

Aplicación de técnicas de virtualización ligera para la evaluación de redes de comunicaciones

Trabajo Final de Estudios

Ingeniería Telemática

Autor: Enrique Fernández Sánchez

Tutor: Josemaria Malgosa Sanahuja

Universidad Politécnica de Cartagena

Revisión Noviembre 2021

Índice

Índice de figuras	4
Índice de tablas	4
Listado de ejemplos	4
Glosario de términos	6
Agradecimientos	7
1 Introducción	8
1.1 Contexto del trabajo	8
1.2 Objetivos	9
1.3 Descripción de los capítulos restantes de la memoria	9
2 Virtualización de Funciones de Red (NFV)	10
2.1 Tecnologías implicadas	12
2.2 Virtualización ligera	13
3 Interfaces de red virtuales en <i>Linux</i>	14
3.1 Nombres predecibles en dispositivos físicos.	14
3.2 MAC compartida <code>enp2s0:{0,1,2...}</code>	16
3.3 VLAN 802.1q <code>enp2s0.{0,1,2...}</code>	16
3.4 VLAN 802.1ad. <code>enp2s0.{0,1,2...}.{0,1,2...}</code>	17
3.5 Pares VETH.	18
3.6 TUN/TAP	20
4 Espacio de nombres en <i>Linux</i>	29
4.1 ¿Qué es un <i>espacio de nombres</i> ?	29
4.2 ¿Cómo crear/acceder a un <i>namespace</i> ?	29
4.3 ¿Cuántos <i>namespaces</i> hay?	30
4.3.1 UTS namespace	31
4.3.2 Mount namespace	32
4.3.3 Process ID namespace	34
4.3.4 Network namespace (netns)	35
4.3.5 User ID (user)	38
4.3.6 Interprocess Communication namespace (IPC)	38
4.3.7 Control group (cgroup)	39
4.3.8 Time	42
4.4 Ejemplo de uso de 'netns' usando comando <code>ip</code>	43
4.5 Ejemplo de uso de 'netns' usando comando <code>unshare</code>	45

5	Virtualización ligera y contenedores	46
5.1	Creando nuestro propio “contenedor”	47
5.2	Contenedores LXC	48
5.3	Contenedores Docker	53
6	Caso práctico: Virtualización para simulación de redes	61
6.1	Interconexión física de diferentes red virtuales	61
6.2	Evaluación de prestaciones	61
7	Conclusiones	62
7.1	Propuestas futuras	62
	Bibliografía	63
	Enlaces y referencias	63
	Imágenes	65
	Anexos	66
	Anexo 1. Instalación ansible	66
	Anexo 2. Configuración de <i>Guest Network</i> para comunicar con la VM	68
	Anexo 3. Playbooks ansible	69

Índice de figuras

1	Comparativa enfoque clásico de las redes contra el enfoque virtualizado. [1]	10
2	Comparativa redes clásicas vs redes basadas en NFV [2]	11
3	Comparativa en capas OSI de las tecnologías NFV y SDN [3]	12
4	Comparativa entre virtualización ligera y virtualización dura [7]	13
5	Regla udev definida por el usuario	15
6	Diagrama conexión VLANs	16
7	Trama ethernet utilizando VLAN 802.1ad [4]	17
8	Ejemplo básico de utilización de pares virtuales ethernet	18
9	Ejemplo avanzado de utilización de pares virtuales ethernet, utilizando bridge	19
10	Comparativa en capa OSI de las interfaces TUN/TAP. [6]	21
11	Uso de la aplicación sock con interfaces tuntap.	27
12	Tabla de encamientamiento para una interfaz tuntap.	27
13	Captura de un paquete emitido por la interfaz tap0.	28
14	Archivos asociados al grupo test_cg una vez lo creamos	39
15	Salida tras ejecutar el programa de Python sin limitar	40
16	Salida tras ejecutar el programa de Python, una vez limitado	40
17	Diferentes namespaces en Linux y su API de acceso. (5)	41
18	Logotipo del proyecto linuxcontainers.org [8]	48
19	Arquitectura Docker [9]	53
20	Logotipo de Docker [10]	54
21	Ejecución comando docker version sin “demonio” en funcionamiento	54
22	Ejecución comando docker run hello-world para comprobar instalación	55
23	Ejecución comando docker run -it ubuntu bash para levantar un contenedor ubuntu	57
24	Comprobación de la versión de ubuntu del contenedor desplegado	57
25	Comprobación de la versión de ubuntu del contenedor desplegado	58
26	Búsqueda en DockerHub de una imagen base para nuestro contenedor.	59
27	Creación de imagen Docker para aplicación Python de pruebas	60
28	Ejecución de la imagen Docker creada, utilizando Dockerfile	60

Índice de tablas

Listado de ejemplos

1	Ejemplo de uso de tunc1 para controlar interfaces TUN/TAP [33]	22
5	Compilar e instalar programa sock (37)	26
6	Creación interfaz TAP	26
7	Uso de aplicacion sock para crear cliente servidor asociado a un puerto	26
8	Ejemplo de un persistencia namespace	29
9	Ejemplo de uso de UTS namespace	31

10	Uso de mount namespace con “bind”	32
11	Uso de mount namespaces con “tmpfs”	33
12	Uso de process id namespace	34
13	Creation persistent network namespace	35
14	Comprobar network namespaces existentes	35
15	Asignar interfaz loopback a un namespace	35
16	Ejecutar cualquier programa con un network namespace	35
17	Ejemplo configuración de NAT entre eth0 y veth	36
18	Configuración de NAT para dar Internet a un network namespace	37
19	Ejemplo de uso UID namespace	38
20	Programa en Python que consume 4 GB de RAM	40
21	Configuración interfaz NAT bridge en LXC	49
22	Configuración contenedor LXC para usar NAT bridge	50
23	Crear un contenedor con privilegios en LXC	50
24	Crear un contenedor con privilegios en LXC, modo no interactivo	50
25	Configuración interfaz NAT bridge y aplicaciones X a un contenedor LXC	51
26	Asignar IP al host LXC y arrancar un contenedor	51
27	Configuración para mapear UID y GID para un contenedor sin privilegios en LXC	52
28	Código Python de ejemplo para crear un Dockerfile	58
29	Contenido del archivo Dockerfile para crear contenedor con código Python	59
30	Instalación ansible en Ubuntu	66
31	Instalación ansible en Fedora	66
32	Instalación ansible en OpenSUSE	66
33	Instalación ansible en Arch Linux	66
34	Crear clave SSH para autenticación en VM	66
35	Copiar clave pública de ansible a VM	67
36	Contenido del archivo inventory de ansible	67

Glosario de términos

NFV (*Network Function Virtualization*). *Network Function Virtualization*, o bien, Virtualización de funciones de red.

SDN (*Software Defined Networks*)

Namespace. *Espacio de nombres*

NS (*Network namespace*). Tipo de espacio de nombres en Linux que tiene por función aislar la parte de red de la máquina host.

Linux. Sistema operativo tipo UNIX, de código abierto, multiplataforma, multiusuario y multitarea.

Unix. Sistema operativo portable, multitarea y multiusuario desarrollado a partir de 1969. Se divide en tres tareas básicas: el núcleo de sistema operativo, el intérprete de comandos y programadas de utilidades.

Kernel de Linux. Núcleo del sistema operativo Linux.

PID (*Process Identifier*). Identificador de procesos que están ejecutándose bajo un sistema tipo Linux.

UID (*User identifier*). Encontrado normalmente como un número o palabra, supone un identificador de usuario dentro del sistema Linux.

GID (*Group Identifier*). Al igual que sucede con el UID, suele aparecer como un número o palabra y hace referencia al identificador de grupo dentro del sistema de Linux.

root. Cuenta superusuario del sistema operativo Linux.

veth-pair (*Virtual Ethernet Pair*). Tipo de interfaz virtual. Funcionan de dos en parejas.

Agradecimientos

1 Introducción

Con el fin de concluir los estudios de grado en ingeniería telemática, es necesaria la investigación y el posterior desarrollo del *Trabajo Fin de Estudios* (TFE). Dicho trabajo, tiene como objetivo enfrentar al alumno a un proceso de investigación en el que pueda aplicar los conceptos que ha ido aprendiendo durante su paso por el grado, además pudiendo añadir puntos de innovación, y aportar soluciones nuevas a un proyecto en específico.

En este documento, recojo lo que sería mi memoria en relación al TFE. En dicho documento detallaremos las diferentes investigaciones realizadas sobre el concepto de virtualización en sistemas Linux, como funcionan los contenedores y como utilizar la virtualización ligera para la evaluación de redes de comunicaciones, en nuestro caso, de conmutación de paquetes.

1.1 Contexto del trabajo

Este proyecto nace con el objetivo de profundizar en conceptos novedosos para el ámbito de red, como por ejemplo podría ser NFV. Definimos NFV (Network Function Virtualization) como la virtualización de hardware físico de red, con el fin de solucionar problemas de escalabilidad y de optimización.

Además, trabajaremos sobre otras tecnologías igual de importantes como pueden ser la virtualización de recursos, o los bien conocidos *contenedores*. Particularizaremos estas tecnologías y las acercamos al campo de conocimiento de la telemática para utilizarlas con el fin de evaluar y simular redes de conmutación de tipo IP.

Partiendo de supuesto de que las herramientas más sencillas de simulación de redes se nos pueden quedar cortas en cuanto queremos escalar el sistema, nos topamos en que otras soluciones pueden ser mucho más *resource hungry* de lo que podríamos imaginar. Además, tampoco podemos contar con realizar dichas pruebas de manera física, ya que el presupuesto del proyecto escalaría de nivel exponencial. Esto sucede ya que deberíamos configurar cada dispositivo de manera específica, y después proceder a interconectarlos con una tecnología de red adecuada, que en nuestro caso, son tecnologías en fase de desarrollo o evaluación de prestaciones.

1.2 Objetivos

Como bien hemos adelantado en el apartado anterior, el objetivo principal de este proyecto es el de analizar la situación en el ámbito de simulación y evaluación de redes de comunicación, además, concretar en aquellas soluciones que estén basadas en virtualización ligera. Por lo tanto, se pretende:

- Aprender los conceptos básicos de la virtualización de sistemas de red (NFV).
- Comprender la diferencia entre virtualización *ligera* y virtualización *dura*.
- Estudiar, dentro del sistema operativo Linux, las diferentes tecnologías que nos permiten adoptar soluciones NFV.
- Definir que es *espacio de nombres* y como podemos aplicarlo para virtualizar redes.
- Profundizar en el concepto de *interfaces virtuales* en Linux.
- Desgranar el concepto amplio de contenedor, relacionandolo con los *espacios de nombres*.
- Desarrollar una aplicación conceptual para la evaluación de un sistema en concreto, utilizando virtualización ligera.

1.3 Descripción de los capítulos restantes de la memoria

En este apartado se comentará brevemente la distribución de capítulos de la memoria. Además, se mencionará que temas se han abordado en cada uno de ellos.

- **Capítulo 1:** Introducción.
- **Capítulo 2:** Virtualización de funciones de red.
- **Capítulo 3:** Interfaces de red virtuales en Linux.
- **Capítulo 4:** Espacio de nombres en Linux.
- **Capítulo 5:** Virtualización ligera y contenedores.
- **Capítulo 6:** Caso práctico: Virtualización para la simulación de redes.
- **Capítulo 7:** Conclusiones.

2 Virtualización de Funciones de Red (NFV)

El punto de partida de “virtualización de funciones de red” surge a partir de las necesidades de las operadoras para solucionar problemas de gran escala de la red. Estos se puede resumir en los siguientes:

- Saturación en la red o en servicios específicos de la red.
- Servicios que requieren una instalación manual o que necesitan una intervención manual.
- Problemas relacionados con operadoras, como puede ser la reducción de costes del servicio.

