Fuente

Dvnet: virtualizando redes

Integrado, sencillo y extensible

Pablo Collado Soto - 24/10/2023

¿Qué queremos lograr?

Fuente

- La **configuración** de redes **no** es **trivial**:
    - Hacen falta **varias máquinas** para «probar».
    - Queremos poder iterar con **rapidez**.
    - Buscamos un entorno **controlado**.
    - **Emular** las redes: **no** queremos hacer **simulaciones.**
    - Queremos algo **sencillo**: la **infraestructura no** es nuestro **interés.**
- Las **soluciones** disponibles **no** nos **convencían**…
- Por ello, **diseñamos** nuestro **propio** sistema: **dvnet**.

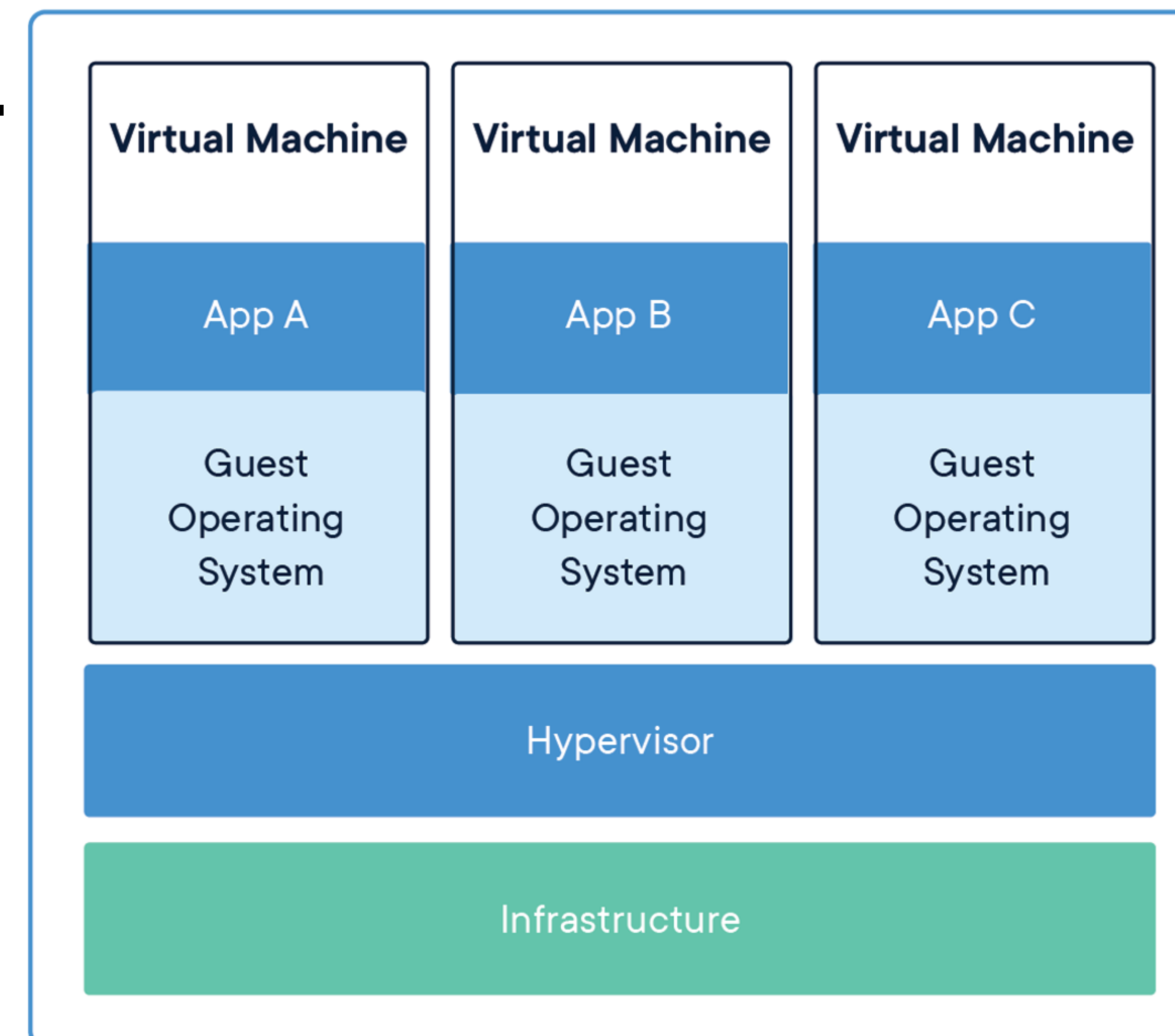
¿Qué ventajas tiene?

Fuente

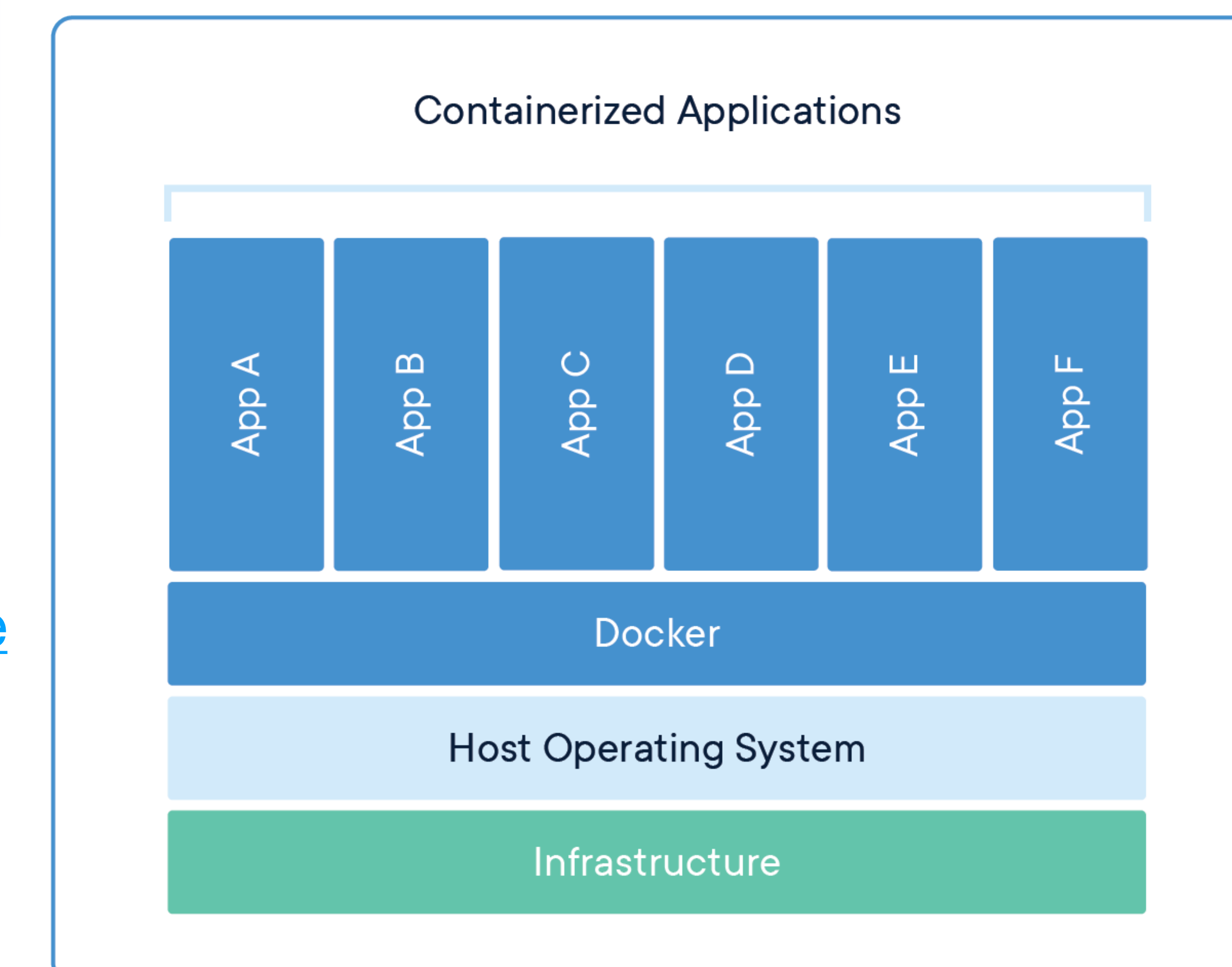
- **Dvnet** emplea **contenedores** en su núcleo:
  - La tecnología es **menos disruptiva**.
  - La **solución** es muy **liviana**.
  - Permite una **gran extensibilidad**.
  - Soporta **redes** de **gran tamaño**.
  - Permite **integrarse** con herramientas conocidas.
  - **Elimina** mucho «**ruido** de **fondo**».

