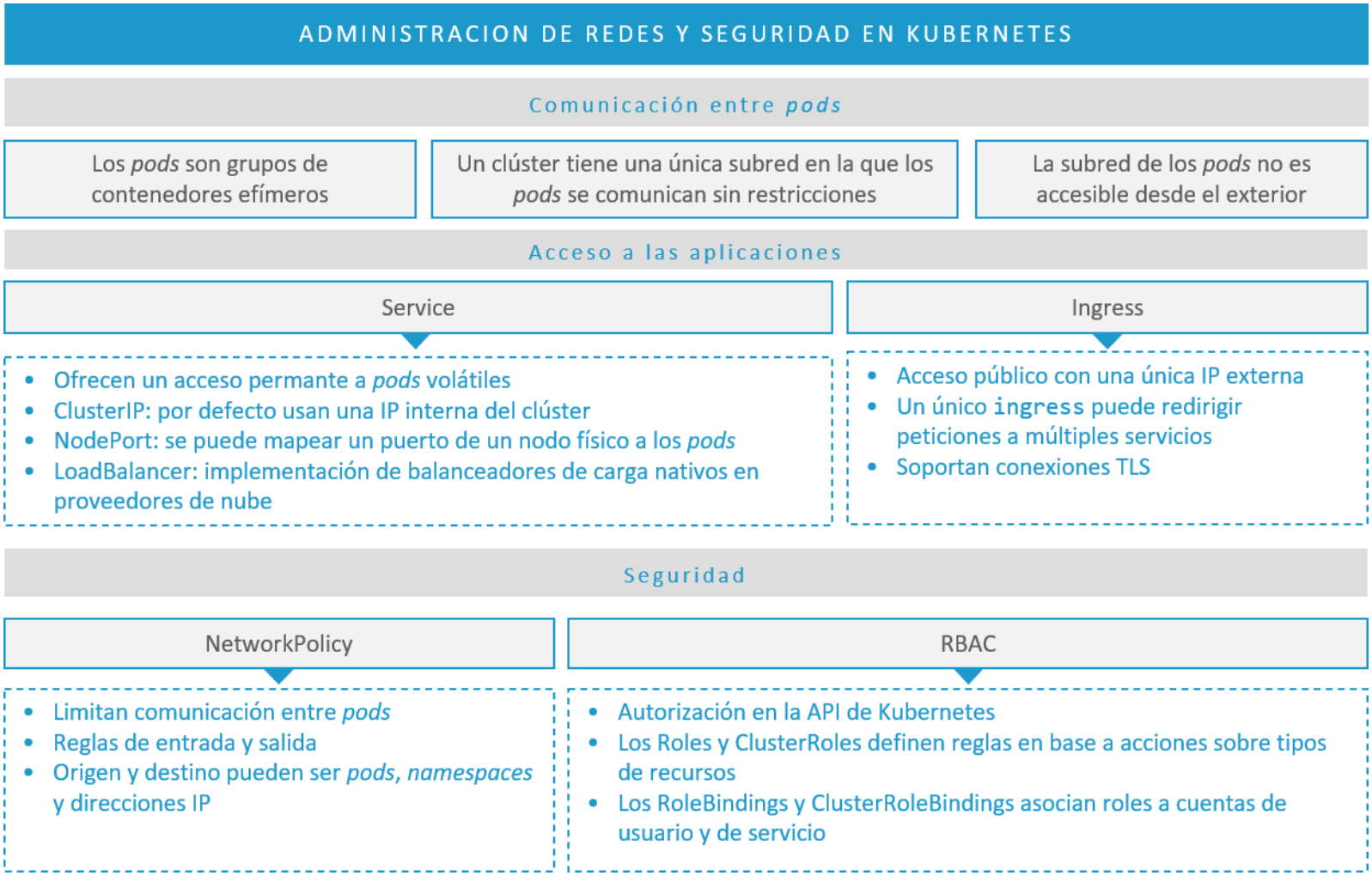
SecDevOps y Administración de Redes para Cloud

Redes y seguridad en Kubernetes

Índice

Esquema



Ideas clave

8.1. Introducción y objetivos

Kubernetes ofrece funcionalidades de **automatización** y **mantenimiento** de aplicaciones con un **paradigma** sustancialmente diferente al tradicional. La virtualización ha adaptado las nociones tradicionales, pero ha mantenido los mismos conceptos. Una **máquina virtual** se despliega más rápido que un servidor físico, pero, básicamente, es el mismo concepto, con sus ventajas y sus desventajas.

Kubernetes hace uso de la **tecnología de contenedores** con un modelo totalmente diferente. La forma de **diseñar aplicaciones** es completamente nueva, y lo mismo ocurre con la administración de las redes.

Este tema presentará los **conceptos esenciales** de Kubernetes, para poder explicar, en detalle, los **elementos de red y de seguridad.** Los **objetivos** que se pretenden conseguir en este tema son:

- Conocer en detalle el funcionamiento de la red de Kubernetes y los objetos implicados.
- Practicar con los recursos en formato YAML.
- Conocer los elementos básicos del control de autorización en Kubernetes.

8.2. Introducción

Kubernetes es un sistema que **automatiza el despliegue,** el **escalado** y la **administración** de aplicaciones diseñadas como contenedores. Un clúster de Kubernetes está formado por **múltiples nodos** (que pueden ser físicos o virtuales) en

los que se despliegan **contenedores** que ejecutan los procesos de las aplicaciones. La comunicación de red y el ciclo de vida de estos contendores se gestionan con tipos de recursos nativos de Kubernetes, como los *pods,* servicios y políticas de red.