Estos problemas radican en una serie de motivos, pero principalmente tiene que ver con el poco crecimiento de la red, y su rápida adopción por la sociedad. Si concretamos estos motivos, podrían ser tal que:

- Falta de estabilidad, debido a la poca flexibilidad de la misma.
- Poca evolución en las redes “core”.
- Falta de innovación, se hace realmente difícil crear nuevos servicios.

En general, se podría resumir en que la poca innovación y la falta de flexibilidad, hacían realmente difícil cambiar ciertos servicios y estructuras críticas de la red “core”. Para ello, para solucionar todos estos problemas, desde los grupos de trabajo de la ITU se empezó a trabajar en nuevas propuestas con el fin de aportar nuevas alternativas. La solución propuesta con más apoyo sería “la virtualización de funciones de red”, que tendría su punto de partida en octubre de 2012, en un grupo de trabajo de la ITU, formado por 13 operadoras internacionales, dando lugar a un paper informativo [19] en el que se detallaba de forma teórica la solución de NFV.

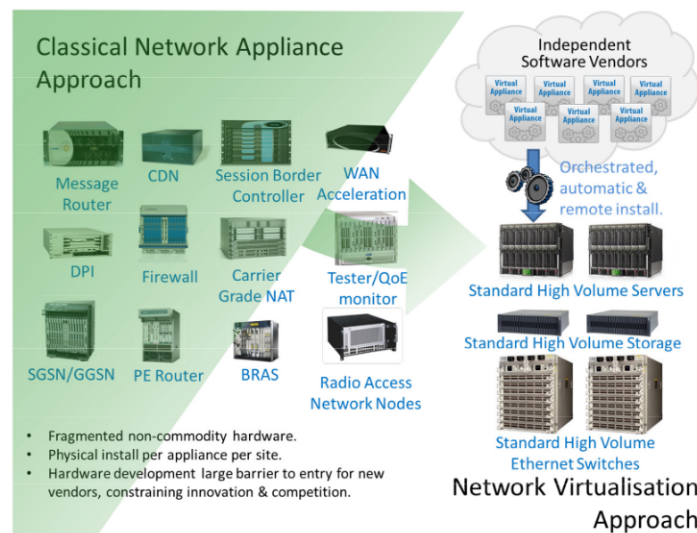


Figura 1: Comparativa enfoque clásico de las redes contra el enfoque virtualizado. [1]

La actualidad es cada dispositivo, corresponde con un aparato físico. Son aparatos embebidos y solo cumplen una función específica. Utilizando la virtualización de red, podemos llegar al punto de tenerlo todo virtualizado. Tenemos una 'imagen' de router que la podemos desplegar en cualquier ordenador de carácter general, y cumplir diferentes funciones a la vez (que un router sea a la vez un firewall, por ejemplo).

Las funciones de red están basadas en tener un software y hardware específico para cada dispositivo. NFV nos aporta que esos recursos software y hardware, se despliegan en servidores físicos de propósito general. Por lo tanto, un mismo nodo físico, puede ser DHCP, router o Firewall. [18]

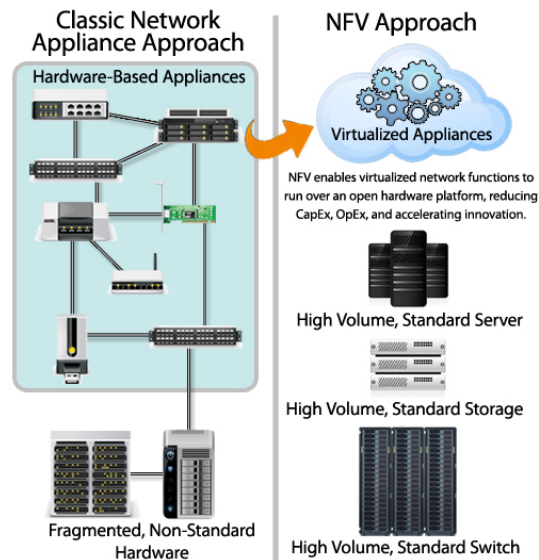


Figura 2: Comparativa redes clásicas vs redes basadas en NFV [2]

La virtualización de funciones de red supone una manera para reducir costes y acelerar el desarrollo de servicios para los operadores de red a costa de desacoplar funciones como pueden ser un "firewall" en un hardware dedicado, moviéndolos a servidores virtuales. Tal y como podemos ver en la comparativa de la Figura 2, sustituimos electrónica de red específica, como podrían ser router, switches, etc; por máquinas virtualizadas que se despliegan en servidores de carácter general, dando lugar a un mayor control y escalabilidad de los sistemas físicos. A consecuencia de esto, podemos ver como las redes toman un camino diferente, dejando a atrás el hardware y software propietario, para centrarse en un enfoque basado en el software.

2.1 Tecnologías implicadas

Para que este cambio de paradigma se materialice, es decir, pase de ser “papers” con soluciones generalistas, y que de verdad estas se materialicen en una solución viable, tienen que realizarse una serie de desarrollos, los cuales provocan que aún a día de hoy sean soluciones experimentales, que aunque son utilizadas en entornos reales, todavía están en continuo desarrollo. Además, aunque estamos hablando de NFV, hay una tecnología extra que también aparece en la ecuación, esta tecnología es el SDN (*Software Defined Networks*). En este caso particular, nos encontramos con que el NFV no puede existir sin el SDN, y viceversa [31]. Si tuviéramos que definir brevemente cada una de dichas tecnologías, podríamos resumirlo en lo siguiente:

- *Software Defined Networks*. Suponen un punto de vista de las redes, en las que las propias redes utilizan una serie de controladores, basados en software o en API de control, con el fin de dirigir el tráfico de red y comunicarse con la infraestructura hardware de capas superiores. (ver Figura 3)
- *Network Functions Virtualizations*. NFV desacopla las funciones de la red de dispositivos de hardware dedicados y las traslada a servidores virtuales, y así se consolidan múltiples funciones en un único servidor físico. Dentro de este servidor físico, podemos distinguir diferentes funciones de red virtuales (VNF), dichas funciones suponen un conjunto de máquinas interconectadas entre sí, y cada una de ellas tiene una función distinta, y además, el conjunto de ellas tienen por objetivo realizar una función que antes era realizada por un equipo físico determinado (un router, firewall o similar). (ver Figura 3)

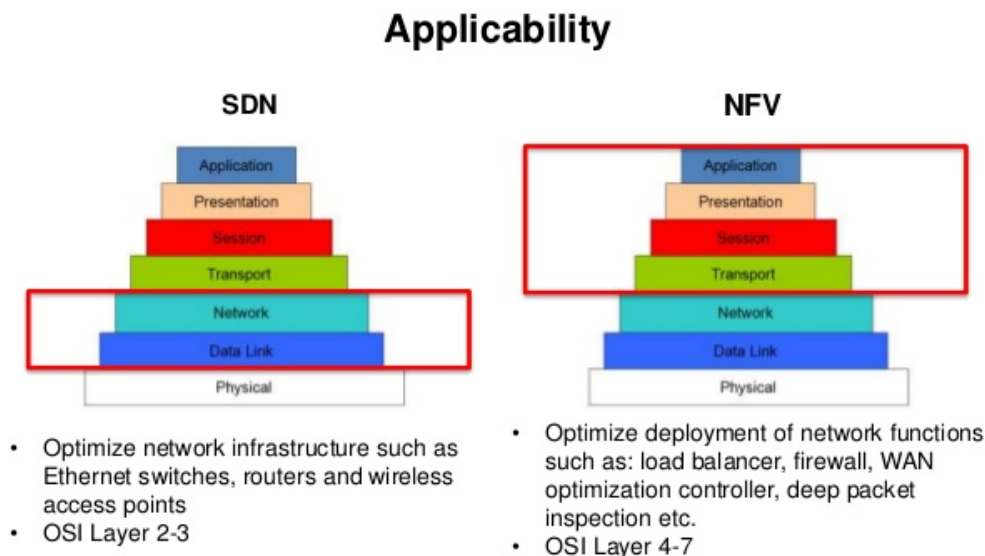


Figura 3: Comparativa en capas OSI de las tecnologías NFV y SDN [3]

2.2 Virtualización ligera

Entendemos la virtualización ligera como un tipo de virtualización de un sistema. Dicha virtualización se hace a nivel de sistema operativo, aportando que existan diferentes espacios de usuarios aislados entre sí, sin embargo, a diferencia del concepto ‘*máquina virtual*’ (virtualización dura), en la virtualización ligera todos los sistemas virtualizados dependen del kernel de la máquina host, mientras que en el caso de una máquina virtual, cada instancia tendría su propio kernel (ver Figura 4). [36]

Por lo tanto, las ventajas destacables sobre la virtualización ligera frente a la virtualización ligera serían:

- Un único kernel para todas las instancias que queramos ejecutar.
- Eliminamos el correspondiente *overhead* al no tener que emular un kernel para cada instancia.
- Menor consumo de CPU y RAM, comparado con las virtualizaciones duras.
- Disponen de la posibilidad de iniciar, mover, parar o borrar la instancia virtualizada.

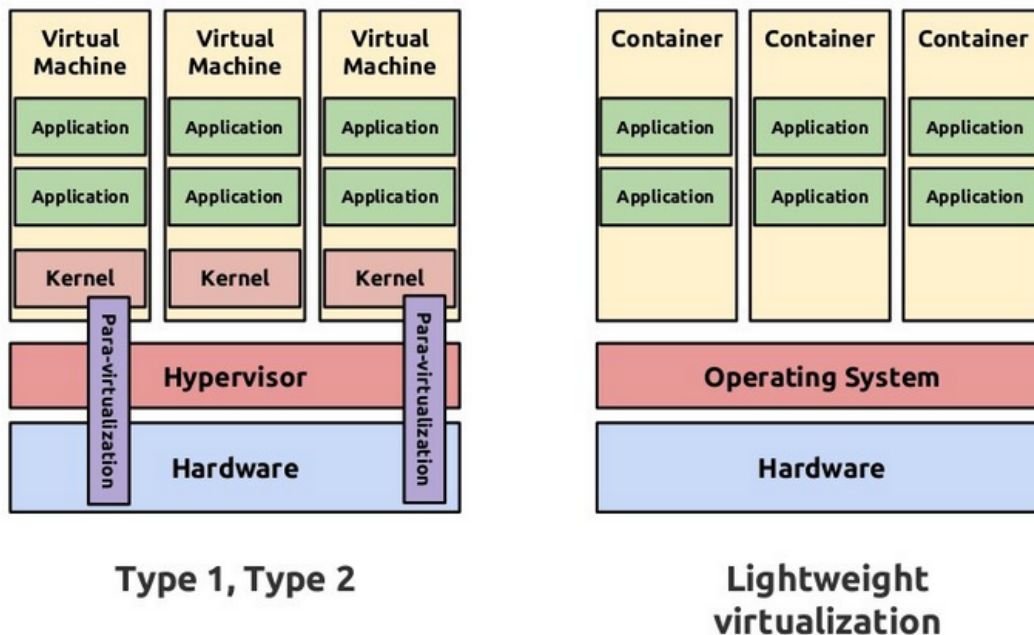


Figura 4: Comparativa entre virtualización ligera y virtualización dura [7]

En consecuencia de tener que utilizar un mismo kernel para todas las instancias, tenemos que profundizar sobre los conceptos de interfaces de red virtuales [3] y espacios de nombres [4], ya que serán piezas clave para tener nuestras instancias aisladas entre sí, pero a su vez conectadas con los diferentes recursos en red que nosotros definamos.