¿Qué es una red en capa 3?

- En **L3**, una red se compone de una serie de ***hosts*** y ***routers***.
- **Hosts** y **routers** son entidades «**virtuales**».
- Dos **candidatos**:
  - **Máquinas Virtuales (VMs):**
    - Consumen **muchos recursos**.
    - Infraestructuras de red «opacas».
  - **Contenedores:**
    - **Livianos.**
    - Infraestructura de red «**estándar**».



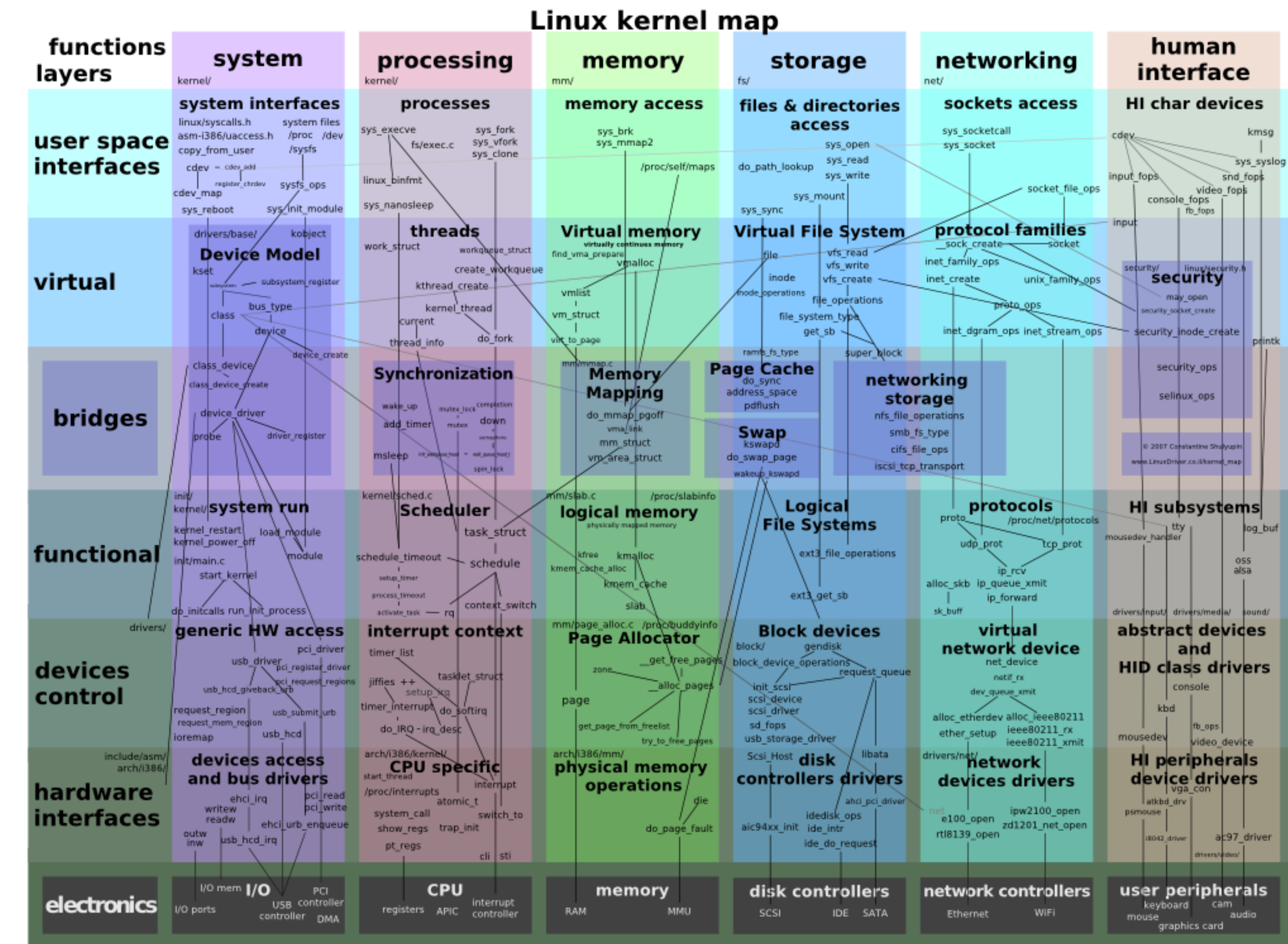
Fuente



Fuente

¿Y en capa 2?