Plantillas YAML

Es necesario explicar un concepto más para poder entender los ejemplos que se exponen a lo largo del tema. Los **objetos** de Kubernetes se pueden crear con la herramienta de línea de comandos `kubectl`, indicando cada uno de los **parámetros** del objeto en cuestión, o se pueden definir en una **plantilla** en formato **YAML** y ejecutar el comando `kubectl apply -f plantilla.yaml`.

Una plantilla puede tener una o más **definiciones de objetos,** y todos se crearán al ejecutar la plantilla. Si la plantilla se modifica y se aplica de nuevo, los recursos se actualizarán, y si se añade un objeto a la plantilla, este se creará. Por ejemplo, un *pod* puede definirse con el siguiente código:

```
apiVersion: v1
kind: Pod
metadata:
  name: bastion
  labels:
    app: admin
spec:
  containers:
  - image: debian:buster-slim
    name: bastion-pod
    args: ['sleep', 'infinity']
```

Este bloque define un recurso de tipo `Pod` con nombre `bastion`, le asigna una **etiqueta** con clave `app` y valor `admin` (todos los objetos tienen nombre y pueden tener etiquetas) y luego especifica que el `Pod` usará la imagen `debian:buster-slim` (disponible en Docker Hub). Ejemplos como este se usarán a lo largo de la explicación para ilustrar los diferentes conceptos.

Pods

Un ***pod*** es un **grupo de contenedores** localizados en el mismo nodo y representa el componente básico de Kubernetes. En lugar de desplegar contenedores individualmente, siempre es obligatorio **desplegar** los contenedores en *pods*. Esto no significa que un *pod* deba incluir más de un contenedor, y, de hecho, es habitual que los *pods* contengan solo un contenedor.

La idea principal de los *pods* es que, cuando un *pod* contiene **múltiples contenedores,** todos ellos se ejecutan siempre en un solo nodo. Los contenedores comparten el mismo nombre de ***host*** y la misma **tarjeta de red.** No comparten el sistema de ficheros, pero pueden usar un mismo directorio mediante un volumen de Kubernetes.

El **concepto relevante** en esta asignatura es que, debido a que los contenedores en un *pod* comparten el mismo interfaz de red, todos ellos comparten igual **dirección IP** y **rango de puertos.** Esto significa que los procesos que se ejecutan en contenedores de un mismo *pod* deben evitar **arrancar** el **proceso** en el mismo número de puerto, o provocarán un conflicto.

Además, todos los contenedores en un *pod* comparten la **interfaz de red** de **bucle invertido** (el *loopback*), por lo que un contenedor puede comunicarse con otros contenedores en el mismo *pod* a través de `localhost`.

Comunicación entre *pods*

Todos los *pods* en un clúster comparten el mismo **espacio de direcciones.** En la Figura 1, los *pods* de los nodos 1 y 2 pueden comunicarse directamente porque están en la misma **subred.** Esto significa que cualquier *pod* puede acceder a cualquier otro, usando su dirección IP. No hay **NAT,** ni **rúters,** ni múltiples subredes: cuando dos *pods* intercambian paquetes de red entre sí, cada uno verá la dirección IP real del otro como la IP de origen en el paquete.

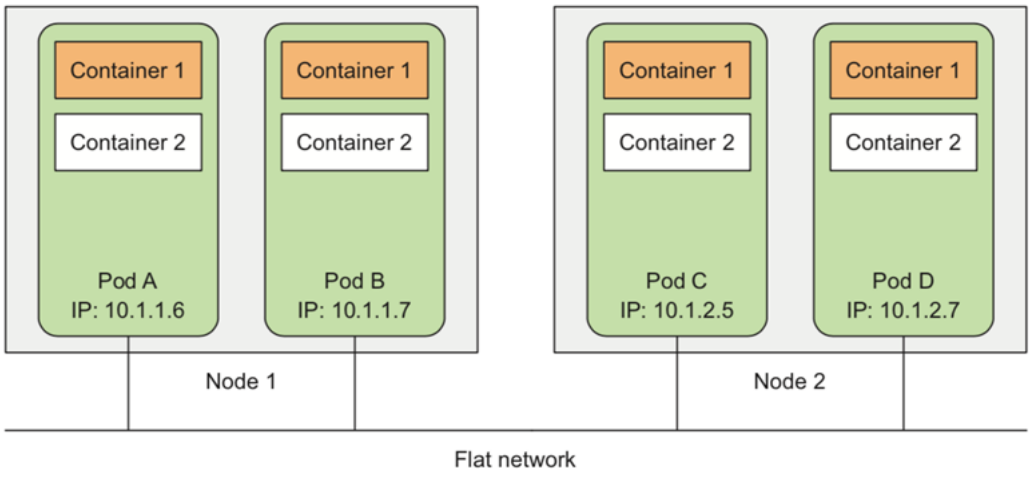


Figura 1. Red de *pods* en un clúster de 2 nodos. Fuente: Lukša, 2018.

En consecuencia, la **comunicación** entre *pods* es muy simple. No importa si dos *pods* se ejecutan en un solo nodo o en nodos diferentes; en ambos casos, los contenedores pueden comunicarse entre sí a través de la **red,** al igual que los ordenadores en una red de área local, independientemente de la **topología de red** que interconecta los nodos del clúster.