3 Interfaces de red virtuales en *Linux*

En este capítulo, vamos a profundizar en el concepto de interfaces de red virtuales, pero las específicas al sistema operativo basados en el kernel de Linux. Hacemos esto ya que los servidores utilizados a gran escala, la mayoría hacen uso de distribuciones de Linux.

Linux dispone de una selección muy diferente de interfaces de red que nos permiten, de manera sencilla, el uso de máquinas virtuales o contenedores. En este apartado vamos a mencionar las interfaces más relevantes de cara a la virtualización ligera que proponemos para el despliegue de una red virtualizada. Para obtener una lista completa de las interfaces disponibles, podemos ejecutar el siguiente comando `ip link help`.

En este trabajo, vamos a comentar la siguientes interfaces [11]:

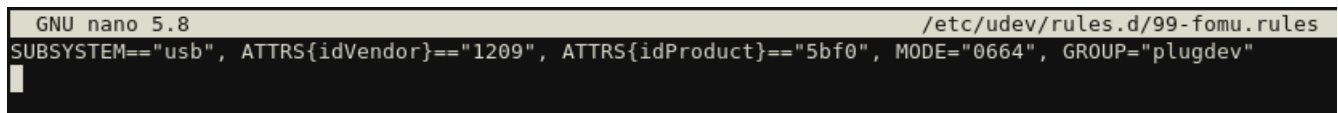
- MAC compartida: `eth0:{0,1,2...}`
- VLAN 802.1q: `eth0.{0,1,2...}`
- VLAN 802.1ad: `eth0.{0,1,2...}.{0,1,2...}`
- Pares de ethernet virtuales (*veth pairs*)
- TUN/TAP

3.1 Nombres predecibles en dispositivos físicos.

Con el fin de comprender como se nombran las diferentes interfaces de red, es necesario que estudiemos el servicio `udev` (*Dynamic device management*). `udev` es el gestor de dispositivos que utiliza el kernel de Linux, su función principal es permitir al administrador del sistema operativo registrar manejadores del espacio de usuario para eventos del sistema. Estos eventos que recibe el servicio `udev` son principalmente generados por el kernel, en respuesta a eventos físicos relacionados con dispositivos periféricos. Por lo tanto, el propósito de `udev` es actuar sobre la detección de periféricos y con conexiones tipo *hotplug*, pudiendo encadenar acciones necesarias que devuelven el control al kernel, como por ejemplo, cargar módulos específicos o un firmware de un dispositivo. [24]

Además, `udev` también se encarga de gestionar los nodos de dispositivos en el directorio `/dev` añadiendo, enlazando simbólicamente y renombrándolos. Por otro lado, una consideración a tener en cuenta es que `udev` maneja los eventos de manera concurrente, lo que aporta una mejora de rendimiento, pero a la vez puede dar problemas a la hora de que el orden de carga de los módulos del kernel no se conserva entre los arranques. Un ejemplo de esto podría ser que en el caso de tener dos discos duros (uno llamado `/dev/sda` y otro `/dev/sdb`), en el siguiente arranque el orden de de arranque puede variar, generando que ambos identificadores se intercambien entre sí, desencadenando una serie de problemas en nuestro sistema. [24]

A modo de ejemplo, el usuario puede crear sus propias reglas, de modo que puede realizar las acciones que ya hemos comentado, de acuerdo a sus propias necesidades. A modo de ejemplo, podemos ver en la siguiente captura (Figura 5) como hemos creado un archivo en la ruta `/etc/udev/rules.d/` en el que definimos la regla para un dispositivo físico en específico (en este caso un USB). Identificamos el dispositivo que queremos a través de los atributos `idVendor` e `idProduct` (atributos necesarios para cualquier dispositivo USB), después le asignamos un "MODE" que corresponde con los permisos que le queremos asignar al dispositivo (en modo numérico) y el grupo al que permitimos acceder al dispositivo.



```
GNU nano 5.8 /etc/udev/rules.d/99-fomu.rules
SUBSYSTEM=="usb", ATTRS{idVendor}=="1209", ATTRS{idProduct}=="5bf0", MODE="0664", GROUP="plugdev"
```

Figura 5: Regla udev definida por el usuario

En el caso de las interfaces físicas de red, vamos a suponer que estamos utilizando el nombrado de interfaces de red antiguo. Esto proviene de que en la últimas versiones del kernel, se ha cambiado la forma en la que las interfaces de red son nombradas por Linux (`systemd networkd v197` [25]). Es por esto por lo que antes podíamos tener interfaces tal que `eth0` y ahora nos encontramos con la siguiente nomenclatura `enps30`. Este cambio surge ya que anteriormente se nombraban las diferentes interfaces conforme el propio ordenador estaba en la etapa de `boot`, por lo que podría pasar que a lo que nosotros entendíamos como `eth1`, en el próximo arranque fuera `eth0`, dando lugar a incontables errores en el sistema. Es por esto por lo que se empezó a trabajar en soluciones alternativas. Por ejemplo, la que acabamos de comentar, utiliza la información aportada por la BIOS del dispositivo para catalogarlo en diferentes categorías, con su formato de nombre para cada categorías. Dichas clasificación corresponde con las siguientes:

1. Nombres incorporados en Firmware/BIOS que proporcionan un número asociado a dispositivos en la placa base. (ejemplo: `eno1`)
2. Nombres incorporados en Firmware/BIOS proveniente de una conexión PCI Express hotplug, con número asociado al conector. (ejemplo: `ens1`)
3. Nombres que incorporan una localización física de un conector hardware. (ejemplo: `enp2s0`)
4. Nombres que incorporan una la MAC de una interfaz. (ejemplo: `enx78e7d1ea46da`)
5. Sistema clásico e impredecible, asignación de nombres nativa del kernel. (ejemplo: `eth0`)

3.2 MAC compartida `enp2s0:{0,1,2...}`

Todas las interfaces asociadas comparten la misma dirección MAC. Cada una de ellas, recibe el nombre de *aliases*. La funcionalidad principal que tienen este tipo de interfaces es la de asignar varias direcciones de IP a una misma interfaz de red.

```
$ ip addr add 192.168.56.151/24 broadcast 192.168.56.255 dev enp2s0  
label enp2s0:1
```

Sin embargo, el comando `iproute2` admite esta misma funcionalidad sin tener que crear interfaces de red extra. Para ello, solo tenemos que asociar cada IP con la interfaz de red deseada.

```
$ ip addr add 192.168.56.151/24 dev enp2s0  
$ ip addr add 192.168.56.251/24 dev enp2s0
```

3.3 VLAN 802.1q `enp2s0.{0,1,2...}`

Siguiendo el mismo concepto que la interfaz anterior, pero en este caso utilizando el estándar 802.1q, que permite etiquetar las tramas, para crear una red lógica independiente. Es necesario que la interfaz a la que estemos asignando, sea un puerto trunk, o bien sea tagged para una VLAN específica.

```
$ ip link add link enp2s0 name enp2s0.{num} type vlan id {num}  
$ ip addr add 192.168.100.1/24 brd 192.168.100.255 dev enp2s0.{num}  
$ ip link set dev enp2s0.{num} up
```

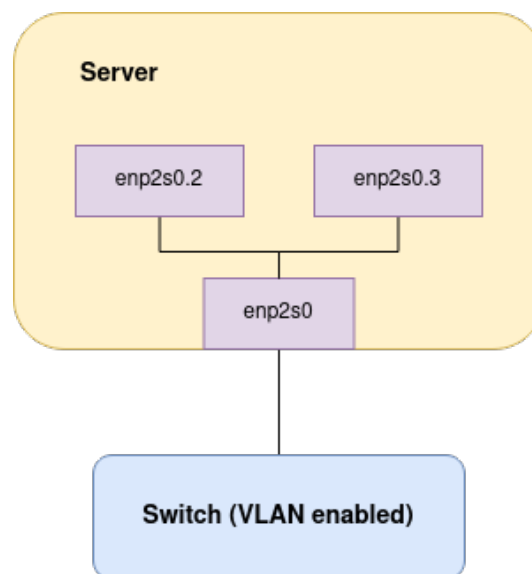


Figura 6: Diagrama conexión VLANs

3.4 VLAN 802.1ad. $\text{enp2s0}\{0,1,2\}\{0,1,2\}$

En este apartado comentamos la interfaz virtual asociada al estándar 802.1ad, dicho estándar supone una actualización respecto a las VLANs basadas en 802.1q, pero añadiendo la posibilidad de tener dos tags dentro de mismo frame ethernet. En el caso del estándar 802.1q, solo podíamos tener un tag. El estándar de vlans 802.1ad es realmente útil cuando el proveedor de red y el usuario de dicha red quieren utilizar VLANs, además que amplía el límite de 4094 VLANs diferentes permitidas por el estándar 802.1Q [28][29]. Otra forma de la que nos podemos encontrar esta interfaz es bajo el nombre “QinQ”.

La estructura de la trama ethernet sigue la siguiente estructura,

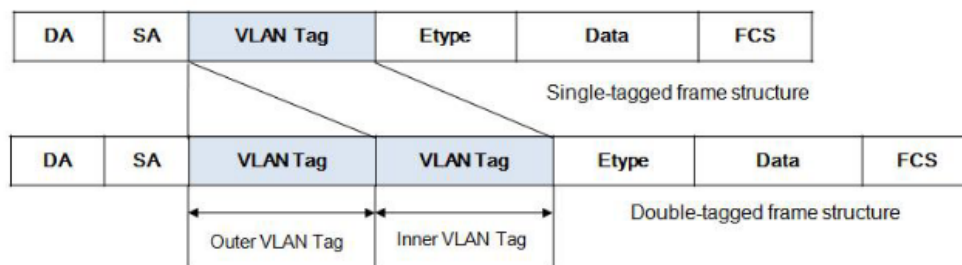


Figura 7: Trama ethernet utilizando VLAN 802.1ad [4]

Para configurar un enlace utilizando este estándar, tenemos que ejecutar los siguientes comandos [30]:

```
$ ip link add link eth0 eth0.1000 type vlan proto 802.1ad id 1000
$ ip link add link eth0.1000 eth0.1000.1000 type vlan proto 802.1q id 1000
```

De esta manera, lo que hacemos es asociar una primera VLAN a una interfaz de red, utilizando 802.1ad, después a esa misma interfaz con identificador, podemos asignar otra nueva VLAN, pero esta vez utilizando el estándar 802.1q. Por lo tanto, al final nos quedaría una interfaz similar a `eth0.1000.1000` en la que podemos distinguir dos identificadores de red virtual.

3.5 Pares VETH.

Los `veth` (Ethernet virtuales) son un dispositivo virtual que forman un túnel local ethernet. El dispositivo se crea en parejas. [11]

Los paquetes transmitidos por un extremo del ethernet virtual se reciben inmediatamente en el otro extremo. Si alguno de ellos se encuentra apagado, decimos que el link de la pareja esta también apagado. A modo de ejemplo, nos fijamos una estructura básica en la que dos aplicaciones se comunican utilizando `veth`, tal y como vemos en la figura (8)

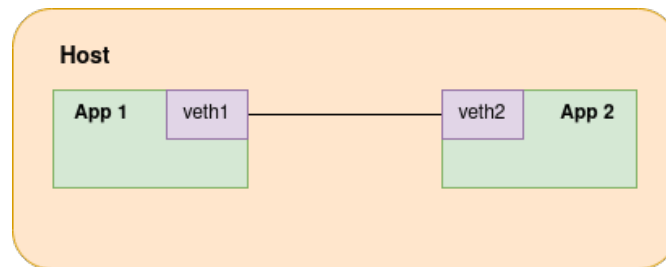


Figura 8: Ejemplo básico de utilización de pares virtuales ethernet

Las interfaces virtuales ethernet se inventaron con el fin de comunicar diferentes *network namespaces*. Aunque profundizaremos en ello más adelante, los namespace de Linux permiten encapsular recursos globales del sistema de forma aislada, evitando que puedan interferir con procesos que estén fuera del namespace.