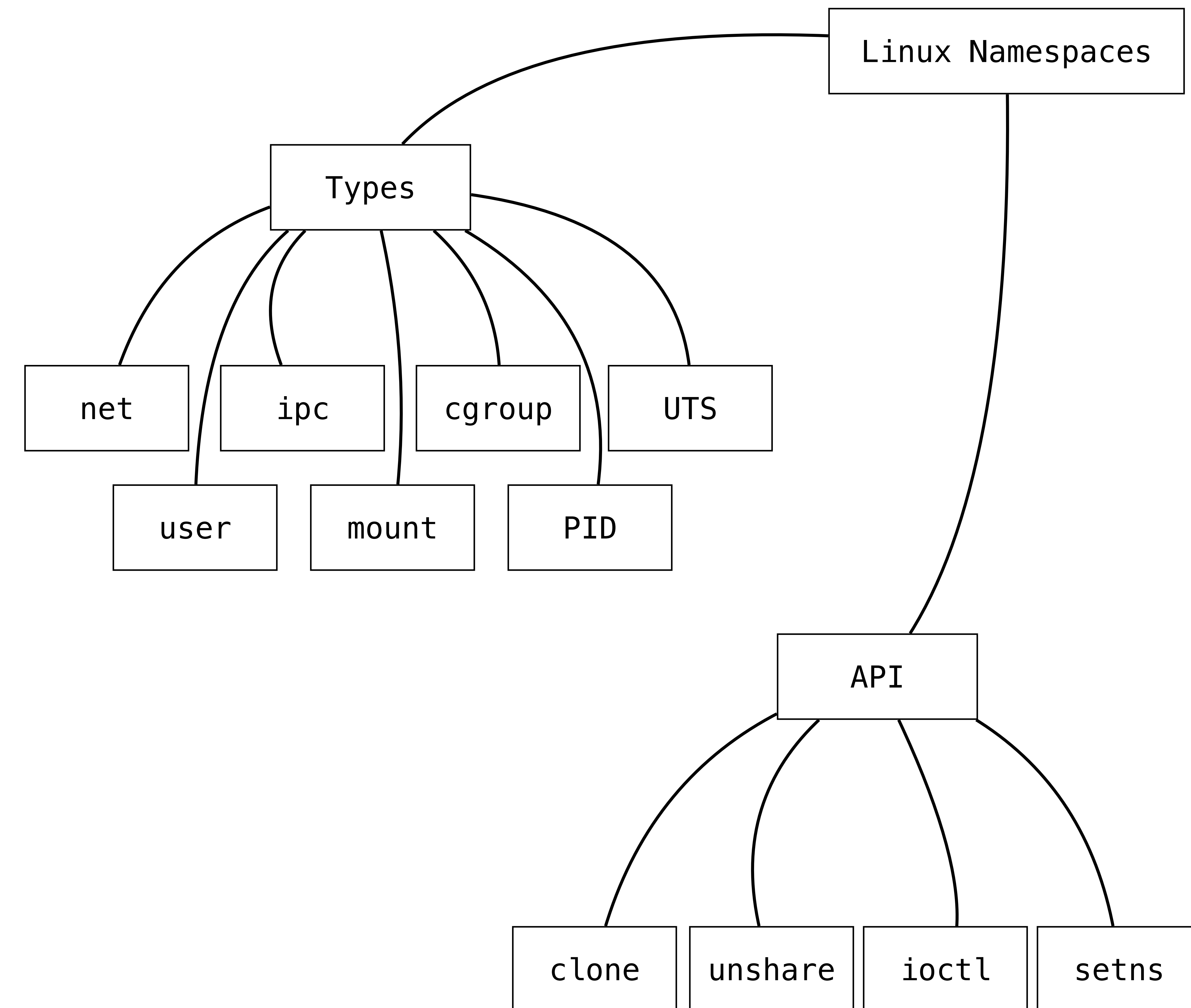
- En **L2**, la red se compone de equipos **«físicamente» conectados.**
- El kernel de **Linux** ofrece alternativas virtuales:
  - **Switches**: **bridges**.
  - **Cables**: Dispositivos de Ethernet Virtuales (**veth**).
- El kernel de **Linux** aúna **L2** y **L3**:
  - Una entidad en **L3** es una «**pila de red**».
  - **Network Namespace:** pila de red virtual.



[Fuente]

La pila de red y los espacios de nombres

- La **pila** de red de Linux no deja de ser un «**trozo**» de código.
- Una **pila** de red equivale a una **entidad lógica** en L3.
- Así, una **pila independiente** es un **nodo virtual**.
- **Linux** ofrece la posibilidad de **interactuar** con la **pila** de red.
- Para ello, **configuramos** el **espacio** de **nombres** de **red** apropiado.
- Para compartimentar recursos, ¡cada **contenedor** tiene su propio **network namespace**!



Basado en esta imagen.

Interactuando con el net namespace

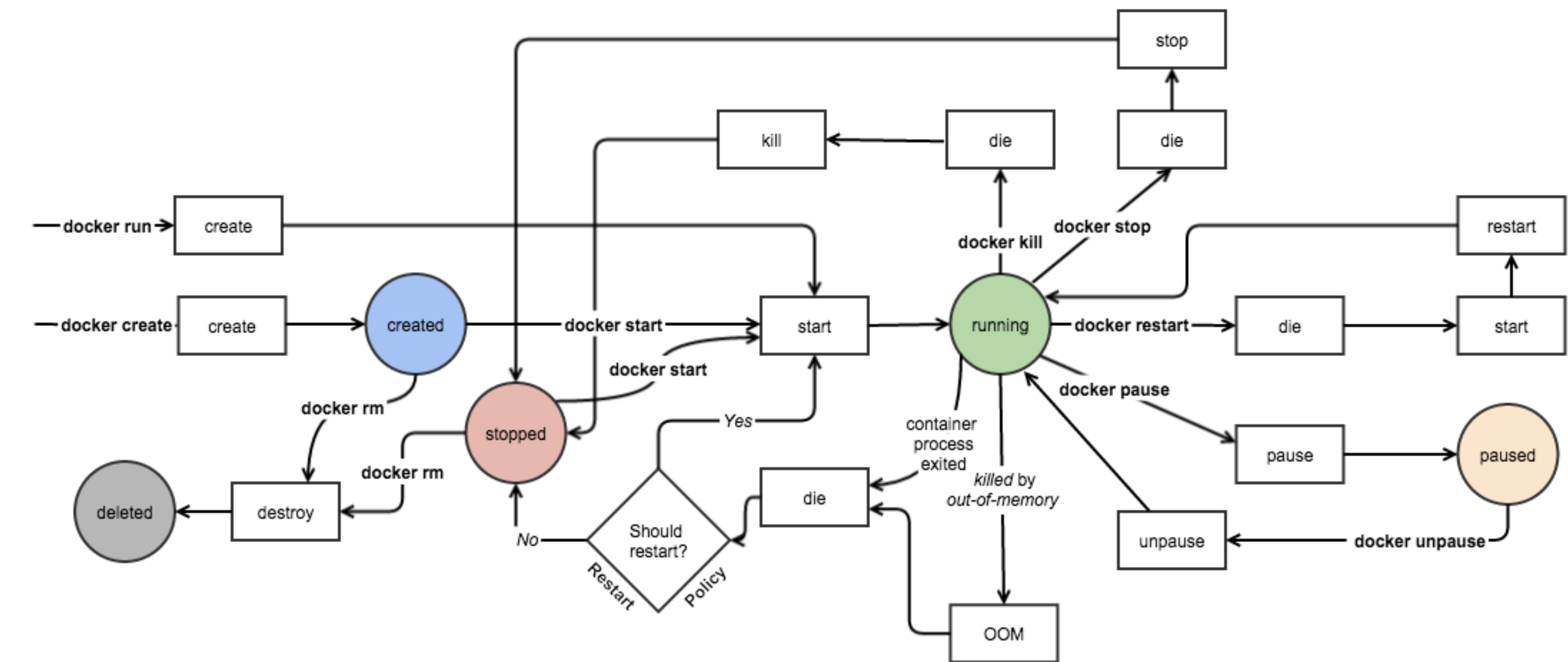
- Todo se soporta sobre sockets `AF_NETLINK(7)`.
- La interfaz aparece envuelta por `iproute2`.
- Los comandos `ip(8)` pueden ejecutarse en cualquier namespace.
- Para **`ip(8)`**, cada namespace es un «contexto».
- Los espacios de nombres de los procesos se exponen con en `proc(5)`.

```
foo$ ls -l /proc/`docker inspect -f {{.State.Pid}} R-1)`/net
-r--r--r--. 1 root root 0 Oct 23 00:54 netlink
-r--r--r--. 1 root root 0 Oct 23 00:54 netstat
-r--r-----. 1 root root 0 Oct 23 00:54 nf_conntrack
-r--r-----. 1 root root 0 Oct 23 00:54 nf_conntrack_expect
-r--r--r--. 1 root root 0 Oct 23 00:54 packet
-r--r--r--. 1 root root 0 Oct 23 00:54 protocols
-r--r--r--. 1 root root 0 Oct 23 00:50 psched
-r--r--r--. 1 root root 0 Oct 23 00:54 ptype
-r--r--r--. 1 root root 0 Oct 23 00:54 raw
-r--r--r--. 1 root root 0 Oct 23 00:54 raw6
-r--r--r--. 1 root root 0 Oct 23 00:54 rout
```

```
foo$ cat /proc/$(docker inspect -f {{.State.Pid}} R-1)/net/route
Iface     Destination   Gateway     Flags   RefCnt  Use Metric      Mask   MTU Window IRTT
ethr-1-a     0000000A   00000000     0001        0    0      0  00FFFFFF    0      0    0
ethr-1-c     0001000A   0202000A     0003        0    0     20  00FFFFFF    0      0    0
ethr-1-c     0002000A   00000000     0001        0    0      0  00FFFFFF    0      0    0
```