Al igual que un ordenador en una LAN, cada *pod* obtiene su propia dirección IP, y es accesible desde todos los demás *pods* a través de esta red establecida específicamente para ellos. Esto, generalmente, se logra a través de una **red adicional** definida por software por encima de la **red subyacente.**

8.3. Objetos de red en Kubernetes

Se usará como ejemplo una aplicación WordPress en una arquitectura de dos niveles, como la de la Figura 2: un *pod* con una base de datos **MySQL** y un `Deployment` con el **servidor Apache** y el código de **WordPress.** El sistema será accesible con un `Ingress` y los *pods* estarán protegidos con varias `NetworkPolicy`.

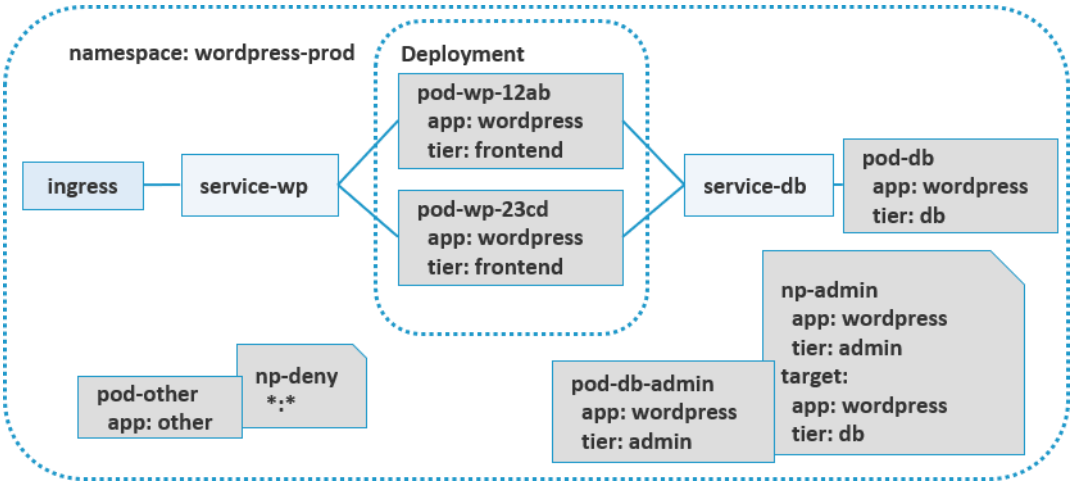


Figura 2. Arquitectura de la aplicación. Fuente: elaboración propia.

La **arquitectura** del ejemplo ha sido adaptada de una serie de **tutoriales.** Se han modificado y simplificado los siguientes elementos:

- El despliegue de MySQL se ha convertido en un único *pod*.
- Se han eliminado los volúmenes persistentes.
- La contraseña se configura de manera estática.
- Se han añadido las políticas de red.
- Hay un *pod* bastión nuevo, que servirá para que los administradores de la base de datos se puedan conectar a esta. Se ha introducido, sobre todo, para demostrar el uso de políticas de red.

En la sección A fondo puedes encontrar varios tutoriales para desplegar aplicaciones sencillas.

En el siguiente vídeo, titulado **«Redes y seguridad en Kubernetes»**, puedes seguir paso a paso el ejemplo que estamos trabajando.

Accede al vídeo

En esta arquitectura hay dos *pods* individuales: el `pod bastion`, cuyo código se ha incluido, y el *pod* de la base de datos MySQL, cuya definición se muestra a continuación. Usa una **imagen** también disponible en Docker Hub y expone el puerto 3306.

```
apiVersion: v1
kind: Pod
metadata:
  name: mysql-pod
  labels:
    app: wordpress
    tier: db
spec:
  containers:
  - image: mysql:5.6
    name: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: mysql-pass
    ports:
    - containerPort: 3306
      name: mysql
```

Para reproducir el ejemplo, no hay más que **copiar las definiciones** de todos los elementos en un fichero YAML y aplicar la configuración con `kubectl apply -f plantilla.yaml`.

Despliegues

Los `Deployments` o despliegues de Kubernetes definen **grupos de *pods*** idénticos. Soportan **autoescalado** y **actualizaciones en ciclo** (o *rolling updates*). A los efectos de este ejemplo, se usan para **desplegar** dos *pods* idénticos y **demostrar** cómo un servicio puede balancear el tráfico entre dos *pods,* sin haber especificado *a priori* el número de *pods*.