La configuración necesaria para implementar el ejemplo de la figura 8 sería el siguiente [14]:

```
$ ip netns add app1
$ ip netns add app2
$ ip link add veth1 netns app1 type veth peer name veth2 netns app2
```

De esta manera, tendríamos creados los namespaces `app1` y `app2`, que estarían interconectados entre sí. Ahora procedemos a asignar una IP a cada interfaz.

```
$ ip netns exec app1 ip addr add 10.1.1.1/24 dev veth1
$ ip netns exec app1 ip link set veth1 up
$ ip netns exec app2 ip addr add 10.1.1.2/24 dev veth2
$ ip netns exec app2 ip link set veth2 up
```

Para comprobar que hay conectividad entre las diferentes aplicaciones (app1 y app2) utilizamos la función del comando `ip` para ejecutar programas dentro de un `network namespace`, en este caso realizar un ping entre ambas aplicaciones:

```
$ ip netns exec app1 ping 10.1.1.2
```

Por otro lado, si quisiéramos una topología más compleja, como por ejemplo que varios namespaces puedan hacer uso de una interfaz física, tendríamos que añadir un elemento extra a nuestro sistema. El diagrama de la topología podría ser tal que así:

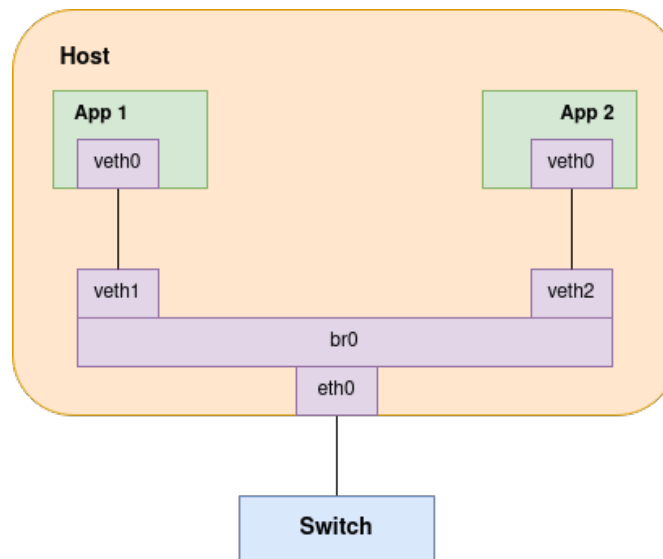


Figura 9: Ejemplo avanzado de utilización de pares virtuales ethernet, utilizando bridge

Como podemos comprobar en la figura 9, es necesario que utilicemos un “bridge” para que podamos conectar ambas interfaces virtuales a una interfaz física, para replicar dicha topología, ejecutaremos los siguientes comandos:

```
$ ip netns add app1
$ ip netns add app2
$ ip link add veth1_br type veth peer name veth0 netns app1
$ ip link add veth2_br type veth peer name veth0 netns app2
```

Para definir un bridge entre las diferentes interfaces virtuales que hemos creado, utilizaremos una interfaz tipo *bridge* de Linux, o bien podemos configurar dicho *bridge* usando *Open vSwitch*. *Open vSwitch* es un programa de código abierto diseñado para ser utilizado como un switch multi-capa virtual [26][27].

```
$ ip link add veth1_br type veth peer name veth0 netns app1
$ ip link add veth2_br type veth peer name veth0 netns app2
$ ovs-vsctl add-br ovsbr0
```

```
$ ovs-vsctl add-port ovsbr0 veth1_br
$ ovs-vsctl add-port ovsbr0 veth2_br
$ ovs-vsctl add-port ovsbr0 eth0
```

Por último, añadimos las direcciones IP que nos faltan en la topología:

```
$ ip addr add 10.1.1.10/24 dev veth1_br
$ ip link set veth1_br up
$ ip addr add 10.1.1.20/24 dev veth2_br
$ ip link set veth2_br up
$ ip netns exec app1 ip addr add 10.1.1.15/24 dev veth0
$ ip netns exec app1 ip link set veth0 up
$ ip netns exec app2 ip addr add 10.1.1.25/24 dev veth0
$ ip netns exec app2 ip link set veth0 up
$ ip netns exec app1 ip link set lo up
$ ip netns exec app2 ip link set lo up
```

De esta manera, ya tendríamos la topología configurada con conectividad entre las diferentes aplicaciones, además de cada aplicación con una interfaz física de la máquina, todo esto utilizando una interfaz tipo bridge.

3.6 TUN/TAP

TUN/TAP son dos interfaces de red virtuales de Linux, que permiten dar conectividad entre programas dentro del espacio de usuario, es decir permiten conectar aplicaciones en específico con el kernel. Esta interfaz es expuesta al usuario mediante la ruta `/dev/net/tun`. Como bien hemos mencionado, existen dos tipos de interfaces virtuales controladas por `/dev/net/tun`:

- TUN. Interfaces encargadas de transportar paquetes IP (trabaja sobre la capa 3).
- TAP. Interfaces encargadas de transportar paquetes Ethernet (trabaja sobre la capa 2).

TUN (Capa 3)

Las interfaces TUN (`IFF_TUN`) transportan paquetes PDU (*Protocol Data Units*) de la capa 3 [32]:

- En la práctica, transporta paquetes IPv4 y/o paquetes IPv6.
- La función `read()` devuelve un paquete de capa 3 PDU, es decir un paquete IP.
- Utilizando la función `write()` podemos enviar un paquete IP.
- No hay capa 2 en esta interfaz, por lo que los mecanismos que se ejecutan en esta capa no estarán presentes en la comunicación. Por ejemplo, no tenemos ARP.
- Pueden funcionar como interfaces tipo *Point to Point*.

TAP (Capa 2)

Las interfaces TAP (IFF_TAP) transportan paquetes de capa 2 32:

- En la práctica, transporta *frames Ethernet*, por lo tanto, actuaría como si fuera un adaptador virtual de Ethernet (“bridge virtual”).
- La función `read()` devuelve un paquete de capa L2, un *frame Ethernet*.
- Utilizando la función `write()` permite enviar un *frame Ethernet*.
- Podemos cambiar la MAC asociada a nuestra interfaz TAP utilizando el parámetro `SIOCSIFHWADDR`, en la función `ioctl()`, la cual usamos para crear un TUN/TAP dentro de nuestra aplicación.

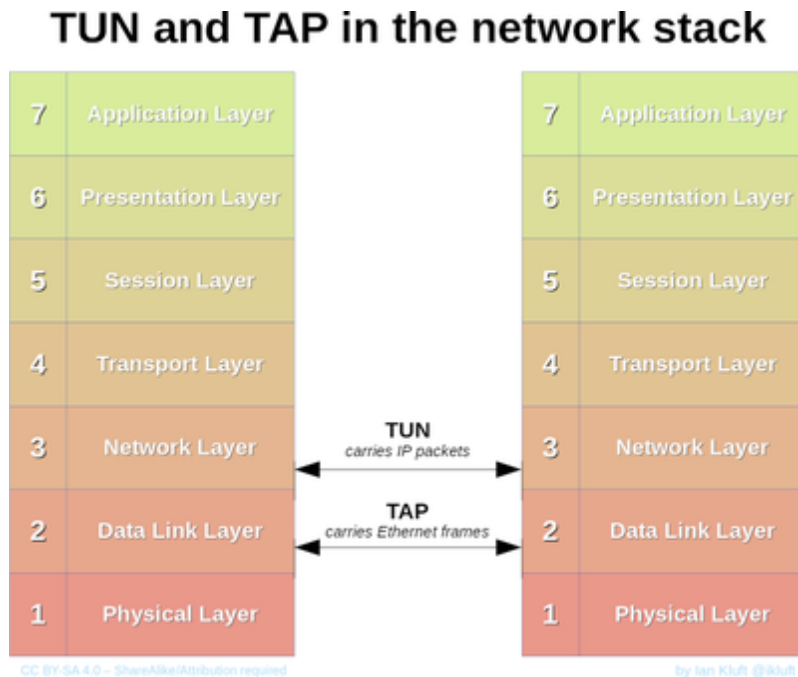


Figura 10: Comparativa en capa OSI de las interfaces TUN/TAP. [6]

Si nos fijamos en la figura 10, podemos ver las diferencias entre ambas interfaces. Aún así, hay una que se utiliza mucho más que la otra, esta interfaz es TAP. Las interfaces tipo TAP son muy ampliamente utilizadas para realizar “túneles virtuales” para una aplicación en concreto. Sería la propia aplicación la encargada de monitorizar dicho “túnel”. Podemos encontrar aplicaciones TAP en hipervisores o en clientes VPN.

En el caso de que queramos crear interfaces TUN/TAP fuera de una aplicación, es decir, desde la línea de comandos, tendremos que utilizar los programas `tunctl` o `ip`. Para ello, podemos revisar los ejemplos 1 y 2 donde se comentan algunos de los comandos más importantes para trabajar con estas interfaces en cada uno de los programas mencionados.

Ejemplo 1: Ejemplo de uso de `tunctl` para controlar interfaces TUN/TAP [33]

```
1 # Create the tap interface by default
2 tunctl
3 # equivalent to
4 tunctl -p
5
6 # For users 'user' create a tap interface
7 tunctl -u user
8
9 # Create Tun interface
10 tunctl -n
11 # Configure IP Address for interface and enable
12 ip addr add 192.168.0.254/24 dev tap0
13 ip link set tap0 up
14 # Add routing to interface
15 ip route add 192.168.0.1 dev tap0
16
17 # Delete interface
18 tunctl -d tap0
19
```

Ejemplo 2: Ejemplo de uso de `ip` para controlar interfaces TUN/TAP [33]

```
20 # Show help
21 ip tuntap help
22
23 # Create tun/tap devices
24 ip tuntap add dev tap0 mod tap # create tap
25 ip tuntap add dev tun0 mod tun # create tun
26
27 # Delete tun/tap devices
28 ip tuntap del dev tap0 mod tap # delete tap
29 ip tuntap del dev tun0 mod tun # delete tun
30
```

Ejemplo de aplicación que crea una interfaz tuntap

Por otro lado, otra manera de trabajar con estas interfaces, es dejar que el programa que estemos utilizando cree dichas interfaces. Es por esto por lo que dentro del programa podemos definir que se refiera a la ruta `/dev/net/tun` para crear la interfaz necesaria. A modo de ejemplo, se implementa un programa en C que crea una interfaz TUN/TAP (en el programa elegimos cual de las dos queremos) y que devuelve el tamaño de los paquetes que reciba en dicha interfaz. Este ejemplo nos sirve para comprobar con una aplicación puede crear y gestionar una interfaz, y además, mediante un programa externo de captura de paquetes (Wireshark, tcpdump, etc...) ver los paquetes que llegan a la interfaz y cual es su estructura.

El código utilizado sería el que vemos en [3]. Es importante comentar que en la línea 80 podemos modificar el tipo de interfaz que vamos a crear, usaremos `IFF_TUN` para crear un TUN y `IFF_TAP` para crear un TAP.

