Estudio y análisis de proyectos sobre redes para la interconexión de contenedores Docker

Departamento de Sistemas Informáticos y Computación

Master Universitario en Computación Paralela y Distribuida

Universitat Politècnica de València

Curso Académico 2017 – 18

Autor

Ke Zhang

Director

Ignacio Blanquer Espert

ÍNDICE

Índice de figura

Índice de tabla

1 Abstract

Las tecnologías de contenedores están actualmente en el centro de la nueva ola, que involucra la construcción, empaquetamiento y entrega de aplicaciones. Las tecnologías de contenedores tienen impacto en todos los aspectos de la tecnología informática, desde el desarrollo de aplicaciones hasta la implementación de las aplicaciones y la gestión de centros de datos de gran tamaño especialmente en cuanto la elasticidad vertical y horizontal. Entre las tecnologías de contenedores destaca Docker [1].

Cuando se comienza a usar el contenedor de Docker en un entorno de producción real para implementar la aplicación, es posible que se tengan que utilizar varios contenedores para implementar una aplicación compleja con varios servicios, en la que cada contenedor implementa un servicio específico. Si utilizamos una infraestructura con múltiples hosts, normalmente no podremos saber de antemano en qué host se creará cada contenedor. Si queremos que los contenedores creados en estos hosts pueden comunicarse entre sí se deberá definir un modelo de comunicaciones adicional. En este trabajo fin de máster se realiza un análisis de los un esquemas de comunicación entre host en la plataforma de contenedores Docker, incluyendo **Docker Networks** [1]**, Calico** [2]**, weave** [3]**, flannel** [4]**, Kubernetes** [5]**, Romana** [6]**, Mesos** [7], y se presentan los principios de cada esquema. Finalmente, se comparan cuatro tipos de modos comunes de red de Calico, Weave, Flannel y overlay (Docker Networks), que pueden servir de referencia para la selección de un entorno de prueba y producción.

2 Introducción

En sus inicios, Docker comenzó como una simple red de host único donde se ejecutan los contendores. Desafortunadamente, esto impide que las aplicaciones de Docker se extienden a múltiples hosts. Con ese planteamiento, aparecieron algunos proyectos como Calico, Flannel y Weave, y desde noviembre de 2015, Docker también ha comenzado a admitir la red de cobertura de múltiples anfitriones.

2.1 ¿Por qué prestar atención a la red de contenedores?

Desde el nacimiento de la tecnología de contenedores, los temas relativos al almacenamiento y la red han sido objeto de trabajo intenso. Los contenedores necesitan acceso vía red y las redes físicas son muy diferentes a las redes virtuales. Se ha trabajado intensamente en el ámbito virtual, introduciendo conceptos como el router virtual y el switch virtual, que deben revisarse en el ámbito de los contenedores. Ya que si bien disponer de cientos de máquinas virtuales en una

misma aplicación es poco frecuente, en el ámbito de las aplicaciones de micro-servicio, cada aplicación puede ser un enorme ecosistema, con muchos servicios y contenedores. La dispersión puede ser mucho más alta que en el ámbito de la máquina virtual tradicional. Entonces, la red virtual tendrá una mayor demanda en el ámbito de los contendores.

En el desarrollo de la tecnología de contenedores hasta la fecha, el trabajo en las redes de comunicaciones ha estado relativamente rezagado. Aunque se han realizado intentos, el nivel de red en Docker no tiene un estándar (únicamente una especificación llamada CNM [8](Container Network Model)), de modo que el desarrollo de red está quedando atrás. Esta situación ha proporcionado un espacio para un gran número de empresas de nueva creación o de colaboraciones de código abierto, desarrollando una variedad de implementaciones de red, con muchas soluciones complejas, para que los clientes puedan elegir.

En conclusión:

- El diseño de la red de la era del contenedor debe ser reconsiderado.
- El desarrollo del micro servicio ha generado una mayor demanda de la red.
- El desarrollo de la tecnología de contenedores en la parte de la red se está quedando atrás.
- Hay variedad de programas y se carece de estándares de la industria.
- La combinación de la tecnología SDN y los contenedores hace que el problema sea más complicado.
- La selección de redes de contenedores afecta a los usuarios y proveedores.

2.2 Características

En comparación con la red de máquinas virtuales, la red de contenedores tiene las siguientes características, así como los desafíos técnicos:

- Las máquinas virtuales tienen un mecanismo de aislamiento perfecto, no hay mucha diferencia de uso en tarjeta de red virtual y tarjeta de hardware, Mientras que el contenedor utiliza el network namespace para proporcionar aislamiento de la red en el kernel, Por lo tanto, para proteger la seguridad del contenedor, el diseño de la red de contenedores requiere una consideración más cuidadosa.

- Por razones de seguridad, en muchos casos el contenedor se implementará dentro de la máquina virtual, este despliegue anidado debe diseñar un nuevo modelo de red.
- La distribución del contenedor es diferente a la de la máquina virtual. Es posible que una máquina virtual que ejecute el negocio necesita dividirse en varios contenedores para ejecutarse. De acuerdo con las diferentes necesidades de los negocios, estos contenedores a veces deben colocarse en un servidor, a veces pueden distribuirse en varios hosts de la red, es probable que el modelo de red para los dos escenarios sea diferente.
- El contenedor no tiene overhead y las aplicaciones se ejecutan más rápido, por eso necesitamos las actualizaciones de políticas de red para mantener la velocidad.
- En el caso de aplicaciones con un gran número de contenedores, las ARP Flooding multi-host causarán una gran cantidad de desperdicio de recursos.
- El ciclo de vida del contenedor es muy corto, se reinicia con mucha frecuencia, la gestión efectiva de la dirección de red (IPAM) es muy crítica.

2.3 La historia del desarrollo de la red de contenedores

El desarrollo de la red de contenedores ha pasado por tres etapas. La primera etapa es acceso directo a los recursos de red. El primer modelo de red de contenedores utiliza la red dentro del host. se quiere exponer el servicio, se necesitará hacer la redirección de puertos. Por ejemplo, un host tiene muchos contenedores Apache, cada Apache lanzar una puertos de 80, entonces, ¿cómo lo hago? Necesito mapear el primer contenedor con el puerto 80 de host, y el segundo con el puerto 81 de host, y el modelo HOST tiene poca capacidad de aislamiento, tanto de tráfico de datos como de seguridad. Falta un modelo intermedio, en el que se utiliza un bridge VXLAN en el que se conectan todos los contendores dentro de un mismo host. Básicamente, ninguna empresas usar con este forma.

Finalmente, se han presentado muchas soluciones, como la implementación de la red basada en IPSec de Rancher, como Flannel basado en la implementación de redes de enrutamiento de tres niveles.

Hoy en día, la red de contenedores apareció dos patrones. Una es arquitectura CNM basada en Docker, El otro es la arquitectura CNI que lideró con Google, Kubernetes y CoreOS.

2.4 Términos técnicos

Escriba algunos términos técnicos antes de comenzar

- **IPAM:** Gestión de direcciones IP. El problema de la gestión de direcciones IP no es exclusiva de los contenedores. Estos servicios existen en las redes tradicionales (el servicio DHCP es un IPAM). El ámbito de los de contenedores, hay dos métodos principales de IPAM: Los segmentos de direcciones IP basados en CIDR o la asignación de una IP a cada contenedor. La asignación de una IP única en un grupo de servidores de contenedores, se resuelve mediante IPAM.
- **Overlay:** Red separada superpuesta sobre los dos o tres niveles existentes. La red usualmente tendrá su propio espacio de direcciones IP independiente, y su propio enrutamiento.
- **IPSesc:** un protocolo de comunicación encriptado de un punto a punto, usualmente usado en el canal de datos en la red Overlay
- **vxLAN:** Es una solución conjunta propuesta por VMware, Cisco, RedHat etc., principalmente para resolver el problema de que la red virtual VLAN es demasiado pequeña (4096). Debido a que cada cliente en la nube pública tiene una VPC diferente, 4096 redes virtuales no es suficiente. vxLAN, puede soportar hasta 16 millones de redes virtuales, suficiente para una nube pública.
- **Bridge:** La conexión de las dos redes entre el equipo de red. En el contexto de este trabajo fin de máster nos referimos a Bridges de Linux, y concretamente Docker0.
- **BGP:** Protocolo de enrutamiento de red autónomo en la red de troncal.

3 Desarrollo

3.1 Docker

3.1.1 Resumen de la red del contenedor Docker

Redes predeterminadas

De forma predeterminada, Docker proporciona tres redes, que son creadas por el proceso de Docker Daemon, correspondientes al "modelo de red" de Docker.

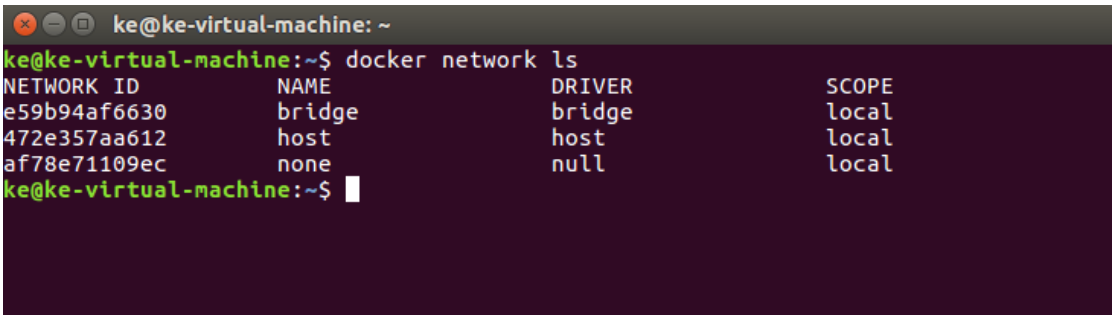


Figura 3-1 Modelo de red de Docker

Cuando se ejecuta un contenedor, se puede usar el indicador **-net** para especificar en qué red desea ejecutar en el contenedor.

```
ke@ke-virtual-machine:~$ docker run --net=host -itd --name=container busybox
fbeeeb4d0860a80f033d3c1a41275359c8aac4b907de7063a7cadf851285c410
```

- **Bridge**: La red bridge representa a la red docker0 que existe en todas las instalaciones de Docker. A menos que use docker run -net =opción, el daemon de Docker conectará el contenedor a esta red de forma predeterminada. Usando el comando **ifconfig** en el host, puede verse que este puente es parte de la pila de red del host. [1]

```
ke@ke-virtual-machine: ~
ke@ke-virtual-machine:~$ ifconfig
docker0   Link encap:以太网  硬件地址 02:42:25:77:f1:98
          inet 地址:172.17.0.1  广播:0.0.0.0  掩码:255.255.0.0
          UP BROADCAST MULTICAST  MTU:1500  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:0 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
          接收字节:0 (0.0 B)  发送字节:0 (0.0 B)

ens33     Link encap:以太网  硬件地址 00:0c:29:14:b3:90
          inet 地址:192.168.141.133  广播:192.168.141.255  掩码:255.255.255.0
          inet6 地址: fe80::b26a:89b9:bd6a:d0b4/64 Scope:Link
```

Figura 3-2 Docker0

- **None**: La red agrega un contenedor a una pila de red específica del contenedor sin conectarlo a ningún interfaz de red. [1]

```
ke@ke-virtual-machine: ~
ke@ke-virtual-machine:~$ docker network inspect none
[
    {
        "Name": "none",
        "Id": "af78e71109ecc21f91092bd7eb9148ab9c57cf9b09edd762223364030f285d9a"
,
        "Scope": "local",
        "Driver": "null",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": []
        },
        "Internal": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Figura 3-3 Docker network inspect none

- **Host**: La red agrega un contenedor a la pila de red del host. Se puede comprobar que la configuración de red en el contenedor es la misma que el host.

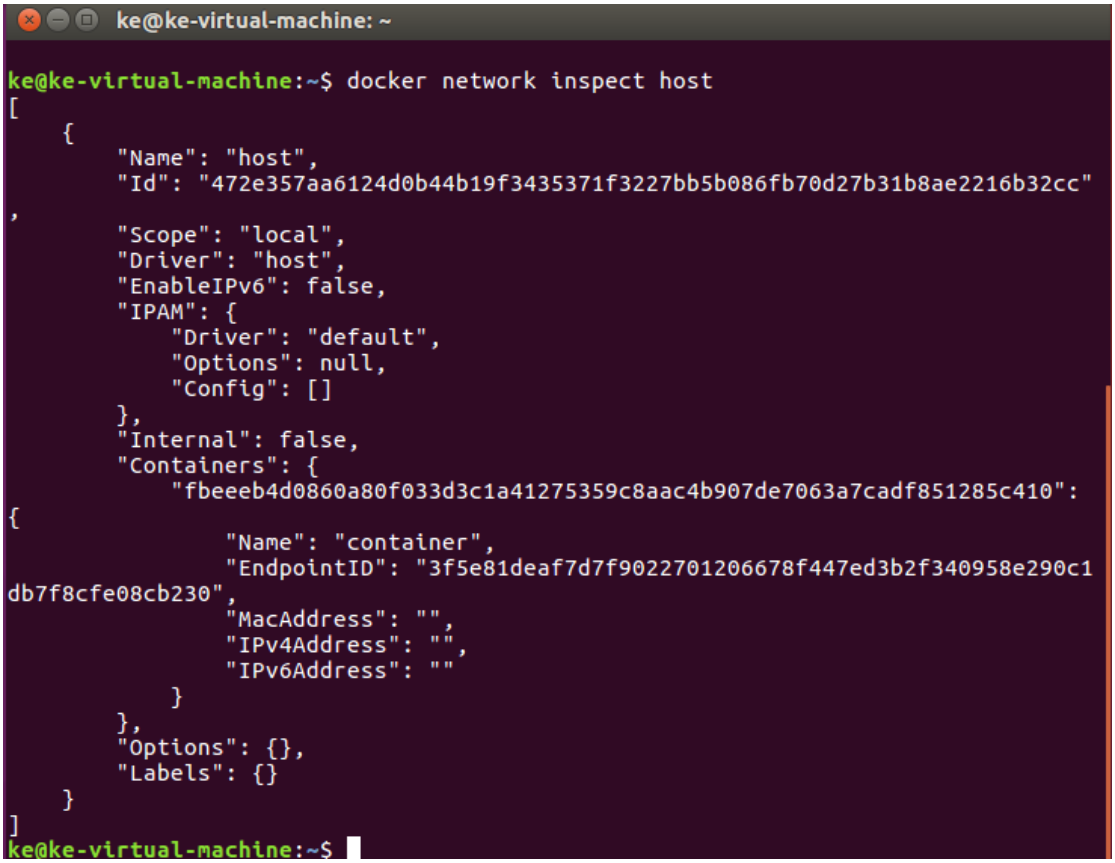


Figure 3-4 Docker network inspect host

También podemos crear la red desde dentro del contenedor. Por ejemplo:

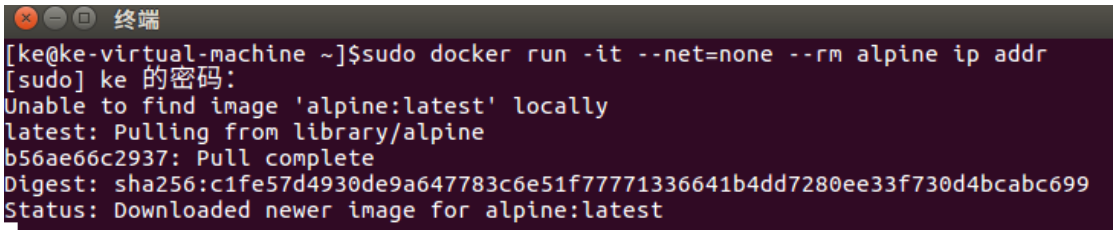


Figura 3-5 La red desde dentro del contenedor

Donde se especifica --net para configurar el tipo de red, que puede ser host, bridge, overlay o none.

Redes personalizadas

Docker permite utilizar redes personalizadas para aislar mejor los contenedores. Docker proporciona algunos controladores predeterminados de red para crear estas redes. Puede crear una nueva red Bridge o utilizar una red existente. También puede crear un complemento de red o una red remota y escribirlo en sus propias especificaciones. Puede crear varias redes del mismo tipo y agregar un contenedor a varias redes. Los contenedores sólo pueden comunicarse dentro de la misma red y no pueden comunicarse entre diferentes redes. Un contenedor

conectado a dos redes puede comunicarse con los miembros de cada red. Cuando un contenedor está conectado a varias redes, la conexión externa se proporciona a través de la primera red que no está interna. [1]

- **Comando de red docker**

```
ke@ke-virtual-machine: ~
ke@ke-virtual-machine:~$ sudo docker network -help
Flag shorthand -h has been deprecated, please use --help
unknown shorthand flag: 'e' in -elp
See 'docker network --help'.

Usage:  docker network COMMAND

Manage Docker networks

Options:
      --help   Print usage

Commands:
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one or more networks
  ls          List networks
  rm          Remove one or more networks

Run 'docker network COMMAND --help' for more information on a command.
```

Figura 3-6 Comendo de red docker

- **Crear una red de prueba de red**