El *pod* de la aplicación web se puede crear con la siguiente definición:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  replicas: 2
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
    spec:
      containers:
      - image: wordpress:4.8-apache
        name: wordpress-pod
        env:
        - name: WORDPRESS_DB_HOST
          value: mysql
        - name: WORDPRESS_DB_PASSWORD
          value: mysql-pass
        ports:
        - containerPort: 80
          name: wordpress
```

Servicios

Un **servicio,** o *service*, de Kubernetes es un recurso que crea un único **punto de entrada** permanente para un *pod,* o para un grupo de *pods,* que proporcionan el mismo servicio.

Cada servicio tiene una **dirección IP** y un puerto que nunca cambian durante la vida del servicio. Los clientes pueden abrir conexiones a esa IP y puerto, y esas conexiones se **enrutan** a uno de los *pods* que respaldan ese servicio. De esta manera, los clientes de un servicio no necesitan conocer la **ubicación,** ni las **direcciones,** de los *pods* individuales que brindan el servicio, lo que facilita su mantenimiento.

En cada **operación de arranque o parada** de un *pod,* el servicio actualiza los registros para determinar el conjunto de *pods* más actualizados a los que redirigir el tráfico. Estas operaciones de parada y arranque pueden deberse a una **tarea iniciada** por un administrador (por ejemplo, una actualización de la imagen del *pod*) o a un evento interno, por ejemplo, una **caída del proceso,** que Kubernetes puede detectar para reiniciar el *pod* automáticamente, o una actualización de la imagen mediante un *rolling update* iniciado por el despliegue.

El siguiente ejemplo describe gráficamente la razón de ser de los servicios. En una **arquitectura,** con una **aplicación web** y una de **base de datos,** puede haber varios *pods* que actúen como servidor web y un único *pod* de base datos (podría haber más, pero este ejemplo está simplificado para hacer más sencilla la explicación). Hay **dos problemas** a solucionar:

- Los **clientes externos** deben ser capaces de conectarse a los *pods* web sin importar el número de *pods* que cumplen esa función (un despliegue puede crear un número arbitrario de *pods* idénticos, que, además, pueden aparecer y desparecer en eventos de autoescalado).

- Los contenedores de la aplicación web deben poder conectarse a la base de datos. Debido a que la base de datos se ejecuta dentro de un *pod*, este puede sufrir **caídas** o **tareas de mantenimiento** que cambien su dirección IP. En estas situaciones no es deseable reconfigurar los *pods* web una y otra vez.

El primer problema se soluciona creando un **servicio** para los *pods* web y configurándolo para que sea accesible desde fuera del clúster. En este caso, se expone una **única dirección IP** constante, a través de la cual, los clientes externos pueden conectarse a los *pods*.

El segundo se resuelve de la misma manera, salvo que el servicio del *pod* de la base de datos no necesita una IP externa, solo necesita una IP fija para que la dirección del servicio no cambie, incluso si cambia la **dirección IP** del *pod*. Los elementos de este ejemplo se muestran en la Figura 3. Los **componentes** de *frontend* serían los *pods* de la aplicación web del ejemplo y el de *backend,* el pod de la base de datos.

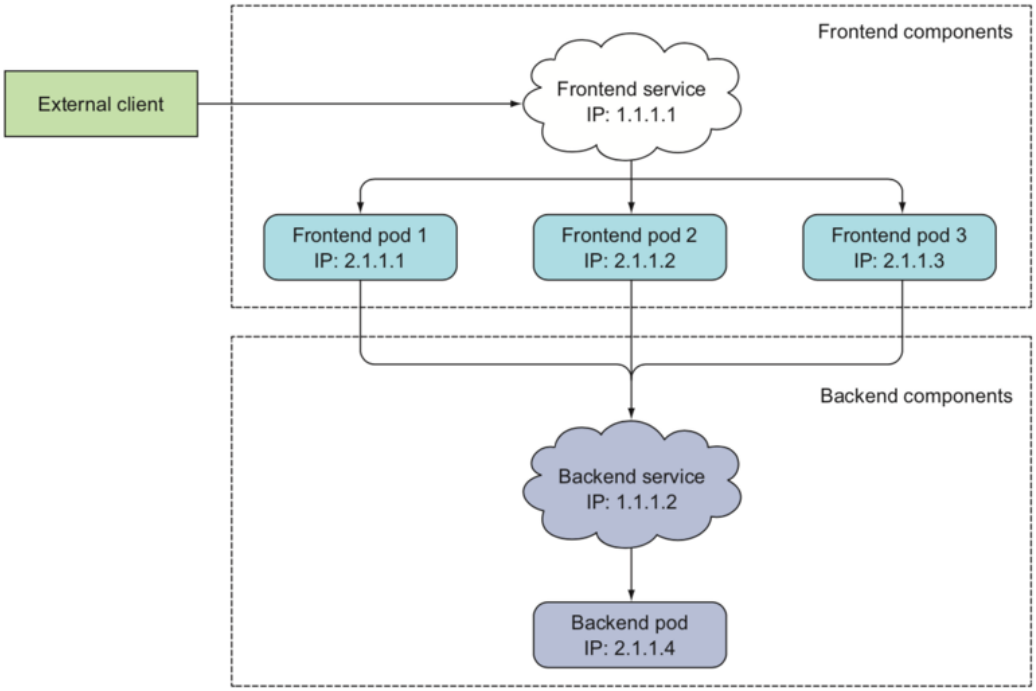


Figura 3. Servicios y *pods* en una arquitectura de dos capas. Fuente: Lukša, 2018.

Por defecto, los servicios están configurados en modo **ClusterIP.** Estos servicios reciben una IP del clúster, similar a la que tienen los *pods*, por lo que solo es accesible por otros **recursos internos.**

Este modo sería el necesario para el servicio de la base de datos del ejemplo. Sin embargo, esta configuración no servirá para el **servicio web** (podría tener sentido en un servicio interno que solo fueran a consumir otros *pods* del clúster, pero no para exponerlo públicamente). Los servicios de **WordPress** y **MySQL** del ejemplo se definirían con el siguiente código.

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
    tier: frontend
---
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: wordpress
spec:
  ports:
    - port: 3306
  selector:
    app: wordpress
    tier: db
  clusterIP: None
```

Ambos servicios son **similares,** excepto en el puerto que exponen y los *pods* que seleccionan. La selección de *pod* se hace con **etiquetas:** el servicio `wordpress` tiene la propiedad `selector`, que indica dos **etiquetas:** `app=wordpress` y `tier=frontend`. La definición de los *pods* de WordPress incluía esas dos etiquetas.

La misma **técnica** se usa con el servicio y el *pod* de MySQL, pero con etiquetas diferentes.