Ejemplo 3: Aplicación de ejemplo para crear tun/tap (tuntap.c) [34]

```
31 /**
32 Receive incoming packages over tun/tap device.
33 stdout -> size of received package
34 **/
35
36 #include <net/if.h>
37 #include <sys/ioctl.h>
38 #include <sys/stat.h>
39 #include <fcntl.h>
40 #include <string.h>
41 #include <sys/types.h>
42 #include <linux/if_tun.h>
43 #include <stdlib.h>
44 #include <stdio.h>
45
46 int tun_alloc(int flags)
47 {
48
49     struct ifreq ifr;
50     int fd, err;
51     char *clonedev = "/dev/net/tun";
52
53     if ((fd = open(clonedev, O_RDWR)) < 0) {
54         return fd;
55     }
56
57     memset(&ifr, 0, sizeof(ifr));
58     ifr.ifr_flags = flags;
59
60     if ((err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0) {
61         close(fd);
62         return err;
63     }
64 }
```

```

65 printf("Open tun/tap device: %s for reading...\n", ifr.ifr_name);
66
67 return fd;
68 }
69
70 int main()
71 {
72
73     int tun_fd, nread;
74     char buffer[1500];
75
76     /* Flags: IFF_TUN   - TUN device (no Ethernet headers)
77      *         IFF_TAP   - TAP device
78      *         IFF_NO_PI - Do not provide packet information
79      */
80     tun_fd = tun_alloc(IFF_TAP | IFF_NO_PI);
81
82     if (tun_fd < 0) {
83         perror("Allocating interface");
84         exit(1);
85     }
86
87     while (1) {
88         nread = read(tun_fd, buffer, sizeof(buffer));
89         if (nread < 0) {
90             perror("Reading from interface");
91             close(tun_fd);
92             exit(1);
93         }
94
95         printf("Read %d bytes from tun/tap device\n", nread);
96     }
97     return 0;
98 }
99

```


Los pasos para realizar este ejemplo serían los siguientes:

Ejemplo 4: Instrucciones para realizar las pruebas con el código [3]

```
100 # Guardamos el codigo anterior en tuntap.c
101 # Compilamos el programa
102 gcc tuntap.c -o tun
103
104 ### Terminal 1
105 # Ejecutamos el binario, este se quedara a la escucha en la interfaz creada
106 ./tun
107
108 ### Terminal 2
109 # Asignamos una IP a la interfaz recién creada
110 ip addr add 192.168.209.138/24 dev tun0
111 ip link set dev tun0 up
112
113 # Capturamos el trafico que viaja por la interfaz
114 tcpdump -i tun0
115
116 ### Terminal 3
117 # Mandamos trafico a la interfaz creada, un ping por ejemplo:
118 ping -c 4 192.168.209.139 -I tun0
119
```

Ejemplo aislamiento entre puertos usando interfaz tuntap

En este ejemplo, queremos comprobar el funcionamiento de una interfaz tap, pero poniéndonos en el caso de que el usuario cree dicha interfaz mediante el comando `ip` y un programa externo se asocie a dicha interfaz. Para ello, vamos a utilizar la aplicación `sock` [37], que nos servirá para crear un socket IP en un puerto específico.

Ejemplo 5: Compilar e instalar programa `sock` (37)

```
1 mkdir ~/tmp
2 cd tmp
3 tar zxvf sock-0.3.2.tar.gz
4 cd sock-0.3.2
5 ./configure
6 make
7 sudo make install
8
```

Una vez tenemos el programa instalado, podemos proceder a crear la interfaz tap.

Ejemplo 6: Creación interfaz TAP

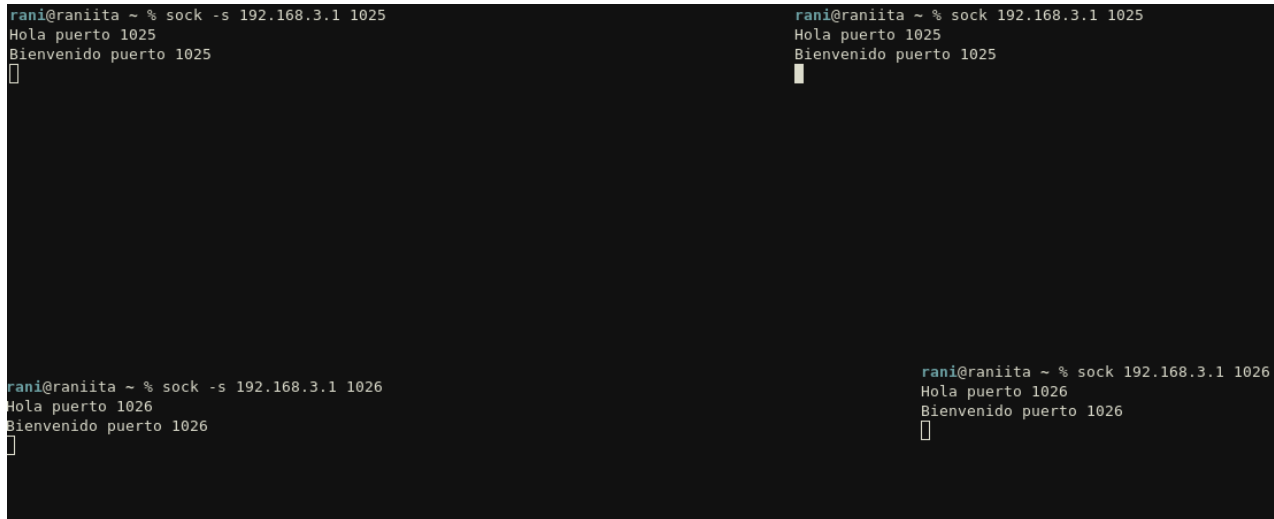
```
1 ip tuntap add dev tap0 mode tap
2 ip address add 192.168.3.1/24 dev tap0
3 ip link set tap0 up
4 ip route add 192.168.3.1 dev tap0
5
```

Una vez ya tenemos creada la interfaz, podemos proceder a comprobar la funcionalidad utilizando el programa `sock`. Para ello, vamos a crear dos instancias cliente-servidor, cada una con un puerto asociado diferente.

Ejemplo 7: Uso de aplicación `sock` para crear cliente servidor asociado a un puerto

```
1 sock -s 192.168.3.1 1025      # Terminal 1 (servidor)
2 sock 192.168.3.1 1025        # Terminal 2 (cliente)
3
4 sock -s 192.168.3.1 1026      # Terminal 3 (servidor)
5 sock 192.168.3.1 1026        # Terminal 4 (cliente)
6
```

De esta manera, podemos escribir en cada una de las terminales, y comprobar que solo son recibidas por el servidor, o cliente, asociado al puerto de la terminal en cuestión. Es decir, en las interfaces tuntap tenemos aislamiento entre los diferentes puertos, por lo que funcionan de manera equivalente a un enlace ethernet físico. En la figura (11) podemos ver el ejemplo realizado.



```
rani@raniita ~ % sock -s 192.168.3.1 1025
Hola puerto 1025
Bienvenido puerto 1025
█

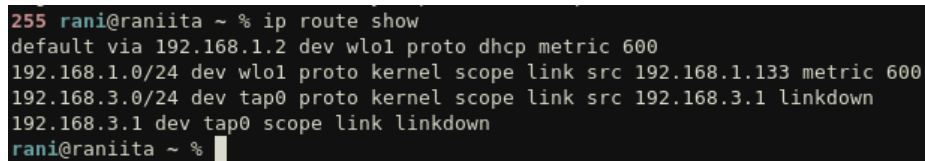
rani@raniita ~ % sock 192.168.3.1 1025
Hola puerto 1025
Bienvenido puerto 1025
█

rani@raniita ~ % sock -s 192.168.3.1 1026
Hola puerto 1026
Bienvenido puerto 1026
█

rani@raniita ~ % sock 192.168.3.1 1026
Hola puerto 1026
Bienvenido puerto 1026
█
```

Figura 11: Uso de la aplicación sock con interfaces tuntap.

Si comprobamos la tabla de enrutamiento de la interfaz creada, `tap0`, podemos ver como aparece asociado el trafico de la IP que le dimos a dicha interfaz, con que utilice la interfaz `tap0`. (Ver figura [12])



```
255 rani@raniita ~ % ip route show
default via 192.168.1.2 dev wlo1 proto dhcp metric 600
192.168.1.0/24 dev wlo1 proto kernel scope link src 192.168.1.133 metric 600
192.168.3.0/24 dev tap0 proto kernel scope link src 192.168.3.1 linkdown
192.168.3.1 dev tap0 scope link linkdown
rani@raniita ~ % █
```

Figura 12: Tabla de encamiamiento para una interfaz tuntap.

Por último, podemos comprobar la estructura de los paquetes enviados utilizando programas como Wireshark, tcpdump, etc. Es importante comentar que aunque los paquetes deberían viajar por la interfaz tap0, el kernel los redirige por loopback. En la figura 13 podemos ver un ejemplo de captura de un paquete enviado por el tap0.

ip.dst == 192.168.3.1						
No.	Time	Source	Destination	Protocol	Length	Info
25	2.849978407	192.168.3.1	192.168.3.1	TCP	74	1025 → 49886 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=8 TSval=605132816 TSecr=605069095
26	2.849998371	192.168.3.1	192.168.3.1	TCP	66	49886 → 1025 [ACK] Seq=1 Ack=9 Win=512 Len=0 TSval=605132816 TSecr=605132816

<ul style="list-style-type: none"> ▶ Frame 26: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface lo, id 0 ▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00) ▶ Internet Protocol Version 4, Src: 192.168.3.1, Dst: 192.168.3.1 ▼ Transmission Control Protocol, Src Port: 49886, Dst Port: 1025, Seq: 1, Ack: 9, Len: 0 <ul style="list-style-type: none"> Source Port: 49886 Destination Port: 1025 [Stream index: 12] [TCP Segment Len: 0] Sequence Number: 1 (relative sequence number) Sequence Number (raw): 3085149397 [Next Sequence Number: 1 (relative sequence number)] Acknowledgment Number: 9 (relative ack number) Acknowledgment number (raw): 1089008822 1000 = Header Length: 32 bytes (8) ▶ Flags: 0x010 (ACK) Window: 512 [Calculated window size: 512] [Window size scaling factor: -1 (unknown)] Checksum: 0x8779 [unverified] [Checksum Status: Unverified] Urgent Pointer: 0 ▶ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps ▶ [SEQ/ACK analysis] ▶ [Timestamps]

Figura 13: Captura de un paquete emitido por la interfaz tap0.

4 Espacio de nombres en *Linux*

4.1 ¿Qué es un *espacio de nombres*?

Los *espacios de nombres*, o también llamados, *namespaces*, son una característica del kernel de Linux que permite gestionar los recursos del kernel, pudiendo limitarlos a un proceso o grupo de procesos. Suponen una base de tecnología que aparece en las técnicas de virtualización más modernas (como puede ser Docker, Kubernetes, etc). A un nivel alto, permiten aislar procesos respecto al resto del kernel.

El objetivo de cada *namespaces* es adquirir una característica global del sistema como una abstracción que haga parecer a los procesos de dentro del *namespace* que tienen su propia instancia aislada del recurso global.

4.2 ¿Cómo crear/acceder a un *namespace*?

Los namespaces normalmente se suelen asociar a procesos o aplicaciones en específico. Para manipular estos namespaces, podemos destacar las siguientes herramientas:

- `unshare`. Permite asociar un namespace a un archivo. Si había un namespace ya en ese archivo, lo sobrescribe. No nos permite reutilizar un namespace.
- `nsenter`. Puede acceder al namespace de un archivo existente. Podemos asociar el namespace a un archivo del sistema, de modo que aunque cerremos el proceso asociado, el archivo sigue existiendo, y por lo tanto puede ser reutilizado.