```
ke@ke-virtual-machine: ~
ke@ke-virtual-machine:~$ sudo docker network create test-network
94551994ed2b822c498d930b9b6e7a3f1569107919ae12a564f6c617184bcae8
ke@ke-virtual-machine:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
0a696b54c614        bridge              bridge              local
472e357aa612        host                host                local
af78e71109ec        none                null                local
94551994ed2b        test-network        bridge              local
ke@ke-virtual-machine:~$
```

Figura 3-7 Crear una red de prueba de red

- **Iniciar el contenedor y conectar a la red de prueba**

Figura 3-8 Iniciar el contenedor y conectar a la red de prueba

3.1.2 Contenedor docker para comunicación de host único

Contenedor docker para comunicación de host único se usando la tecnología de interconexión de contenedores, es decir, **-link**, el puerto de red de mapeo no es

la única forma de conectarse entre sí. El sistema de **linking** de Docker le permite conectar varios contenedores juntos y permitirles interactuar entre ellos.

Crear un contenedor de base de datos:

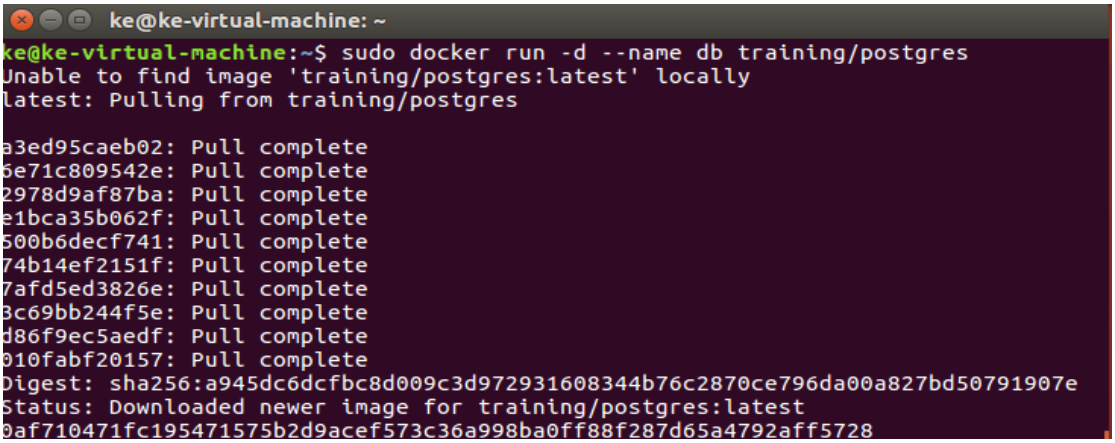


Figura 3-9 Crear un contenedor de base de datos

A continuación, se crea un nuevo contenedor de web y se conecta al contenedor DB.

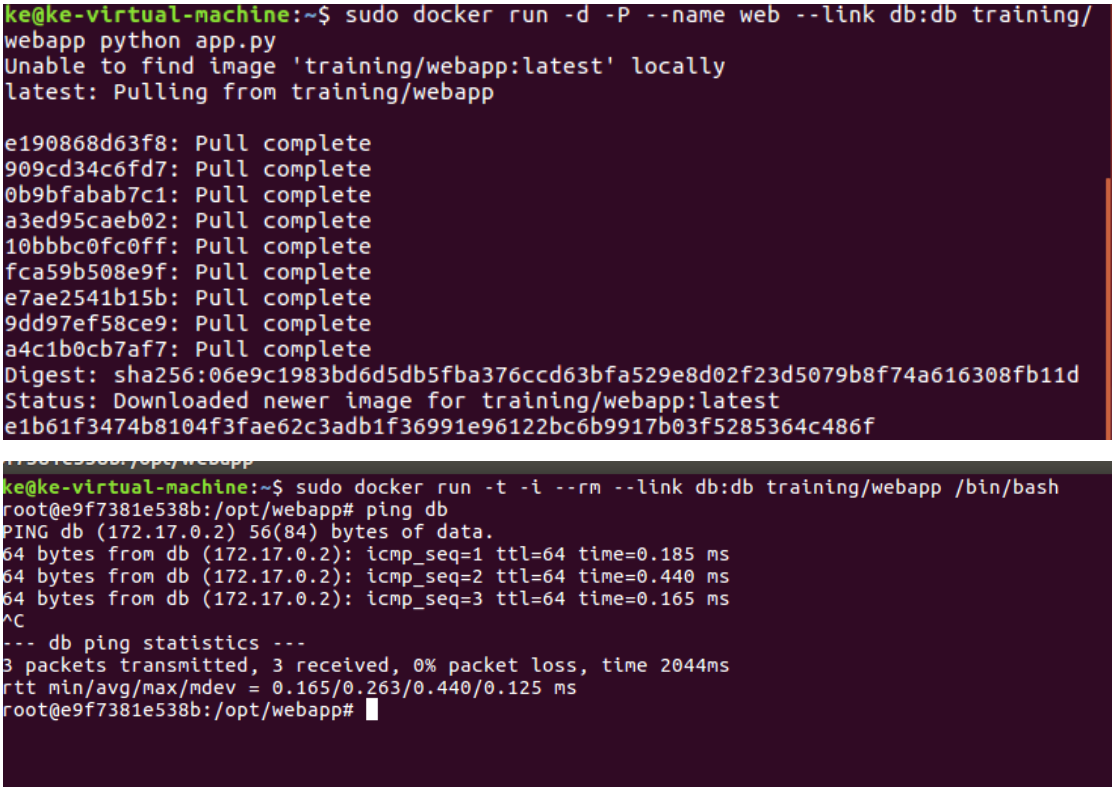


Figura 3-10 Crea un nuevo contenedor de web y conecta al contenedor DB

En este punto, contenedores DB y contenedores web han establecido una relación de Internet. El formato del parámetro -link es -link name: alias, donde name es el nombre del contenedor a enlazar, y alias es el alias de la conexión.

Hay 2 hosts, el primero es un contenedor web, el segundo es el contenedor de DB. El comando ping se puede probar la conexión dentro del host y el contenedor DB.

3.1.3 Comunicación entre contenedores Docker en diferentes hosts

En primer lugar, tenemos los siguientes esquemas para comunicar entre los hosts.

- **Contenedor usando el modo host**: El contenedor utiliza la red del host directamente, así puede soportar la comunicación entre hosts. Aunque puede resolver problemas de comunicación entre hosts, este enfoque es muy limitado, propenso a conflictos en los puertos e ignora el aislamiento de red. Es probable que un fallo de contenedor haga que todo el host se bloquee.
- **Enlace de puerto:** Al establecer el puerto del contenedor con el puerto de host, cuando se comunica entre hosts, se debe usar el par IP de host + puerto para acceder a los servicios en el contenedor. Obviamente, este enfoque se puede admitir la pila de red de cuatro niveles y más, y como el contenedor y el host están estrechamente acoplados, es difícil lidiar con escalabilidad y flexibilidad.
- **Personalice la red de contenedores fuera de docker:** Después de crear el contenedor mediante Docker, modificar el espacio de nombres de la red del contenedor para definir la red del contenedor. Lo típico es **pipework**, El contenedor se crea en modo none, **pipework** pasa por el espacio de nombres de la red del contenedor para reconfigurar la red para el contenedor, así, la red de contenedores puede tener IP estáticas, comunicarse a través de vxlan y otros métodos, por lo que la red es muy flexible, aunque durante un periodo de tiempo el contenedor no tendrá IP. Obviamente no se puede usar en escenarios de gran escala, solo se puede usar en la prueba de laboratorio.
- **SDN de terceros define la red de contenedores:** Utilizando herramientas de SDN de terceros, como Open vswitch o Flannel, para crear un entorno de red para contenedores que puedan comunicarse desde diferentes hosts. Estos esquemas generalmente requieren que el CIDR del puente docker0 en cada host sea diferente para evitar el problema de colisión IP y limitar el rango de IP disponible del contenedor en el host, además de la necesidad de una configuración más compleja cuando el contenedor requiere proporcionar servicios fuera del clúster. Desde el Docker de versión 1.9, se ha incluido la implementación de una red de overlay basada en la Vxlan, que se puede soportar la comunicación de contenedores entre los hosts. Al mismo tiempo, también se soporte el mecanismo de **plugin** a través del **libnetwork** para integrar una variedad de implementaciones de terceros, para lograr la comunicación entre hosts.

Los esquemas para realizar la comunicación entre los hosts más populares en la actualidad son:

- **Red basada en el túnel de Overlay**: Según el tipo de túnel, diferentes empresas u organizaciones tienen diferentes implementaciones. La red de Overlay de Docker se basa en la implementación del túnel de vxlan. La OVN debe lograrse a través de un túnel de **geneve** o de **stt**. La última versión de **flannel** también se inició realizar la red de OVERLAY basando vxlan predeterminada.
- **RED de Overlay basado en la encapsulación de paquetes**: Sobre la base de la encapsulación UDP y otros paquetes, cluster de Docker para lograr una red que comunicación entre los hosts, Las implementaciones típicas incluyen **weave**, y **flannel** de versión anterior.
- **Basado en la red SDN de implementación de tres niveles**: Basado en el acuerdo y el enrutamiento de tres niveles, implementar la comunicación entre los hosts directamente en la red de tres niveles, y lograr la seguridad del aislamiento de la red a través de iptables, como por ejemplo en el Proyecto Calico. Al mismo tiempo, para el entorno que no soporta el enrutamiento de tres niveles, Calico también proporciona una implementación de red entre host basada en encapsulación IPIP.

3.1.4 CNI y CNM

- **CNI (Container Network Interface)**

Container Network Interface (CNI), es una especificación de red de contenedores iniciada por CoreOS, es la base del complemento de red de Kubernetes. La idea básica es: Container Runtime crear un contenedor, primero crea un network namespace y luego llama al complemento de CNI para configurar la red para este netns, al final inicie el proceso dentro del contenedor.

El complemento CNI incluye dos partes:

1) **CNI Plugin** es responsable de configurar la red para el contenedor, que incluye dos interfaces básicas:

   AddNetwork(net NetworkConfig, rt RuntimeConf).

   DelNetwork(net NetworkConfig, rt RuntimeConf).

2) **IPAM Plugin** es responsable de asignar direcciones IP al contenedor, que incluye principalmente host-local y dhcp.

Por ejemplo tiene:

Figura 3-11 Los complementos de CNI

- **CNM (Container Network Model)**

Container network model (CNM) es el modelo de red de Docker, compuesto principalmente de Sandbox, Network y Endpoint

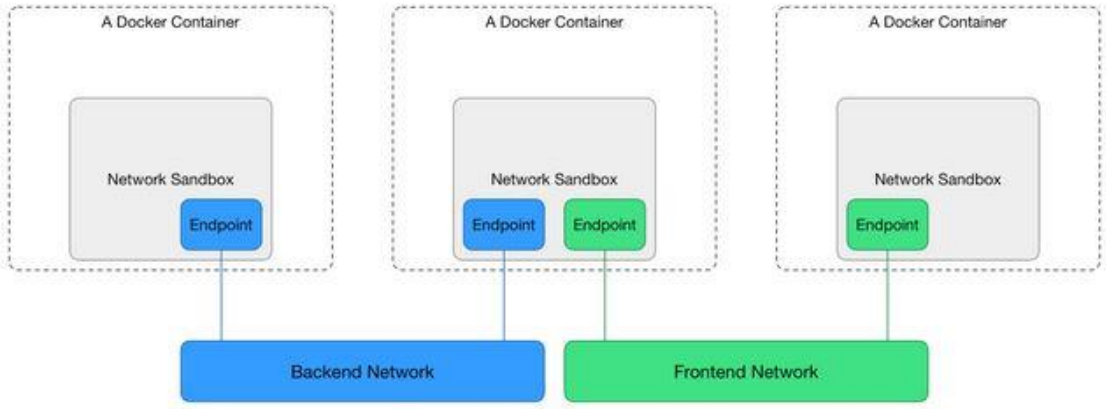


Figura 3-12 Container Network Model

1) **Sandbox**: Un Sandbox correspondiente a una pila de red de contenedores, puede gestionar la interfaz, la ruta, los dns y otros parámetros del contenedor. Un Sandbox puede tener múltiples Endpoints, estos Endpoints pueden pertenecer a una Red diferente.
2) **Endpoint**: Sandbox access network a través de Endpoint, un Endpoint solo puede pertenecer a una red. La implementación del endpoint puede ser un veth pair, un puerto interno de Open vSwitch o algún otro dispositivo similar.
3) **Network**: Una Network consta de un grupo de Endpoint, Estos Endpoint se pueden comunicar directamente entre sí, la comunicación entre diferentes Endpoint de red está aislada entre sí.

Los controladores de red en modo CNM no pueden acceder al espacio de nombres de red del contenedor. El beneficio de hacerlo es que libnetwork puede proporcionar el arbitraje para la resolución de conflictos.

CNI admite la integración con IPAM de terceros y se puede usar en cualquier contenedor.

3.2 Flannel

3.2.1 Introducción de Flannel

Una de las implementaciones de red existentes para contenedores es Flannel. Flannel está desarrollado por CoreOS y se usa en Kubernetes. Flannel puede resolver los dos problemas de la asignación de la dirección IP y la comunicación entre hosts:

- Para el problema de la asignación de IP: Utiliza un CIDR para que cada host asigna un segmento de dirección, tal como un segmento de dirección de una máscara de 24 bits, de forma que un host puede admitir 254 contenedores. Cada host se dividirá en un segmento de dirección IP de subred, para resolver el problema de la asignación de direcciones IP. Una vez que este segmento de dirección se asigna al Docker Daemon, Docker Daemon puede asignar IP al contenedor.
- El problema del intercambio de paquetes entre host, es adoptado por el enrutamiento de tres niveles: Al igual que con las prácticas tradicionales, todos los contenedores estarán conectados a Docker0, entre Docker0 y la tarjeta de red del host se inserta en un dispositivo virtual Flannel0, este dispositivo virtual le da a Flannel tiene mucha flexibilidad, puede implementar diferentes paquetes, protocolos de túnel como VXLAN. l paquete está encapsulado como un paquete de UDP de VXLAN por el dispositivo Flannel0. Es decir, Flannel0 puede hacer la adaptación del acuerdo, que es las características de Flannel, es una ventaja.

3.2.2 El principio de funcionamiento de la Flannel

Flannel es esencialmente una "overlay network", Es decir, los datos de TCP se empaquetarán en otro nivel de enrutamiento, reenvío y comunicación de paquetes de red. Actualmente tienen soporte para el reenvío de datos de enrutamiento UDP, VxLAN, AWS VPC y GCE.

La comunicación de datos predeterminada entre los nodos es el reenvío por UDP, en la página GitHub de Flannel tiene el siguiente esquema:

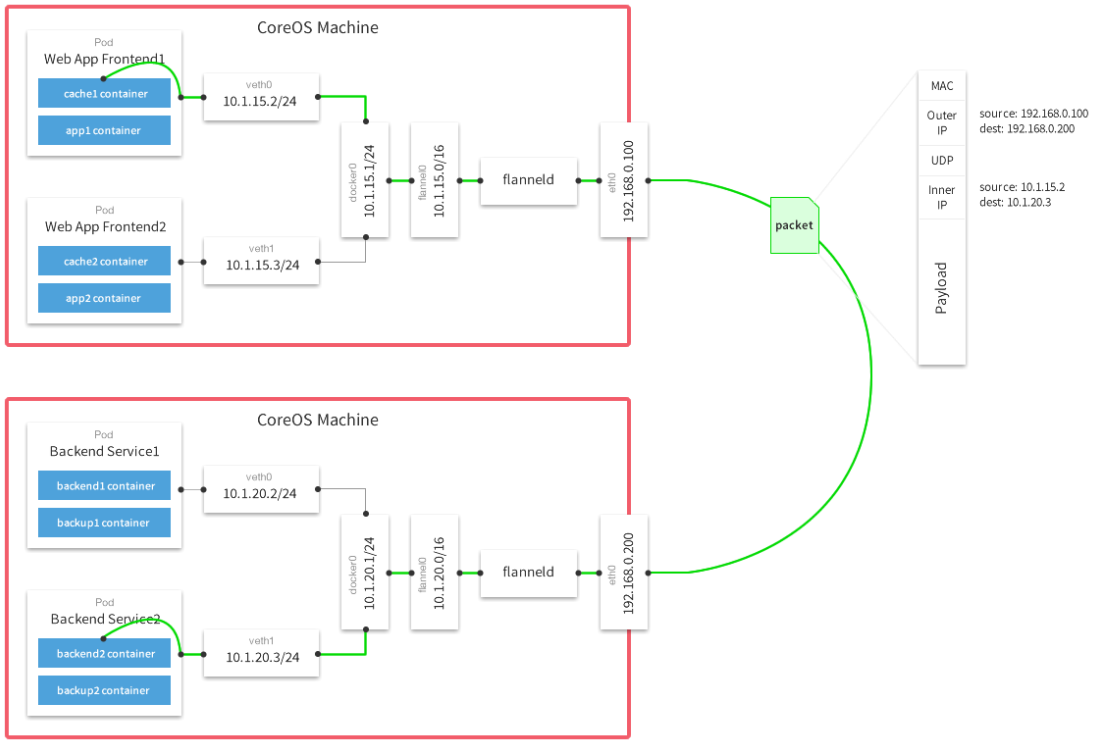


Figura 3-13 Esquema de Flannel [4]

Los datos del contenedor de origen emitidos por el host de la tarjeta de red virtual docker0 se reenvían a la tarjeta de red virtual flannel0, esta es una tarjeta de red virtual P2P, monitorización del servicio de flanneld en el otro extremo de la tarjeta de red.

Flannel mantiene una tabla de enrutamiento entre nodos a través del servicio **Etcd**.

El servicio de flanneld del host de origen encapsula el contenido de datos original a un paquete de UDP, de acuerdo con su propia tabla de enrutamiento entrega de al servicio de flanneld de nodo de destino, los datos llegaron después del desembalaje, y luego directamente en la tarjeta de red virtual de flannel0 del nodo de destino, y luego reenviado a la tarjeta de red virtual docker0 del host de destino, y finalmente a través de docker0 para llegar al contenedor de destino.

Así que completamos la transmisión del paquete de datos, y tenemos que explicar dos cuestiones.

- **Encapsulación UDP**

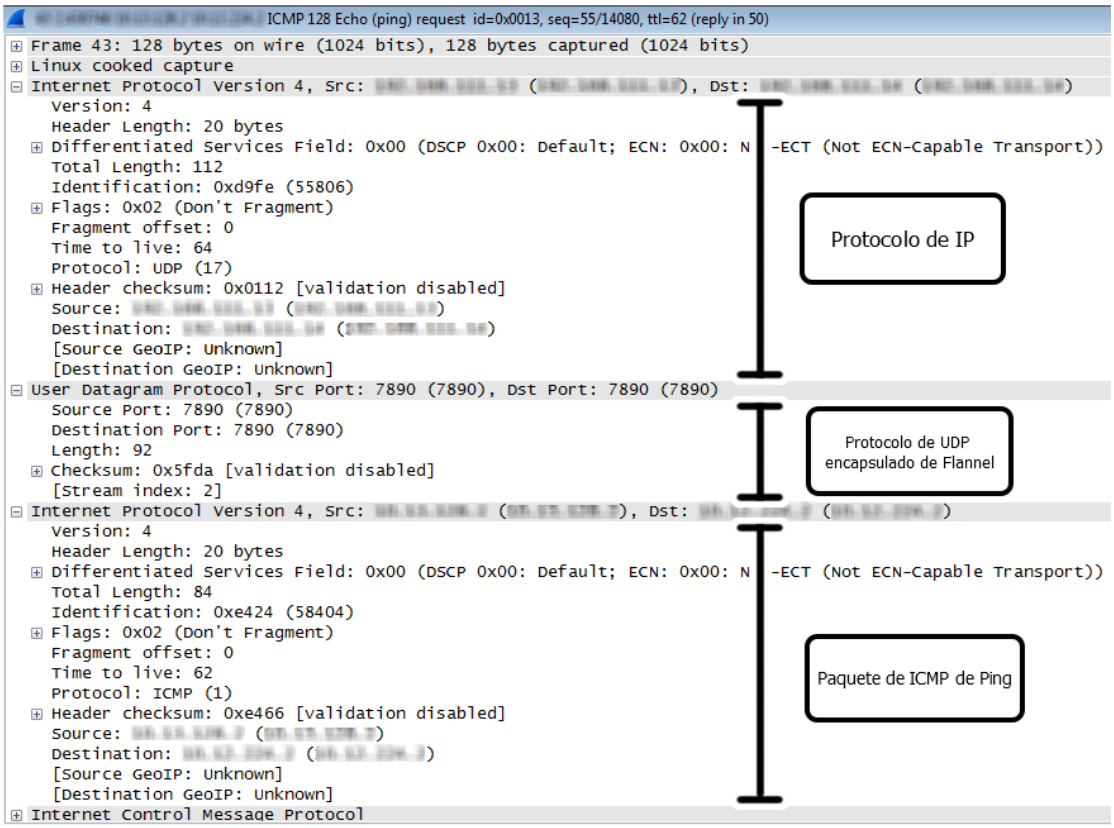


Figura 3-14 El paquete de Dato de Flannel

Este es el paquete de tráfico del comando ping obtenido en uno de los nodos de comunicación. Se puede ver en la parte de contenido de datos de UDP que en realidad es otro paquete de ICMP. Los datos origen están encapsulados en el servicio de UDP de Flannel del nodo de origen, después de la entrega al nodo de destino por el otro extremo del servicio Flannel, revertido al paquete de datos original. De esta forma este proceso es transparente para ambos lados del servicio Docker.

- **Docker en cada nodo utiliza un segmento de dirección IP diferente**

La Flannel a través de la Etcd asignada al segmento de dirección IP disponible en cada nodo, modifica los parámetros de inicio de Docker, como se puede ver a continuación.