Creando los nodos

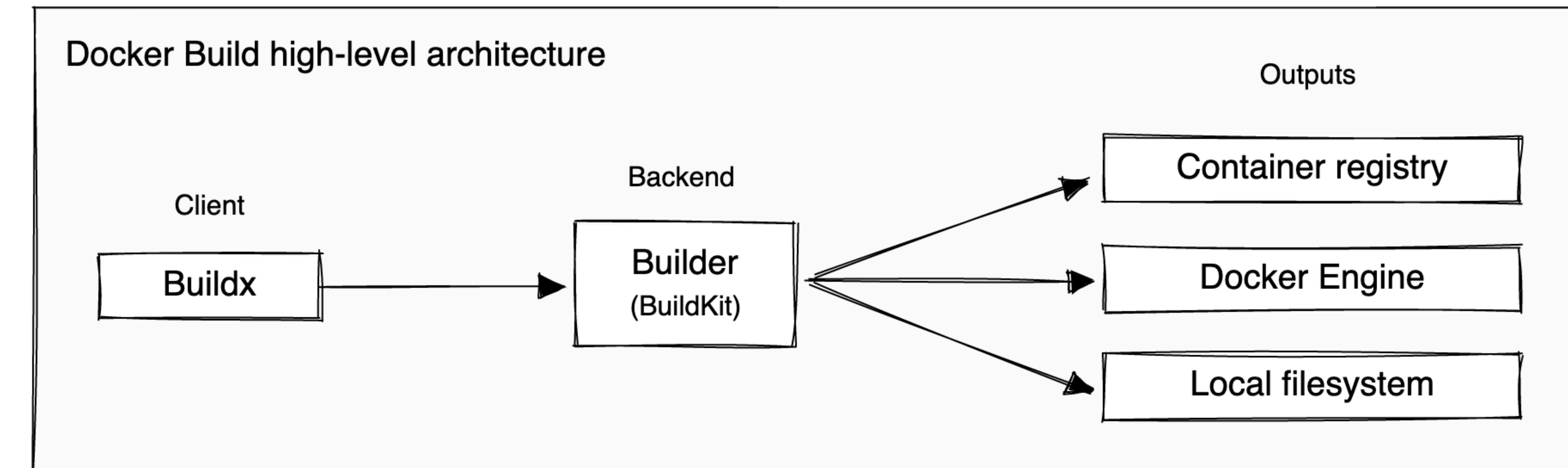
- Los **nodos** de red serán **contenedores**.
- Un contenedor **persiste** mientras lo haga su **PID 1**.
- Un contenedor **no** suele tener un **daemon** del **sistema**.
  - Aunque ¡`tini` se incluye ya desde hace un tiempo!
- Dado que buscamos instanciar redes, ejecutamos `sshd(8)` como **PID 1**.



Fuente

Definiendo los nodos

- Un **contenedor** puede entenderse como una **imagen** en **ejecución**.
- Nosotros definimos **imágenes** en un **Dockerfile**.
- La **imagen** se **construye** y luego se **ejecuta**.
- Estas **imágenes** son mucho más **livianas** que una **VM**.
- Además, se pueden **compartir** de manera sencilla en **registros**.
- **Definir** nuevas **imágenes** es muy **fácil**.

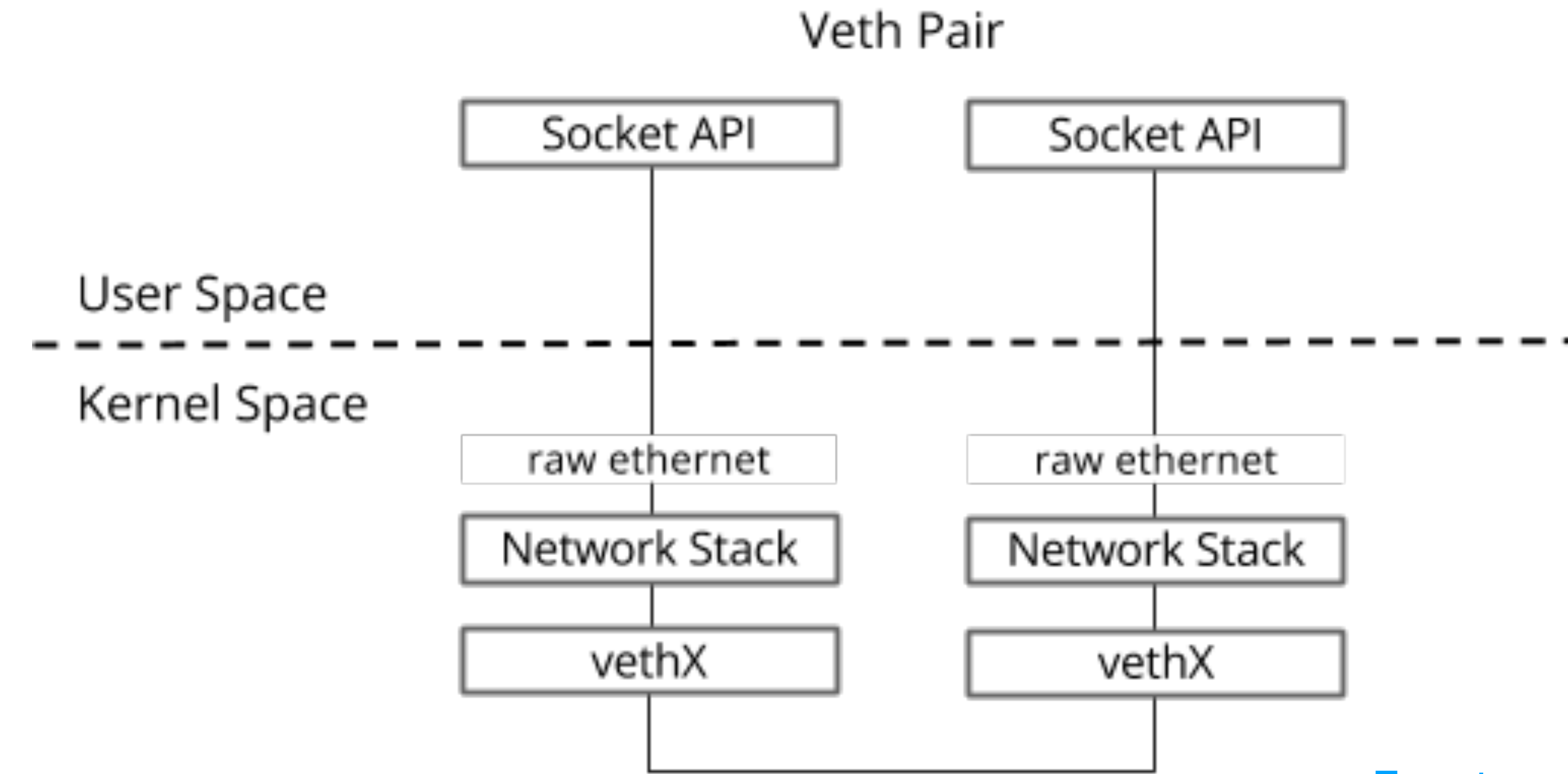


Fuente

```
FROM ubuntu:22.04

RUN \
    apt-get update && \
    apt-get install -y iproute2 && \
    apt-get install -y iputils-ping && \
    apt-get install -y openssh-server && \
    apt-get install -y iptables && \
    apt-get install -y tcpdump && \
    apt-get install -y frr && \
    mkdir /run/sshd && \
    echo "PermitRootLogin yes" >> /etc/ssh/sshd_config

RUN ["/bin/bash", "-c", "echo -e '1234\n1234' | passwd root"]

CMD ["/usr/sbin/sshd", "-D"]
```

