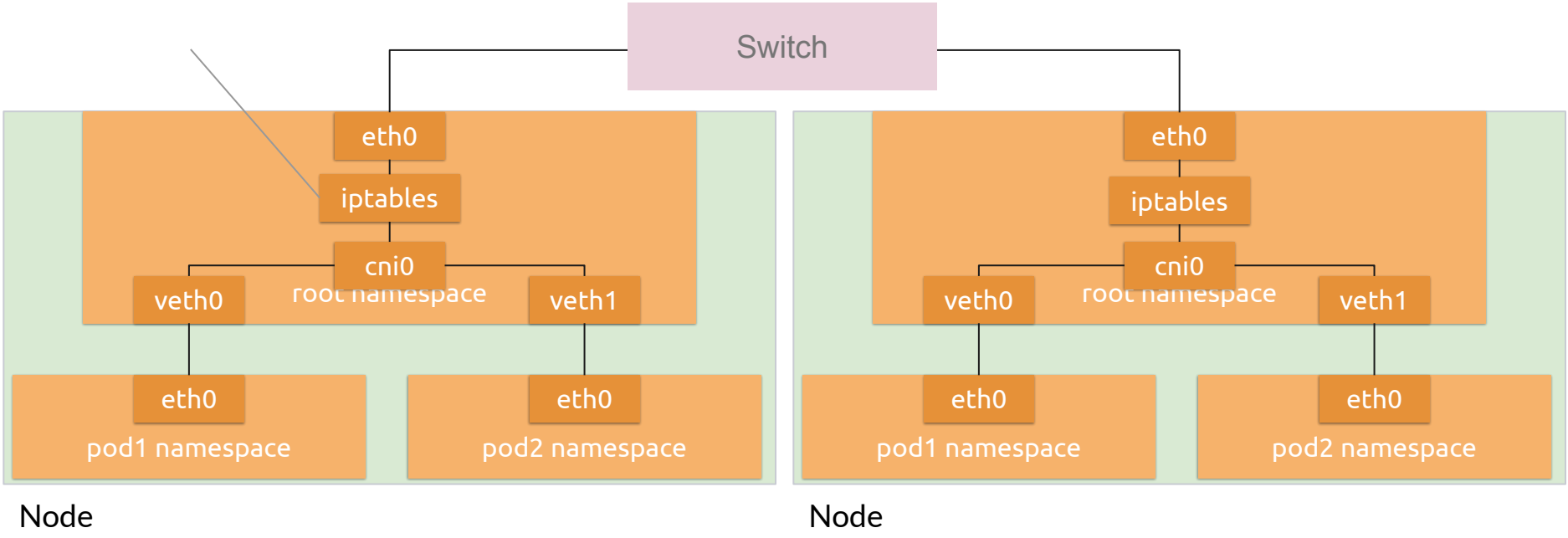


Overlay Networking - Ejemplo

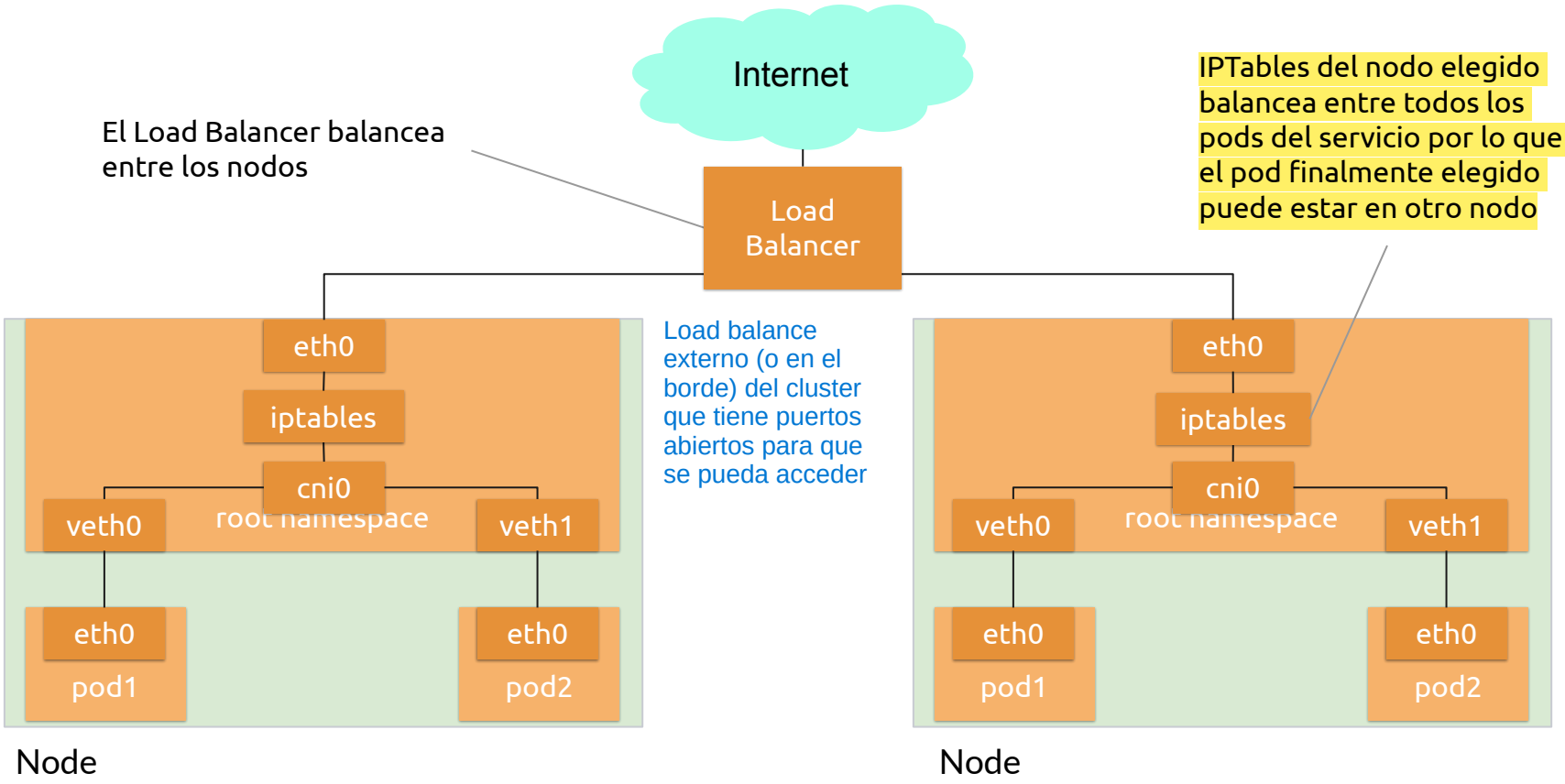
- Plugin de networking
- Genera un Overlay Network encapsulando los paquetes internos con VXLAN
- Mantiene la lista de nodos en un etcd
- Tiene un demonio flanneld que actúa como interfaz virtual y que consulta el etcd para conocer las IPs externas de los nodos y las redes internas de los mismos.

Comunicación de un Pod a un Service

IPTables elige un Pod al azar de los que están asociados al servicio y cambia la IP de destino del paquete