```
core@ip-172-31-14-97 / $ ps aux | grep '172.17.18.1' | grep docker
root     21740  0.0  1.7 215388 17604 ?        Ssl  00:40   0:00 docker --daemon --host=fd:// --bip=172.17.18.1/24 --mtu=8973 --ip-masq=false
core@ip-172-31-14-97 / $
```

Este es el parámetro de ejecución del proceso de servicio de Docker que se visualiza en el nodo donde se ejecuta el servicio Flannel.

Hay que tener en cuenta que el parámetro "--bip = 172.17.18.1 / 24", limita el rango de IP que recibe el contenedor de nodos.

Este rango IP es asignado automáticamente por Flannel, y Flannel garantiza que no se dupliquen manteniendo registros en el servicio Etcd.

Las características de Flannel son:

1 Muy adecuado para usar en Kubernetes;
2 Debido a que cada host tiene una subred, la falta de flexibilidad.

3.3 Weave

3.3.1 Introducción de Weave

Weave fue desarrollado por Zett.io y permite a partir de una red virtual conectar contenedores Docker implementados en varios hosts. El proceso es transparente para el contenedor, por lo que las aplicaciones que usan la red no necesitan configurar información como mapeos de puertos y enlaces.

Los dispositivos externos tienen acceso a los servicios proporcionados por el contenedor de la aplicación en la red Weave, y los sistemas internos existentes también pueden estar expuestos al contenedor de la aplicación. Además, la comunicación de Weave admite el cifrado, por lo que los usuarios pueden conectarse al host desde una red que no es de confianza.

Weave crea una red virtual que conecta contenedores de Docker que implementados por varios hosts.

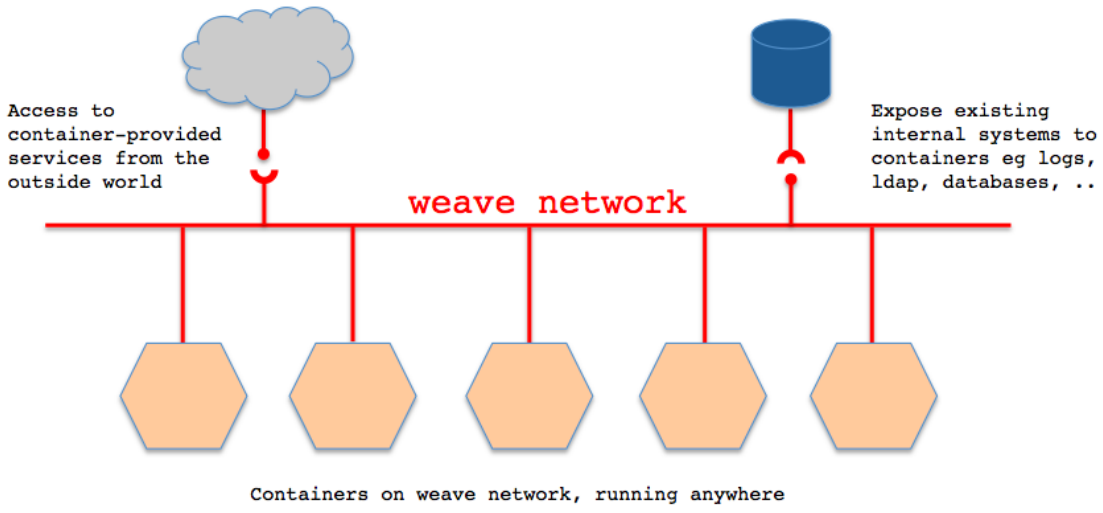


Figura 3-15 Los contenedores conectados al mismo switch de red [9]

Las aplicaciones usan la red como si todos los contenedores estuvieran conectados al mismo switch de red, no hacer falta para configurar los puertos, enlaces, etc. Los servicios proporcionados por los contenedores en la red de weave pueden hacerse accesibles al mundo exterior, independientemente de dónde se estén ejecutando esos contenedores. También los sistemas existentes pueden exponerse a los contenedores independientemente de su ubicación.

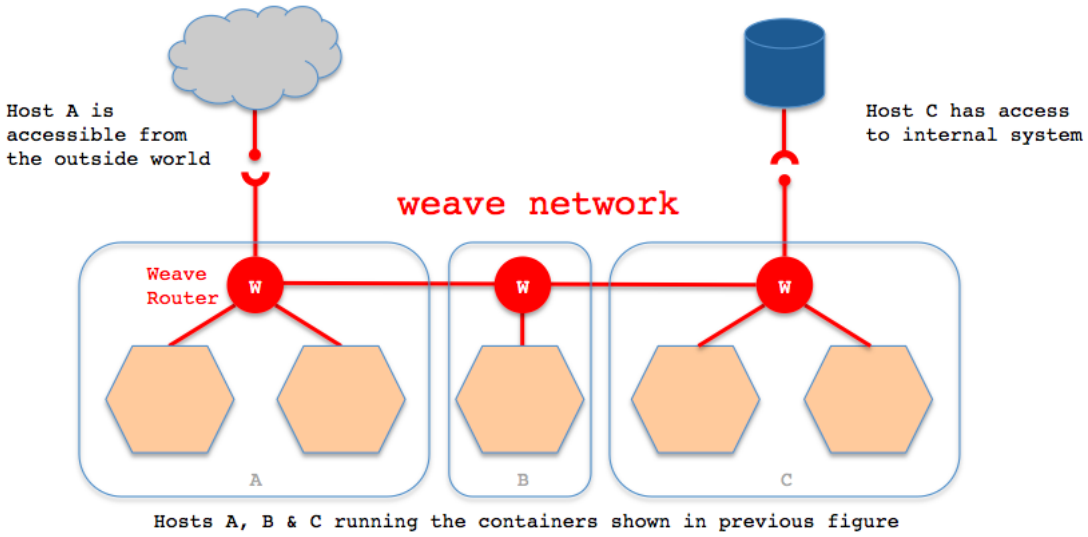


Figura 3-16 los contenedores operan en las redes conectadas parcialmente [9]

Weave puede atravesar firewalls y operar en las redes conectadas parcialmente. Con Weave puedes construir aplicaciones que consisten en múltiples contenedores, que se puede ejecutar en cualquier lugar.

3.3.2 Principio de implementación de Weave

Weave creará un bridge en el host, Cada contenedor está conectado al bridge a través de un par veth, mientras que el bridge tiene un contenedor de Weave Router conectado con él. El Router buscará el paquete de red conectándose a la interfaz en bridge (La interfaz funciona en modo promiscuo), utilizando pcap.

La relación entre los módulos:

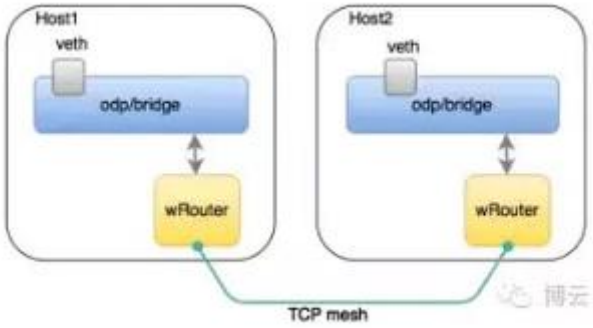


Figura 3-17 La relación entre los módulos [10]

Canal de datos:

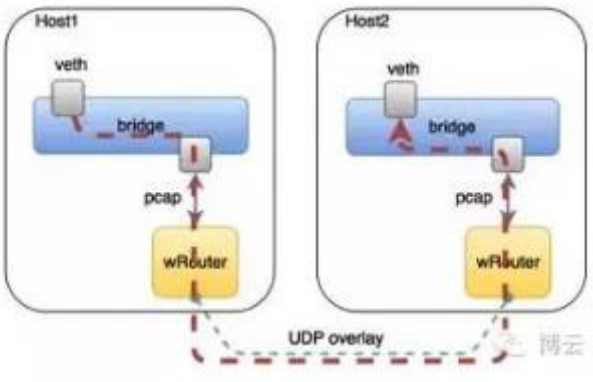


Figura 3-18 Canal de datos [10]

En cada contenedor habrá dos tarjeta de red, una tarjeta de red conectada a sí mismo, que puede conectarse con otro bridge de host; Otra tarjeta de red conectada a un bridge en Docker, y en este puente para monitorear un servicio DNS. El DNS está realmente integrado en el Router interno, es decir. Entonces, cuando el contenedor realice una consulta de DNS, en realidad, los pedidos se enruta al servidor DNS del host.

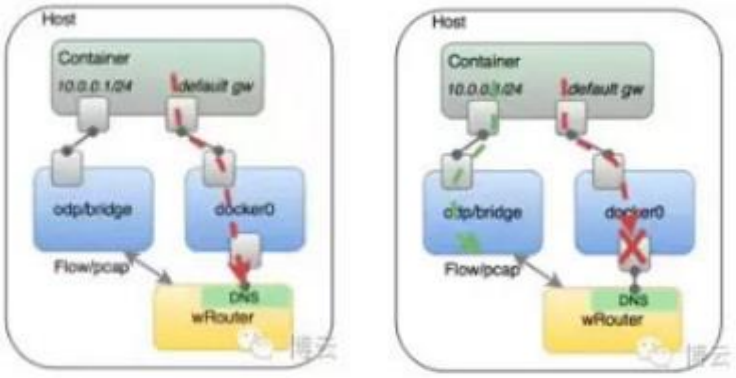


Figura 3-19 El Principio de funcionamiento y el aislamiento de Weave [10]

El aislamiento de Weave es un aislamiento de nivel de subred. Por ejemplo, hay dos contenedores en el segmento de red 10.0.1-24, luego en todos los contenedores añadirá una ruta de adp/bridge (de izquierda). Pero todo el tráfico del segmento que no pertenece a la red se convertirá en Docker0, en este momento Docker0 no está conectado con otros, por lo que llegó a un efecto de aislamiento.

- Ventajas de Weave

1. Soporte de comunicación cifrado entre hosts;
2. Soporte los contenedores unirse a la red o pelar la red dinámicamente;
3. Admite comunicación multi-subred en los hosts.

- Desventajas de Weave

1. No es compatible con el descubrimiento de servicios, el host no puede unirse dinámicamente a la red del nodo;
2. Solo a través de weave launch o weave connect para unirse a la red weave.

3.4 Calico

3.4.1 Introducción de Calico

Project Calico es una implementación de SDN de tres capas, sin usar redes sobrecargadas. Calico se basa en el protocolo BPG y en el propio mecanismo de enrutamiento de Linux, sin requerir de hardware especial. Calico no usa tecnología NAT o de túnel. Se puede implementar fácilmente en el servidor físico, la máquina virtual (como OpenStack) o el entorno contenedor. Al mismo tiempo, viene con los componentes de administración de ACLs basados en Iptables que son muy flexibles, para satisfacer las necesidades de aislamiento de seguridad más complejas.

Todos los contenedores están configurados para usar calico-node para lograr la interoperabilidad de la red y el acceso a Ethernet.

3.4.2 Principio de implementación de Calico

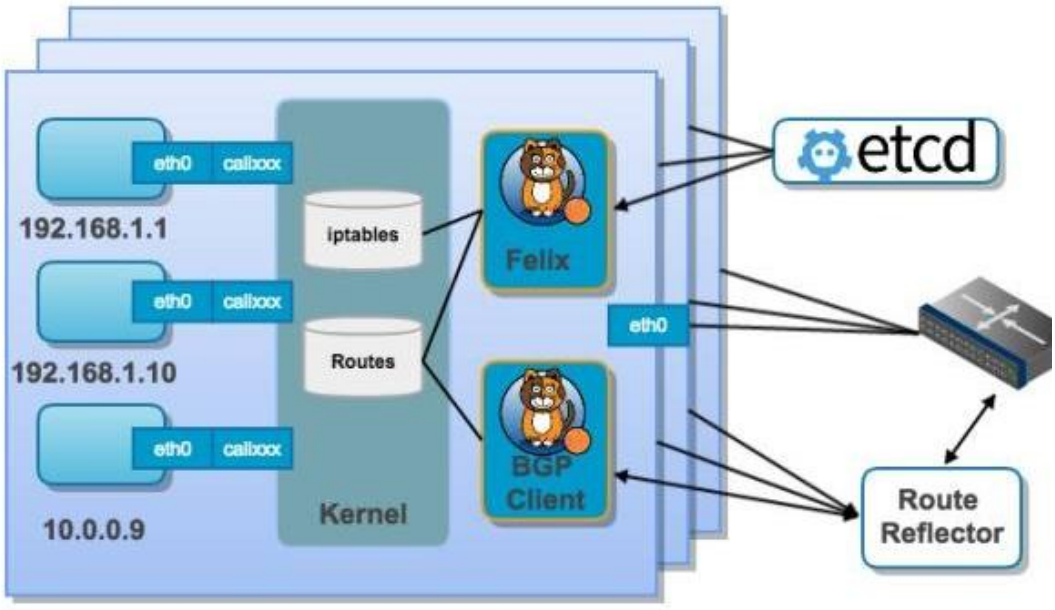


Figura 3-20 Principio de implementación de Calico [11]

Calico considera cada pila de protocolos del sistema operativo como un Router, y luego todos los contenedores que están conectados al Router en la terminal de red, entre enrutadores ejecutan el protocolo de enrutamiento estándar-protocolo BGP, y luego, que aprendan cómo reenviar la topología de red. El programa Calico es en realidad un programa puro de tres niveles, es decir, la pila

de tres capas de cada máquina para garantizar que los dos contenedores, entre la conectividad de tres niveles entre el contenedor de host. Para el plano de control, ejecuta dos programas principales en cada nodo, uno es Felix, que supervisa el almacenamiento del centro de etcd, obtiene eventos del mismo, como añadir un IP en la máquina o asignar un contenedor en la máquina, etc. Y luego en esta máquina para crear un contenedor, y configurar su tarjeta de red, IP, MAC, y luego escribe una anotación dentro de la tabla de enrutamiento, lo que indica que la IP debe estar en esta tarjeta. Una parte es un programa de enrutamiento estándar, que obtendrá los cambios de IP en enrutamiento del núcleo, y luego, extendido al otro host a través del protocolo de enrutamiento BGP estándar, para que el mundo exterior sepa que el IP está aquí.

Dado que Calico es una implementación pura de tres niveles, puede evitar la operación de encapsulación de paquetes relacionada con el programa de dos niveles, sin NAT y sin ninguna overlay. Por lo tanto, su eficacia de reenvío puede ser la más alta de todos los planes porque los paquetes van directamente a la pila del protocolo TCP / IP, y su aislamiento también es más bueno. Debido a que la pila del protocolo TCP / IP proporciona un conjunto de reglas de firewall, puede lograr una lógica de aislamiento más compleja a través de las reglas de IPTABLES.

Combinado con el cuadro anterior, comprendemos los componentes de Calico:

Felix, **Calico Agent**, ejecutándose en cada nodo que necesita ejecutar carga de trabajo, es principalmente el responsable de configurar el enrutamiento y las ACL y otra información para garantizar la conectividad del Endpoint;

etcd, almacenamiento de valor clave distribuido, es principalmente responsable de la consistencia de los metadatos de la red, para garantizar la precisión del estado de la red de Calico;

BGP Client (BIRD), Calico despliega los clientes de BGP en cada nodo que aloja a Felix. La función del cliente BGP es leer el estado de enrutamiento programado en el núcleo por Felix y distribuirlo alrededor del centro de datos.

BGP Route Reflector (BIRD), Para las implementaciones más grandes, un BGP simple puede convertirse en un factor delimitador porque requiere que cada cliente de BGP se conecte con cualquier otro cliente de BGP en una topología de malla. Esto requiere de un número creciente de conexiones que se vuelven difíciles de mantener con rapidez debido a la naturaleza N ^ 2 del aumento. Por esa razón, en implementaciones más grandes, Calico desplegará un BGP route reflector. Este componente, comúnmente utilizado en Internet, actúa como un punto central al que se conectan los clientes de BGP, evitando que la comunicación llegue a todos los clientes de BGP en el clúster.

- No es compatible con VRF (Virtual Routing and Forwarding [12]).

- No es compatible con la función de aislamiento de redes de múltiples usuarios, en el escenario de múltiples “tenants” habrá problemas de seguridad de la red.
- Los requisitos de diseño del plano de control Calico de la red física fueron L2 Fabric.

3.5 Romana

3.5.1 Introducción de Romana

Después del flat L3 de Calico, revisamos en esta sección la solución de Hierarchy L3 dada por Romana. Romana es un nuevo proyecto de código abierto propuesto por Panic Networks en 2016. Diseñado para resolver el problema de los costos de red del programa Overlay, aunque el objetivo básicamente es igual con Calico, pero a partir de una idea muy diferente.

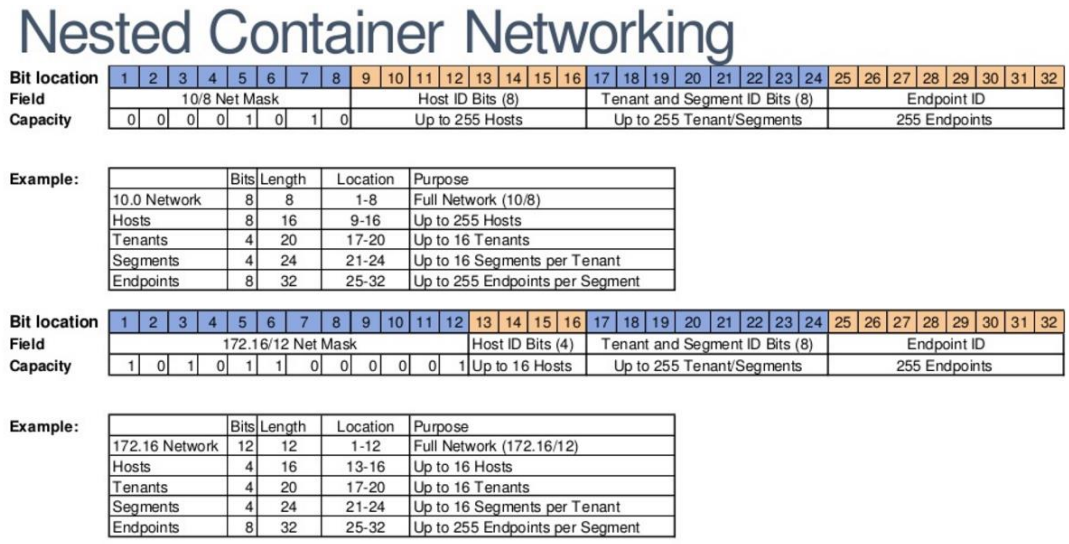


Figura 3-21 Los Redes de contenedores anidado [13]

Romana funciona en conjunto con los sistemas de orquestación en la nube y emite direcciones IP utilizando un sistema inteligente de administración de direcciones IP (IPAM) con reconocimiento de topología. Cada microsegmento está compuesto por uno o más bloques de direcciones de red. Romana luego configura rutas a estos bloques entre los hosts y dispositivos de red para que puedan reenviar el tráfico directamente a los puntos finales y aplicar la política de red sin la sobrecarga de la encapsulación. [14]

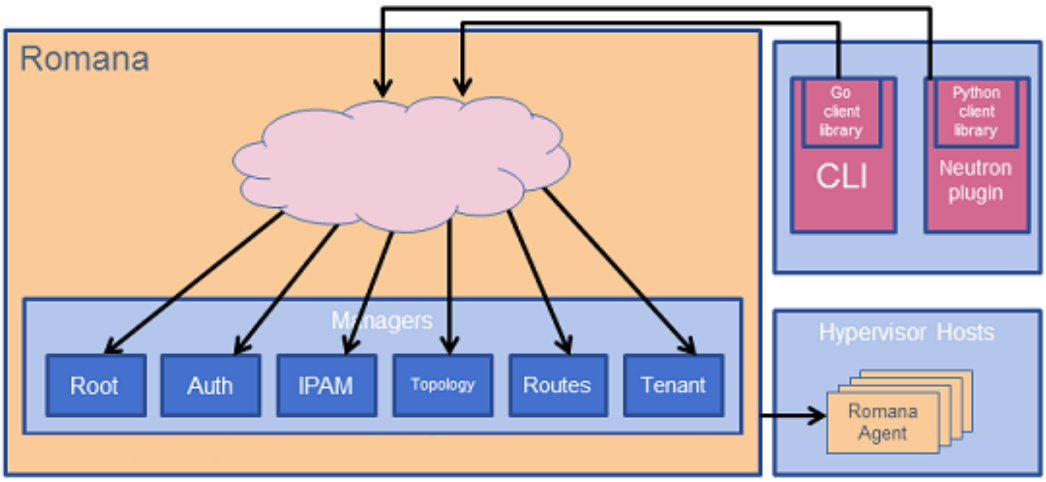


Figura 3-22 El principio de funcionamiento de romana [15]

Romana aísla los puntos finales con filtros de tráfico aplicados a los puntos finales de la red. Las interfaces de red se conectan directamente con el kernel y proporcionan aislamiento de otros puntos finales en el host que, de otro modo, podrían compartir un puente de Linux en donde los marcos de Ethernet fluyen libremente. Esto permite que el filtrado de paquetes en la capa 3 mediante el uso de reglas de iptable imponga el aislamiento y la política de seguridad de la red.

Romana admite las API de Kubernetes NetworkPolicy para el aislamiento del espacio de nombres, así como su propia capacidad ampliada de aplicación de políticas de red para el filtrado del tráfico de entrada y salida y las políticas de protección del host.

La política de red basada en metadatos y etiquetas de pod permite implementar políticas de seguridad flexibles en colecciones arbitrarias de puntos finales de red, independientemente de la topología o el direccionamiento de la red.

Las ventajas de Romana es la red pura de tres niveles que tiene buen rendimiento. Pero se administra a los clientes basado en IP, hay restricciones a escala, y es difícil para cambiar los dispositivos físicos o cambiar la planificación de direcciones.

3.6 Mesos

Apache Mesos es un kernel administrador de clúster de código abierto desarrollado en la Universidad de California, Berkeley. Mesos corre en cada nodo del cluster y provee aplicaciones (como Hadoop, Spark, Kafta entre otras) con API's para el manejo de recursos y planificación de tareas de todo el datacenter.

3.6.1 Elementos básicos de Mesos

Mesos-master: Coordina todos los slave y determina los recursos disponibles para cada nodo. Agrega informes que calculan todos los recursos disponibles en

todos los nodos y luego emite invitaciones de recursos al Framework registrado al Master.

Mesos-slave: Informe al master de sus propios recursos gratuitos y administra cada mesos-tarea en este nodo, después del framework, solicita el recursos del Maestro, el slave recibirá el mensaje para iniciar el framework del executor.

Framework: Por ejemplo, Hadoop, Spark, etc., accede a Mesos a través de MesosSchedulerDiver

Executor: Ejecutor, utilizado para iniciar la tarea en el marco informático

El objetivo de Mesos es compartir recursos hardware de manera eficiente entre diferentes frameworks, simplificando su propia lógica de programación para maximizar la compatibilidad y la escalabilidad para garantizar la solidez de los entornos de uso de clústeres a gran escala y la aplicabilidad universal a una variedad de posibles entornos informáticos. Un gran valor de Mesos es la capacidad de administrar un conjunto de recursos de diferente naturaleza (cpu, memoria, red, disco, dispositivos especiales) entre un conjunto de sistemas de ejecución heterogéneos llamados frameworks.

3.6.2 Plataforma de computación distribuida basada en Mesos

- **Arquitectura de mesos**

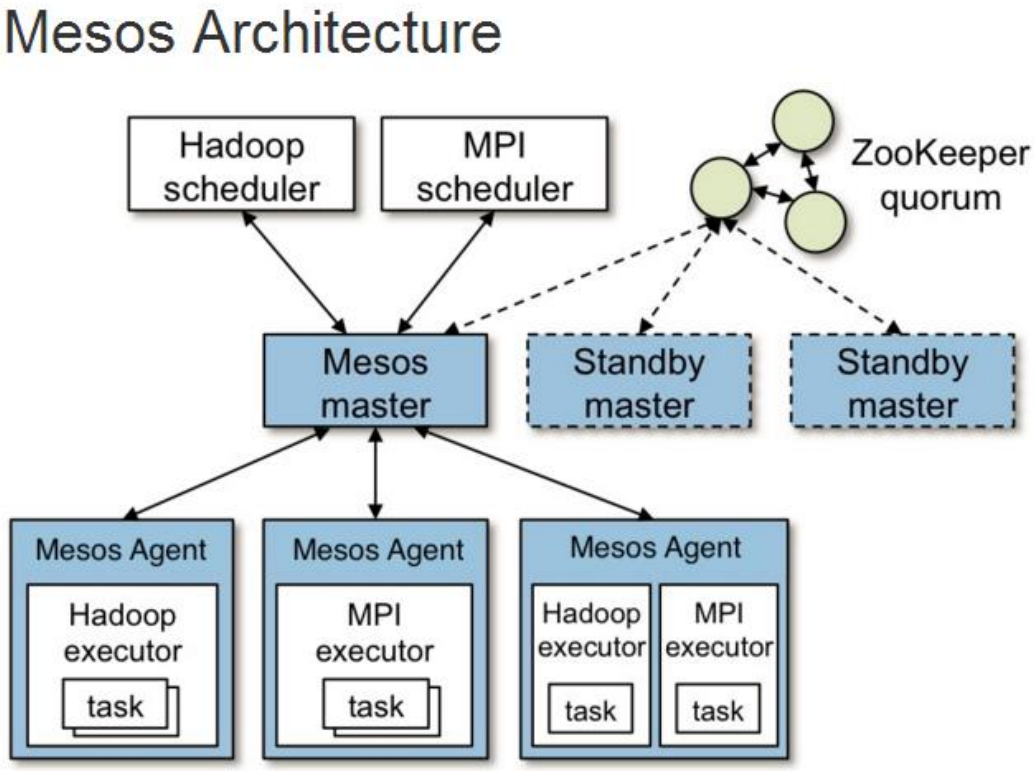


Figura 3-23 Arquitectura de Mesos [7]

En general, Mesos es una estructura master / slave, donde el master es muy ligero, almacenando los datos referentes al framework y al estado de los esclavos Mesos. Mesos utiliza zookeeper para resolver el punto único de fallo del maestro de Mesos y el descubrimiento de los servicios.

Mesos Master es en realidad un gestor de recursos global, usando una estrategia para asignar un recurso inactivo en un slave a un framework. Varios frameworks se registran con el maestro de Mesos a través de su propio planificador. La función principal del Mesos slave es informar de los recursos disponibles, del estado de la tarea e iniciar el ejecutor del framework.

- **Distribución de grano fino**

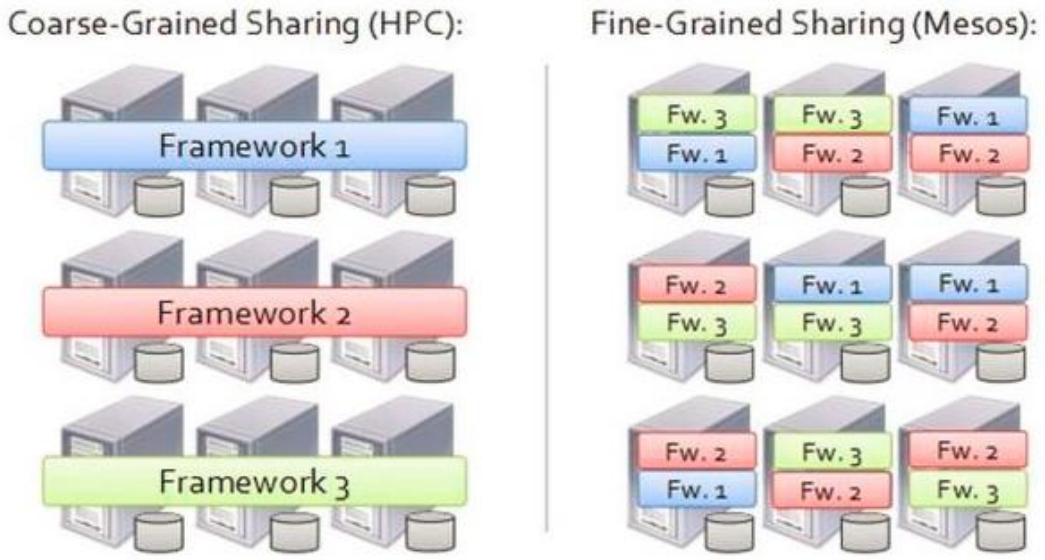


Figura 3-24 Distribución de grano fino de Mesos [16]

El mayor beneficio de Mesos es su capacidad de hacer una asignación de recursos detallada para clústeres distribuidos. Como se muestra en la figura 3-22, la parte izquierda es la asignación de recursos de grano grueso, la derecha es la asignación de recursos de grano fino. La asignación precisa de recursos se refiere a la asignación directa de recursos de acuerdo con las necesidades reales de la tarea, este mecanismo de asignación puede mejorar la utilización de los recursos.

En la figura 3-22, se muestran tres clústeres a la izquierda, en el que en cada uno hay tres servidores del clúster, se instalaron tres plataformas informáticas distribuidas. Por ejemplo, los tres anteriores instalados Hadoop de los tres medios son Spark, los siguientes tres son Storm; se manejaron tres marcos diferentes.

A la derecha hay nueve servidores administrados unificados por clúster de Mesos, todos de las tareas Spark, Hadoop o Storm se mezclan en nueve servidores en ejecución. Mesos mejora la redundancia de recursos. La administración de

recursos de grano grueso puede implicar el malgastar recursos por lo que la administración detallada mejora las capacidades de administración de recursos. La máquina Hadoop está muy inactiva, Spark no está instalado, pero Mesos puede responder tan pronto como cualquiera de las programaciones. Lo último es la estabilidad de los datos, porque las nueve máquinas son administradas centralmente por Mesos, por ejemplo para Hadoop. Este recurso informático no se comparte y el almacenamiento se comparte poco. Si esto está ejecutando Spark para hacer una migración de datos de red, obviamente afectará la velocidad. Entonces, el método de asignación de recursos será resource offers, que puede elegir los recursos, como en Mesos es Spark o Hadoop, y así sucesivamente

La lógica de distribución de Mesos es muy simple, realizando de forma continua ofertas de los recursos disponibles a los frameworks que se subscriben. Los métodos de asignación de recursos de Mesos también tienen un inconveniente potencial, no hay distribución centralizada, por tanto, puede que no sea una forma óptima en su conjunto. Pero las deficiencias de esta fuente de datos no son muy graves en este momento. Ahora es difícil llegar al 50% de la tasa de contribución de recursos de un centro de cómputo, la gran mayoría de los centros de cómputo están muy ocupados.

- **Proceso de Mesos**

La siguiente figura muestra un ejemplo de un framework para ejecutar una tarea.

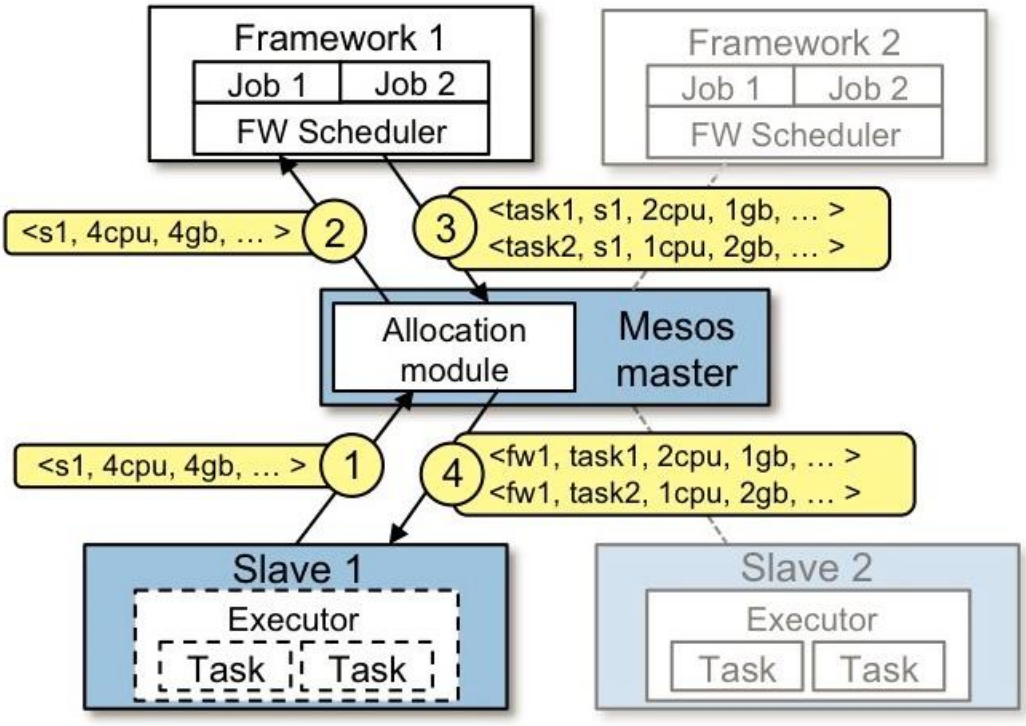


Figura 3-25 Proceso de Mesos [7]

1 Slave1 informa al Master que tiene recursos ociosos por un total de <4CPU, 4 GB de RAM>, el Máster llama al módulo de asignación y le comunica al Framework1 todos los recursos disponibles.

2 El Máster envía una Oferta de Recursos describiendo el recurso libre actual para el Slave1 al Framework1.

3 El planificador de Framework1 responde al Master, indicando que necesita ejecutar dos tareas en Slave1. La primera tarea necesita como recursos <2CPU, 1GBRAM>, y la segunda tarea requiere como recursos <1CPU, 2GB RAM>.

4 El máster asigna los recursos de la tarea a Slave1, Slave1 asigna los recursos apropiados al executor de Framework1. Entonces el executor comienza a realizar estas dos tareas, debido a que Slave1 deja sin utilizar los recursos de <1CPU, 1G RAM> por lo que el módulo de asignación puede proporcionar estos recursos a Framwork2.

Cuando la tarea finalice, el contenedor será "destruido", liberando los recursos asignados.

- **Enviar la tarea Docker en Mesos**

Mesos y Docker ya se combinan a la perfección. Mesos puede ejecutar aplicaciones de servicio y aplicaciones por lotes a través de Marathon y Chronos. Marathon y Chronos envían la tarea a través de RESTful, donde describen mediante JSON las características de la aplicación el número de instancias, los parámetros de la aplicación y las características del contenedor Docker a utilizar.

3.6.3 Ventajas Mesos

- **Eficiencia**

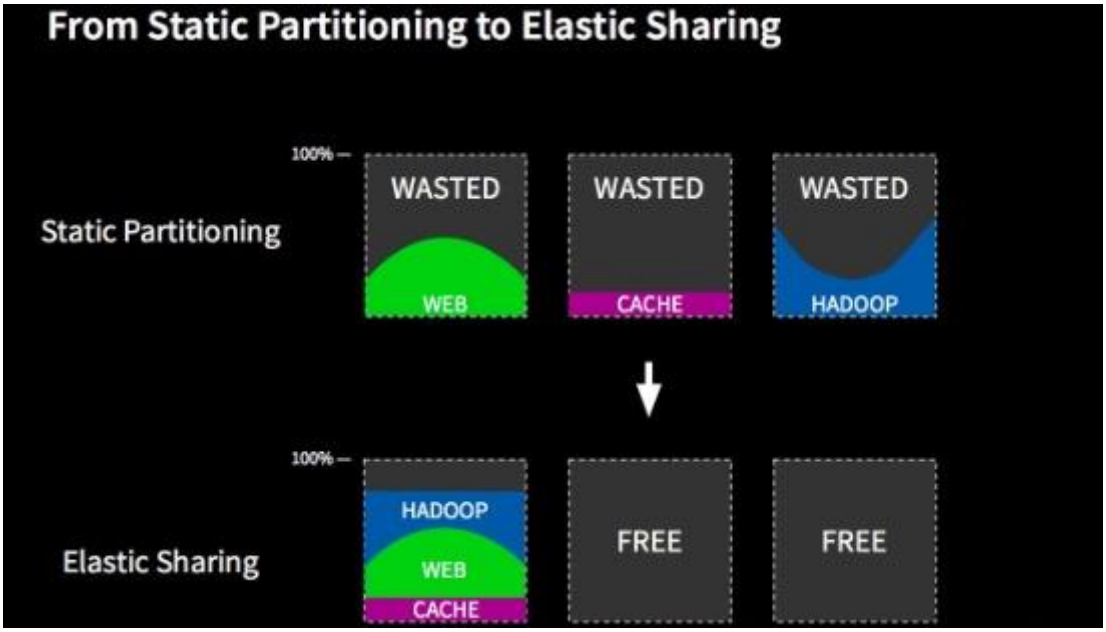


Figura 3-26 Eficiencia de Mesos [16]

La imagen proviene del sitio web de Mesosphere, que muestra los beneficios de Mesos para la eficiencia. Al usar Mesos explorer para arbitrar diferentes planificadores, ingresaremos el modo de compartimiento dinámico / compartición flexible, y todas las aplicaciones pueden usar el conjunto común de nodos para maximizar de forma segura el uso de los recursos. Para maximizar el uso de los recursos de forma segura. Mesos tiene una granularidad más fina porque Mesos asigna recursos en la capa de aplicación en lugar de la capa de máquina, asignando tareas a través del contenedor en lugar de la máquina virtual (VM) completa.

- **Escalabilidad**

El diseño escalable de Mesos proviene principalmente del uso de una arquitectura de programación de dos niveles. Al usar frameworks como proxy la programación real de tareas, el Master se puede implementar con un código muy ligero, debido a que el Máster no tiene que conocer la lógica de programación para cada tipo de aplicación que es compatible. Esto hace que sea es más fácil escalar el clúster. Además, dado que el Master no tiene que hacer la programación para cada tarea, no se convierte en un cuello de botella en el rendimiento.

- **Modular**

Cada acceso por un nuevo Framework, no requiere cambios en el código del Máster ni de los Esclavos, lo que hace que Mesos pueda ser soportado en un área amplia, y ha sido la razón de su creciente popularidad. En cambio, los desarrolladores pueden enfocarse en sus aplicaciones y en la elección de Framework. El marco que se admite actualmente se muestra a continuación:

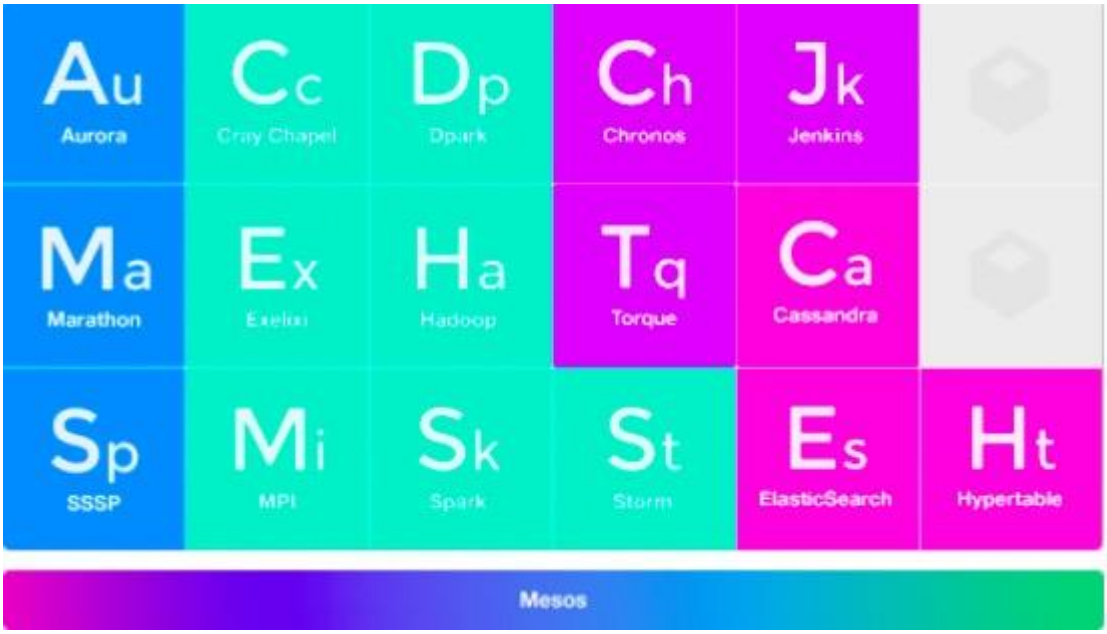


Figura 3-27 Modular de Mesos [16]

3.6.4 Algunos problemas

- Mesos no admite la definición de frameworks prioritarios, no puede establecer la prioridad de la tarea.
- Fragmentación de los archivos en ejecución. Spark shuffle producirá una gran fragmentación de archivos en la máquina esclava, si la configuración del esclavo no es suficiente, conducirá directamente al inodo de la máquina al 100%. Para mejorar el rendimiento del sistema de archivos, se debe modificar su configuración de spark y establecer **park.shuffle.consolidateFiles** en verdadero.
- Mesos es el más adecuado para tareas de trabajo de corta duración, frameworks flexibles como MapReduce, etc. Para trabajos que requieren una gran cantidad de tipos de recursos durante mucho tiempo, su planificación de recursos no global puede dificultar una distribución óptima. El marco de programación de Omega de Google está tratando de resolver estos problemas al mismo tiempo.

3.7 Kubernetes

Kubernetes es el sistema de gestión de un clúster de contenedores de código abierto de Google. Basado en Docker para construir un servicio de programación de contenedores, en su versión 1.8 proporciona programación de recursos, proporciona recuperación ante fallos, registro de servicios, capacidad de expansión dinámica entre otras capacidades.

3.7.1 Arquitectura de Kubernetes

Kubernetes define un conjunto de bloques de construcción (primitivas) que conjuntamente proveen los mecanismos para el despliegue, mantenimiento y escalado de aplicaciones. Los componentes que forman Kubernetes están diseñados para estar débilmente acoplados pero a la vez ser extensibles para que puedan soportar una gran variedad de flujos de trabajo. La extensibilidad es provista en gran parte por la API de Kubernetes, que es utilizada por componentes internos así como extensiones y contenedores ejecutando sobre Kubernetes

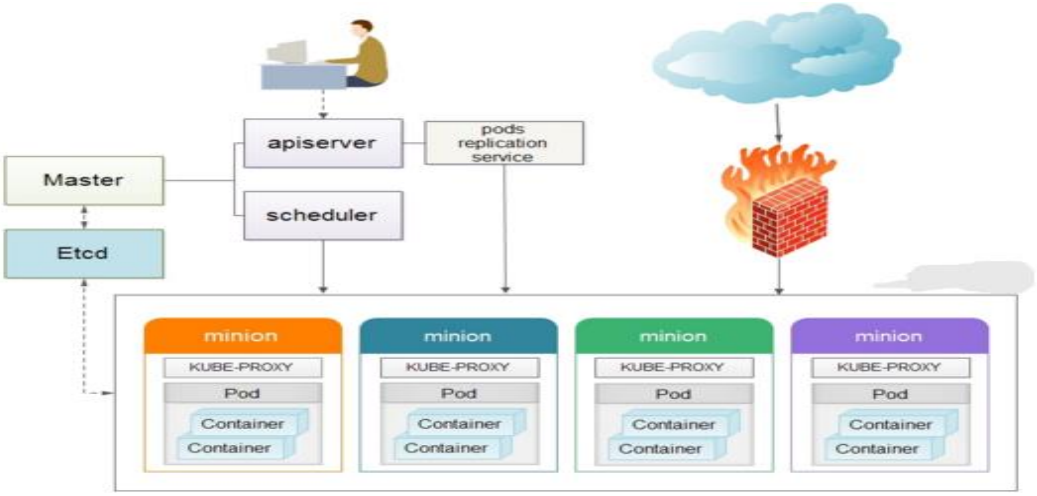


Figura 3-28 Arquitectura de Kubernetes [17]

Pods

En el sistema de Kubernetes, el elemento más pequeño no es un contenedor, sino el Pod, una abstracción de un grupo de contenedores que se puede crear, destruir, despachar y administrar en conjunto.

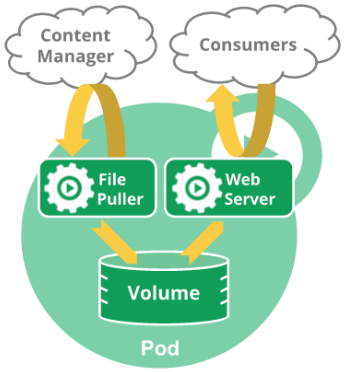


Figura 3-29 Pods de kuberneters [18]

Controllers

Un controlador es un bucle de reconciliación que lleva al estado real del clúster hacia el estado deseado.18 Hace esto mediante la administración de un conjunto de pods. Un tipo de controlador es un "Replication Controller", que se encarga de la replicación y escala mediante la ejecución de un número especificado de copias de un pod a través de un clúster. También se encarga de crear pods de reemplazo si un nodo subyacente falla.16 Otros controladores que forma parte del sistema central de Kubernetes incluye al "DaemonSet Controller" para la ejecución de exactamente un pod en cada máquina (o algún subconjunto de máquinas), y un "Job Controller" para ejecutar pods que ejecutan hasta su finalización, por ejemplo como parte de un trabajo batch.19 El conjunto de pods que un

controlador administra está determinado por los selectores de etiquetas que forman parte de la definición del controlador.

Services

Un Servicio de Kubernetes es una abstracción que define un conjunto lógico de Pods y una política para acceder a ellos, siguiendo un modelo de micro servicios. El conjunto de Pods apuntado por un Servicio es determinado por un Label Selector. Como ejemplo, un back-end de procesamiento de imágenes que se ejecuta con 3 réplicas. Esas réplicas pueden ser reemplazados: a las interfaces no les importa cuál es el programa de fondo que usan. Si bien los Pods reales que componen el conjunto backend pueden cambiar, los clientes de frontend no deberían tener que estar al tanto de eso o llevar un registro de la lista de los backend. La abstracción del servicio permite este desacoplamiento.

Labels and selectors

Kubernetes permite a los clientes (usuarios o componentes internos) vincular pares clave-valor llamados etiquetas (en inglés label) a cualquier objeto API en el sistema, como pods o nodos. Correspondientemente, selectores de etiquetas son consultas contra las etiquetas que resuelven a los objetos que las satisfacen.

Las etiquetas y los selectores son el mecanismo principal de agrupamiento en Kubernetes, y son utilizados para determinar los componentes sobre los cuales aplica una operación.

etcd

etcd es un almacén de datos persistente, liviano, distribuido de clave-valor desarrollado por CoreOS que almacena de manera confiable los datos de configuración del clúster, representando el estado general del cluster en un punto del tiempo dado. Otros componentes escuchan por cambios en este almacén para avanzar al estado deseado

Proxy

Proxy no solo resuelve el conflicto de puerto de servicio en el mismo host, también proporciona el puerto del servicio de reenvío de servicios para proporcionar capacidades de servicio externo.

3.7.2 Modelo de red Kubernetes

En la red de Kubernetes hay dos tipos de IP (IP de Pod y IP de Clúster de Servicio), la dirección IP de Pod está realmente presente en una tarjeta de red, Service Cluster IP Es una IP virtual que se redirige del kube-proxy a su puerto local utilizando las reglas de Iptables y luego se equilibra al Pod de fondo.

- **Modelo de Docker**

Antes de analizar el enfoque de Kubernetes para la creación de redes, vale la pena revisar la forma "normal" en que la red funciona con Docker. Por defecto, Docker usa una red de host-privada. Crea un bridge virtual, llamado docker0 de forma predeterminada, y asigna una subred desde uno de los bloques de direcciones privados definidos en RFC1918 para ese bridge. Para cada contenedor que Docker crea, asigna un dispositivo Ethernet virtual (llamado veth) que se adjunta al bridge. El veth está mapeado para aparecer como eth0 en el contenedor, utilizando espacios de nombres de Linux. La interfaz eth0 del contenedor recibe una dirección IP del rango de direcciones del bridge.

El resultado es que los contenedores Docker pueden comunicarse con otros contenedores solo si están en la misma máquina (y, por lo tanto, en el mismo bridge virtual). Los contenedores en diferentes máquinas no pueden alcanzarse, de hecho, pueden terminar con los mismos rangos de red y direcciones IP.

- **Modelo de Kubernetes**

IP-per-Pod, cada Pod tiene una dirección IP separada, todos los contenedores dentro del Pod comparten un espacio de nombres de red

Todos los Pods del clúster están en una red plana directamente conectada, se puede acceder directamente a través de IP

- Se puede acceder a todos los contenedores directamente sin NAT
- Se puede acceder a todos los nodos y todos los contenedores directamente sin NAT
- El contenedor vio IP como se ha visto con otros contenedores

Se puede acceder a la IP del clúster de servicios dentro del clúster y se debe acceder a las solicitudes externas a través de NodePort, LoadBalance o Ingress.

Cómo lograr esto

1) El siguiente es implementación de Kube-proxy con el modo de iptables:

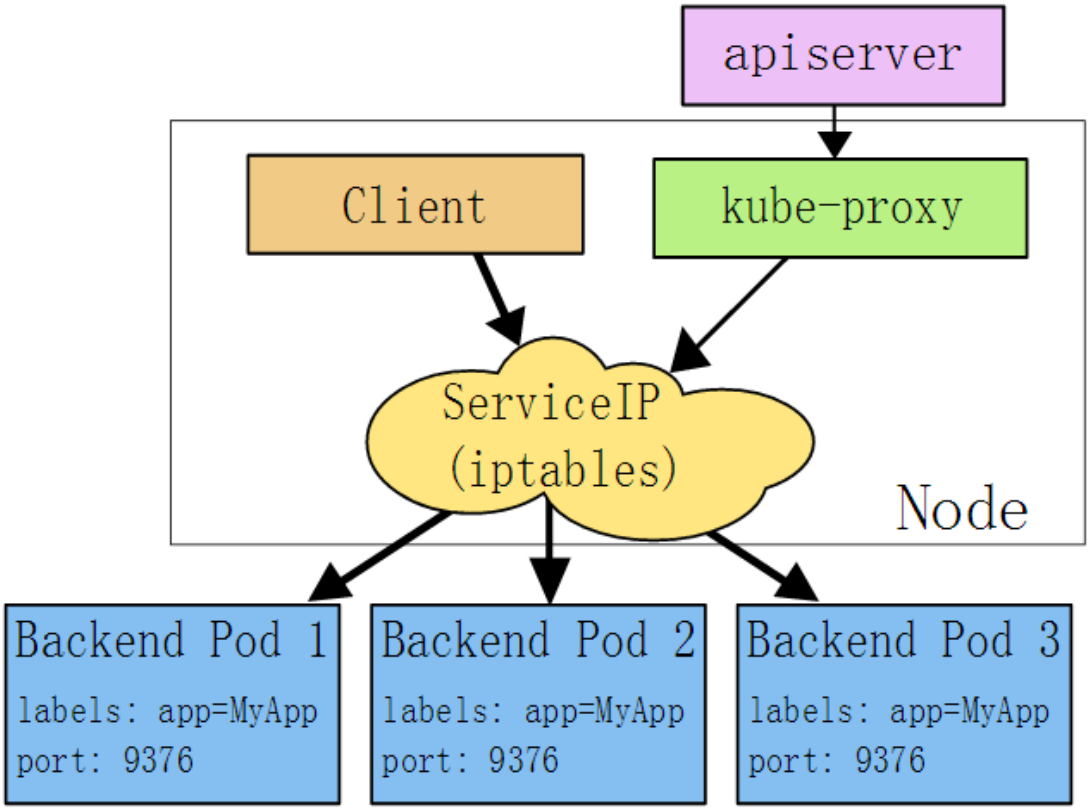


Figura 3-30 Proxy-mode [19]

En este modo, el kube-proxy monitorea el servidor maestro de Kubernetes que agregar y eliminar servicios y objetos de punto final. Para cada servicio, instala la regla de iptables, captura el tráfico al clusterIP (virtual) y el tráfico del puerto, y redirige el tráfico a una de las colecciones de servicio del back-end. Para cada objeto de Endpoints, instala la regla de iptables para seleccionar el Pod de back-end.

Al igual que con el agente de espacio del usuario, el resultado final está vinculado a la IP del servicio: cualquier tráfico en el puerto se envía al backend apropiado, y el cliente no sabe nada sobre Kubernetes o servicios o pod. Esto debería ser más rápido y más confiable que el agente espacial del usuario. Sin embargo, a diferencia del agente de espacio del usuario, si el Pod seleccionado inicialmente es no responde, el agente de iptables no puede volver a intentar automáticamente a otro pod, por lo que depende de tener una sonda de trabajo preparado.

2) Componentes de código abierto de la red Kubernetes:

Hay varias formas en que este modelo de red puede implementarse con ayudar de código abierto, principalmente dividido en dos partes: Overlay Networking y Esquema de enrutamiento.

Overlay Networking en la capa de IaaS de la red es también más aplicaciones, estamos de acuerdo en que con aumento del tamaño del nodo aumentará la complejidad, Y además de los problemas de red para rastrear más difícil, en caso de clúster a gran escala, este es un punto para considerar. Principalmente el siguiente programa: Weave, Open vSwitch(OVS), Flannel y Racher.

El esquema de enrutamiento generalmente es de 3 o 2 capas para lograr el aislamiento y la interoperabilidad entre el contenedor host, también es muy fácil para solucionar un problema, por ejemplo: Calico, Macvlan, etc.

4 Despliegue

En este capítulo, implementaremos estos escenarios y compararemos sus diferencias

4.1 Describe de entorno experimental

Hay tres hosts de Ubuntu en funcionamiento en el entorno experimental.

Instalaremos Docker Machine y desplegaremos Docker en todos los hosts mediante el comando docker-machine, siguiendo el documento de instalación oficial en https://docs.docker.com/machine/install-machine.

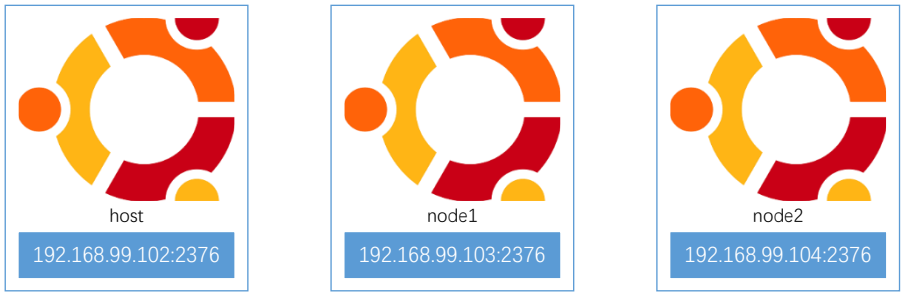


Figura 4-1 El entorno experimental

El método de instalación es muy simple y basta ejecutar los siguientes comandos:

```
curl -L
https://github.com/docker/machine/releases/download/v0.9.0/docker-machine-`uname -
s`-`uname -m` >/tmp/docker-machine &&
chmod +x /tmp/docker-machine &&
sudo cp /tmp/docker-machine /usr/local/bin/docker-machine
```

El ejecutable descargado se coloca en **/usr/local/bin**

Finalmente, ejecutamos **docker-machine version** para verificar que el comando se encuentra disponible.

```
[root@ubuntu ~]#
[root@ubuntu ~]# docker-machine version
docker-machine version 0.9.0-rc2, build 7b19591
[root@ubuntu ~]#
```

A continuación, se agrega el siguiente código a `$HOME/.bashrc`

```
PS1='[\u@\h \W$(__docker_machine_ps1)]\$'
```

4.2 Docker

El tráfico de red entre hosts ha sido durante mucho tiempo una característica muy esperada en Docker, y ya existen muchas versiones de herramientas o métodos de terceros en la comunidad antes de la versión 1.9 para tratar de resolver dicho problema. Aunque hay muchas diferencias en los detalles de la implementación de estos programas, sus ideas se pueden dividir en dos tipos: red VLAN de Capa 2 y red de Overlay. La red de Overlay se refiere a un nuevo formato de datos que encapsula paquetes de Capa 2 sobre paquetes IP a través de un protocolo de comunicación acordado sin cambiar la infraestructura de red existente. Esto no solo aprovechará la distribución de datos del proceso del protocolo de enrutamiento IP, sino que también la tecnología overlay para expandir el número de identificadores diferentes. La limitación de 4000 de VLAN se amplia para admitir hasta 16 millones de usuarios y, cuando sea necesario, el tráfico de difusión puede transformarse en tráfico de multidifusión, que evita inundar los datos de difusión. Por lo tanto, la red Overlay es en realidad la solución de enrutamiento y transmisión de datos de nodo cruzado más convencional. Oficialmente se incorporó a las soluciones de comunicaciones entre nodos compatibles oficialmente en Docker 1.9.

4.2.1 Creación de una red de overlay de Docker

En este apartado ilustramos como implementar varios escenarios de red de hosts cruzados en los host de Docker (192.168.99.102) node1 (192.168.99.103) y node2 (192.168.99.104), con Consul. Consul tiene múltiples componentes, pero en conjunto, es una herramienta para descubrir y configurar servicios en su infraestructura.

La forma más fácil es ejecutar Consul como un contenedor:

```
docker run -d -p 8500:8500 -h consul --name consul progrium/consul -server –bootstrap
```

```
[ke@ke-virtual-machine ~]$clear

docker@host:~$ docker run -d -p 8500:8500 -h consul --name consul progrium/consul -server -bootstrap
066c86551c830ef12eb3ec59b41d7ddc3c600006436e2bdac0436e9da15fb91d
docker@host:~$ exit
[ke@ke-virtual-machine ~]$docker-machine ls
NAME     ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        ERRORS
host     -        virtualbox   Running   tcp://192.168.99.103:2376           v17.10.0-ce
node1    -        virtualbox   Running   tcp://192.168.99.104:2376           v17.10.0-ce
node2    -        virtualbox   Running   tcp://192.168.99.102:2376           v17.10.0-ce
[ke@ke-virtual-machine ~]$
```

Después de que se haya iniciado el contenedor, puede acceder al Consul en la URL http://192.168.99.102:8500.

Figura 4-2 La página de web de Consul

A continuación, se modifica el archivo de configuración del demonio Docker para node1 y node2, ubicado en **/etc/docker/daemon.json**

```
{
¨cluster-store¨:¨consul://192.168.99.102:8500¨,
¨cluster-advertise¨:”10.0.2.15:2376”(a mi caso)
}
```

Donde indicamos en `cluster-store` la dirección de Consul y en `cluster-advertise` indicamos a Consul la propia dirección de enlace del nodo. Posteriormente se deber reiniciar el demonio de Docker.

```
systemctl daemon-reload
systemctl restart docker.service
```

node1 y node2 se registrarán automáticamente en la base de datos Consul.

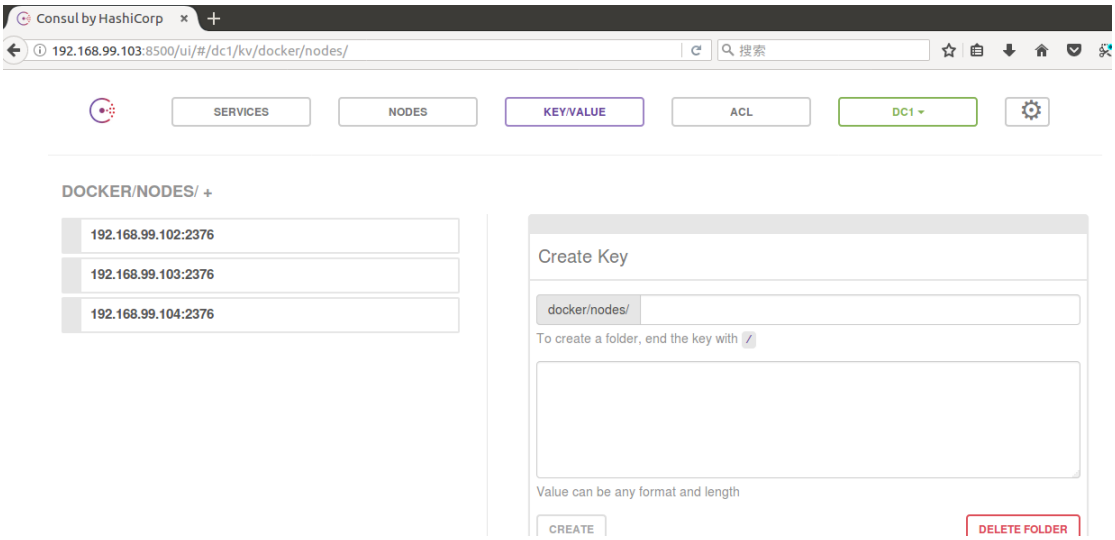


Figura 4-3 La base de datos Consul

El entorno listo y experimental es el siguiente.

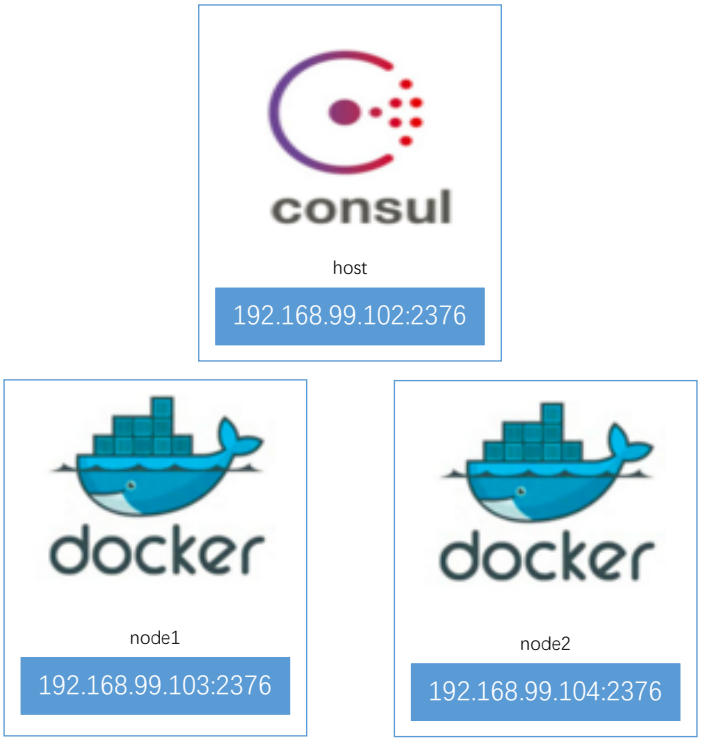


Figura 4-4 El entorno listo y experimental de DockerNetwork

Una vez configurado el entorno experimental, configurado y ejecutado el Consul, procedemos a crear una red de overlay.

Creamos una red de overlay llamada **demonet** con el siguiente comando:

```
docker@host:~$ ls
docker@host:~$ docker network create --driver overlay --subnet 10.10.3.0/24 demonet
58ffff4ec3817ec22f65de3c485c19f9361e0457a28e9fcbb219fccc15a8b770
docker@host:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
5fc032a7614b        bridge              bridge              local
58ffff4ec381        demonet             overlay             global
2fd03dccb931        host                host                local
af3d75b7ceb5        none                null                local
docker@host:~$
```

`-d overlay indica` que se utiliza el driver como overlay. Podemos comprobar la creación de la red con `docker network ls`

```
docker@node1:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
d283eadaf99d        bridge              bridge              local
58ffff4ec381        demonet             overlay             global
3154d5cfd61c        host                host                local
12378c76247b        none                null                local
docker@node1:~$
```

Hay que tener en cuenta que la red demonet tiene como ámbito global y el resto de redes son locales. Veamos la red existente en node1.