Creando el «hardware»

- Podemos lograr con **`ip(8)`**:
  - **Bridges**:
    - `ip link add foo-brd type bridge`
  - **Veths**:
    - `ip link add v-x type veth peer name v-y`
    - `ip link set v-[x | y] master foo-brd`
    - `ip link set netns v-[x | y] foo-netns`



Fuente

Direccionando, encaminando y filtrando

Fuente

- Al igual que antes, con **`ip(8)`** podemos:
  - **Direccionar**:
    - **`ip a add <IPv4>/<netmask> brd + dev <veth>`**
  - **Encaminar:**
    - **`ip route add default via <gateway-IPv4>`**
- Con [**`iptables(8)`**]() podemos gestionar los firewalls:
  - **`iptables -t INPUT -A -s <src-cidr> -j [DROP | ACCEPT]`**

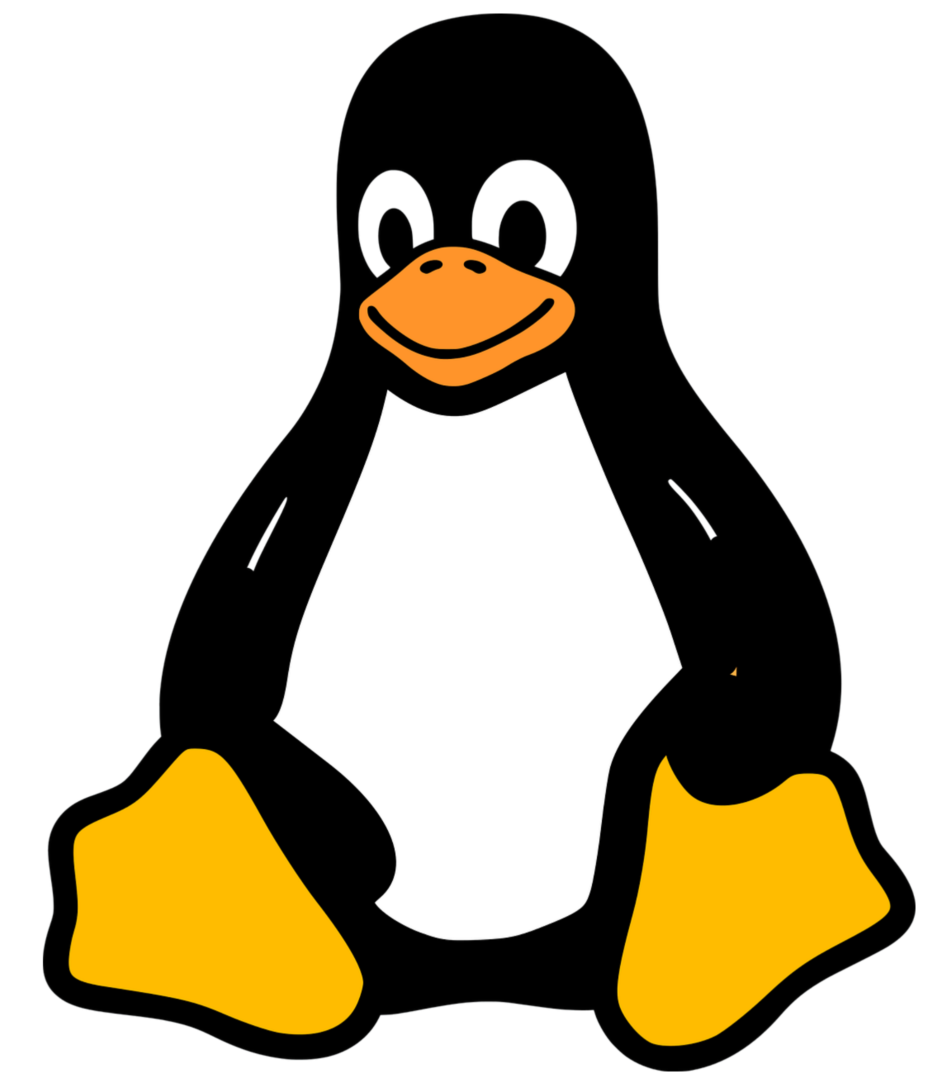
Sutilezas

- Este caso de uso obliga a **configurar** el **kernel**.
- Podemos hacerlo con [**`sysctl(8)`**]:

  - `sysctl -w net.bridge.bridge-nf-call-iptables=0`

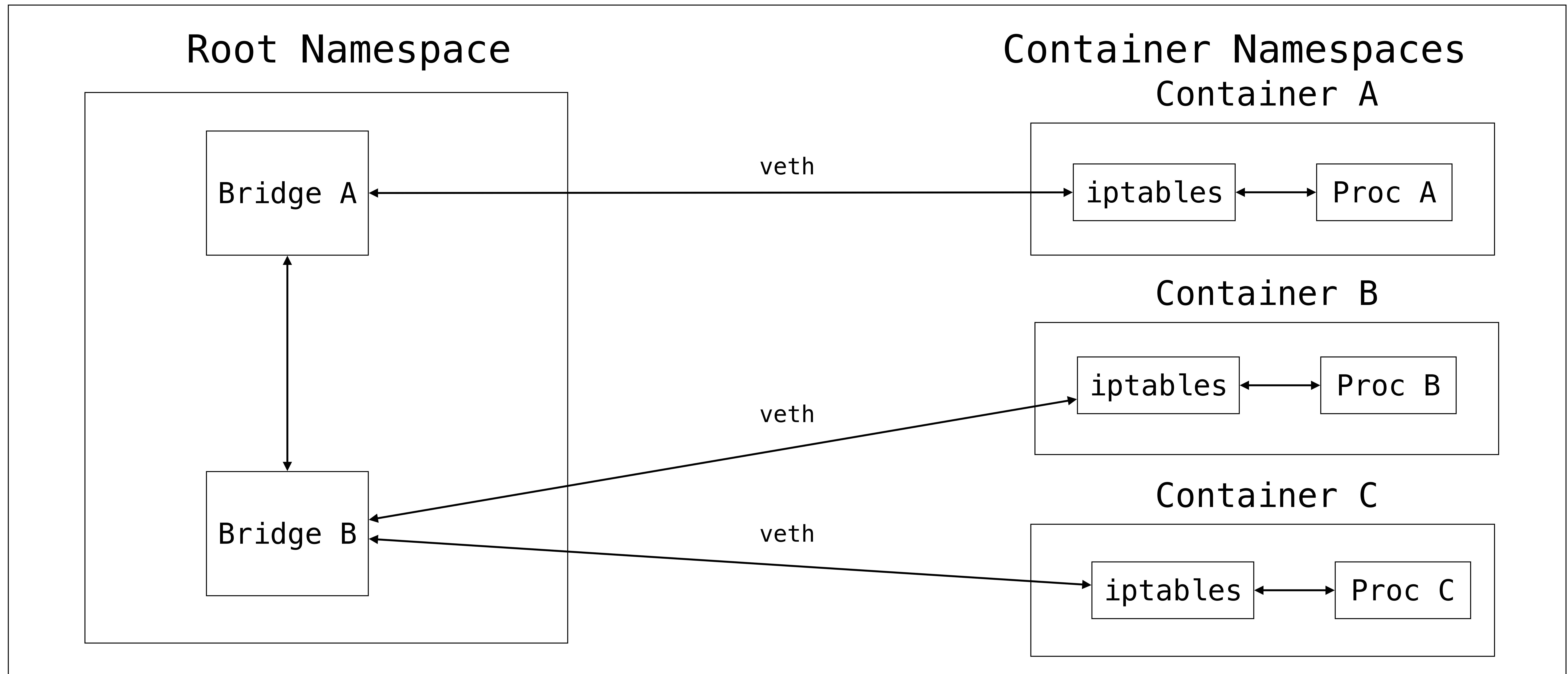
  - `sysctl -w net.ipv4.ip_forward=1`

- Para que los cambios persistan se debe editar **`/etc/sysctl.conf`**.
- Los **contenedores** deben tener asignadas unas [**`capabilities(7)`**]:
  - **`CAP_NET_ADMIN`**
  - **`CAP_SYS_ADMIN`**



Fuente

Después de todo esto



Automatizando el proceso

Fuente

- Con `ip(8)` se puede lograr **todo**.
- Pero… ¡es una **locura**! Además, **no** es **persistente**…
- Hay que **automatizar** estos despliegues.
- Primera estrategia: **Python**.
    - Una **red** se define como una **clase**.
    - El **intérprete** de Python se hace **necesario.**
    - **No** se desarrolla un **lenguaje** de **configuración**.
    - El **código** es una **pesadilla**…

Reescribiendo el código

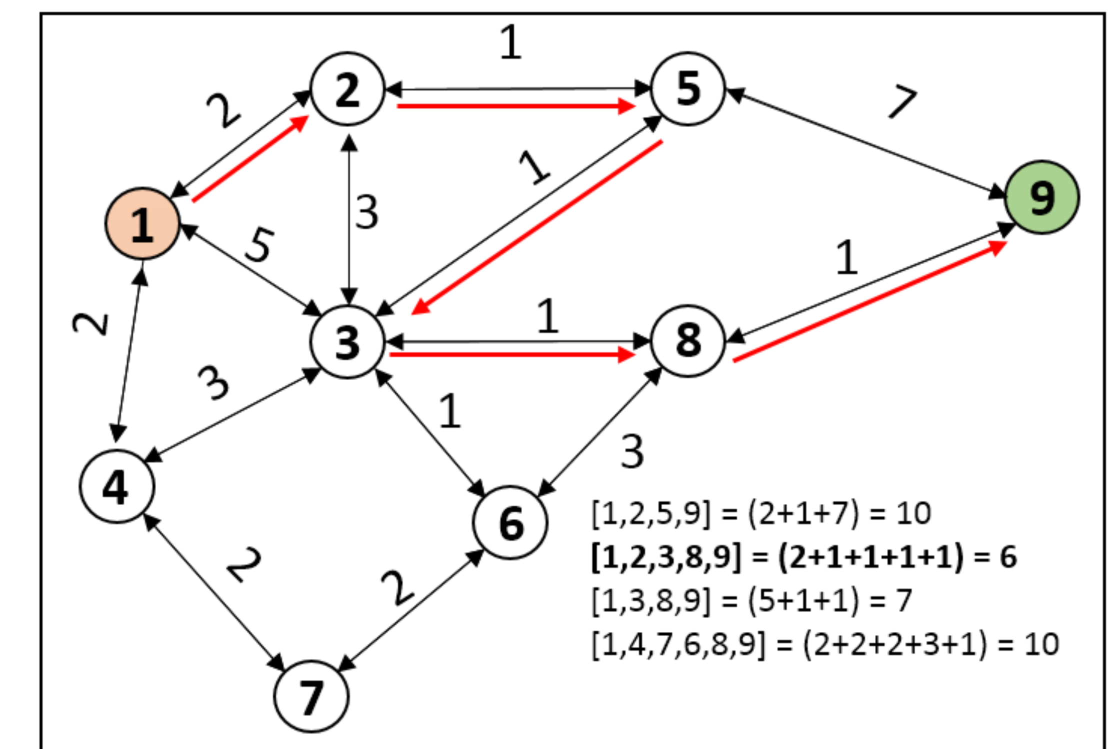
Fuente

- **Ken Thomson** dijo:

«Uno de mis días más productivos consistió descartar 1000 líneas de código».

- Con ese espíritu, reescribimos **dvnet** en Go ya que:
  - Es un lenguaje **tipado**.
  - Se compila **estáticamente**.
  - La **compilación cruzada** es trivial.
  - Es tremendamente **legible** y **compacto**.
  - Es el **lenguaje** en el que está escrito **Docker**.

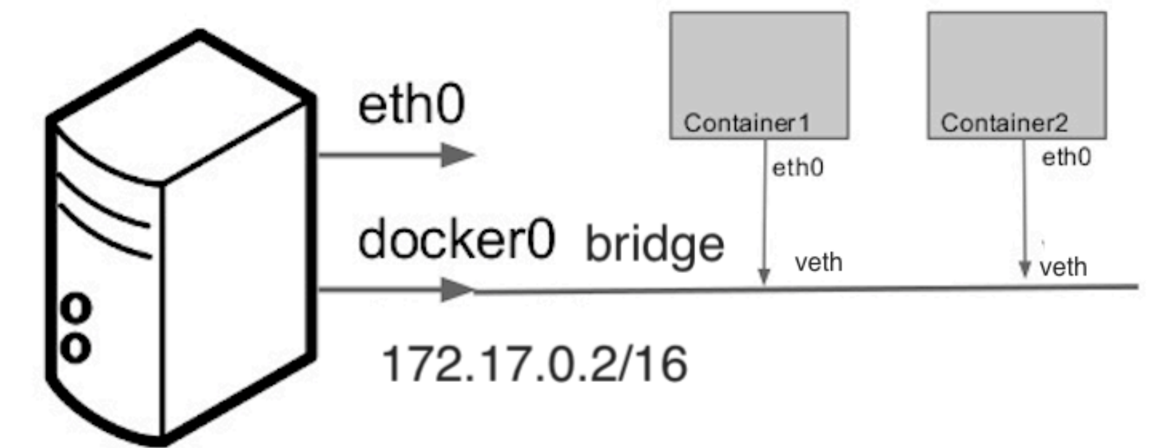
La estructura de dvnet



Fuente

- Dvnet es un **plugin** nativo de **Docker**.
- La comunicación se emplea un socket UNIX en **`/run/docker/plugins`**.
- Para escribir el plugin nos basamos en el ***helper*** de [**`docker/go-plugins`**].
- El plugin es un simple **binario** que se ejecuta en **segundo plano**.
- El daemon se gestiona a través de [**`systemctl(1)`**].
- **No** se requiere ninguna **configuración**.
- Las **bitácoras** se pueden consultar con [**`journalctl(1)`**].
- El **encaminamiento** de la red emplea el algoritmo de Dijkstra.