En resumen, si quisiéramos mantener “vivo” un namespace, sería necesario que lo asociemos con un archivo del sistema, y después volver a “crear” el namespace con la herramienta `nsenter` apuntando a dicho archivo. A modo de ejemplo, sería tal que así:

Ejemplo 8: Ejemplo de un persistencia namespace

```
1 touch /root/ns-uts # Creamos un archivo
2 unshare --uts=/root/ns-uts /bin/bash # Asociamos namespace UTS al archivo
3 hostname FooBar
4
5 # Salimos del namespace
6 exit
7
8 # Volvemos a entrar al namespace
9 nsenter --uts=/root/ns-uts /bin/bash
10 hostname # Nos devuelve 'FooBar'
11
12 # Salimos del namespace
13 exit
14
15 umount /root/ns-uts # Eliminamos el namespace definitivamente
16
```

Tal y como vemos en el ejemplo (8), utilizamos los comandos `unshare` y `nsenter` para manipular un namespace de tipo UTS (`hostname`), veremos más en profundidad este namespace en el

apartado 4.3.1. Lo importante es comprobar que si asociamos un namespace a un archivo, podemos recuperar dicho namespace si utilizamos el comando `nsenter`. Además, si quisieramos eliminar permanentemente dicho namespace, tendríamos que hacer uso del comando `umount`.

4.3 ¿Cuántos *namespaces* hay?

El kernel ha estado en constante evolución desde que 1991, cuando Linus Torvalds comenzó el proyecto, actualmente sigue muy activo y se siguen añadiendo nuevas características. El origen de los namespaces se remonta a la versión del kernel 2.4.19, lanzada en 2002. Conforme fueron pasando los años, más tipos diferentes de namespaces se fueron añadiendo a Linux. El concepto de *User namespaces*, se consideró terminado con la versión 3.9. [1]

Actualmente, tenemos 8 tipos diferentes de namespaces, siendo el último añadido en la versión 5.8 (lanzada el 2 de Agosto de 2020).

1. UTS (hostname)
2. Mount (mnt)
3. Process ID (pid)
4. Network (net)
5. User ID (user)
6. Interprocess Communication (ipc)
7. Control group (cgroup)
8. Time [3]

4.3.1 UTS namespace

El tipo más sencillo de todos los namespaces. La funcionalidad consiste en controlar el hostname asociado del ordenador, en este caso, del proceso o procesos asignados al namespace. Existen tres diferentes rutinas que nos permiten obtener y modificar el hostname:

- *sethostname()*
- *setdomainname()*
- *uname()*

En una situación normal sin namespaces, se modificaría una String global, sin embargo, si estamos dentro de un namespace, los procesos asociados tienen su propia variable global asignada.

Un ejemplo muy básico de uso de este namespace podría ser el siguiente [2]:

Ejemplo 9: Ejemplo de uso de UTS namespace

```
1 $ sudo su      # super user
2 $ hostname     # current hostname
3 > arch-linux
4 $ unshare -u /bin/sh  # shell with UTS namespace
5 $ hostname new-hostname # set hostname
6 $ hostname     # check hostname of the shell
7 > new-hostname
8 $ exit        # exit shell and namespace
9 $ hostname    # original hostname
10 > arch-linux
11
```

En el ejemplo planteado, vemos que utilizamos el comando `unshare`. Utilizando la documentación de dicho comando, `man unshare`. Podemos deducir lo siguiente:

- Ejecuta un programa con algunos namespaces diferentes del host.
- En los parámetros podemos especificar cual o cuales namespaces queremos desvincular.
- Tenemos que especificar la ruta del ejecutable que queremos aislar
- La sintaxis sería tal que: `unshare [options] <program> [<argument>...]`

4.3.2 Mount namespace

Este namespace fue el primero en aparecer en el Kernel de Linux, apareció en la versión 2.4.19, en 2002. El objetivo era restringir la visualización de la jerarquía global de archivos, dando lugar a que cada namespace tenga su propio “set” de puntos de montaje (directorio, archivo o enlace simbólico, que al crearse forma parte del sistema de archivos del sistema). [45][46]

Una vez iniciamos el sistema, solo existe un único mount namespace, al que llamamos “namespace inicial” (*default*). Los nuevos mount namespaces que creamos los haremos utilizando las siguientes llamadas al sistema: `clone()`, para crear un nuevo proceso asociado en el nuevo namespace; o bien utilizando `unshare()`, para mover un proceso dentro del nuevo namespace. Cuando creamos un nuevo mount namespace, este recibe una copia del punto de montaje inicial, replicado por el namespace que lo llama. [46]

Un concepto importante es que los mount namespaces, por defecto tienen activado la funcionalidad del kernel llamada `shared subtree`. Esto permite que cada punto de montaje que tengamos en nuestra máquina pueda tener su propio tipo de propagación asociado. Esta información permite que nuevos puntos de montaje en ciertas rutas se propagen a otros puntos de montaje. A modo de ejemplo, si conectamos un USB a nuestro sistema, y este se monta automáticamente en la ruta `/MI_USB`, el contenido solo estará visible en otros namespaces si la propagación está configurada correctamente. [7]

Por lo tanto, un namespace de tipo *mount* nos permite modificar un sistema de archivos en concreto, sin que el host, o en otros namespaces, pueda ver y/o acceder a dicho sistema de archivos. Podríamos concluir que el objetivo de este espacio de nombres es el de permitir que diferentes procesos puedan tener “vistas” diferentes de los puntos del montaje de nuestro sistema.

Un ejemplo básico de esta funcionalidad podría ser la siguiente:

Ejemplo 10: Uso de mount namespace con “bind”

```
1 $ sudo su      # run a shell in a new mount namespace
2 $ unshare -m /bin/sh
3 $ mount --bind /usr/bin/ /mnt/
4 $ ls /mnt/cp
5 > /mnt/cp
6 $ exit        # exit the shell and close namespace
7 $ ls /mnt/cp
8 > ls: cannot access '/mnt/cp': No such file or directory
9
```

Como vemos en el ejemplo, dentro del namespaces lo que hacemos es crear un *mount* de tipo *bind*, que tiene por función que un archivo de la máquina host se monte en un directorio en específico, en este caso, un directorio únicamente del programa que hemos asignado al namespace. Otro ejemplo de uso de estos namespaces es crear un sistema de archivos temporal que solo sea visible para ese proceso.

tmpfs

De cara al siguiente ejemplo, es interesante que repasemos la funcionalidad del kernel del linux llamada tmpfs. Dicha funcionalidad permite crear un sistema de archivos temporal dentro de nuestro sistema. El sistema de archivo creado reside completamente en la memoria y/o “swap” de nuestro sistema. Este tipo de montajes suele ser muy interesante cuando necesitamos agilizar el acceso a ciertos archivos, ya que al estar en la memoria RAM, la lectura es mucho más rápida que si lo comparamos con un disco duro e incluso un disco duro de estado solido. Sin embargo, el inconveniente que tiene es que el contenido de dicho sistema de archivos se elimina una vez reiniciemos el sistema, ya que este está almacenado en la memoria RAM de nuestro dispositivo. [47]

Este tipo de archivos es comunmente utilizado en los siguientes directorios: /tmp, /var/lock, /var/run, entre otros muchos. Sin embargo, tiene una utilidad muy importante para nuestra propuesta de sistema de archivos dentro de un mount namespace, ya que además de poder comprobar el aislamiento entre los montajes del mount namespace *default*, nos permite crear un montaje que puede ser heredado entre los diferentes *namespaces* que creemos, aprovechando la funcionalidad de *shared tree*.

Un ejemplo de esto podría ser el siguiente: [48]

Ejemplo 11: Uso de mount namespaces con “tmpfs”

```
1 # Creamos un directorio para nuestro sistema de archivos
2 $ mkdir /tmp/mount_ns
3
4 # Creamos el mount namespaces usando unshare
5 $ unshare -m /bin/bash
6
7 # Utilizamos tmpfs para crear un punto de montaje dentro del namespaces
8 $ mount -n -t tmpfs tmpfs /tmp/mount_ns
9
10 # Comprobamos que el montaje se ha creado correctamente
11 $ df -h | grep mount_ns
12 > tmpfs    7.8G  0 7.8G  0%  /tmp/mount_ns
13 $ cat /proc/mounts | grep mount_ns
14 > tmpfs    /tmp/mount_ns tmpfs rw,relatime    0 0
15
16 # En una terminal aparte (fuera del namespaces creado)
17 $ cat /proc/mounts | grep mount_ns
18 >
19 $ df -h | grep mount_ns
20 >
21 # Comprobamos que en el default namespace no tenemos acceso
22 # al montaje tmpfs que hemos creado
23
```

4.3.3 Process ID namespace

Para entender en que consiste este namespace, primero tenemos que conocer la definición de *process id* dentro del Kernel. En este caso, *process id* hace referencia a un número entero que utiliza el Kernel para identificar los procesos de manera unívoca. [8]

Concretando, aísla el namespace de la ID del proceso asignado, dando lugar a que, por ejemplo, otros namespaces puedan tener el mismo PID. Esto nos lleva a la situación de que un proceso dentro de un *PID namespace* piense que tiene asignado el ID "1", mientras que en la realidad (en la máquina host) tiene otro ID asignado.

Ejemplo 12: Uso de process id namespace

```
1 $ echo $$      # PID de la shell
2 $ ls -l /proc/$$/ns # ID espacios de nombres
3 $ sudo unshare -f --mount-proc -p /bin/sh
4 $ echo $$      # PID de la shell dentro del ns
5 $ ls -l /proc/$$/ns # nuevos ID espacio de nombres
6 $ ps
7
8 $ ps -ef      # ejecutar en una shell fuera del ns. Comparar PID
9 $ exit
10
```

Si ejecutamos el ejemplo, lo que podemos comprobar es que el ID del proceso que está dentro del namespaces (`echo $$`), no coincide con el proceso que podemos ver de la máquina host (`ps -ef | grep /bin/sh`). Más concretamente, el primer proceso creado en un PID namespace recibirá el pid número 1, y además de un tratamiento especial ya que supone un `init process` dentro de ese namespace [2].

4.3.4 Network namespace (netns)

Este namespace nos permite aislar la parte red de una aplicación o proceso que nosotros elijamos. Con esto conseguimos que el *stack* de red de la máquina host sea diferente al que tenemos en nuestro namespace. Debido a esto, el namespace crea una interfaz virtual, conjunto con el resto de necesidades para conformar un stack de red completo (tabla de enrutamiento, tabla ARP, etc...).

Para crear un *namespace* de tipo *network*, y que este sea persistente, utilizamos la *tool* *ip* (del *package* *iproute2*).

Ejemplo 13: Creation persistent network namespace

```
1 $ ip netns add ns1
2
```

Este comando creará un network namespace llamado *ns1*. Cuando se crea dicho namespace, el comando *ip* realiza un montaje tipo *bind* en la ruta */var/run/netns*, permitiendo que el namespace sea persistente aún sin tener un proceso asociado.

Ejemplo 14: Comprobar network namespaces existentes

```
1 $ ls /var/run/netns
2 or
3 $ ip netns
4
```

Como ejemplo, podemos proceder a añadir una interfaz de *loopback* al namespace que previamente hemos creado:

Ejemplo 15: Asignar interfaz loopback a un namespace

```
1 $ ip netns exec ns1 ip link dev lo up
2 $ ip netns exec ns1 ping 127.0.0.1
3 > PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
4 > 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.115 ms
5
```

La primera línea de este ejemplo, corresponde con la directiva que le dice al namespace que "leve" la interfaz de *loopback*. La segunda línea, vemos como el namespace *ns1* ejecuta el *ping* a la interfaz de *loopback* (el *loopback* de ese namespace).

Es importante mencionar, que aunque existen más comandos para gestionar las redes dentro de linux (como pueden ser *ifconfig*, *route*, etc), el comando *ip* es el considerado sucesor de todos estos, y los anteriores mencionados, dejarán de formar parte de Linux en versiones posteriores. Un detalle a tener en cuenta con el comando *ip*, es que es necesario tener privilegios de administrador para poder usarlo, por lo que deberemos ser *root* o utilizar *sudo*.

Por lo tanto, utilizando el comando *ip*, podemos recapitular que si utilizamos la siguiente directiva, podemos ejecutar el comando que nosotros indiquemos, pero dentro del network namespace que previamente hemos creado.