```
docker@node1:~$ docker network inspect demonet
[
    {
        "Name": "demonet",
        "Id": "58ffff4ec3817ec22f65de3c485c19f9361e0457a28e9fcbb219fccc15a8b770"
        "Created": "2017-11-13T18:44:05.669293179Z",
        "Scope": "global",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "10.10.3.0/24"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Node1 también puede ver a **demonet**. Esto se debe a que al crear la red global **demonet**, el host guarda la información de red de overlay en Consul, y node1 lee los datos de la nueva red desde Consul. Después, cualquier cambio de **demonet** se sincronizará con node1 y node2

Ahora podemos ejecutar un contenedor conectado a la red overlay.

4.3 Flannel

Flannel es una solución de red de contenedores desarrollada por CoreOS. Flannel asigna una subred a cada host, el contenedor obtiene direcciones IP de esta subred, y estas IPs se pueden enrutar entre hosts y los contenedores se pueden comunicar a través de hosts sin NAT y port mapping.

Este capítulo recrea un entorno experimental como a continuación se muestra:

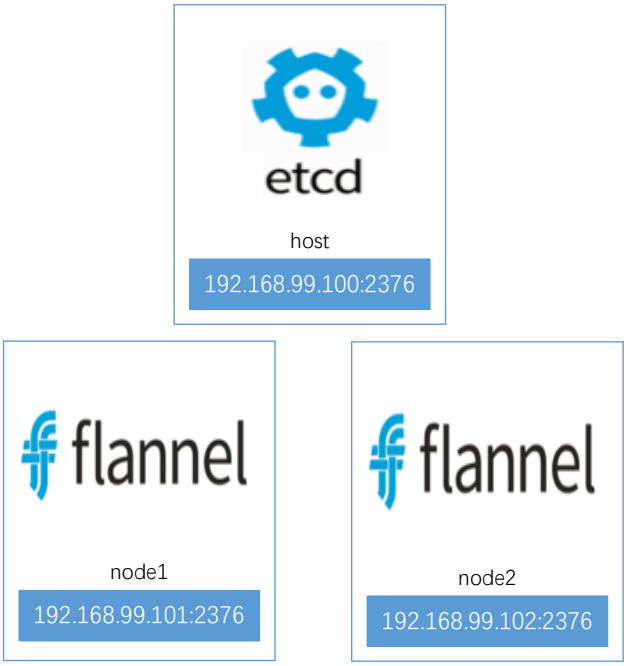


Figura 4-5 El entorno listo y experimental de flannel

```
[ke@ke-virtual-machine ~]$docker-machine ls
NAME     ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        ERRORS
host     -        virtualbox   Running   tcp://192.168.99.100:2376           v17.10.0-ce
node1    -        virtualbox   Running   tcp://192.168.99.101:2376           v17.10.0-ce
node2    -        virtualbox   Running   tcp://192.168.99.102:2376           v17.10.0-ce
```

4.3.1 Instalación y configuración de etcd

Para ello se ejecuta la siguiente secuencia de comandos en 192.168.99.100:

```
ETCD_VER=v2.3.7
DOWNLOAD_URL=https://github.com/coreos/etcd/releases/download
curl -L ${DOWNLOAD_URL}/${ETCD_VER}/etcd-${ETCD_VER}-linux-amd64.tar.gz -o
/tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz
mkdir -p /tmp/test-etcd && tar xzvf /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz -C
/tmp/test-etcd --strip-components=1
cp /tmp/test-etcd/etcd* /usr/local/bin/
```

```
docker@host:~$ sudo sh start
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   620    0   620    0     0   1064      0 --:--:-- --:--:-- --:--:--  1063
100 8347k  100 8347k    0     0   531k      0  0:00:15  0:00:15 --:--:--  635k
etcd-v2.3.7-linux-amd64/Documentation/
etcd-v2.3.7-linux-amd64/Documentation/runtime-configuration.md
etcd-v2.3.7-linux-amd64/Documentation/admin_guide.md
etcd-v2.3.7-linux-amd64/Documentation/tuning.md
etcd-v2.3.7-linux-amd64/Documentation/glossary.md
etcd-v2.3.7-linux-amd64/Documentation/rfc/
etcd-v2.3.7-linux-amd64/Documentation/rfc/v3api.md
etcd-v2.3.7-linux-amd64/Documentation/discovery_protocol.md
etcd-v2.3.7-linux-amd64/Documentation/errorcode.md
etcd-v2.3.7-linux-amd64/Documentation/metrics.md
etcd-v2.3.7-linux-amd64/Documentation/security.md
etcd-v2.3.7-linux-amd64/Documentation/configuration.md
etcd-v2.3.7-linux-amd64/Documentation/docker_guide.md
etcd-v2.3.7-linux-amd64/Documentation/dev/
etcd-v2.3.7-linux-amd64/Documentation/dev/release.md
etcd-v2.3.7-linux-amd64/Documentation/members_api.md
etcd-v2.3.7-linux-amd64/Documentation/auth_api.md
etcd-v2.3.7-linux-amd64/Documentation/backward_compatibility.md
etcd-v2.3.7-linux-amd64/Documentation/platforms/
```

Este script se descarga de github, y se debe almacenar en /usr/local/bin/. Después iniciamos etcd y abrimos el puerto de 2379.

```
etcd -listen-client-urls http://192.168.99.100:2379 -advertise-client-urls http://192.168. 99. 100:2379
```

```
docker@host:~$ etcd -listen-client-urls http://192.168.99.100:2379 -advertise-client-urls http://192.168.99.100:2379
2017-11-15 16:11:44.205914 I | etcdmain: etcd Version: 2.3.7
2017-11-15 16:11:44.207232 I | etcdmain: Git SHA: fd17c91
2017-11-15 16:11:44.208698 I | etcdmain: Go Version: go1.6.2
2017-11-15 16:11:44.210386 I | etcdmain: Go OS/Arch: linux/amd64
2017-11-15 16:11:44.211150 I | etcdmain: setting maximum number of CPUs to 1, total number of available CPUs is 1
2017-11-15 16:11:44.211164 W | etcdmain: no data-dir provided, using default data-dir ./default.etcd
2017-11-15 16:11:44.211510 I | etcdmain: listening for peers on http://localhost:2380
2017-11-15 16:11:44.211550 I | etcdmain: listening for peers on http://localhost:7001
2017-11-15 16:11:44.211581 I | etcdmain: listening for client requests on http://192.168.99.100:2379
2017-11-15 16:11:44.211864 I | etcdserver: name = default
2017-11-15 16:11:44.212620 I | etcdserver: data dir = default.etcd
2017-11-15 16:11:44.212631 I | etcdserver: member dir = default.etcd/member
2017-11-15 16:11:44.212637 I | etcdserver: heartbeat = 100ms
2017-11-15 16:11:44.212643 I | etcdserver: election = 1000ms
2017-11-15 16:11:44.212649 I | etcdserver: snapshot count = 10000
2017-11-15 16:11:44.212658 I | etcdserver: advertise client URLs = http://192.168.99.100:2379
2017-11-15 16:11:44.212666 I | etcdserver: initial advertise peer URLs = http://localhost:2380,http://localhost:7001
2017-11-15 16:11:44.212678 I | etcdserver: initial cluster = default=http://localhost:2380,default=http://localhost:7001
2017-11-15 16:11:44.212894 I | etcdserver: starting member ce2a822cea30bfca in cluster 7e27652122e8b2ae
2017-11-15 16:11:44.212947 I | raft: ce2a822cea30bfca became follower at term 0
2017-11-15 16:11:44.212965 I | raft: newRaft ce2a822cea30bfca [peers: [], term: 0, commit: 0, applied: 0, lastindex: 0, lastterm: 0]
2017-11-15 16:11:44.212977 I | raft: ce2a822cea30bfca became follower at term 1
2017-11-15 16:11:44.213181 I | etcdserver: starting server... [version: 2.3.7, cluster version: to be decided]
2017-11-15 16:11:44.214847 N | etcdserver: added local member ce2a822cea30bfca [http://localhost:2380 http://localhost:7001] to cluster 7e27652122e8b2ae
2017-11-15 16:11:44.613591 I | raft: ce2a822cea30bfca is starting a new election at term 1
2017-11-15 16:11:44.616444 I | raft: ce2a822cea30bfca became candidate at term 2
2017-11-15 16:11:44.616486 I | raft: ce2a822cea30bfca received vote from ce2a822cea30bfca at term 2
2017-11-15 16:11:44.616521 I | raft: ce2a822cea30bfca became leader at term 2
2017-11-15 16:11:44.616545 I | raft: raft.node: ce2a822cea30bfca elected leader ce2a822cea30bfca at term 2
2017-11-15 16:11:44.616950 I | etcdserver: published {Name:default ClientURLs:[http://192.168.99.100:2379]} to cluster 7e27652122e8b2ae
2017-11-15 16:11:44.617052 I | etcdserver: setting up the initial cluster version to 2.3
2017-11-15 16:11:44.617202 N | etcdserver: set the initial cluster version to 2.3
```

Se puede comprobar si etcd está disponible

```
etcdctl --endpoints=192.168.56.101:2379 set foo "bar"
etcdctl --endpoints=192.168.56.101:2379 get foo
```

```
docker@host:~$ etcdctl --endpoints=192.168.99.100:2379 set foo "bar"
bar
docker@host:~$ etcdctl --endpoints=192.168.99.100:2379 get foo
bar
docker@host:~$
```

4.3.2 Configuración de Flannel

Primero descargamos y renombramos la imagen

```
docker pull cloudman6/kube-cross:v1.6.2-2
docker tag cloudman6/kube-cross:v1.6.2-2 gcr.io/google_containers/kube-cross:v1.6.2-2
```

```
docker@host:~$ docker pull cloudman6/kube-cross:v1.6.2-2
v1.6.2-2: Pulling from cloudman6/kube-cross
8b87079b7a06: Pull complete
a3ed95caeb02: Pull complete
1bb8eaf3d643: Pull complete
3e04171ce2e5: Pull complete
9f9bcb469766: Pull complete
70062a14b40c: Pull complete
21c92d96cb05: Pull complete
9fb07296d135: Pull complete
a2284bceb471: Pull complete
bdc030014e01: Pull complete
27e746ee3136: Pull complete
050bdedc4a72: Pull complete
57772763ed27: Pull complete
2a70edb2bbaf: Pull complete
Digest: sha256:ad8d5cac561c70d181ec9a75e00e8d01fd2de02648860d3a93be74f365ae408d
Status: Downloaded newer image for cloudman6/kube-cross:v1.6.2-2
docker@host:~$ docker tag cloudman6/kube-cross:v1.6.2-2 gcr.io/google_containers/kube-cross:v1.6.2-2
```

Después, descargamos el código de Flannel

```
git clone https://github.com/coreos/flannel.git
```

```
docker@host:~$ git clone https://github.com/coreos/flannel.git
Cloning into 'flannel'...
remote: Counting objects: 23877, done.
remote: Compressing objects: 100% (21/21), done.
remote: Total 23877 (delta 7), reused 10 (delta 1), pack-reused 23855
Receiving objects: 100% (23877/23877), 43.70 MiB | 5.45 MiB/s, done.
Resolving deltas: 100% (7889/7889), done.
Checking connectivity... done.
```

Finalmente, copiamos el ejecutable de flanneld a node1 y node2.

```
cd flannel
make dist/flanneld-amd64
docker-machine scp dist/flanneld-amd64 node1:/usr/local/bin/flanneld
docker-machine scp dist/flanneld-amd64 node2:/usr/local/bin/flanneld
```

4.3.3 Almacenamiento de la información de configuración de red de flannel a etcd.

Primero debemos escribir la información de configuración en el archivo flannel-config.json, de la siguiente manera:

```
etcdctl --endpoints=192.168.56.101:2379 set /docker-test/network/config < flannel-config.json
```

```
docker@host:~/flannel$ etcdctl --endpoints=192.168.99.100:2379 set /docker-test/network/config < flannel-config.json
{
  "Network": "10.4.0.0/16",
  "SubnetLen": 24,
  "Backend": {
    "Type": "vxlan"
  }
}
```

Donde `Network` define el grupo de IP de la red como 10.4.0.0/16.

`SubnetLen` especifica que a cada host se le asigne un tamaño de subnet de 24 bits (10.2.X.0/24) y `Backend` indica que los hosts comunican a través de la vxlan

4.3.4 Comenzar flannel

Para ello ejecutamos el siguiente comando en node1 y node2:

```
flanneld -etcd-endpoints=http://192.168.56.101:2379 -iface=enp0s8 -etcd-prefix=/docker-test/network
```

```
root@node1:~# flanneld -etcd-endpoints=http://192.168.99.100:2379 -iface=eth1 -etcd-prefix=/docker-test/network
I1115 17:06:56.663312    2524 main.go:483] Using interface with name eth1 and address 192.168.99.101
I1115 17:06:56.666961    2524 main.go:500] Defaulting external address to interface address (192.168.99.101)
I1115 17:06:56.668186    2524 main.go:235] Created subnet manager: Etcd Local Manager with Previous Subnet: 0.0.0.0/0
I1115 17:06:56.670656    2524 main.go:238] Installing signal handlers
I1115 17:06:56.673906    2524 main.go:348] Found network config - Backend type: vxlan
I1115 17:06:56.674651    2524 vxlan.go:119] VXLAN config: VNI=1 Port=0 GBP=false DirectRouting=false
I1115 17:06:56.686472    2524 local_manager.go:234] Picking subnet in range 10.4.1.0 ... 10.4.255.0
I1115 17:06:56.689023    2524 local_manager.go:220] Allocated lease (10.4.41.0/24) to current node (192.168.99.101)
I1115 17:06:56.691041    2524 main.go:295] Wrote subnet file to /run/flannel/subnet.env
I1115 17:06:56.692688    2524 main.go:299] Running backend.
I1115 17:06:56.693248    2524 vxlan_network.go:56] watching for new subnet leases
I1115 17:06:56.700763    2524 main.go:391] Waiting for 23h0m5.093073531s to renew lease
```

```
root@node2:~# flanneld -etcd-endpoints=http://192.168.99.100:2379 -iface=eth1 -etcd-prefix=/docker-test/network
I1115 17:11:25.302060    4615 main.go:483] Using interface with name eth1 and address 192.168.99.102
I1115 17:11:25.308204    4615 main.go:500] Defaulting external address to interface address (192.168.99.102)
I1115 17:11:25.309507    4615 main.go:235] Created subnet manager: Etcd Local Manager with Previous Subnet: 0.0.0.0/0
I1115 17:11:25.311595    4615 main.go:238] Installing signal handlers
I1115 17:11:25.317344    4615 main.go:348] Found network config - Backend type: vxlan
I1115 17:11:25.318741    4615 vxlan.go:119] VXLAN config: VNI=1 Port=0 GBP=false DirectRouting=false
I1115 17:11:25.336739    4615 local_manager.go:234] Picking subnet in range 10.4.1.0 ... 10.4.255.0
I1115 17:11:25.342276    4615 local_manager.go:220] Allocated lease (10.4.8.0/24) to current node (192.168.99.102)
I1115 17:11:25.344365    4615 main.go:295] Wrote subnet file to /run/flannel/subnet.env
I1115 17:11:25.345764    4615 main.go:299] Running backend.
I1115 17:11:25.347408    4615 vxlan_network.go:56] watching for new subnet leases
I1115 17:11:25.349530    4615 main.go:391] Waiting for 23h0m0.571105205s to renew lease
```

Donde `-etcd-endpoints` especifica la URL de etcd, `-iface` indica la interfaz utilizada para la transferencia de datos entre hosts y `-etcd-prefix` denota la clave de configuración de red de flannel que se almacena en etcd.

Como se presenta en el gráfico arriba eth1 se erige como la interfaz para comunicarse con el host externo, el grupo de redes de flannel es 10.4.1.0/16, y la subnet es 10.4.8.0/24.

Después de que se inicie flanneld, hay algunos cambios en la red interna del nodo1

```
docker@node1:~$ ip addr show flannel.1
6: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether fa:dc:3c:e1:0f:fd brd ff:ff:ff:ff:ff:ff
    inet 10.4.41.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::f8dc:3cff:fee1:ffd/64 scope link
       valid_lft forever preferred_lft forever
docker@node1:~$ ip route
default via 10.0.2.2 dev eth0  metric 1
10.0.2.0/24 dev eth0  proto kernel  scope link  src 10.0.2.15
10.4.8.0/24 via 10.4.8.0 dev flannel.1 onlink
127.0.0.1 dev lo  scope link
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1
192.168.99.0/24 dev eth1  proto kernel  scope link  src 192.168.99.101
docker@node1:~$
```

Como se puede comprobar, se ha creado una nueva interfaz llamada flannel.1 y se ha configurado la primera IP 10.4.41.0 en la subred.

4.3.5 Configurar flannel para la conexión a Docker

Para ello debemos editar el archivo de configuración Docker de host1, /etc/docker/daemon.json, y configurar `--bip` y `--mtu`.

Los valores de estos dos parámetros deben ser los mismos que `FLANNEL_SUBNET` y `FLANNEL_MTU` en el fichero /run/flannel/subnet.env.

```
root@node1:/mnt/sda1/var/lib/boot2docker/etc/docker# cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.4.0.0/16
FLANNEL_SUBNET=10.4.41.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=false
```

Una vez hechos los cambios debemos reiniciar el demonio de Docker.

Docker configurará la IP 10.4.40.1 en el Linux bridge docker0 y agregará la ruta. La topología de red de entorno actual queda como se muestra:

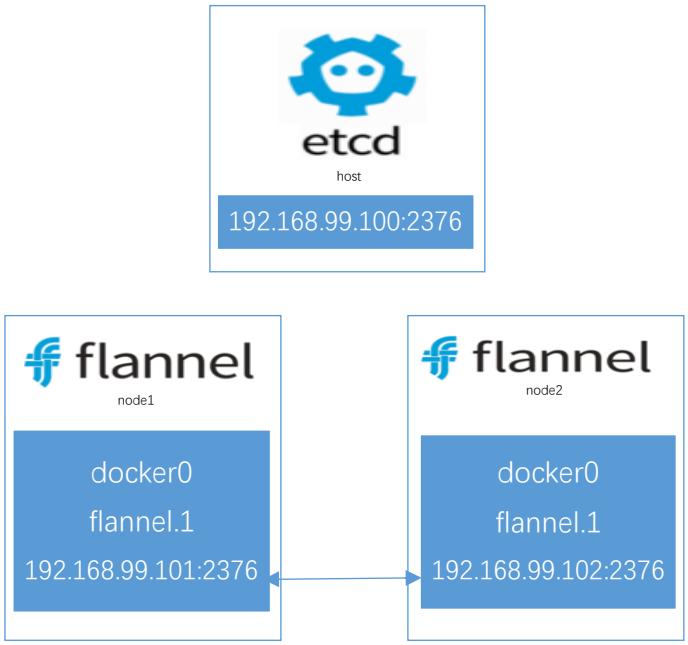


Figura 4-6 La topología de red de entorno de flannel

4.4 Weave

4.4.1 Cómo usar la red Weave

Weave es una solución de red de contenedores desarrollada por Weaveworks. Weave crea una red virtual que conecta contenedores desplegados en múltiples hosts. Para los contenedores, Weave es como un switch de Ethernet, al que todos los contenedores están conectados. Cada contenedor puede comunicarse directamente sin NAT y sin asignación de puertos. Además, el módulo DNS de Weave permite el acceso a los contenedores a través de hostname.

- **Entorno experimental como se muestra a continuación**

Weave no se basa en bases de datos distribuidas (como etcd y Consul) para intercambiar información de red, simplemente ejecuta los componentes de Weave en cada host para construir una red de contenedores de host cruzado. Implementamos Weave en host1 y host2 y los configuramos adecuadamente.

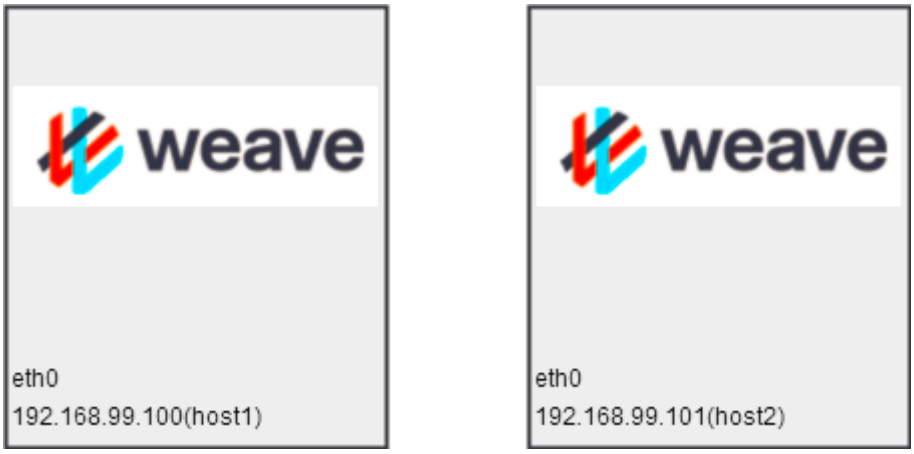


Figura 4-7 El entorno de experimental de Weave

- **Instalar y desplegar Weave**

Para ello ejecutamos el siguiente comando en host1 y host2:

```
curl -L git.io/weave -o /usr/local/bin/weave
chmod a+x /usr/local/bin/weave
```

```
root@host1:~# curl -L git.io/weave -o /usr/local/bin/weave
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100   595    0   595    0     0    383      0 --:--:--  0:00:01 --:--:--  116k
100 52453  100 52453    0     0  20816      0  0:00:02  0:00:02 --:--:--  417k
root@host1:~# chmod a+x /usr/local/bin/weave
```

- **Activamos Weave en host1**

Para ello ejecutamos el comando `weave launch` en el host1 que iniciará el servicio de Weave. Todos los componentes de Weave se basan en contenedores, Weave descarga la última imagen por Docker hub e inicia el contenedor.

```
root@host1:/usr/local/bin# docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
weaveworks/weaveexec    2.0.5               90e2b740a10c        5 weeks ago         108MB
weaveworks/weave        2.0.5               655c700f50d3        5 weeks ago         58.3MB
weaveworks/weavedb      latest              3f610642de09        6 months ago        252B
```

Weave creará una nueva red Docker `weave`:

```
root@host1:/usr/local/bin# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
097449491f19        bridge              bridge              local
7b69abed1b3c        host                host                local
a7a69913da55        none                null                local
ab01b130fe5f        weave               weavemesh           local
```

El rango de IP es 10.32.0.0/12

```
root@host1:/usr/local/bin# docker network inspect weave
[
    {
        "Name": "weave",
        "Id": "ab01b130fe5fc04698be3faaf759a5453a59cf10d4e66d30e90bdc45be20671c",
        "Created": "2017-11-16T09:49:27.538759131Z",
        "Scope": "local",
        "Driver": "weavemesh",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "weavemesh",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.32.0.0/12"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "works.weave.multicast": "true"
        },
        "Labels": {}
    }
]
```

4.4.2 Análisis de estructura de red de Weave

Para ello ejecutamos el contenedor bbox1 en host1:

```
eval $(weave env)
docker run --name bbox1 -itd busybox
```

```
root@host1:/usr/local/bin# eval $(weave env)
root@host1:/usr/local/bin# docker run --name bbox1 -itd busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
0ffadd58f2a6: Pull complete
Digest: sha256:bbc3a03235220b170ba48a157dd097dd1379299370e1ed99ce976df0355d24f0
Status: Downloaded newer image for busybox:latest
5638e401f1e0ba957795375467ecb7e45f568190afb6f2cd66b90e207b7553e0
root@host1:/usr/local/bin#
```

Es importante ejecutar primero `eval$(weave env)` para activar las variables de entorno., que envía los siguientes comandos del docker al weave proxy.