Requisitos e instalación



Fuente

- La instalación consiste en «**copiar**» el binario y la unidad de SystemD.
- Ofrecemos un **script** que lo hace **automáticamente**:

```
$ curl -sSL https://raw.githubusercontent.com/pcolladosoto/dvnet/main/install.sh | sudo bash
```

- También ofrecemos un **script** que lo **desinstala**:

```
$ curl -sSL https://raw.githubusercontent.com/pcolladosoto/dvnet/main/uninstall.sh | sudo bash
```

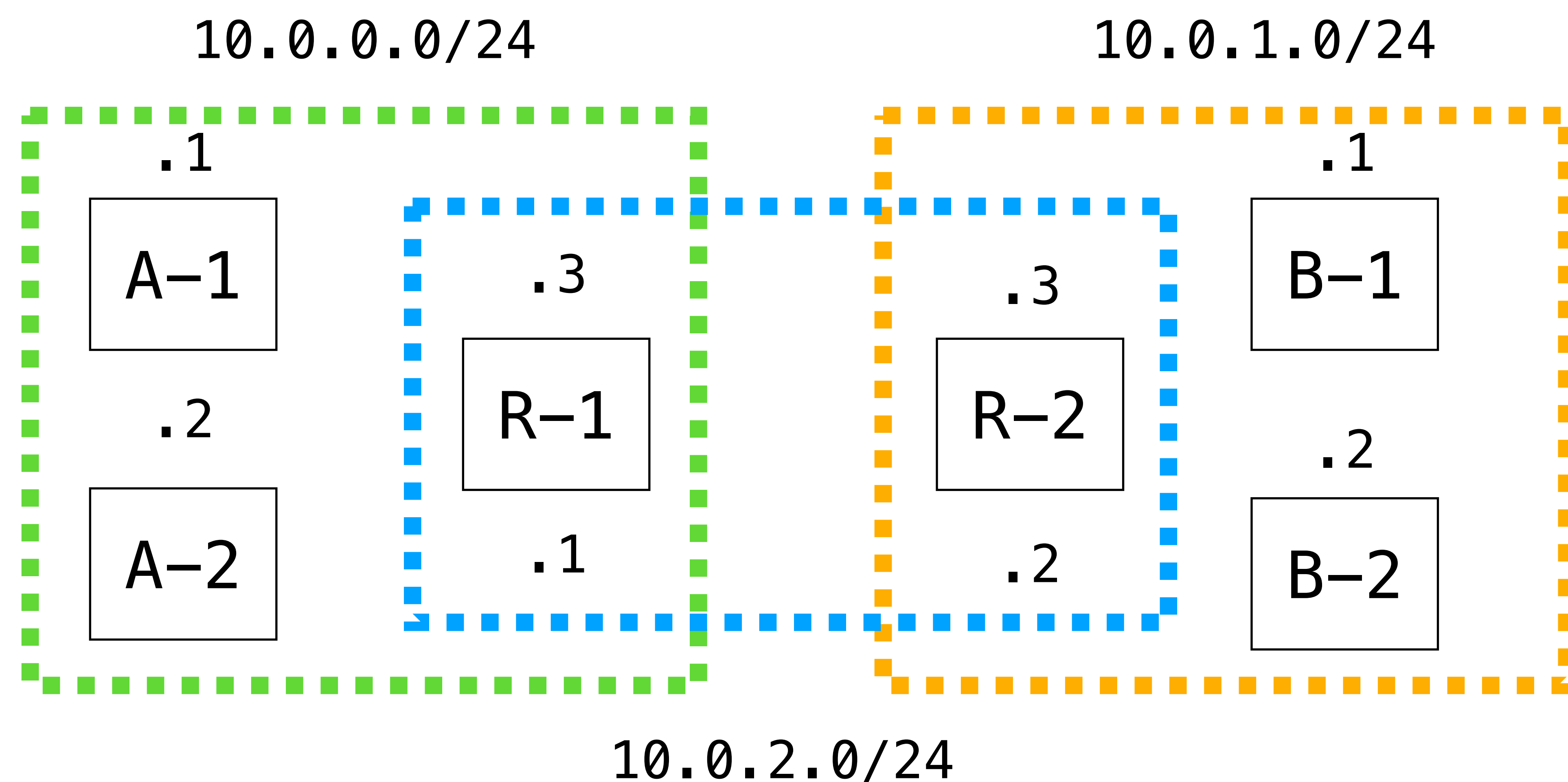
- Además, se puede **instalar** en distribuciones derivadas de **RHEL** con:

```
$ yum install https://github.com/pcolladosoto/dvnet/releases/download/v0.2/dvnet-0.2-1.el9.x86_64.rpm
```

- En cualquier caso, dvnet **debe** ejecutarse sobre **Linux**: ¡no es una limitación!

A por un ejemplo

- Vamos a definir una red con **4 hosts** y **2 routers**.
- La idea es configurar [OSPF]() para que los routers se descubran las subredes a las que no pertenecen.
- Usaremos [FRRouting]() como suite de encaminamiento.
    - Es el sucesor de [Quagga]().




```
{
    "name": "Test Net 0",
    "automatic_routing": false,
    "outbound_access": {
        "enabled": true,
        "cidr": "192.168.240.0/24"
    },
    "update_hosts": true,
    "subnets": {
        "A": {
            "cidr": "10.0.0.0/24",
            "hosts": {
                    "A-1": {"image": "pcollado/dhost"},
                    "A-2": {"image": "pcollado/dhost"}
            }
        },
        "B": {
            "cidr": "10.0.1.0/24",
            "hosts": {
                    "B-1": {"image": "pcollado/dhost"},
                    "B-2": {"image": "pcollado/dhost"}
            }
        },
        "C": {
            "cidr": "10.0.2.0/24",
            "hosts": {}
        }
    },
    "routers": {
        "R-1": {
            "fw_rules": {"POLICY": "ACCEPT", "ACCEPT": [], "DROP": []},
            "subnets": ["A", "C"],
            "image": "pcollado/drouter"
        },
        "R-2": {
            "fw_rules": {"POLICY": "ACCEPT", "ACCEPT": [], "DROP": []},
            "subnets": ["C", "B"],
            "image": "pcollado/drouter"
        }
    }
}
```

Levantando la red

- Una vez hemos definido la red a través de un **`JSON`** debemos levantarla.
- Todo ello se hace a través del cliente nativo de **`Docker`**:

```
$ docker network create --driver dvnet \
        --opt net.dvnet.def=/ruta/de/la/definición.json \
        nombre-de-la-red
$ docker network remove nombre-de-la-red
$ docker network ls
$ docker ps
```

¡Ruta absoluta! (→ /ruta/de/la/definición.json)