Kubernetes ofrece varios métodos para exponer servicios al exterior: mediante un servicio de tipo `NodePort`, un servicio de tipo `LoadBalancer` o mediante un `Ingress`, que es otro tipo de recurso que trabaja junto al servicio.

Servicios NodePort

El primer método para exponer un conjunto de *pods* a clientes externos es **crear un servicio** y establecer su **tipo** como `NodePort`. Al crear un servicio `NodePort`, Kubernetes **reserva** un **puerto** en todos sus nodos, el mismo número de puerto en todos ellos, y **reenvía** las conexiones entrantes en esos puertos a los *pods* que forman parte del servicio.

Esta **reserva del puerto** es similar a un servicio normal de tipo `ClusterIP`, pero un servicio `NodePort` es accesible no solo a través de la IP interna del clúster, sino también a través de la IP de **cualquier nodo.**

Estas IP no pertenecen a la red de los *pods*, sino a la **subred** a la que están conectadas las interfaces de los nodos y, por tanto, son accesible desde fuera del clúster.

Un ejemplo de servicio `NodePort` sería el siguiente (este ejemplo no pertenece al caso práctico y no debe incluirse para replicar la arquitectura):

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: MyApp
  ports:
      port: 80
      targetPort: 80
      nodePort: 30007
```

Servicios con balanceador de carga

Los clústeres de Kubernetes que se **ejecutan** en **proveedores de la nube,** generalmente, admiten la provisión automática de un balanceador de carga desde la infraestructura de la nube. Esto es posible si el clúster se aprovisiona como un **servicio nativo** del proveedor, por ejemplo, mediante el servicio EKS en AWS, AKS en Azure o GKE en Google Compute Cloud .

Si se **configura** un clúster de Kubernetes manualmente a partir de **instancias de cómputo** (EC2, por ejemplo), el servicio de balanceador no tiene por qué estar disponible, a menos que el proveedor ofrezca la posibilidad de instalar los *plugins* de su entorno en un clúster propio.

Si el clúster lo soporta, todo lo que se necesita hacer es establecer el **tipo de servicio** en `LoadBalancer` en lugar de `NodePort`. El **balanceador de carga** tendrá su propia dirección IP única, y de acceso público, y redirigirá todas las conexiones a su servicio.

De este modo, los clientes pueden acceder al servicio a través de la dirección IP del balanceador. Este balanceador será un **recurso** de tipo ELB en AWS, *Azure Load Balancer* en Azure o *Cloud Load Balancing* en GCE .

Si Kubernetes se **ejecuta** en un entorno que no admite los servicios de `LoadBalancer`, el **balanceador** no se aprovisionará y el servicio seguirá comportándose como si fuera de tipo `NodePort`. Esto se debe a que un servicio `LoadBalancer` es una extensión de un servicio `NodePort`.

El siguiente **código** define un servicio de tipo `LoadBalancer` (tampoco pertenece al caso práctico).

```
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
```

Kubernetes Ingress

Los **recursos** `Ingress` varían de un clúster a otro ya que, aunque son un **objeto nativo** de Kubernetes, la implementación se basa en un **controlador** de `Ingress`, que no se instala por defecto en todos los clústeres. Los controladores disponibles suelen estar

basados en algún producto que ya ofrezca las funcionalidades previstas para un `Ingress`, como nginx o envoy, o con un **servicio propio de un proveedor de nube,** como el *Application Load Balancer* de AWS .

Hay **dos razones** principales para usar un objeto `Ingress` en vez de un servicio. La primera es que cada servicio de tipo `LoadBalancer` requiere su propio **balanceador de carga** con su propia **dirección IP pública,** mientras que un `Ingress` solo requiere uno, incluso cuando proporciona **acceso** a más de un servicio. Cuando un cliente envía una petición HTTP al `Ingress`, el **nombre de *host*** y la **ruta** en la solicitud determinan a qué servicio se reenvía el tráfico.

La otra razón es que los `Ingress` ofrecen más **funcionalidades** que un servicio: afinidad de sesión basada en *cookie*, terminación de conexiones seguras con certificados TLS, etc.

El funcionamiento de un `Ingress` es el siguiente:

- El cliente realiza una búsqueda DNS del equipo al que quiere enviar la petición. El nombre estará registrado con la IP pública del `Ingress`.
- El cliente envía una petición HTTP al controlador `Ingress` y especifica el encabezado `Host` con el nombre que acaba de resolver mediante DNS.
- A partir de ese encabezado, el controlador determina a qué servicio está intentando acceder el cliente.
- El controlador redirige la petición a la IP de uno de los *pods* asociados al servicio.

Los objetos `Ingress` pueden tener **múltiples asociaciones** entre nombres de *host* y servicios, consumiendo una única IP pública. Esta **característica** no es exclusiva de Kubernetes: los productos en los que se basan los controladores la han ofrecido desde hace tiempo.

El objeto `Ingress` del caso práctico se definiría con el siguiente código. Las **peticiones** con el nombre de host `demo-site.local`, en la cabecera `Host`, se dirigirán al servicio wordpress, a la ruta raíz.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: wordpress
spec:
  rules:
  - host: demo-site.local
    http:
      paths:
        - path: /
          backend:
            serviceName: wordpress
            servicePort: 80
```

Seguridad de red

Los *pods* reciben una IP de una **única subred** en todo el clúster, al margen del número de nodos que lo compongan. Los *pods* pueden **comunicarse** entre sí sin restricciones. Esto puede no ser relevante si el clúster está dedicado a una única aplicación y la seguridad de red está correctamente aplicada en los servicios expuestos al exterior.