Verificamos la configuración actual de la red del contenedor bbox1.

```
root@host1:/usr/local/bin# docker exec -it bbox1 ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
15: eth0@if16: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
17: ethwe@if18: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1376 qdisc noqueue
    link/ether 1e:68:24:bd:42:47 brd ff:ff:ff:ff:ff:ff
    inet 10.32.0.1/12 scope global ethwe
       valid_lft forever preferred_lft forever
```

bbox1 tiene dos interfaces de red, eth0 y ethwe, donde eth0 es la red puente predeterminada bridge docker0. Por otro lado, ethwe tiene asignada la IP 10.32.0.1/12 por lo que podemos comprobar que ethwe está relacionado con weave, y `ethwe@if18` nos dice que el número de interfaz es 18 que corresponde a ethwe.

Weave también creo el bridge `vethwe-bridge`, como puede comprobarse con `ip -d link` output:

```
    link/ether 08:00:27:ff:ca:7e brd ff:ff:ff:ff:ff:ff promiscuity 0
4: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:11:b3:34 brd ff:ff:ff:ff:ff:ff promiscuity 0
6: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:14:c5:6e:ac brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
7: datapath: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1
    link/ether da:5c:85:bd:96:16 brd ff:ff:ff:ff:ff:ff promiscuity 1
    openvswitch
9: weave: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc noqueue state UP mode DEFAULT group default qlen 1000
    link/ether 96:67:26:29:4c:a2 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
11: vethwe-datapath@vethwe-bridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc noqueue master datapath state UP mode DEFAULT group defaul
t
    link/ether ca:9f:f6:40:73:bc brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
    openvswitch_slave
12: vethwe-bridge@vethwe-datapath: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc noqueue master weave state UP mode DEFAULT group default
    link/ether 3e:39:77:d0:23:0a brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
    bridge_slave
14: vxlan-6784: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 65485 qdisc noqueue master datapath state UNKNOWN mode DEFAULT group default qlen 1000
    link/ether e6:3d:d2:1b:5b:26 brd ff:ff:ff:ff:ff:ff promiscuity 1
    vxlan id 0 srcport 0 0 dstport 6784 nolearning ageing 300
    openvswitch_slave
16: veth57dcf2f@if15: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default
    link/ether a6:69:7d:7b:f5:1c brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
    bridge_slave
18: vethwepl4176@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc noqueue master weave state UP mode DEFAULT group default
    link/ether 12:dc:6e:f8:7e:10 brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
    bridge_slave
```

Weave ha creado varias interfaces nuevas:

1 `vethwe-bridge` y `vethwe-datapath` es un par veth.
2 El master de `vethwe-datapath` es `datapath.`
3 `datapath` es un openvswitch.
4 `vxlan-6784` es un interfaz vxlan cuyo master es `datapath.`

La estructura de red host1 queda como se muestra a continuación.

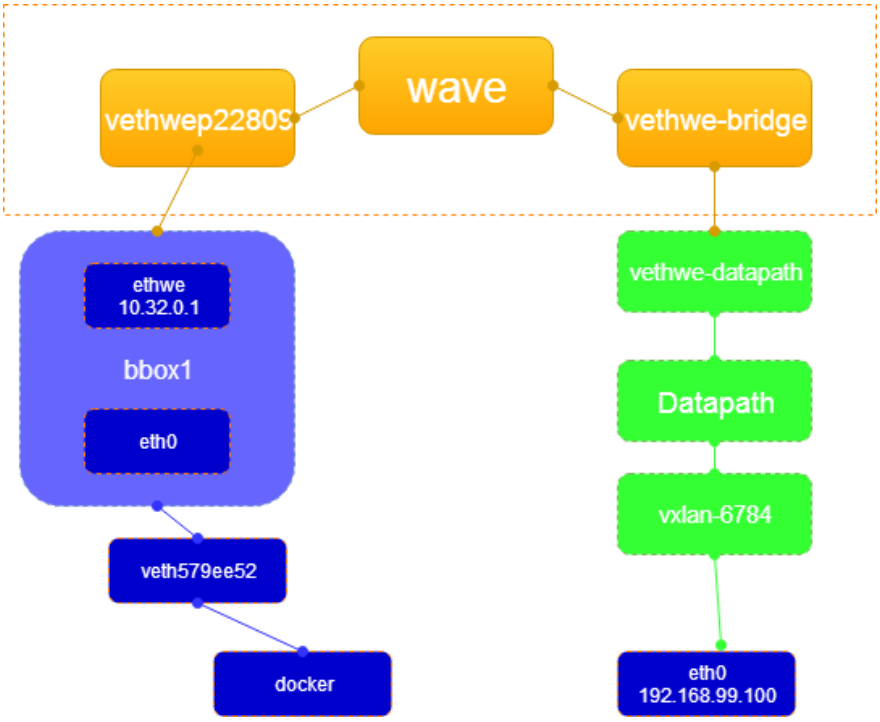


Figura 4-8 La estructura de la red de host1

La red Weave contiene dos conmutadores virtuales Linux bridge `weave` y Open vSwitch `datapath`.

Ejecutamos un contenedor bbox2 nuevamente.

```
docker run --name bbox2 -itd busybox
```

Weave DNS crea el dominio predeterminado weave.local para el contenedor, y bbox1 puede comunicarse con bbox2 directamente a través del hostname.

```
root@host1:~# docker exec bbox1 hostname
bbox1.weave.local
```

```
root@host1:~# docker exec bbox1 ping -c 2 bbox2
PING bbox2 (10.32.0.2): 56 data bytes
64 bytes from 10.32.0.2: seq=0 ttl=64 time=0.054 ms
64 bytes from 10.32.0.2: seq=1 ttl=64 time=0.177 ms

--- bbox2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.054/0.115/0.177 ms
```

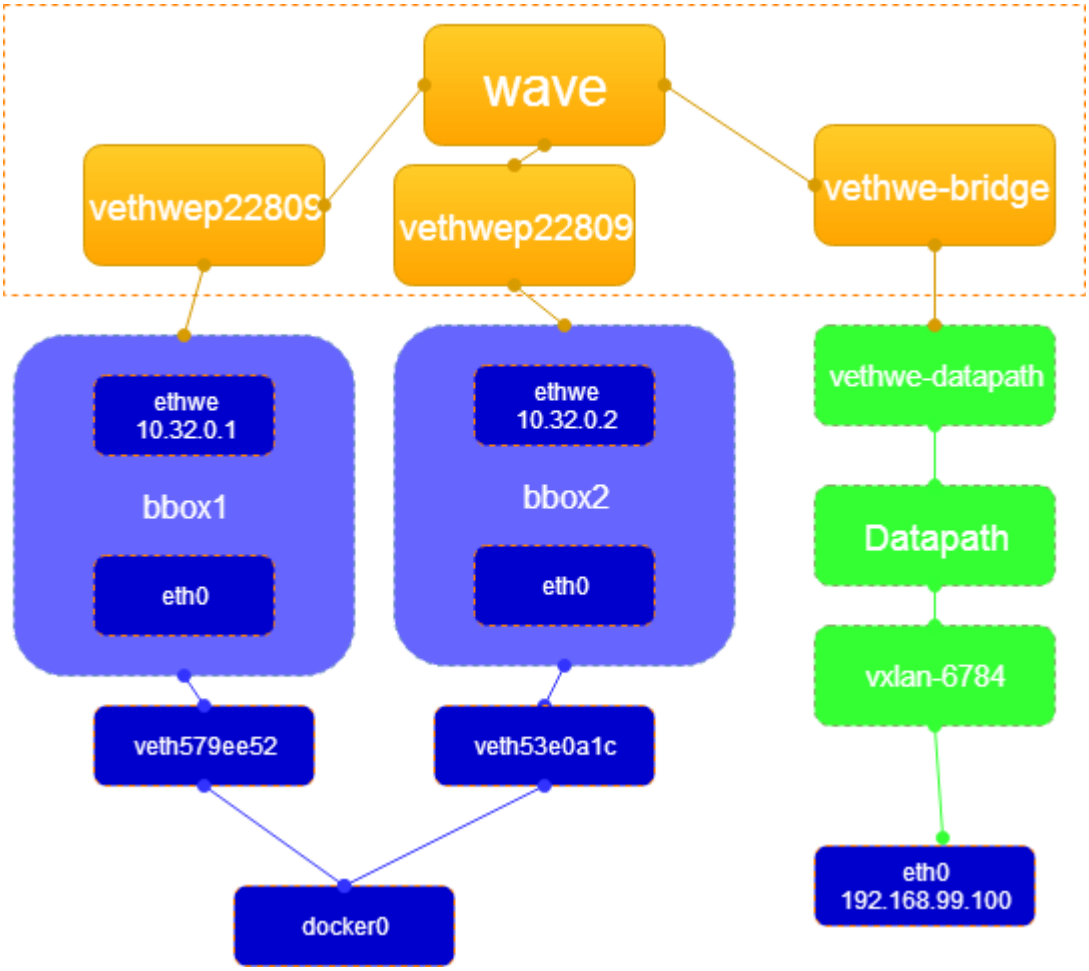


Figura 4-9 La estructura de la red de host1 y host2

4.4.3 Comunicación y aislamiento en Weave

En Primer lugar ejecutamos en el host2 el siguiente comando:

```
weave launch 192.168.99.100
```

Aquí se debe especificar la IP del host1 para que el host1 y el host2 puedan unirse a la misma red de weave.

Después ejecutamos el contenedor bbox3:

```
root@host2:~# eval $(weave env)
root@host2:~# docker run --name bbox3 -itd busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
0ffadd58f2a6: Pull complete
Digest: sha256:bbc3a03235220b170ba48a157dd097dd1379299370e1ed99ce976df0355d24f0
Status: Downloaded newer image for busybox:latest
7546549ea5c338b85cb598a3bececb0f48f4a148aa0db146b1a890061561f8bf
```

- **Conectividad de red de weave**

bbox3 puede hacer ping a bbox1 y bbox2 directamente.

```
root@host2:~# docker exec bbox3 ping -c 2 bbox1
PING bbox1 (10.32.0.1): 56 data bytes
64 bytes from 10.32.0.1: seq=0 ttl=64 time=3.707 ms
64 bytes from 10.32.0.1: seq=1 ttl=64 time=4.294 ms

--- bbox1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 3.707/4.000/4.294 ms
```

A través del túnel VxLAN entre host1 y host2, los tres contenedores están lógicamente en la misma LAN y, por supuesto, pueden comunicarse directamente

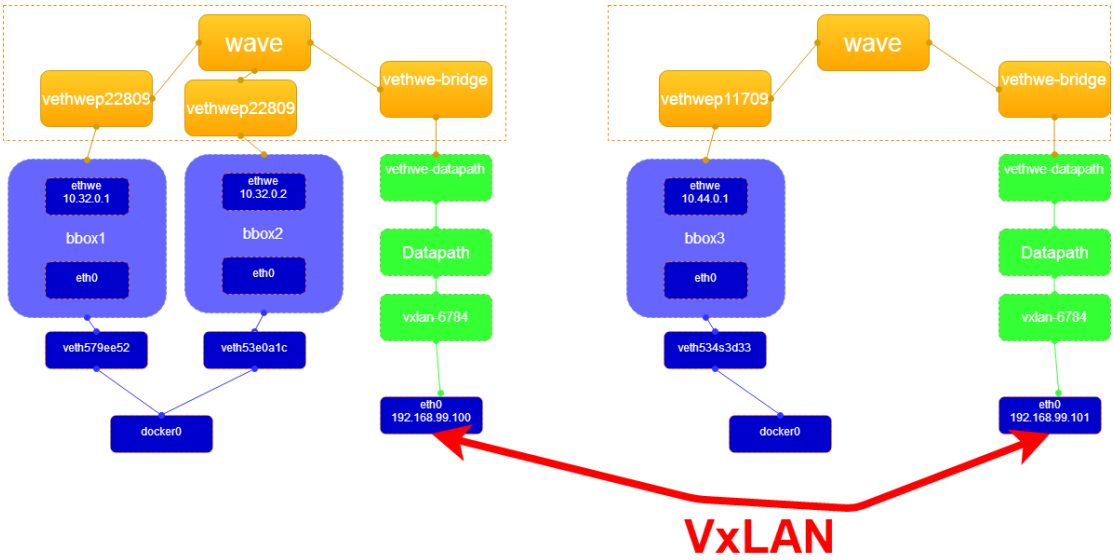


Figura 4-10 La estructura de la red de host1 host2 y host3

```
root@host2:~# docker exec bbox3 ip route
default via 172.17.0.1 dev eth0
10.32.0.0/12 dev ethwe scope link  src 10.44.0.0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2
224.0.0.0/4 dev ethwe scope link
```

- **Aislamiento de red de weave**

De forma predeterminada, Weave usa una subred (por ejemplo, 10.32.0.0/12). Todos los contenedores de los hosts obtienen IPs de este espacio de direcciones porque la misma subred permite que el contenedor se comunique directamente. Si desea lograr el aislamiento de red, puede asignar diferentes IP de subred al contenedor a través de la variable de entorno WEAVE_CIDR, por ejemplo:

```
root@host1:~# docker run -e WEAVE_CIDR=net:10.32.2.0/24 -ti busybo
x
/ # ip r
default via 172.17.0.1 dev eth0
10.32.2.0/24 dev ethwe scope link  src 10.32.2.1
172.17.0.0/16 dev eth0 scope link  src 172.17.0.4
224.0.0.0/4 dev ethwe scope link
/ # ping -c 2 bbox1
PING bbox1 (10.32.0.1): 56 data bytes

--- bbox1 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
/ #
```

Aquí la función de `WEAVE_CIDR = net:10.32.2.0/24` asigna la IP 10.32.2.2 al contenedor. Como 10.32.0.0/12 y 10.32.2.0/24 están en subredes diferentes, no es posible conectar con bbox1.

4.4.4 Comunicación con redes externas

Para unirse al host de weave, ejecutar `weave expose.`

```
root@host1:~# weave expose
10.32.0.3
```

Esta IP e configurará en el bridge de Weave del host1

```
root@host1:~# ip addr show weave
8: weave: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc noqueue state UP group default qlen 1000
    link/ether 96:67:26:29:4c:a2 brd ff:ff:ff:ff:ff:ff
    inet 10.32.0.3/12 scope global weave
       valid_lft forever preferred_lft forever
    inet6 fe80::9467:26ff:fe29:4ca2/64 scope link
       valid_lft forever preferred_lft forever
```

```
root@host1:~# ping -c 2 10.32.0.1
PING 10.32.0.1 (10.32.0.1): 56 data bytes
64 bytes from 10.32.0.1: seq=0 ttl=64 time=0.545 ms
64 bytes from 10.32.0.1: seq=1 ttl=64 time=0.143 ms

--- 10.32.0.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.143/0.344/0.545 ms
```

Luego para otros hosts que no tienen Weave accedan a bbox1 y bbox3, simplemente apuntamos a la puerta de enlace del host1. Por ejemplo, agregue la siguiente ruta a 192.168.99.100.

```
ip route add 10.32.0.0/12 via 192.168.56.104
```

```
[root@ubuntu ~]#
[root@ubuntu ~]# ip route
default via 10.0.2.1 dev eth0
10.0.2.0/24 dev eth0  proto kernel  scope link  src 10.0.2.4
10.1.24.0/24 dev docker0  proto kernel  scope link  src 10.1.24.1
10.32.0.0/12 via 192.168.56.104 dev eth1
192.168.56.0/24 dev eth1  proto kernel  scope link  src 192.168.56.101
192.168.122.0/24 dev virbr0  proto kernel  scope link  src 192.168.122.1
[root@ubuntu ~]#
[root@ubuntu ~]# ping 10.44.0.0
PING 10.44.0.0 (10.44.0.0) 56(84) bytes of data.
64 bytes from 10.44.0.0: icmp_seq=1 ttl=63 time=0.774 ms
64 bytes from 10.44.0.0: icmp_seq=2 ttl=63 time=0.774 ms
```

4.5 Calico

Calico es una solución de red virtual pura de tres niveles. Calico asigna una IP para cada contenedor. Cada host es un enrutador que conecta contenedores de diferentes hosts. A diferencia de VxLAN, Calico no tiene extra paquetes, no requiere NAT y la asignación de puertos, la escalabilidad y el rendimiento son buenos.

4.5.1 Implementar la red de Calico

- **Entorno experimental de Calico**

Calico confía en etcd para compartir e intercambiar información entre diferentes hosts, almacenando el estado de la red Calico. Vamos a ejecutar etcd en el host 192.168.56.101. Cada host de la red de Calico necesita ejecutar los componentes de Calico para proporcionar administración de la interfaz del contenedor, enrutamiento dinámico, ACL dinámicas, estado de los informes y más.

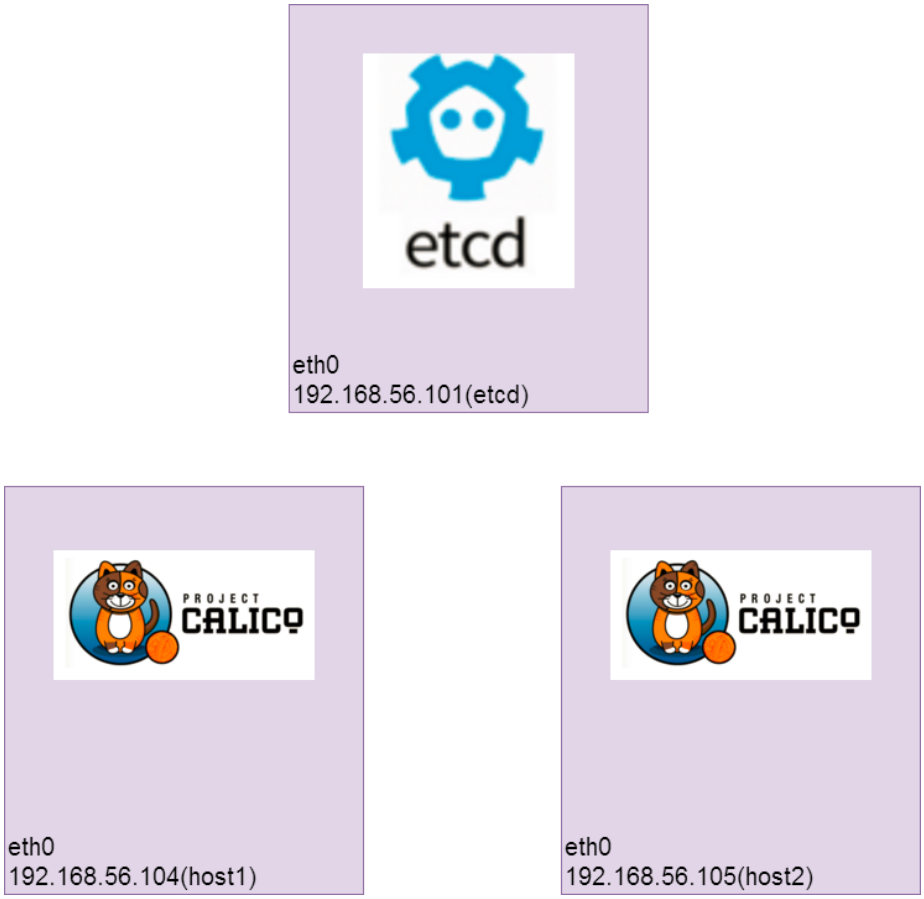


Figura 4-11 Ambiente experimental de Calico

- **Inicio de etcd**

En el host 192.168.56.101 ejecutamos el siguiente comando para iniciar etcd:

```
etcd-listen-client-urls      http://192.168.56.101:2379      -advertise-client-urls
http://192.168.56.101:2379
```

Modificamos el archivo de configuración del demonio deDocker para host1 y host2 y reiniciamos el demonio de Docker.

- **Implementar calico**

Primero procedemos a descargar calicoctl:

```
wget                                                                    -O
/usr/local/bin/calicoctl https://github.com/projectcalico/calicoctl/releases/downlo
ad/v1.0.2/calicoctl
chmod +x calicoctl
```

Iniciamos calico en host1 y host2:

```
root@host1:~#
root@host1:~# calicoctl node run
Running command to load modules: modprobe -a xt_set ip6_tables
Enabling IPv4 forwarding ①
Enabling IPv6 forwarding
Increasing conntrack limit
Removing old calico-node container (if running).
Running the following command to start calico-node: ②

docker run --net=host --privileged --name=calico-node -d --restart=always -e NODENAME=host1 -e CALICO_
NETWORKING_BACKEND=bird -e NO_DEFAULT_POOLS= -e CALICO_LIBNETWORK_ENABLED=true -e CALICO_LIBNETWORK_IF
PREFIX=cali -e ETCD_ENDPOINTS=http://192.168.56.101:2379 -e ETCD_AUTHORITY= -e ETCD_SCHEME= -v /var/ru
n/calico:/var/run/calico -v /lib/modules:/lib/modules -v /var/log/calico:/var/log/calico -v /run/docke
r/plugins:/run/docker/plugins -v /var/run/docker.sock:/var/run/docker.sock calico/node:v1.0.1

Image may take a short time to download if it is not available locally.
Container started, checking progress logs.
Waiting for etcd connection... ③
Using configured IPv4 address: 192.168.56.104
No IPv6 address configured
Using global AS number
Calico node name:  host1
CALICO_LIBNETWORK_ENABLED is true - start libnetwork service
Calico node started successfully ④
root@host1:~#
```

① Establecemos una red de host.

② Descargamos e iniciamos el contenedor calico-node, y calico se ejecuta como un contenedor.

③ Conectamos etcd.

- **Creación de una red en Calico**

Para crear una red de Calico llamada cal_ent1, ejecutamos el siguiente comando en host1 o host2:

```
docker network create --driver calico --ipam-driver calico-ipam cal_net1
```

Donde, `--driver calico` indica el driver de libicotwork de CNM (Container Network Mode) usando calico y especificamos el IPAM driver para IP mediante la opción `--ipam-driver calico-ipam`.

```
root@host1:~#
root@host1:~# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c609e2f471b8        bridge              bridge              local
f54f08f7eb1a        cal_net1            calico              global
e5b3c2e071e3        host                host                local
c16af7d078b1        none                null                local
root@host1:~#
```

4.5.2 Estructura de las redes de Calico

Primero ejecutamos el contenedor bbox1 en el host1 y lo conectamos a cal_net1:

```
docker container run --net cal_net1 --name bbox1 -tid busybox
```

```
root@host1:~#
root@host1:~# docker exec bbox1 ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
10: cali0@if11  <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
    inet 192.168.119.2/32 scope global cali0
       valid_lft forever preferred_lft forever
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
root@host1:~#
```

`cali0` es calico interface, la IP asignada es `192.168.119.2`, cali0 corresponde a la interfaz cali5f744ac07f0 del host1 número 11.

host1 actuará como un enrutador responsable de reenviar el paquete destinado a bbox1.

```
root@host1:~#
root@host1:~# ip route
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3  proto kernel  scope link  src 10.0.2.5
10.2.79.0/24 dev docker0  proto kernel  scope link  src 10.2.79.1 linkdown
192.168.56.0/24 dev enp0s8  proto kernel  scope link  src 192.168.56.104
blackhole 192.168.119.0/26  proto bird
192.168.119.2 dev cali5f744ac07f0  scope link
192.168.122.0/24 dev virbr0  proto kernel  scope link  src 192.168.122.1 linkdown
```

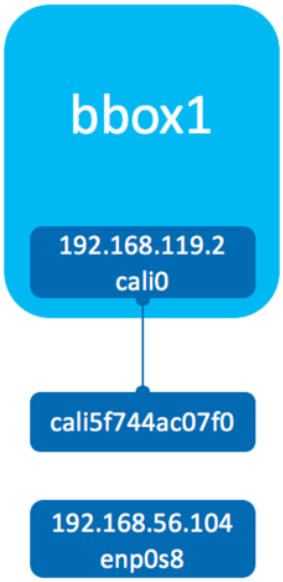


Figura 4-12 La estructura de la red de host1

A continuación, ejecutamos el contenedor bbox2 en host2 y nos conectamos a cal_net1:

```
docker container run --net cal_net1 --name bbox2 -tid busybox
```

La IP es `192.168.183.65`

```
root@host2:~#
root@host2:~# docker exec bbox2 ip address show cali0
10: cali0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
    inet 192.168.183.65/32 scope global cali0
       valid_lft forever preferred_lft forever
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
root@host2:~#
```

Agregamos al host2 dos rutas:

```
root@host2:~#
root@host2:~# ip route
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3  proto kernel  scope link  src 10.0.2.6
10.2.17.0/24 dev docker0  proto kernel  scope link  src 10.2.17.1 linkdown
192.168.56.0/24 dev enp0s8  proto kernel  scope link  src 192.168.56.105
192.168.119.0/26 via 192.168.56.104 dev enp0s8  proto bird
192.168.122.0/24 dev virbr0  proto kernel  scope link  src 192.168.122.1 linkdown
blackhole 192.168.183.64/26  proto bird
192.168.183.65 dev calif666a18df71  scope link
root@host2:~#
```

Del mismo modo, host1 agrega automáticamente la ruta a 192.168.183.64/26.

```
root@host1:~#
root@host1:~# ip route
default via 10.0.2.1 dev enp0s3
10.0.2.0/24 dev enp0s3  proto kernel  scope link  src 10.0.2.5
10.2.79.0/24 dev docker0  proto kernel  scope link  src 10.2.79.1 linkdown
192.168.56.0/24 dev enp0s8  proto kernel  scope link  src 192.168.56.104
blackhole 192.168.119.0/26  proto bird
192.168.119.2 dev cali5f744ac07f0 scope link
192.168.122.0/24 dev virbr0  proto kernel  scope link  src 192.168.122.1 linkdown
192.168.183.64/26 via 192.168.56.105 dev enp0s8  proto bird
root@host1:~#
```

4.5.3 La conectividad predeterminada de Calico

Probamos la conectividad desde bbox1 a bbox2:

```
root@host1:~#
root@host1:~# docker exec bbox1 ping -c 2 bbox2
PING bbox2 (192.168.183.65): 56 data bytes
64 bytes from 192.168.183.65: seq=0 ttl=62 time=0.367 ms
64 bytes from 192.168.183.65: seq=1 ttl=62 time=0.446 ms
```

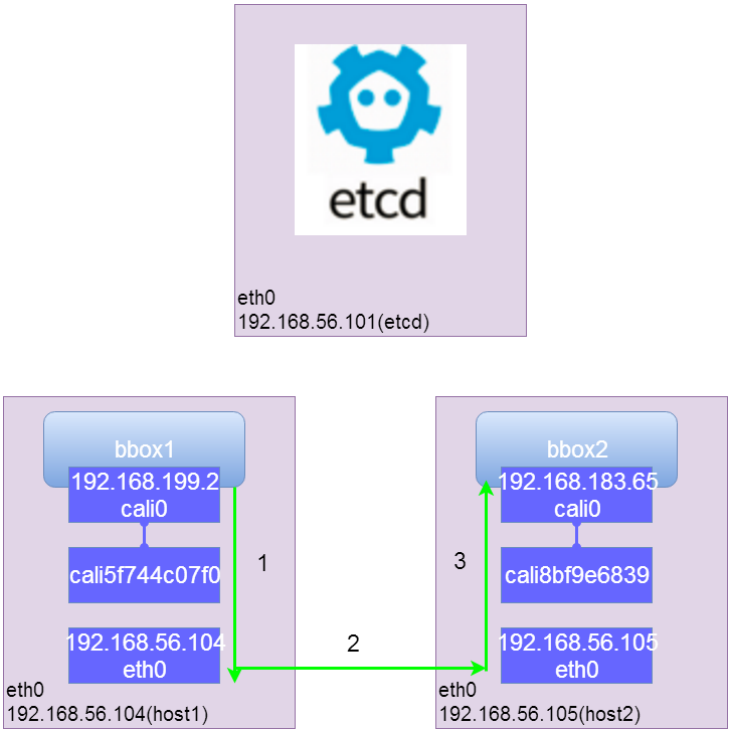


Figura 4-13 La estructura de la red de host1 y host2

① De acuerdo con la tabla de enrutamiento de bbox1, el paquete enviado desde cal0.