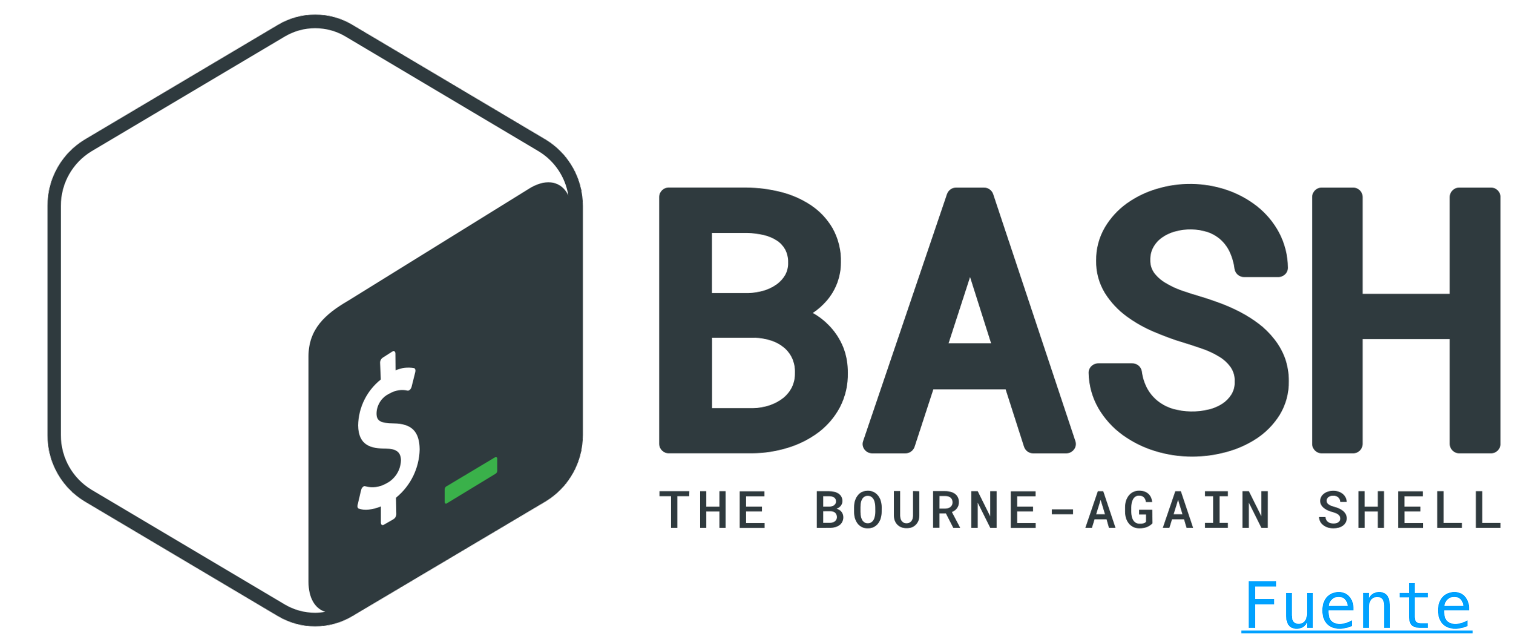
nombre-de-la-red ← Creación de la red

nombre-de-la-red ← Eliminación de la red

network ls ← Listado de las redes

ps ← Listado de los nodos

Interactuando con los nodos



Fuente

- Los nodos son en el fondo contenedores de **Docker**.
- Podemos usar los comandos normales para lanzar una **shell** interactiva:

```
$ docker exec -it nombre-del-contenedor bash
```

- Para ello, nuestras imágenes deben ofrecer una **shell**:
  - Muchas imágenes se construyen sobre **Ubuntu**, por lo que tienen **bash**.
  - Las imágenes «de fábrica» que ofrecemos tienen **bash**:
    - **pcollado/dhost**: Ubuntu + **iproute2**, **tcpdump**, **traceroute**, **ping** y **ssh**.
    - **pcollado/drouter**: Ubuntu + **iproute2**, **tcpdump**, **iptables** **ping** y **ssh**.

Configurando FRR

Fuente

- FRR define un **daemon «básico»** que siempre debe estar activo: **`zebra`**.
- Este daemon se encarga, entre otros, de **gestionar** las **rutas** en el **kernel**.
- Cada **protocolo** de encaminamiento tiene su propio **daemon**.
- Nosotros emplearemos **`ospfd`**, el daemon que implementa **OSPF**.
- Para **lanzar** los **daemons** emplearemos el siguiente comando:

```
/usr/lib/frr/<daemon> --limit-fds 1000 --config_file /ruta/de/config
```

- Si **no limitamos** los descriptores de archivo ¡nos quedamos **sin memoria**!
- Las configuraciones las copiamos a través de **`docker cp ...`**
- En el contenedor se deben ajustar los permisos con **`chown frr:frr ...`**

Las configuraciones

	R-1	R-2
Zebra	See code below	See code below
OSPFd	See code below	See code below

R-1 — Zebra

```
log stdout debugging
interface ethr-1-a
interface ethr-1-c
```

R-2 — Zebra

```
log stdout debugging
interface ethr-2-b
interface ethr-2-c
```

R-1 — OSPFd

```
log stdout debugging
interface ethr-1-c
  ip ospf hello-interval 5
  ip ospf area 0.0.0.1

router ospf
  ospf router-id 10.0.2.1
  redistribute connected
```

R-2 — OSPFd

```
log stdout debugging
interface ethr-2-c
  ip ospf hello-interval 5
  ip ospf area 0.0.0.1

router ospf
  ospf router-id 10.0.2.2
  redistribute connected
```

Capturando tráfico OSPF

OSPF.pcap

Apply a display filter ... <⌘/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.2.2	224.0.0.5	OSPF	82	Hello Packet
2	1.362121	10.0.2.1	224.0.0.5	OSPF	82	Hello Packet
3	5.001072	10.0.2.2	224.0.0.5	OSPF	82	Hello Packet
4	6.364990	10.0.2.1	224.0.0.5	OSPF	82	Hello Packet
5	10.002153	10.0.2.2	224.0.0.5	OSPF	82	Hello Packet
6	11.366739	10.0.2.1	224.0.0.5	OSPF	82	Hello Packet

```
Open Shortest Path First
  OSPF Header
    Version: 2
    Message Type: Hello Packet (1)
    Packet Length: 48
    Source OSPF Router: 10.0.2.2
    Area ID: 0.0.0.1
    Checksum: 0xccac [correct]
    Auth Type: Null (0)
    Auth Data (none): 0000000000000000
  OSPF Hello Packet
    Network Mask: 255.255.255.0
    Hello Interval [sec]: 5
    Options: 0x02, (E) External Routing
    Router Priority: 1
    Router Dead Interval [sec]: 20
    Designated Router: 10.0.2.1
    Backup Designated Router: 10.0.2.2
    Active Neighbor: 10.0.2.1
```


```
0000  01 00 5e 00 00 05 6a 03  09 4d f4 32 08 00 45 c0
0010  00 44 be a3 00 00 01 59  0d f7 0a 00 02 02 e0 00
0020  00 05 02 01 00 30 0a 00  02 02 00 00 00 01 cc ac
0030  00 00 00 00 00 00 00 00  00 00 ff ff ff 00 00 05
0040  02 01 00 00 00 14 0a 00  02 01 0a 00 02 02 0a 00
0050  02 01
```


La traza se genera con:

```
$ tcpdump -i bth-r-1-c -w OSPF.pcap
```

Ejecutado en el anfitrión.

OSPF.pcap Packets: 6 · Displayed: 6 (100.0%) Profile: Default

¿Qué más se puede probar?

- Implementar enlaces inalámbricos con `mac80211_hwsim`.
- Configurar y verificar técnicas de QoS con `tc(8)`.
  - `LARTC` tiene una documentación fantástica.
- Implementar `MPLS` con **`tc(8)`** y `LDP` con `FFRouting`.
- Configurar técnicas de transición de **`IPv4`** a **`IPv6`**.
- Poner a prueba conjuntos de reglas de **`iptables(8)`**.
- ¡Hay **muchas opciones**!

Resumiendo...

- **Dvnet**:
  - Hace uso de **tecnologías** con **gran** base de **usuarios**.
  - Nos hace **transparente** la **complejidad** de la virtualización.
  - Nos permite trabajar con redes emuladas de **gran escala.**
  - No requiere ser experto de **`Docker`**: ¡lo que hemos visto vale!
  - Permite **emular** cualquier **red**.
- La implementación está disponible en `pcolladadosoto/dvnet`.

¿Preguntas?

Fuente