En clústeres compartidos (o *multi-tenant*), esto puede no ser deseable. Para aplicar **reglas de seguridad,** en la red interna, hay que usar recursos de tipo `NetworkPolicy`.

Estos objetos pueden no estar disponibles en función del ***plugin* de red del clúster.** Al igual que el controlador de `Ingress`, el *plugin* de red es configurable y puede no soportar una NetworkPolicy.

El concepto de *multi-tenant* se debería traducir como **multi-inquilino** o, simplemente, **compartido.** Sin embargo, multi-inquilino apenas aparece en la documentación, incluso cuando está traducida al español, y compartido puede dar a entender el concepto contrario, en el que los objetos se comparten entre inquilinos, en vez de los **recursos subyacentes.** Por tanto, el resto del tema seguirá usando ***multi-tenant*** donde sea necesario.

Las `NetworkPolicy` se **aplican** a nivel de *pod* y especifican qué orígenes pueden acceder a los *pods,* o a qué destinos se puede acceder desde los *pods*. Esto se configura a través de las **reglas de entrada y salida,** respectivamente. Ambos tipos de reglas se puede aplicar a un conjunto de *pods*, a todos los *pods* de un espacio de nombres (o *namespace*) o a un bloque de IP utilizando la notación CIDR.

Los *namespaces* permiten aislar objetos en grupos distintos, lo que permite operar solo en aquellos que pertenecen al espacio de nombres especificado, pero no proporcionan ningún tipo de aislamiento de los objetos en ejecución.

Por ejemplo, **diferentes usuarios** pueden tener permiso para desplegar *pods* en espacios de nombres diferentes (convirtiendo el clúster en un entorno *multi-tenant*), pero esos *pods* no están aislados unos de otros y pueden **comunicarse.** Para aplicar medidas de seguridad de red entre *pods* es necesario usar `NetworkPolicies`.

Un **primer paso** puede ser **prohibir el acceso** a todos los *pods* de un *namespace.* Si la `NetworkPolicy` no define ninguna regla, ni selecciona ningún grupo de *pods,* por defecto prohibirá todo el tráfico dirigido a los *pods* donde se ha desplegado esa política.

Esta política se definiría con el siguiente código:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Para permitir el acceso desde un *pod* cliente a un *pod* servidor, es necesario indicar explícitamente qué *pods* de **origen** pueden **conectarse** a qué *pods* de **destino** y en qué puerto, mediante una regla de entrada (o *ingress*; no confundir con el objeto `Ingress` discutido con anterioridad). La **política de seguridad** se aplicará, incluso aunque los *pods* cliente se conecten a través de un servicio. Para el caso práctico se define una política como la siguiente, que permite el tráfico de los *pod* con etiqueta `app=admin` (es decir, el bastión) al puerto 3306 del **pod** de MySQL, identificado con las etiquetas `app=wordpress` y `tier=db`.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: admin-netpolicy
spec:
  podSelector:
    matchLabels:
      app: wordpress
      tier: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: admin
    ports:
    - port: 3306
```

También es posible **aislar** los *pods* de cada *namespace* para evitar el **tráfico** entre un *namespace* y otro. Esto, junto con el **acceso basado en roles,** permite que Kubernetes se comporte como un entorno *multi-tenant* de verdad.

En este caso, la regla de entrada puede seleccionar un conjunto de *namespaces,* en vez de un conjunto de *pods.* Una política como esta se puede **implementar** con el siguiente **código** (este recurso no forma parte del caso práctico).

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-netpol
spec:
  podSelector:
    matchLabels:
      app: MyApp
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          tenant: tenantA
    ports:
    - port: 80
```

Las **reglas** también pueden **seleccionar** direcciones IP de origen y destino, especificando las **subredes** con notación CIDR, por ejemplo, `192.168.1.0/24`.

Las políticas de red pueden combinar la forma de seleccionar orígenes y destinos en una misma regla.