② Los datos llegan a host1 a través del par veth, verifican la tabla de enrutamiento y los datos se envían a host2 por eth0 (192.168.56.105).

③ Por otra parte host2 recibe el paquete, lo envía a calic8bf9e68397 de acuerdo con la tabla de enrutamiento, y luego llega a bbox2 a través del par veth cali0.

```
root@host1:~#
root@host1:~# docker exec bbox1 ip route
default via 169.254.1.1 dev cali0
169.254.1.1 dev cali0
root@host1:~#
```

A continuación, observamos la conectividad entre diferentes redes de Calico.

Crear cal_net2:

```
docker network create --driver calico --ipam-driver calico-ipam cal_net2
```

Para ello se puede ejecutar sobre el contenedor bbox3 en host1, la información del adaptador virtual cal_net2:

```
root@host1:~#
root@host1:~# docker exec bbox3 ip address show cali0
18: cali0@if19: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
    inet 192.168.119.5/32 scope global cali0
       valid_lft forever preferred_lft forever
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
root@host1:~#
```

A continuación procedemos a verificar el aislamiento entre bbox1 y bbox3.

```
root@host1:~#
root@host1:~# docker exec bbox1 ping -c 2 192.168.119.5
PING 192.168.119.5 (192.168.119.5): 56 data bytes

--- 192.168.119.5 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
root@host1:~#
```

Tanto bbox1 como bbox3 están en el host1 y están en una subred de 192.168.119.0/26, pero al pertenecer cada contenedor a diferentes redes de Calico, no existe conectividad de forma predeterminada. La política predeterminada de aislamiento de Calico es que los contenedores sólo pueden comunicarse con contenedores en la misma red de Calico.

4.5.4 Personalización de la Política de la red de Calico

Calico permite a los usuarios definir reglas de políticas flexibles y decontrol detallado del tráfico de contenedores. El siguiente ejemplo muestra cómo puede realizarse para el caso particular de permitir la conexión a través de un único puerto.

1) Primero creamos una nueva red Calico `cal_web` y desplegamos un contenedor con un servidor web (imagen httpd) activo con el nombre `web1`

2) Definimos una política que permita que un contenedor en `cal_net2` acceda al puerto 80 de `web1`

```
docker network create --driver calico --ipam-driver calico-ipam cal_web
docker container run --net cal_web --name web1 -d httpd
```

La IP de web1 es `192.168.119.7`

```
root@host1:~#
root@host1:~# docker container exec web1 ip address show cali0
22: cali0@if23: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
    inet 192.168.119.7/32 scope global cali0
       valid_lft forever preferred_lft forever
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
root@host1:~#
```

Antes de definir la política, bbox3 aún no puede acceder al puerto 80 de web1.

```
root@host1:~# docker container exec bbox3 wget 192.168.119.7
Connecting to 192.168.119.7 (192.168.119.7:80)
wget: can't connect to remote host (192.168.119.7): Connection timed out
root@host1:~#
```

Creamos el archivo de política web.yml

```
- apiVersion: v1
  kind: profile
  metadata:
    name: cal_web ①
  spec:
    ingress:
    - action: allow
      protocol: tcp
      source:
        tag: cal_net2 ②
      destination:
        ports:
        - 80 ③
```

1 Perfil y red `cal_web` tiene mismo nombre de red, todos los contenedores de `cal_web` (web1) usarán este perfil en la política

2 `ingress` permite el acceso al contenedor `cal_net2` (bbox3)

3 Sólo permite la conexión al puerto `80`

`calicoctl apply -f web.yml`

Ahora bbox3 puede acceder al servicio http de web1.

```
root@host1:~#
root@host1:~# docker container exec bbox3 wget 192.168.119.7
Connecting to 192.168.119.7 (192.168.119.7:80)
index.html           100% |*******************************|    45   0:00:00 ETA
```

Sin embargo, no se puede hacer ping, porque solo se abrió el puerto 80.

```
root@host1:~#
root@host1:~# docker container exec bbox3 ping -c 2 192.168.119.7
PING 192.168.119.7 (192.168.119.7): 56 data bytes

--- 192.168.119.7 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
root@host1:~#
```

4.5.5 Personalización del grupo de IP de Calico

Primero definimos un ¨pool¨ de IPs, por ejemplo:

```
cat << EOF | calicoctl create -f -
- apiVersion: v1
 kind: ipPool
 metadata:
   cidr: 17.2.0.0/16
EOF
```

Usamos este IP pool para crear una red de Calico.

```
docker network create --driver calico --ipam-driver calico-ipam --subnet=17.2.0.0/16 my_net
```

El contenedor ahora tiene direcciones IP asignadas en la subred especificada

```
root@host1:~#
root@host1:~# docker container run --net my_net -ti busybox
/ #
/ # ip address show cali0
26: cali0@if27: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
    inet 17.2.119.1/32 scope global cali0
       valid_lft forever preferred_lft forever
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
/ #
```

4.6 Mesos

Con ayuda de **digitalocean** (Es una plataforma de cloud IaaS pública) simulamos el entorno experimental, E implementamos tres máquinas virtuales de sistema de Ubuntu, host (192.168.56.103), node1 (192.168.56.1) y node2 (192.168.56.102), como se presenta en la figura 4-14, a los que nos conectamos mediante ssh.

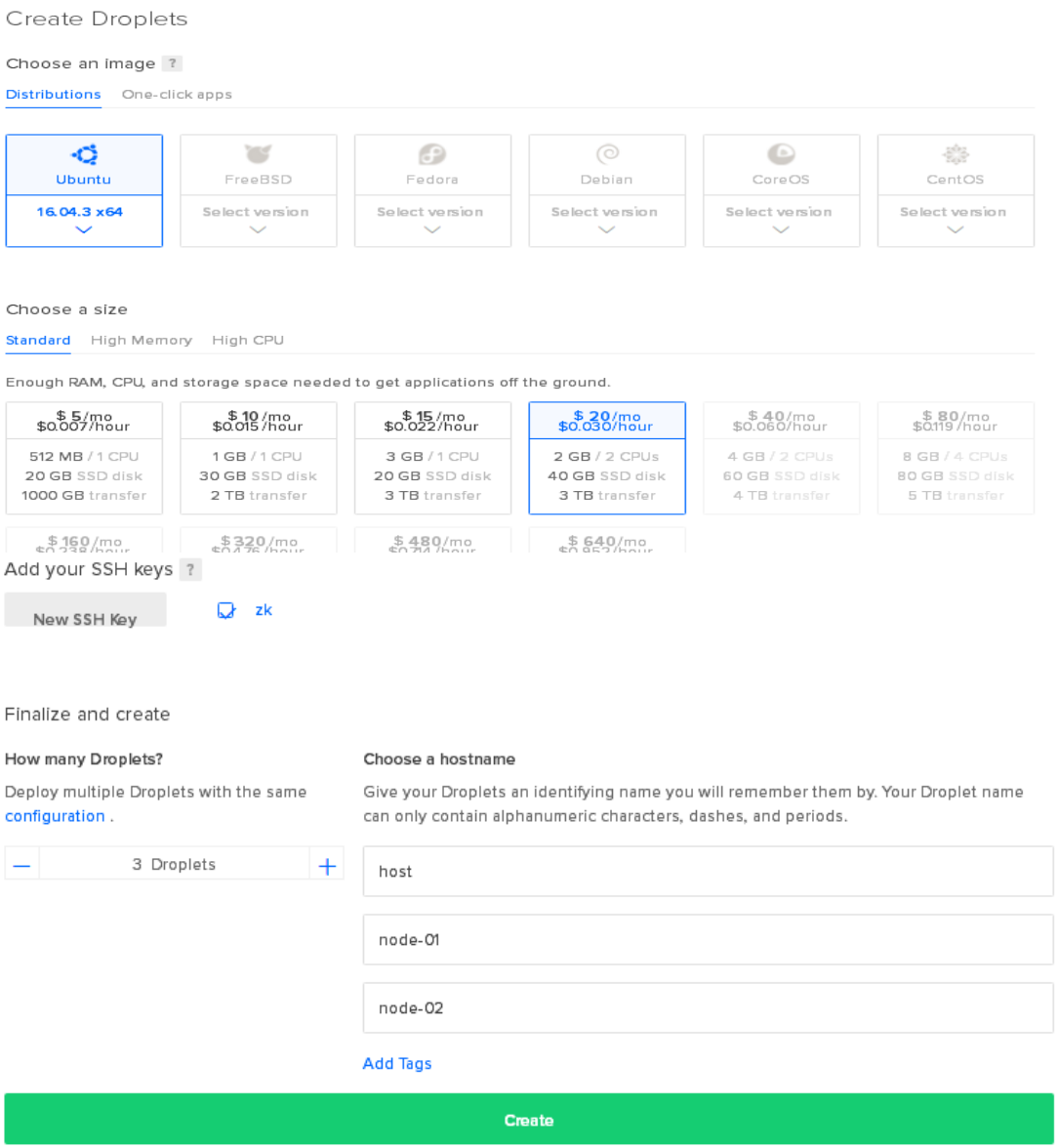


Figura 4-14 La página de web de digitalocean para crear las máquinas virtuales

4.6.1 Instalación los imágenes a los contenedores

Procedemos a realizar una instalación basada en contenedores para facilitar su despliegue. Para ello, descargamos las siguientes imágenes de Docker:

```
docker pull mesoscloud/zookeepe
docker pull mesoscloud/mesos-master
docker pull mesoscloud/mesos-slave
docker pull mesoscloud/marathon
docker pull davidcaste/alpine-tomcat
```

Ponemos en marcha el sistema, ejecutando zookeeper, mesos-master, mesos-slave y marathon.

Ejecutar zookeeper:

```
docker run -d \
-e MYID=1 \
-e SERVERS=192.168.56.103,192.168.56.101,192.168.56.102 \
--name=zookeeper \
--net=host \
--restart=always \
mesoscloud/zookeeper
```

Ejecutar mesos-master

```
docker run -d \
-e MESOS_HOSTNAME=192.168.56.103 \
-e MESOS_IP=192.168.56.103 \
-e MESOS_QUORUM=2 \
-e MESOS_ZK=zk://192.168.56.103:2181,192.168.56.101:2181,192.168.56.102:2181/mesos \
--name mesos-master \
--net host \
--restart=always \
mesoscloud/mesos-master
```

Ejecutar mesos-slave:

```
docker run -d \
-e MESOS_HOSTNAME=192.168.56.103 \
-e MESOS_IP=192.168.56.103 \
-e
MESOS_MASTER=zk://192.168.56.103:2181,192.168.56.101:2181,192.168.56.102:2181/mesos
\
-v /sys/fs/cgroup:/sys/fs/cgroup \
-v /var/run/docker.sock:/var/run/docker.sock \
--name mesos-slave \
--net host \
--privileged \
--restart=always \
mesoscloud/mesos-slave
```

Ejecutar marathon:

```
docker run -d \
-e MARATHON_HOSTNAME=192.168.56.103 \
-e MARATHON_HTTPS_ADDRESS=192.168.56.103 \
-e MARATHON_HTTP_ADDRESS=192.168.56.103 \
-e
MARATHON_MASTER=zk://192.168.56.103:2181,192.168.56.101:2181,192.168.56.102:2181/me
sos \
-e
MARATHON_ZK=zk://192.168.56.103:2181,192.168.56.101:2181,192.168.56.102:2181/marath
on \
--name marathon \
--net host \
--restart=always \
mesoscloud/marathon
```

4.6.2 Publicación de una aplicación a través de Marathon

En el primer host accedemos al servicio Marathon en la URL http://192.168.56.103:8080. Para ello, desde el interfaz gráfico de Marathon,

creamos la aplicación, haciendo clic en “Create Application”, y completando la información de la aplicación de acuerdo con los siguientes datos en formato JSON:

```
{
  "id": "tomcat",
  "cmd": "/opt/tomcat/bin/catalina.sh run",
  "cpus": 1,
  "mem": 512,
  "disk": 512,
  "instances": 1,
  "container": {
    "docker": {
      "image": "davidcaste/alpine-tomcat",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 8080,
          "protocol": "tcp",
          "name": null
        }
      ],
      "parameters": []
    },
    "type": "DOCKER",
    "volumes": [
      {
        "hostPath": "/var/dockertest/logs",
        "containerPath": "/logs",
        "mode": "RW"
      }
    ]
  },
  "env": {},
  "labels": {},
  "healthChecks": []
}
```

4.6.3 Verificación del estado de la aplicación

La figura 4-15 muestra la página principal del interfaz gráfico de marathon (http://192.168.56.103:8080)

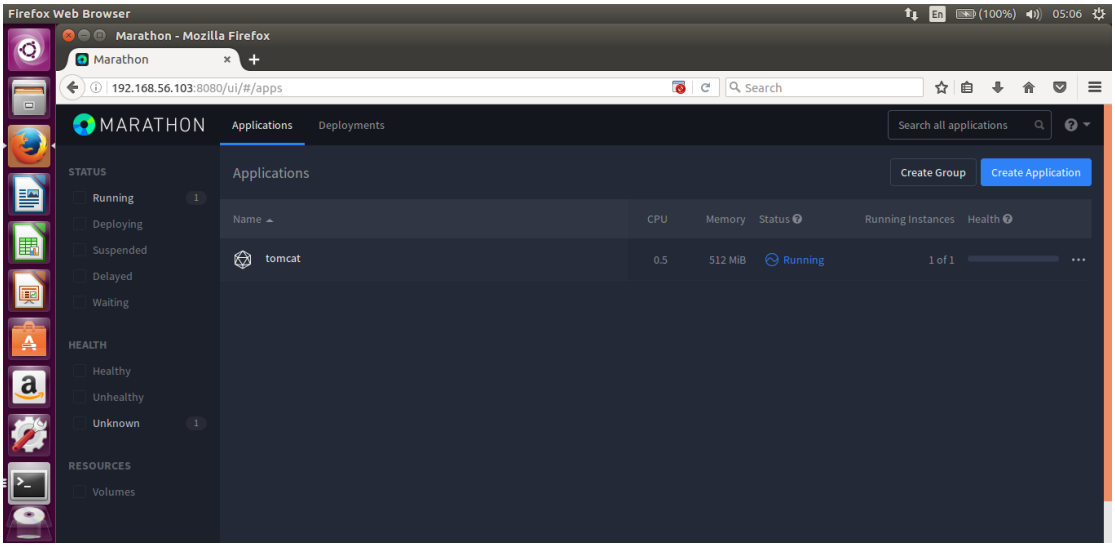


Figura 4-15 La página de web de marathon

Tras hacer clic para entrar en la página de la aplicación, comprobamos que solo hay una instancia (figura 4-16).

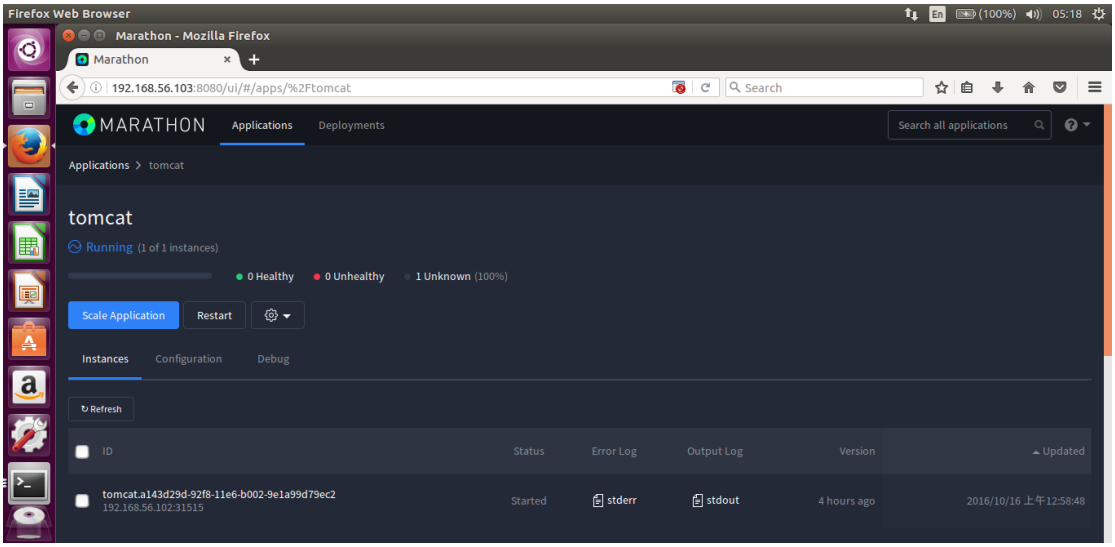


Figura 4-16 La página de web de la aplicación de marathon

Escalamos la application en tres instancias:

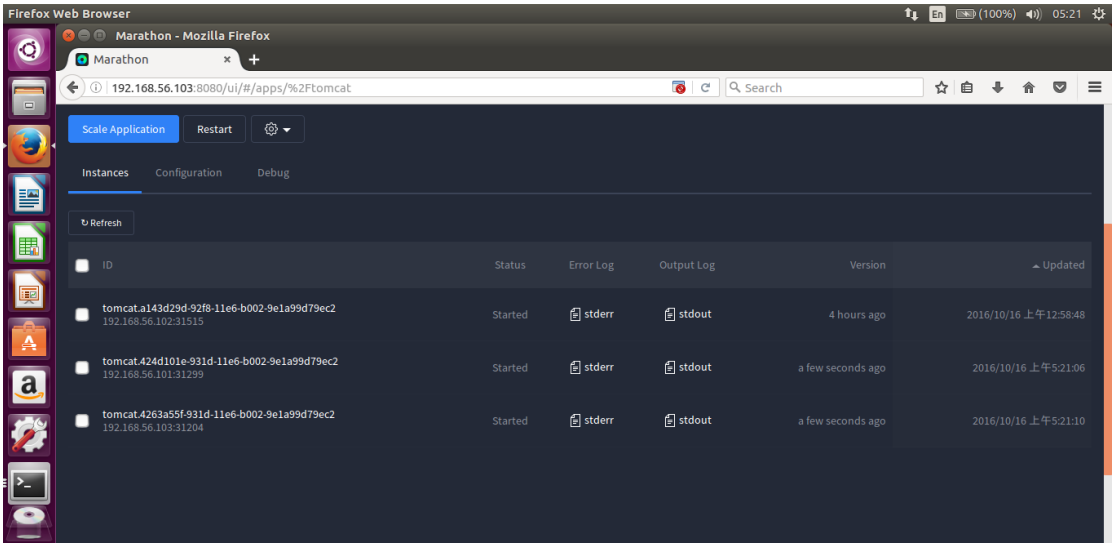


Figura 4-17 La página de web de los instances

Podemos comprobar en la página de "slaves" dentro del interfaz gráfico de mesos que hay tres mesos-slave que están trabajando:

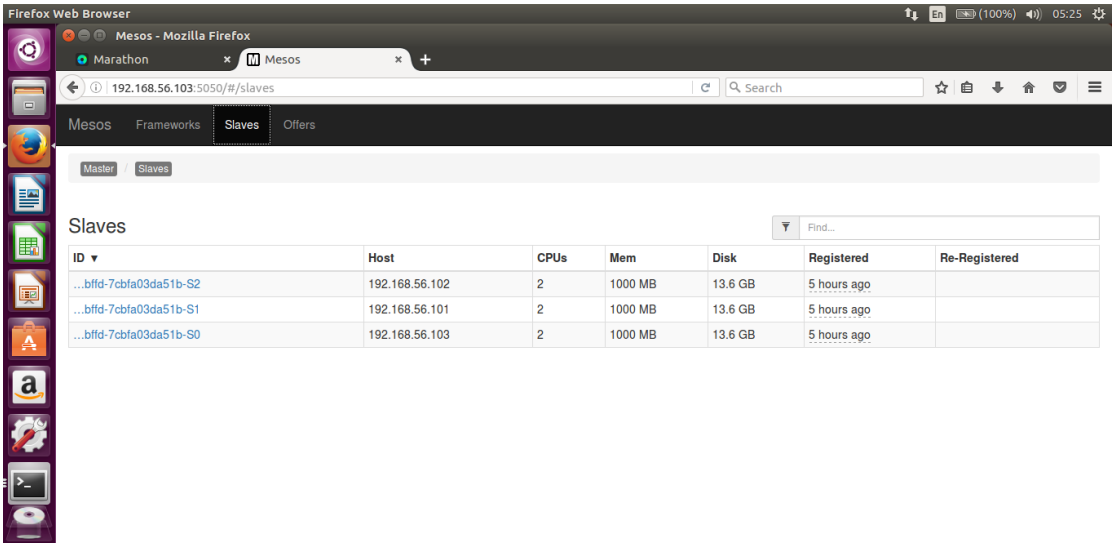


Figura 4-18 La página de web de mesos-slaves

Las redes anteriores (flannel, Calico, Docker Network, etc.) se soportan desde Mesos utilizando el tipo de red "USER" en la descripción del trabajo.

4.7 Kubernetes

4.7.1 Ambiente experimental

Usamos **digitalocean** para simular el entorno experimental e implementamos tres máquinas virtuales, ubuntu-2gb-lon1-01 (master), ubuntu-2gb-lon1-02(node1), ubuntu-2gb-lon1-03(node2).