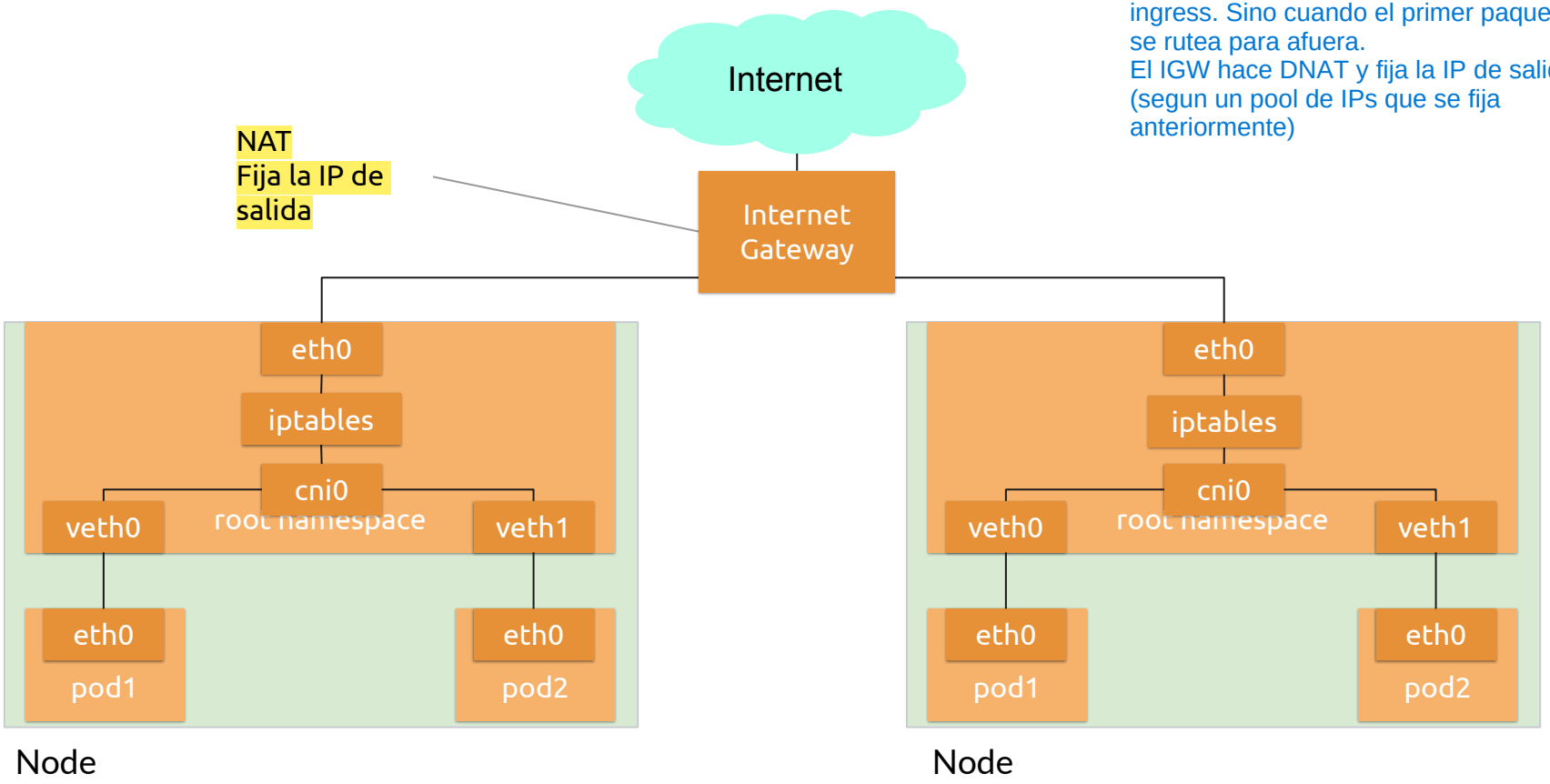


Tráfico Externo a un Service (Ingress)



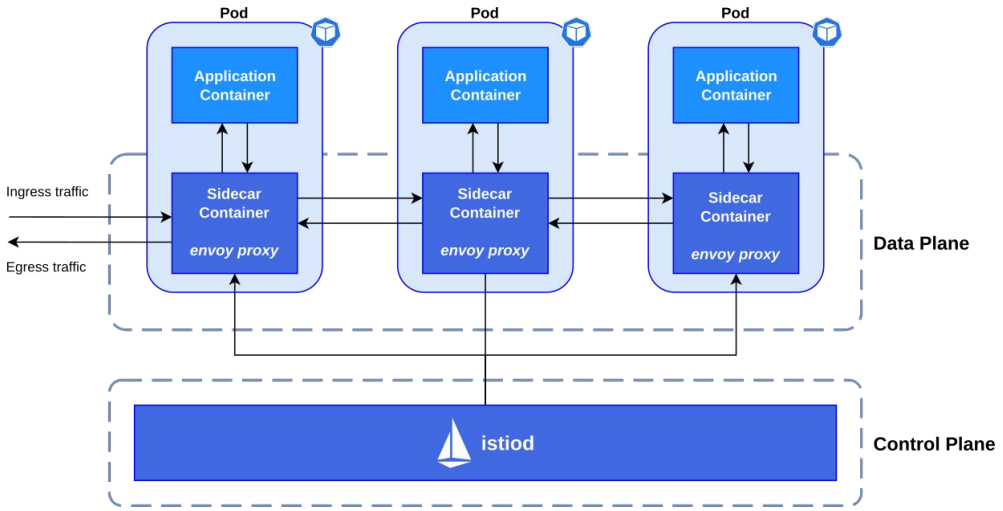
Tráfico Externo desde un Pod (Egress)

No es cuando es una respuesta a ingress. Sino cuando el primer paquete se rutea para afuera.
El IGW hace DNAT y fija la IP de salida (segun un pool de IPs que se fija anteriormente)



Istio

- Istio agrega una capa de management y observabilidad para kubernetes que permite configuraciones con más control que Kubernetes base



Kiali + Istio

- Kiali es una herramienta de visualización integrada con Istio