La política anterior podría **especificar** el **origen** con un bloque de IP con la siguiente sintaxis:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-netpol
spec:
  podSelector:
    matchLabels:
      app: MyApp
  ingress:
  - from:
    - ipBlock:
        cidr: 192.168.1.0/24
    ports:
    - port: 80
```

En los casos anteriores se ha hablado en exclusiva de reglas de entrada, pero una `NetworkPolicy` puede definir también **reglas de salida** (o *egress*). Como la palabra indica, la regla permitirá el tráfico de salida con destino en el objeto, objetos o rango de IP seleccionado.

Ejecución del caso práctico

Para reproducir el caso práctico en un entorno de laboratorio, no hay más que co**piar los bloques de código** en un único **fichero YAML** y ejecutarlo con `kubectl apply`. El vídeo mencionado al principio de este capítulo demuestra el **despliegue** en un **clúster de prueba.**

8.4. Control de acceso basado en roles

Una vez **analizados** los **recursos** de Kubernetes para administrar y proteger la capa de red, ahora le toca el turno a la **protección del acceso** a Kubernetes como tal. Revisaremos el **flujo habitual de autenticación y autorización** en Kubernetes y los **objetos básicos** del módulo de control de acceso basado en roles o RBAC (*role-base access control*).

API de Kubernetes

Cualquier interacción con Kubernetes se lleva a cabo en la API mediante **peticiones REST,** ya sea desde la línea de comandos con kubectl, en el panel web (o *dashboard*) o desde una aplicación ejecutándose en un *pod*. Cada llamada atraviesa las fases de **autenticación, autorización** y **control de admisión.**

En un **clúster típico,** la API escucha en el puerto 6443 mediante HTTPS. Una vez que se establece la conexión TLS, la petición HTTP se mueve al paso de autenticación. Kubernetes no realiza la autenticación por sí mismo, sino que lo delega a **módulos externos.**

Algunos de los módulos disponibles (Kubernetes, s. f.) son **certificados** de cliente, contraseña, tokens simples y tokens JWT (utilizados para cuentas de servicio). Se pueden especificar **varios módulos de autenticación,** en cuyo caso, cada uno se prueba en secuencia, hasta que uno de ellos tenga éxito. Si la solicitud no se puede autenticar, se rechaza con el código de estado HTTP 401. De lo contrario, el usuario se autentica como un nombre de usuario específico, y el nombre de usuario está disponible para los pasos posteriores.

Si bien Kubernetes usa nombres de usuario para las decisiones de control de acceso, no almacena nombres de cuenta ni ninguna otra información sobre los usuarios.

El paso de autorización también se soporta con **módulos externos,** aunque el módulo que se analizará en detalle, **RBAC,** está incluido y habilitado por defecto desde la **versión 1.6.** Al igual que con los módulos de autenticación, se pueden **encadenar** varios módulos de autorización y se procesarán hasta que uno autorice la **petición REST.** Si todos los módulos rechazan la petición, esta se termina con el código de estado HTTP 403.

Si las fases de autenticación y autorización se completan con éxito, la petición pasa a la fase de **control de admisión.** Un controlador de admisión es un **fragmento de código** que intercepta las solicitudes al servidor API de Kubernetes antes de persistir el objeto en el clúster. Los controladores de admisión pueden ser de **validación,** de **mutación** o **híbridos.** Los controladores de mutación pueden **modificar** los objetos que admiten; los controladores de validación solo pueden **leer** el objeto y rechazar la petición si el objeto no es válido.

Módulo RBAC

Desde **Kubernetes 1.6** en adelante, las **políticas RBAC** están habilitadas por defecto. Las políticas de RBAC permiten especificar qué tipos de acciones están permitidas según el usuario y su función en la organización. Algunas comprobaciones posibles con RBAC son:

▪ Protección del clúster permitiendo operaciones privilegiadas (por ejemplo, acceso a secretos) solo a usuarios administradores.
▪ Forzar la autenticación de usuarios en todas las peticiones a la API.
▪ Limitar la creación de recursos (como *pods,* volúmenes, despliegues) a *namespaces* específicos, incluso con cuotas para garantizar que el uso de recursos sea limitado y esté bajo control.
▪ Prohibir la lectura de objetos fuera de los *namespaces* permitidos. Esto permite aislar recursos dentro de la organización e implementar un clúster *multi-tenant.*

Tabla 1: Comprobaciones posibles con RBAC. Fuente: elaboración propia.

RBAC usa los siguientes elementos en su funcionamiento:

- **Recursos:** este elemento se refiere a los tipos de objetos que se han comentado en el tema (y cualquier otro disponible), como *pod,* despliegues, `NetworkPolicy`, `Ingress`, etc.

- **Operaciones:** las acciones sobre los recursos son las habituales de un sistema de autorización: crear, leer, borrar, listar, actualizar, etc. Estas acciones están mapeadas a los verbos típicos de HTTP: `POST`, `GET`, `DELETE`, `PUT`, etc.

- **Reglas:** una regla es un conjunto de operaciones (verbos) que se pueden llevar a cabo en uno o varios tipos de recursos.

- **Roles y roles de clúster:** ambos consisten en reglas, pero mientras que las reglas de un rol son aplicables a un solo *namespace*, las reglas de un rol de clúster abarcan todo el clúster, por lo que son aplicables a más de un espacio de nombres. Ambos elementos tienen un tipo de recurso específico en Kubernetes: `Role` y `ClusterRole`.

- **Sujetos:** corresponden a la entidad que intenta una operación en el clúster. Hay **tres tipos de sujetos** y dependen del módulo de autenticación:

  - **Cuentas de usuario:** son globales, están destinadas a humanos o procesos que viven fuera del clúster y no se almacenan en ningún objeto contenido en el clúster.
  - **Cuentas de servicio:** están circunscritas a un *namespace* y se destinan a llamadas de API desde aplicaciones que se ejecutan en *pods* del propio clúster.
  - **Grupos:** se utilizan para referirse a varias cuentas.

- **Asociaciones de roles:** además de los objetos de `Role` y `ClusterRole`, RBAC define dos objetos más: `RoleBinding` y `ClusterRoleBinding`. Se usan para asignar un rol a un sujeto.

Los siguientes **ejemplos** hacen uso de los **recursos** explicados anteriormente.

Rol para administrador de despliegues.

El primero de ellos define un rol para un administrador de despliegues. Todas las acciones están habilitadas sobre los recursos `Pod`, `Deployment` y `ReplicaSet` (un objeto asociado a un despliegue). Un usuario con este rol no podría, sin embargo, cambiar las políticas de red ni crear objetos `Ingress` o `Service`, por lo que no podría habilitar el acceso a la aplicación desde el exterior.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
 namespace: ns
 name: deployment-manager
rules:
- apiGroups: ["", "extensions", "apps"]
 resources: ["deployments", "replicasets", "pods"]
 verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

Rol para leer y listar recursos

Este otro rol permite leer y listar cualquier tipo de recurso, pero no puede aplicar cambios:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: ns
  name: read-all
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["get", "list"]
```