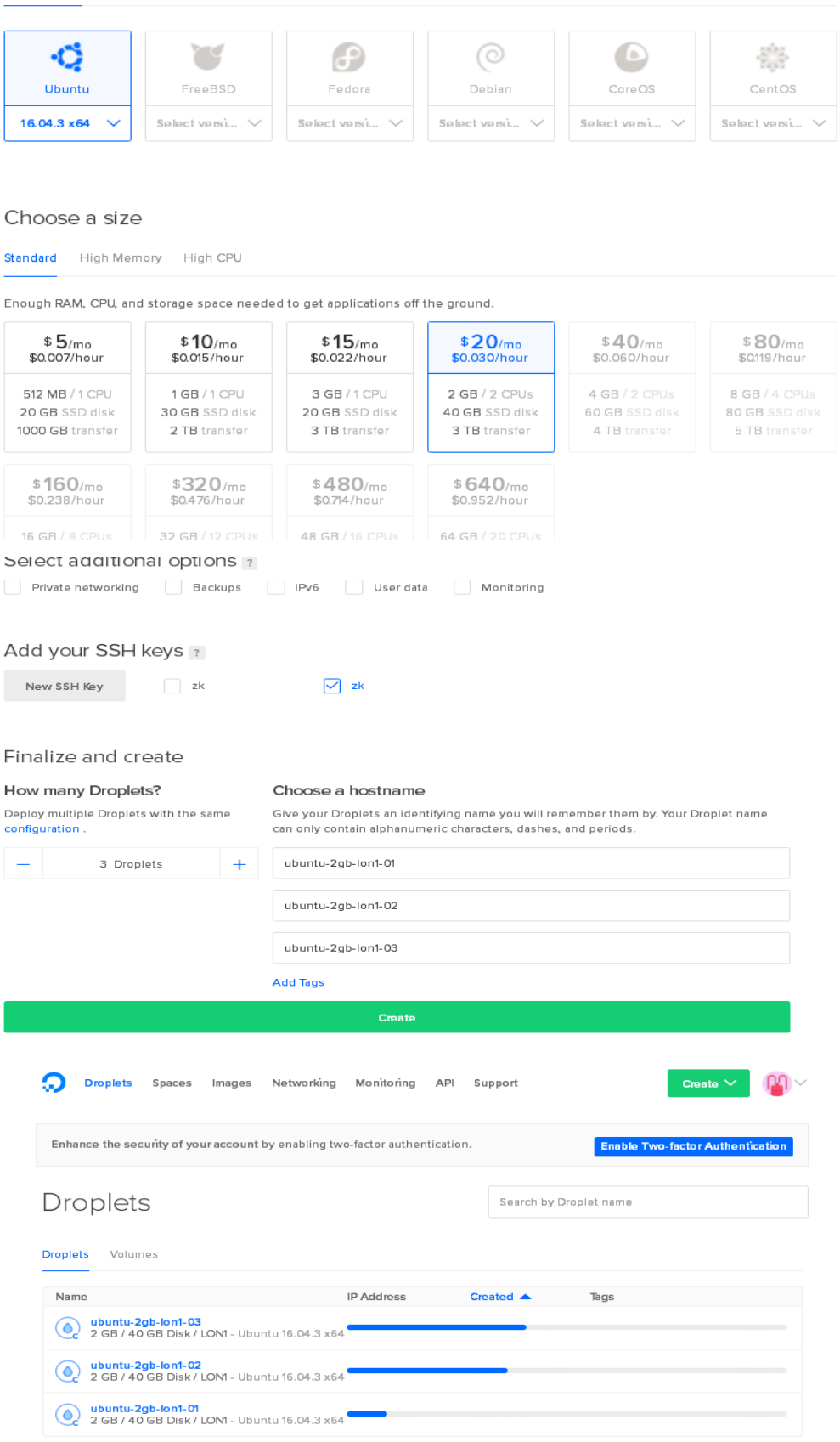


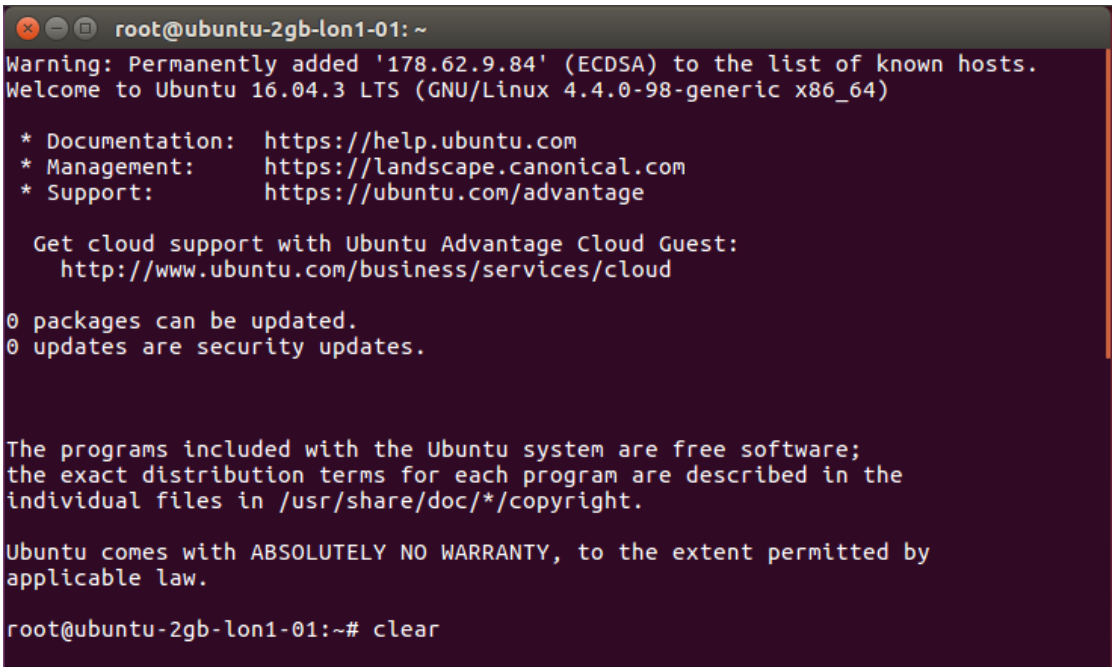
Figura 4-19 La página de web de digitalocean para crear las máquinas virtuales

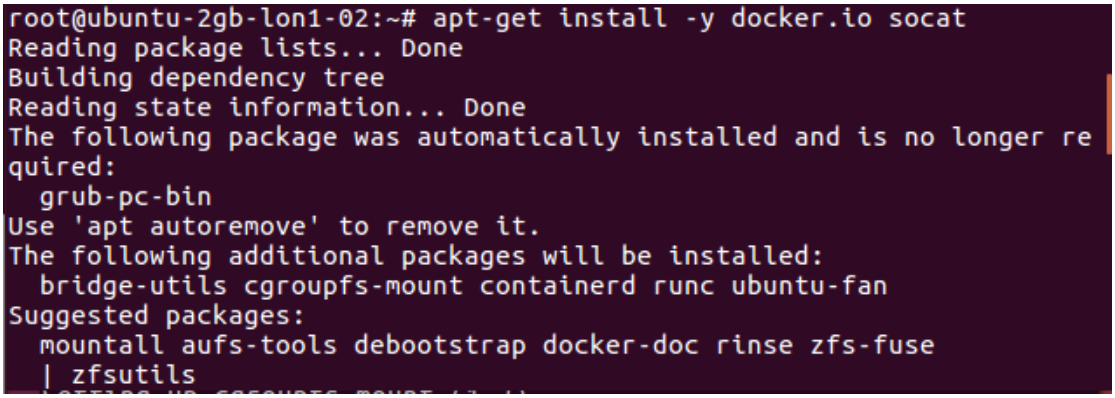
4.7.2 Instalando Docker

En cada una de sus máquinas, instalamos Docker. Se recomienda la versión v1.12, pero se sabe que v1.11, v1.13 y 17.03 también funcionan. Las versiones 17.06+ podrían funcionar, pero aún no han sido probadas y verificadas por el equipo de desarrollo de Kubernetes.

Instalamos Docker desde los repositorios de Ubuntu:

```
apt-get update
apt-get install -y docker.io
```





4.7.3 Installing kubeadm, kubelet and kubectl

La instalación de los servicios principales de Kubernentes la realizamos apartir de los instaladores oficiales, ejecutando los siguientes comandos:

```
apt-get update && apt-get install -y apt-transport-https curl -s
https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add - cat
<<EOF >/etc/apt/sources.list.d/kubernetes.list deb http://apt.kubernetes.io/
```

```
kubernetes-xenial main EOF apt-get update apt-get install -y kubelet kubeadm
kubectl
```

```
root@ubuntu-2gb-lon1-02:~# curl -s -L "https://www.dropbox.com/s/t
so6dc7b94ch2sk/debs-5ab576.txz?dl=1" | tar xJv

debian/bin/unstable/xenial/kubeadm_0.1.0-c0d5c62-00_amd64.deb
debian/bin/unstable/xenial/kubectl_1.4.0-beta.8-00_amd64.deb
debian/bin/unstable/xenial/kubelet_1.4.0-beta.8-00_amd64.deb
debian/bin/unstable/xenial/kubernetes-cni_0.3.0.1-07a8a2-00_amd64.
deb
```

```
root@ubuntu-2gb-lon1-01:~# dpkg -i debian/bin/unstable/xenial/*.de
b
Selecting previously unselected package kubeadm.
(Reading database ... 54515 files and directories currently instal
led.)
Preparing to unpack .../kubeadm_0.1.0-c0d5c62-00_amd64.deb ...
Unpacking kubeadm (0.1.0-c0d5c62-00) ...
Selecting previously unselected package kubectl.
Preparing to unpack .../kubectl_1.4.0-beta.8-00_amd64.deb ...
Unpacking kubectl (1.4.0-beta.8-00) ...
Selecting previously unselected package kubelet.
Preparing to unpack .../kubelet_1.4.0-beta.8-00_amd64.deb ...
Unpacking kubelet (1.4.0-beta.8-00) ...
Selecting previously unselected package kubernetes-cni.
Preparing to unpack .../kubernetes-cni_0.3.0.1-07a8a2-00_amd64.deb
 ...
Unpacking kubernetes-cni (0.3.0.1-07a8a2-00) ...
Setting up kubectl (1.4.0-beta.8-00) ...
Setting up kubernetes-cni (0.3.0.1-07a8a2-00) ...
Setting up kubelet (1.4.0-beta.8-00) ...
Setting up kubeadm (0.1.0-c0d5c62-00) ...
```

4.7.4 Inicializando tu maestro

Para inicializar el maestro, elegimos una de las máquinas en la que instalamos previamente kubeadm y ejecutamos:

```
kubeadm init
```

```
root@ubuntu-2gb-lon1-01:~# kubeadm init
<master/tokens> generated token: "b57f3a.942861ec76b728f7"
<master/pki> created keys and certificates in "/etc/kubernetes/pki
"
<util/kubeconfig> created "/etc/kubernetes/kubelet.conf"
<util/kubeconfig> created "/etc/kubernetes/admin.conf"
<master/apiclient> created API client configuration
<master/apiclient> created API client, waiting for the control pla
ne to become ready
<master/apiclient> all control plane components are healthy after
31.089386 seconds
<master/apiclient> waiting for at least one node to register and b
ecome ready
<master/apiclient> first node is ready after 3.514742 seconds
<master/discovery> created essential addon: kube-discovery, waitin
g for it to become ready
<master/discovery> kube-discovery is ready after 39.005495 seconds
<master/addons> created essential addon: kube-proxy
<master/addons> created essential addon: kube-dns

Kubernetes master initialised successfully!

You can now join any number of machines by running the following o
n each node:

kubeadm join --token b57f3a.942861ec76b728f7 178.62.9.84
```

Registre el resultado de ejecutar el comando `kubeadm init`, lo necesitarás para conecta al maestro con el comendo de `kubeadm join`.

```
root@ubuntu-2gb-lon1-03:~# kubeadm join --token b57f3a.942861ec76b
728f7 178.62.9.84
<util/tokens> validating provided token
<node/discovery> created cluster info discovery client, requesting
 info from "http://178.62.9.84:9898/cluster-info/v1/?token-id=b57f
3a"
<node/discovery> cluster info object received, verifying signature
 using given token
<node/discovery> cluster info signature and contents are valid, wi
ll use API endpoints [https://178.62.9.84:443]
<node/csr> created API client to obtain unique certificate for thi
s node, generating keys and certificate signing request
<node/csr> received signed certificate from the API server, genera
ting kubelet configuration
<util/kubeconfig> created "/etc/kubernetes/kubelet.conf"

Node join complete:
* Certificate signing request sent to master and response
  received.
* Kubelet informed of new secure connection details.

Run 'kubectl get nodes' on the master to see this machine join.
```

```
root@ubuntu-2gb-lon1-01:~# kubectl get nodes
NAME                  STATUS    AGE
ubuntu-2gb-lon1-01    Ready     2m
ubuntu-2gb-lon1-02    Ready     45s
ubuntu-2gb-lon1-03    Ready     32s
root@ubuntu-2gb-lon1-01:~#
```

4.7.5 Instalar una red de pod (romana)

Como se ha descrito en la sección anterior, Kubernetes ofrece un modelo de trabajos basado en Pods. Los contendores que se ejecutan en un Pod comparten la red, y en el caso particular, utilizaremos Romana para crearlo.

Primero necesitamos configurar el tráfico punteado de IPv4 a las cadenas de iptables, y para ello establecemos /proc/sys/net/bridge/bridge-nf-call-iptables en 1 ejecutando `sysctl net.bridge.bridge-nf-call-iptables = 1`. Este es un requisito para que algunos complementos de CNI funcionen.

```
root@ubuntu-2gb-lon1-01:~# sysctl net.bridge.bridge-nf-call-iptabl
es=1
net.bridge.bridge-nf-call-iptables = 1
```

Romana solo funciona en amd64:

```
kubectl apply -f
https://raw.githubusercontent.com/romana/romana/master/containerize/specs/romana-
kubeadm.yml
```

```
root@ubuntu-2gb-lon1-01:~# kubectl apply -f https://raw.githubuser
content.com/romana/romana/master/containerize/specs/romana-kubeadm
.yml
clusterrole "romana-listener" created
serviceaccount "romana-listener" created
clusterrolebinding "romana-listener" created
clusterrole "romana-agent" created
serviceaccount "romana-agent" created
clusterrolebinding "romana-agent" created
service "romana-etcd" created
deployment "romana-etcd" created
service "romana" created
deployment "romana-daemon" created
deployment "romana-listener" created
daemonset "romana-agent" created
```

Después de esperar durante un tiempo, la red de pod basada en romana se implementó con éxito

```
root@ubuntu-2gb-lon1-01:~# kubectl get pods --all-namespaces
NAMESPACE     NAME                                        READY
   STATUS              RESTARTS   AGE
kube-system   etcd-ubuntu-2gb-lon1-01                     1/1
   Running             0          4m
kube-system   kube-apiserver-ubuntu-2gb-lon1-01           1/1
   Running             0          3m
kube-system   kube-controller-manager-ubuntu-2gb-lon1-01  1/1
   Running             0          4m
kube-system   kube-dns-545bc4bfd4-7vzjt                   0/3
   Pending             0          4m
kube-system   kube-proxy-4r5wv                            1/1
   Running             0          4m
kube-system   kube-proxy-mtqlz                            1/1
   Running             0          2m
kube-system   kube-proxy-vh5vf                            1/1
   Running             0          1m
kube-system   kube-scheduler-ubuntu-2gb-lon1-01           1/1
   Running             0          4m
kube-system   romana-agent-8lv2g                          1/1
   Running             0          44s
kube-system   romana-agent-qmwqb                          1/1
   Running             0          44s
kube-system   romana-agent-zm4xr                          1/1
   Running             0          44s
kube-system   romana-daemon-767cf7849f-h5s89              1/1
   Running             0          45s
kube-system   romana-etcd-544d8bc9d4-7fpv9                1/1
   Running             0          45s
kube-system   romana-listener-78d676d8f9-2tsp8            0/1
   ContainerCreating   0          45s
root@ubuntu-2gb-lon1-01:~#
```

5 Conclusiones

En las secciones anteriores, analizamos 4 diferentes soluciones de red de múltiples hosts Docker: Calico, Weave, Flannel y Docker Overlay Network. De acuerdo con las cuatro características principales de soporte técnico de red las siguientes áreas para comparar:

- **Modelo de red**: qué tipo de modelo de red se usa para construir redes de hosts múltiples.
- **Aislamiento de aplicaciones**: qué niveles y tipos de aplicaciones de contenedores son compatibles para el aislamiento.
- **Servicio de nombres**: si se utiliza un nombre de host simple o reglas de DNS para búsquedas de DNS.
- **Requisitos de almacenamiento distribuido**: si se requiere de un almacenamiento distribuido externo como etcd o Consul.
- **Canal encriptado**: si la transferencia de datos e información puede colocarse en un canal encriptado.
- **Soporte de red parcialmente conectado**: Si el sistema puede ejecutarse en redes host parcialmente conectadas.

- **Límite de subred de contenedor**: si la subred del contenedor debe ser diferente de la subred del host.

5.1 Modelo de red

Calico implementa un enfoque puro de Capa 3 para redes más simples de mayor escala, de mejor rendimiento y mayor eficacia. Por tanto, no se puede considerar que Calico es tipo de overlay network. Un enfoque puro de Capa 3 evita la encapsulación de paquetes asociados con las soluciones de Capa 2 lo que simplifica los diagnósticos, reduce la sobrecarga de transmisión y mejora el rendimiento. Calico también implementa el protocolo BGP para el enrutamiento junto con redes IP puras lo que permite la expansión de Internet de redes virtuales.

Flannel tiene dos modelos de red diferentes para elegir. El se llama el backend UDP, que es una solución simple de IP sobre IP que utiliza dispositivos TUN (Es un dispositivo virtual de tres niveles que maneja paquetes de IPs) para encapsular cada fragmento de IP en un paquete UDP formando una red superpuesta. El segundo es el back-end VxLAN al igual que Docker Overlay Network. VxLAN es mucho más rápido que el backend UDP y obtiene un buen soporte del kernel de Linux. Flannel necesita un clúster Etcd para almacenar la configuración de red, asignar subredes y datos auxiliares (como direcciones IP de host). El enrutamiento de paquetes también necesita la colaboración del clúster Etcd. Además, Flannel ejecuta un proceso de flanneld separado en el entorno de host para admitir la conmutación de paquetes. Además de Docker, Flannel también se puede usar con máquinas virtuales tradicionales.

Weave también tiene dos formas diferentes de conectarse: el modo sleeve y el modo fastdp. El modo sleeve implementa un canal UDP para renviar paquetes IP desde el contenedor. La principal diferencia entre el modo de Weave sleeve y el modo de back-end de Flannel UDP es que Weave combina paquetes de varios contenedores en un paquete y transmite por UDP por lo que en la mayoría de los casos el modo de Weave sleeve es más rápido que el modo back end de Flannel UDP.

La otra conexión de Weave, llamada modo de fastdp, también implementa la solución de VxLAN. Aunque no existe documentación oficial sobre el uso de VxLAN, el uso de VxLAN todavía se puede encontrar en el código de Weave.

Un breve resumen del modelo de red se muestra en la siguiente tabla:

	Calico	Flannel	Weave	Romana	DockerOverlay
Tipo de red	puro de Capa 3	VxLAN o UDP	VxLAN o UDP	puro de Capa 3	VxLAN

Tabla 1 Modelo de red

5.2 Aislamiento de aplicaciones

Las redes Flannel, Weave y Docker Overlay Network utilizan el mismo modo de aislamiento de aplicaciones: aislamiento tradicional de CIDR. El aislamiento tradicional de CIDR usa máscaras de red para identificar diferentes subredes y las máquinas en diferentes subredes no pueden comunicarse entre sí.

Calico, permite por defecto solo la comunicación entre contenedores en la misma red, pero a través de su potente sistema de gestión de políticas puede definirse casi cualquier control de acceso a la escena.

Para Romana supone aislamiento de red basado en iptables ACL y administrar Host/Tenant/Segment ID basadas en jerarquía CIDR.

	Calico	Flannel	Weave	Romana	DockerOverlay
Aislamiento	Perfil de esquema	CIDR	CIDR	CIDR	CIDR

Tabla 2 Aislamiento de aplicaciones

5.3 Soporte de protocolo

Debido a que Calico es una solución pura de tres niveles, no es compatible con todos los protocolos Layer 3 o Layer 4. Desde los foros oficiales de github, los desarrolladores de Calico afirman que Calico solo soporta TCP, UDP, ICMP e ICMPv6. Otras soluciones admiten todos los protocolos. Romana basada en Kubernetes puede soportar los protocolos requeridos.

	Calico	Flannel	Weave	Romana	DockerOverlay
Protocolo	TCP, UDP, ICMP & ICMPv6	Todos	Todos	Todos	Todos

Tabla 3 Soporte de protocolo

5.4 Servicio de nombre

Weave admite servicios de nombres entre contenedores. Al crear un contenedor Weave lo coloca en el servicio de nombres de DNS en el formato {nombre de host} .weave.local. Por tanto, puede acceder a cualquier contenedor usando {hostname} .weave.local o simplemente {hostname}.

	Calico	Flannel	Weave	Romana	DockerOverlay
Servicio de nombre	No	No	Si	No	No

Tabla 4 Servicio de nombre

5.5 Almacenamiento distribuido

Docker Overlay, Flannel y Calico requieren etcd o Consul. Weave es responsable de intercambiar información de configuración de red entre hosts y no necesita un sistema de almacenamiento distribuido. Romana es una red local simple que no necesita guardar y compartir información de red.

	Calico	Flannel	Weave	Romana	DockerOverlay
Almacenamiento de distribuido	Si	Si	No	Si	Si

5.6 Canal de cifrado

Flannel admite el cifrado de TLS entre Flannel y Etcd. Weave se puede configurar para cifrar datos de control a través de conexiones TCP y cifrar la carga de paquetes UDP enviados entre pares. Las redes superpuestas Calico, Docker y Romana no admiten ningún tipo de método de encriptación.

	Calico	Flannel	Weave	Romana	DockerOverlay
Canal de cifrado	No	TLS	NaCl Library	No	No

Tabla 5 Canal de cifrado

5.7 Soporte de red parcialmente conectado

Weave se puede implementar en una red parcialmente conectada que permite a Weave conectar hosts separados por un firewall conectando hosts en diferentes centros de datos con direcciones IP internas. Ninguna de las otras soluciones permite este tipo de conectividad.

	Calico	Flannel	Weave	Romana	DockerOverlay

Soporte de red parcialmente conectado	No	No	Si	No	No

Tabla 6 Soporte de red parcialmente conectado

5.8 Restricción de subred de contenedor

Todos los hosts en una red Docker Overlay comparten la misma subred. Las IP se asignan secuencialmente a los contenedores cuando se inician. Este espacio IP se puede personalizar a través de `--subnet`.

Flannel asigna a cada host una subred separada automáticamente, y los usuarios solo necesitan especificar un grupo de IP. La información de enrutamiento entre diferentes subredes también se genera y configura automáticamente por Flannel.

Con la configuración predeterminada de Weave, todos los contenedores utilizan la subred 10.32.0.0/12. Si este espacio de direcciones entra en conflicto con las direcciones IP existentes se puede asignar una subred específica con `--ipalloc-range`.

Calico asigna una subred a cada host desde el IP Pool (personalizable).

Romana utiliza la administración de direcciones IP, junto con el anuncio de rutas, para eliminar la necesidad de una red superpuesta incluso a través de subredes VPC. El IPAM con reconocimiento de topología de Romana reduce la necesidad de actualizaciones de rutas cuando se agregan nuevos puntos finales y no requiere el peering completo de los nodos.

	Calico	Flannel	Weave	Romana	DockerOverlay
Restricción de subred de contenedor	Cada host tiene una subred	Cada host tiene una subred	subred individual	N	subred individual

Tabla 7 Restricción de subred de contenedor

6 Referencias

[1] "Docker container networking," [Online]. Available: https://docs.docker.com/engine/userguide/networking/#default-networks.

[2] Calico, «Project Calico,» [En línea]. Available: https://www.projectcalico.org/.

[3] Weave, «weaveworks,» [En línea]. Available: https://www.weave.works/.

[4] CoreOS, «flannel,» [En línea]. Available: https://coreos.com/flannel/docs/latest/.

[5] kubernetes, «keberneters,» [En línea]. Available: https://kubernetes.io/.

[6] Romana, «Romana,» [En línea]. Available: http://romana.io/.

[7] Mesos, «Apache Mesos,» [En línea]. Available: http://mesos.apache.org/.

[8] nuagenetworks, «The Tale of Two Container Networking Standards: CNM v. CNI,» [En línea]. Available: http://www.nuagenetworks.net/blog/container-networking-standards/.

[9] feisky, «http://www.cnblogs.com/feisky/p/4093717.html,» [En línea].

[10] Dockerinfo, «http://www.dockerinfo.net/3738.html,» [En línea].

[11] «Docker Mesos 在生产环境的应用,» [En línea]. Available: http://www.sohu.com/a/140681480_468650.

[12] wikipedia, «VRF,» [En línea]. Available: https://en.wikipedia.org/wiki/Virtual_routing_and_forwarding.

[13] «Romana Project,» [En línea]. Available: https://www.slideshare.net/RomanaProject.

[14] «Just the Basics,» Romana, [En línea]. Available: http://romana.io/how/romana_basics/.

[15] «Gitbook,» [En línea]. Available: https://feisky.gitbooks.io/sdn/container/romana/.

[16] «mesos 概述,» [En línea]. Available: http://blog.csdn.net/lsshlsw/article/details/47086869.

[17] «基于 Kubernetes 构建 Docker 集群管理详解,» [En línea]. Available: http://www.csdn.net/article/2014-12-24/2823292-Docker-Kubernetes.

[18] «London Adapt or Die: Kubernetes, Containers and Cloud - The MoD Story,» [En línea]. Available: https://www.slideshare.net/apigee/london-adapt-or-die-kubernetes-containers-and-cloud-the-mod-story.

[19] «Kubernetes,» [En línea]. Available: https://kubernetes.io/docs/concepts/services-networking/service/.