Roles asociados a un `RoleBinding`

Estos roles no se aplican por defecto a ninguna cuenta, a menos que se asocien con un `RoleBinding`. La siguiente definición asocia el rol `deployment-manager` a la cuenta de usuario `employee`.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: deployment-manager-binding
  namespace: ns
subjects:
- kind: User
  name: employee
  apiGroup: ""
roleRef:
  kind: Role
  name: deployment-manager
  apiGroup: ""
```

Roles de clúster con RBAC

Los clústeres con RBAC habilitado tienen roles de clúster definidos por defecto. Estos roles se pueden asignar con un `ClusterRoleBinding` directamente. Por ejemplo, para dar permisos de administrador a una cuenta de servicio, habría que definir una asignación como la siguiente:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: plugin-admin-binding
subjects:
- kind: ServiceAccount
  name: custom-plugin
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: ""
```

El rol `cluster-admin` es uno de los `ClusterRoles` por defecto. Además, se aplica en el `namespace kube-system`, que es el *namespace* que contiene los **pods** de sistema que gestionan el funcionamiento del clúster.

8.5. Referencias bibliográficas

Kubernetes. (S. f.). *Authenticating.* https://kubernetes.io/docs/reference/access-authn-authz/authentication/

Lukša, M. (2018). *Kubernetes in Action.* Manning Publications.

Vaswani, V. (S. f.). *Configure RBAC in your Kubernetes Cluster.* Bitnami. https://docs.bitnami.com/tutorials/configure-rbac-in-your-kubernetes-cluster/

A fondo

Ejemplos de despliegue en Kubernetes

Github. (S. f.). Kubernets / examples. https://github.com/kubernetes/examples

La documentación oficial de Kubernetes incluye varios tutoriales para desplegar aplicaciones sencillas y mantienen el código en este repositorio. El caso práctico de este tema se basa en uno de los ejemplos. Se recomienda una revisión en detalle de los tutoriales y, a poder ser, la ejecución de estos en un entorno de laboratorio.

RBAC en Kubernetes

School of Devops. (2018, julio 17). *Role Based Access Control (RBAC) with Kubernetes* [Vídeo]. YouTube. https://www.youtube.com/watch?v=BLktpM--0jA

Este vídeo ofrece una explicación de los conceptos básico del control acceso basado en roles aplicado a Kubernetes. Es una explicación a alto nivel, sin ejemplos prácticos, pero es muy didáctica gracias a los diagramas del profesor.

Test

1. ¿En cuántas subredes se dividen los *pods* de un clúster de Kubernetes?

A. En tantas como *namespaces*.

B. Una única, común a todos los *pods*.

C. Es configurable por el administrador.

D. Cada nodo define una red local para los *pods* que se ejecutan en él.

2. ¿Por defecto, qué restricciones de tráfico hay en la red de los *pods*?

A. Ninguna.

B. Los *pods* solo pueden conectarse a *pods* del mismo *namespace*.

C. Los *pods* solo pueden conectarse a servicios, no a *pods*.

D. Por defecto no hay comunicación, hay que definirla expresamente.

3. Relaciona el recurso con su descripción.

Role	1
NetworkPolicy	2
Pod	3
RoleBinding	4

A	Asocia un Role con una cuenta de usuario o de servicio.
B	Unidad básica de ejecución en Kubernetes
C	Permite o deniega tráfico a, o desde, *pods*
D	Define unas acciones permitidas sobre unos recursos

4. ¿Por qué es necesario crear un servicio para exponer los puertos de un *pod*?

A. No es necesario, se puede trabajar sin un servicio.

B. Por diseño, al crear un *pod,* se crea automáticamente un servicio.

C. Los *pods* son efímeros por definición, por lo que una aplicación que quiera conectarse a un *pod* necesita disponer de un mecanismo que mantenga un nombre fijo para poder acceder, incluso aunque el *pod* desaparezca.

D. Ninguna de las anteriores.

5. ¿Qué cabecera estándar HTTP usa un objeto `Ingress` para redirigir las peticiones?

A. `Location`.

B. `Path`.

C. `Site`.

D. `Host`.

6. ¿En qué formato se definen las plantillas de Kubernetes?

A. Python.

B. YAML.

C. HTML.

D. XML.

7. ¿En qué casos de los siguientes un servicio con balanceador de carga desplegará un balanceador de carga nativo?

A. EKS.

B. AKS.

C. GKE.

D. Todos los anteriores.

8. ¿Cómo se pueden identificar destinos en una `NetworkPolicy`?

A. Con etiquetas de *pods.*

B. Con un nombre de *namespace*.

C. Con una subred IP.

D. Todos los anteriores.

9. ¿Qué tipos de reglas aceptan las `NetworkPolicy`?

A. `Ingress`.

B. `Egress`.

C. Todas las anteriores.

D. Ninguna de las anteriores.

10. ¿Qué diferencia hay entre un `Role` y un `ClusterRole`?

A. Ambos definen acciones permitidas sobre tipos de recursos, pero los `Roles` están restringidos a un *namespace* y los `ClusterRoles` aplican a todo el clúster completo.

B. Ambos definen acciones permitidas sobre tipos de recursos, pero los Roles se aplican a usuarios y los `ClusterRoles` a cuentas de servicios.

C. Los `Roles` están incluidos en el módulo RBAC, mientras que los `ClusterRoles` son nativos de Kubernetes.

D. Ninguna de las anteriores.