Ejemplo 16: Ejecutar cualquier programa con un network namespace

```
1 $ ip netns exec <network-namespace> <command>
2
```

Ejemplo práctico de **network namespaces** con NAT

Una de las problemáticas que supone el uso de los network namespaces, es que solo podemos asignar **una interfaz real a un namespace**. Suponiendo el caso en el que el usuario root tenga asignada la interfaz eth0 (identificador de una interfaz de red física), significaría que solo los programas en el namespace de root podrán acceder a dicha interfaz. En el caso de que eth0 sea la salida a Internet de nuestro sistema, pues eso conllevaría que no podríamos tener conexión a Internet en nuestros namespaces. La solución para esto reside en los **veth-pair**.

Un veth-pair funciona como si fuera un cable físico, es decir, interconecta dos dispositivos, en este caso, interfaces virtuales. Consiste en dos interfaces virtuales, una de ellas asignada al root namespace, y la otra asignada a otro network namespace diferente. Si a esta arquitectura le añadimos una configuración de IP válida y activamos la opción de hacer NAT en el eth0 del host, podemos dar conectividad de Internet al network namespace que hayamos conectado.

Ejemplo 17: Ejemplo configuración de NAT entre eth0 y veth

```
1 # Remove namespace if exists
2 $ ip netns del ns1 &>/dev/null
3
4 # Create namespace
5 $ ip netns add ns1
6
7 # Create veth link
8 $ ip link add v-eth1 type veth peer name v-peer1
9
10 # Add peer-1 to namespace.
11 $ ip link set v-peer1 netns ns1
12
13 # Setup IP address of v-eth1
14 $ ip addr add 10.200.1.1/24 dev v-eth1
15 $ ip link set v-eth1 up
16
17 # Setup IP address of v-peer1
18 $ ip netns exec ns1 ip addr add 10.200.1.2/24 dev v-peer1
19 $ ip netns exec ns1 ip link set v-peer1 up
20 # Enabling loopback inside ns1
21 $ ip netns exec ns1 ip link set lo up
22
23 # All traffic leaving ns1 go through v-eth1
24 $ ip netns exec ns1 ip route add default via 10.200.1.1
25
```

Si siguiendo el ejemplo propuesto, llegamos hasta el punto en el que el tráfico saliente del namespace ns1, será redirigido a v-eth1. Sin embargo, esto no es suficiente para tener conexión a Internet. Tenemos que configurar el NAT en el eth0.

Ejemplo 18: Configuración de NAT para dar Internet a un network namespace

```
1 # Share internet access between host and NS
2
3 # Enable IP-forwarding
4 $ echo 1 > /proc/sys/net/ipv4/ip_forward
5
6 # Flush forward rules, policy DROP by default
7 $ iptables -P FORWARD DROP
8 $ iptables -F FORWARD
9
10 # Flush nat rules.
11 $ iptables -t nat -F
12
13 # Enable masquerading of 10.200.1.0 (ip of namespaces)
14 $ iptables -t nat -A POSTROUTING -s 10.200.1.0/255.255.255.0 -o eth0
15   -j MASQUERADE
16
17 # Allow forwarding between eth0 and v-eth1
18 $ iptables -A FORWARD -i eth0 -o v-eth1 -j ACCEPT
19 $ iptables -A FORWARD -o eth0 -i v-eth1 -j ACCEPT
20
```

Si todo lo hemos configurado correctamente, ahora podríamos realizar un ping hacia Internet, y este nos debería resultar satisfactorio.

```
$ ip netns exec ns1 ping google.es
> PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
> 64 bytes from 8.8.8.8: icmp_seq=1 ttl=50 time=48.5ms
> 64 bytes from 8.8.8.8: icmp_seq=2 ttl=50 time=58.5ms
```

Aún así, no resulta muy cómodo el utilizar `ip netns exec` seguido de la aplicación a utilizar. Es por esto por lo que es común ejecutar dicho comando para asignar el network namespace a una shell. Esto sería tal que así:

```
$ ip netns exec ns1 /bin/bash
```

Utilizaremos `exit` para salir de la shell y abandonar el network namespace.

4.3.5 User ID (user)

Cada sistema dispone de una manera de monitorizar que usuario es el dueño de cada archivo. Esto permite al sistema restringir el acceso a aquellos archivos que consideramos sensibles. Además, bloquea el acceso entre diferentes usuarios dentro del mismo sistema. Para el usuario, este identificador de usuarios se muestra como el usuario que en ese momento está conectado, sin embargo, para nuestro sistema, el identificador de usuario esta compuesto por una combinación arbitraria de caracteres alfanuméricos. Con el fin de mantener el monitoreo correctamente, hay un proceso encargado de transformar esos caracteres a un número específico de identificación (UID), como por ejemplo sería 1000. Es este valor el que se asocia con los archivos creados por este usuario. Esto nos aporta la ventaja de que, si un usuario cambia su nombre, no es necesario reconstruir el sistema de archivos, ya que su UID sigue siendo 1000.

Si por ejemplo queremos ver el UID del usuario que estamos usando en este momento, podemos ejecutar: `echo $UID`, el cual nos devolverá el número asociado a nuestro usuario, en mi caso es el 1000.

Además de diferenciar entre los IDs de usuarios (UID), también se nos permite separar entre IDs de grupos (GID). En linux, un grupo sirve para agrupar usuarios de modo que un grupo puede tener asociado un privilegio que le permite usar un recurso o programas.

Por lo tanto, el namespace de UID, lo que nos permite es tener un UID y GID diferente al del host.

Ejemplo 19: Ejemplo de uso UID namespace

```
1 $ ls -l /proc/$$/ns # espacios de nombres originales
2 $ id
3 > uid=1000(user) gid=1000(user) groups=1000(user), ...
4 $ unshare -r -u bash # Crea un namespace de tipo usuario, programa bash
5 $ id
6 > uid=0(root) gid=0(root) groups=0(root),65534(nobody)
7 $ cat /proc/$$/uid_map
8 >          0          1000          1
9 $ cat /etc/shadow # No nos deja acceder
10 > cat: /etc/shadow: Permission denied
11 $ exit
12
```

Como vemos en el ejemplo, el UID de usuario difiere de la máquina host. Dentro del namespace, tenemos UID 0, sin embargo, eso no significa que podamos acceder a los archivos con UID 0 de la máquina host, ya que en verdad lo que hace el namespace es *mapear* el UID 1000 al 0. [2]

4.3.6 Interprocess Communication namespace (IPC)

Este namespace supone uno de los más técnicos, complicados de entender y explicar. IPC (Inter-process communication) controla la comunicación entre procesos, utilizando zonas de la memoria que están compartidas, colas de mensajes, y semáforos. La aplicación más común para este tipo de gestión es el uso en bases de datos.

4.3.7 Control group (cgroup)

Los grupos de control, o `cgroups`, de Linux suponen un mecanismo para controlar los diferentes recursos de nuestro sistema. Cuando un `cgroup` está activo, puede controlar la cantidad de CPU, RAM, acceso I/O, o cualquier faceta que un proceso puede consumir. Además, permiten definir jerarquías en las que se agrupan, de manera en la que el administrador del sistema puede definir como se asignan los recursos o llevar la contabilidad de los mismos. `Cgroups` permite las siguientes funcionalidades [35]:

- **Limitar recursos.** Podemos configurar un grupo para limitar un recurso (o varios de ellos) para cada proceso que asignemos.
- **Priorizar tareas.** Podemos controlar cuantos recursos utiliza un proceso, comparándolo con otro proceso en un grupo diferente.
- **Monitorización.** Los límites establecidos para los recursos son monitorizados y son reportados al usuario.
- **Control.** Podemos controlar el estado de los procesos asociados a un grupo con un solo comando, pudiendo elegir entre “congelado”, “parado” o “reiniciado”.

La primera versión de `cgroups` aparece en el Kernel en 2007, siendo esta la versión más estandarizada por la mayoría de distribuciones. Sin embargo, en 2016 aparece `cgroups v2` en el Kernel, aportando mejoras en la simplificación de los arboles de jerarquías de la ruta `/sys/fs/cgroup`, además de nuevas interfaces, aportando las bases para contenedores que utilizan el concepto de “*rootless*”.

En el caso de la versión `v1`, los `cgroups` se crean en el sistema de archivos virtual en la ruta `/sys/fs/cgroup`. Para crear un nuevo grupo, en nuestro caso con el objetivo de limitar un proceso en memoria, tendríamos que ejecutar lo siguiente:

```
mkdir /sys/fs/cgroup/memory/<NombreGrupo>
```

De esta manera, ya tendríamos un nuevo grupo creado, asociado al `cgroup` de `memory`. Si ejecutamos el comando `ls` en el directorio que acabamos de crear, podemos comprobar como se han generado una serie de recursos:

```
root@rani-arch /home/rani # cd /sys/fs/cgroup/memory/test_cg
root@rani-arch /sys/fs/cgroup/memory/test_cg # ls
cgroup.procs          memory.kmem.tcp.failcnt  memory.kmem.max_usage_in_bytes  memory.max_usage_in_bytes  memory.oom_control      tasks
cgroup.clone_children memory.memsw.failcnt     memory.kmem.tcp.limit_in_bytes  memory.memsw.limit_in_bytes memory.pressure_level
cgroup.event_control  memory.stat              memory.kmem.tcp.max_usage_in_bytes memory.memsw.max_usage_in_bytes memory.soft_limit_in_bytes
memory.failcnt         memory.swappiness         memory.kmem.tcp.usage_in_bytes  memory.memsw.usage_in_bytes  memory.usage_in_bytes
memory.kmem.failcnt    memory.force_empty       memory.kmem.usage_in_bytes      memory.move_charge_at_immigrate memory.use_hierarchy
memory.kmem.slabinfo   memory.kmem.limit_in_bytes memory.limit_in_bytes           memory.numa_stat             notify_on_release
```

Figura 14: Archivos asociados al grupo `test_cg` una vez lo creamos

Si quisiéramos establecer un límite en el uso de memoria, tendríamos que escribir en el archivo `memory.limit_in_bytes`, a modo de ejemplo, establecemos un límite de 50MB. Esto se haría tal que:

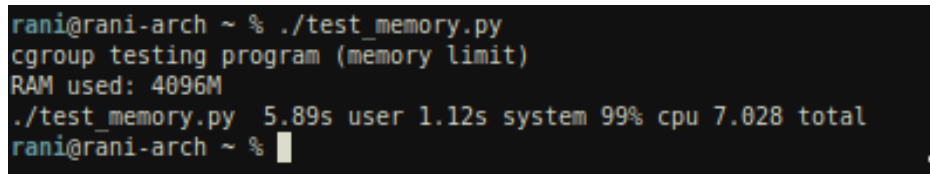
```
echo 50000000 > /sys/fs/cgroup/memory/<NombreGrupo>/memory.limit_in_bytes
```

Para comprobar el funcionamiento de este límite de memoria, vamos a asociarle un programa escrito en Python que consume mucha memoria RAM de golpe. El programa en cuestión vendría dado por el siguiente código:

Ejemplo 20: Programa en Python que consume 4 GB de RAM

```
1 #!/usr/bin/python
2 import numpy
3
4 print("cgroup testing program (memory limit)")
5 result = [numpy.random.bytes(1024*1024) for x in range(1024*4)]
6 print("RAM used: {}M".format(len(result)))
7
```

Ejecutamos el programa, y comprobamos como la salida por consola es tal que:



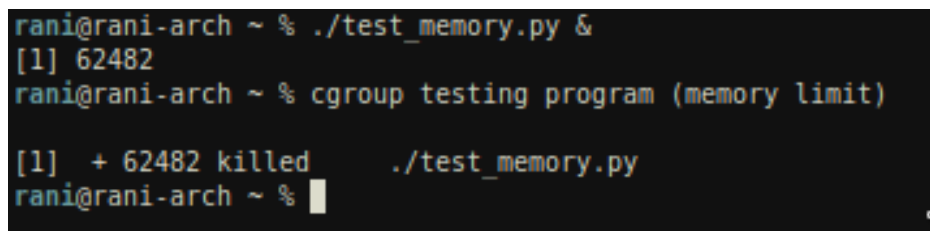
```
rani@rani-arch ~ % ./test_memory.py
cgroup testing program (memory limit)
RAM used: 4096M
./test_memory.py 5.89s user 1.12s system 99% cpu 7.028 total
rani@rani-arch ~ %
```

Figura 15: Salida tras ejecutar el programa de Python sin limitar

Sin embargo, ahora vamos a proceder a limitar ese mismo programa en memoria. Para ello, vamos a añadir el PID asociado a la ejecución de dicho programa al siguiente archivo (ejecutando el script `./test_memory.py &`, nos aparece el PID):

```
echo 62482 > /sys/fs/cgroup/memory/test_cg/cgroup.procs
```

De esta manera ya estaríamos limitando la memoria de ese programa en concreto. Al limitarlo, podemos comprobar como la salida en consola es tal que:



```
rani@rani-arch ~ % ./test_memory.py &
[1] 62482
rani@rani-arch ~ % cgroup testing program (memory limit)

[1] + 62482 killed ./test_memory.py
rani@rani-arch ~ %
```

Figura 16: Salida tras ejecutar el programa de Python, una vez limitado

Como podemos comprobar en las imágenes [15] y [16], la salida del programa no coincide. Esto es debido a que como el programa Python ha superado el límite establecido para su grupo, `cgroups` cerró bruscamente dicho programa, por lo tanto no nos aparece la memoria consumida por el programa, solo se nos notifica que el proceso con PID 62482 ha pasado a estado “*killed*”.

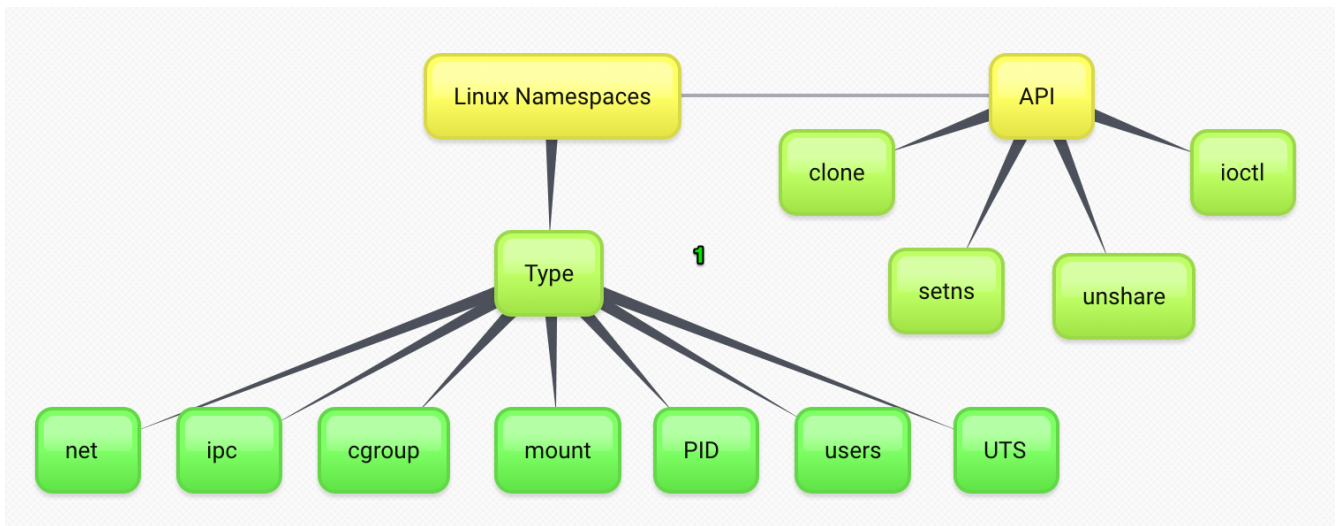


Figura 17: Diferentes namespaces en Linux y su API de acceso. (5)

4.3.8 Time

Por último, nos queda el namespaces asociado al tiempo. Este namespace fue propuesto para que se incorporara al kernel de Linux en 2018 y en enero de 2020 fue añadido a la versión mainline de Linux. Apareció en la release 5.6 del kernel de Linux.

El namespace time, permite que por cada namespace que tengamos, podamos crear desfases entre los relojes monotónicos (CLOCK_MONOTONIC) y de boot (CLOCK_BOOTTIME), de la máquina host. Esto permite que dentro de los contenedores se nos permita cambiar la fecha y la hora, sin tener que modificar la hora del sistema host. Además, supone una capa más de seguridad, ya que no estamos vinculando directamente la hora a los relojes físicos de nuestro sistema. [13]

Un namespace de tipo time, es muy similar al namespace de tipo PID en la manera de como lo creamos. Utilizamos el comando `unshare -T`, y mediante una `systemcall` se nos creará un nuevo time namespace, pero no lo asocia directamente con el proceso. Tenemos que utilizar `setns` para asociar un proceso a un namespace, además todos los procesos dependientes también tendrán asignado dicho namespace.

4.4 Ejemplo de uso de 'netns' usando comando **ip**

En este apartado, vamos a detallar un ejemplo de como funciona el comando **IP** manejando los 'network namespaces'.

Creamos los *network namespaces*, en este caso, con nombre h1 y h2. El sistema no crea directamente el namespace, lo que en realidad hace es definirlos en el sistema. El *network namespace* se crea cuando una aplicación se asocia a el.

```
$ ip netns add h1
$ ip netns add h2
```

Si utilizamos el comando `ip netns`, nos mostrará los netns existentes. Como es un sub-comando de `ip`, muestra los ns que el comando `lsns` no muestra.

Procedemos a asociar una aplicación a cada netns. Utilizamos `bash`.

```
$ ip netns exec h1 bash
$ ip netns exec h2 bash
```

Ahora, si que podemos utilizar el comando `lsns`. Comprobamos que si nos aparecen los ns que hemos creado, cosa que antes de asociar una aplicación al ns, no pasaba.

El comando `ip` crea automaticamente un *nsfs* para poder colocar los archivos de configuración del *netns*. Para ello, debe crearse el directorio `/etc/netns/h1` y poner en el los archivos de configuración de la red.

```
$ mkdir /etc/netns/h1
$ echo "nameserver 8.8.8.8" > /etc/netns/h1/resolv.conf
```

En este momento, tenemos el ns configurado con DNS. Nos quedaría realizar la conexión entre el ethernet físico de nuestro *host* y las interfaces de nuestros namespaces. Para ello, vamos a utilizar un conmutador virtual, en este caso Open vSwitch.

```
$ systemctl enable --now openvswitch.service
```

Creamos un *brige* utilizando el OpenvSwitch.

```
$ $ ovs-vsctl add-br s1
```

Utilizando el comando `ip`, creamos las interfaces virtuales de ethernet y las asignamos a sus namespaces.

```
$ ip link add h1-eth0 type veth peer name s1-eth1
$ ip link add h2-eth0 type veth peer name s1-eth2
$ ip link set h1-eth0 netns h1
$ ip link set h2-eth0 netns h2
```

Utilizando el comando `ovs-vsctl`, asignamos al *bridge* el otro par ethernet que hemos creado para cada namespace.

```
$ ovs-vsctl add-port s1 s1-eth1
$ ovs-vsctl add-port s1 s1-eth2
```

Verificamos que el controlador sea *standalone*, así el switch se comportará como un *learning-switch*.

```
$ ovs-vsctl set-fail-mode br0 standalone
```

Como la conexión es desde localhost al exterior, entendemos que es una conexión fuera de banda.

```
$ ovs-vsctl set controller br0 connection-mode=out-of-band
```

En este momento, tenemos todos configurado a falta de habilitar las diferentes interfaces de nuestra topología.

```
$ ip netns exec h1 ip link set h1-eth0 up
$ ip netns exec h1 ip link set lo up
$ ip netns exec h1 ip add add 10.0.0.1/24 dev h1-eth0
$ ip netns exec h2 ip link set h2-eth0 up
$ ip netns exec h2 ip link set lo up
$ ip netns exec h2 ip add add 10.0.0.2/24 dev h2-eth0
$ ip link set s1-eth1 up
$ ip link set s1-eth2 up
```

Ahora tenemos todas las interfaces configuradas, el switch activado y el sistema interconectado, por lo que podemos ejecutar un ping en una de las terminales de los namespaces para verificar la topología.

```
$ ip netns exec h1 ping -c4 10.0.0.2
```

Si queremos revertir todas las configuraciones que hemos hecho, lo que tenemos que hacer es ejecutar los siguientes comandos:

```
$ ovs-vsctl del-br s1
$ ip link delete s1-eth1
$ ip link delete s1-eth2
$ ip netns del h1
$ ip netns del h2
```

4.5 Ejemplo de uso de 'netns' usando comando **unshare**

En el ejemplo anterior, utilizabamos el comando `ip` para manejar los `netns`, sin embargo, eso nos limitaba los tipos namespaces que queriamos asignar a nuestro namespace. En contra partida a esto, el comando `unshare` nos da más libertad a la hora de crear los namespaces.

Utilizando `unshare`, no podemos ponerle un nombre, pero sí que nos permite asociarlo a un archivo, que montará con tipo `bind`. Esto nos permitirá utilizar en namespace aunque no haya ningún proceso corriendo en el, para ello podemos utilizar el comando `nsenter`.

```
$ touch /var/net-h1
$ touch /var/uts-h1
$ unshare --net=/var/net-h1 --uts=/var/uts-h1 /bin/bash
```

Utilizando el comando `nsenter` podemos ejecutar comandos dentro del namespace.

```
$ nsenter --net=/var/net-h1 --uts=/var/uts-h1 hostname h1
$ nsenter --net=/var/net-h1 --uts=/var/uts-h1 ip address
```

Para destruir el namespace, lo que tendremos que hacer es desmontar los archivos asignados a dicho namespace.

```
$ umount /var/net-h1
$ umount /var/uts-h1
```

Como será común necesitar más de un namespace, de ahora en adelante tendremos que utilizar los comandos `unshare` y `nsenter`.

5 Virtualización ligera y contenedores

En este capítulo vamos a trabajar el concepto de virtualización ligera, que ya introdujimos en el apartado 2.2, pero esta vez aplicando los conceptos de namespaces del apartado anterior. Además, presentaremos el concepto de *contenedor*, y como supone una de las piezas más importantes para la virtualización tal y como hoy día la conocemos.

Recuperando la definición de **virtualización ligera**, entendemos este tipo de virtualización como aquella que **se realiza a nivel de sistema operativo**, permitiendo la coexistencia de diferentes espacios aislados entre sí. En dichos espacios, podremos ejecutar de manera aislada nuestras aplicaciones. Como punto en común, todos los espacios aislados que creemos, **utilizarán** como base **el mismo kernel**. La tecnología clave para realizar esta virtualización serán los diferentes namespaces, comentados en el apartado 4, que podremos combinar a nuestro gusto con el fin de crear un espacio aislado con todos los recursos necesarios para satisfacer las necesidades de nuestra aplicación.

La principal ventaja que encontramos al utilizar namespaces, respecto a otro tipo de virtualizaciones como podrían ser las máquinas virtuales, es el aprovechamiento de los recursos de la máquina host. Al tener todos el mismo kernel, evitamos tener por duplicados los kernels para cada una de las instancias a realizar, ahorrando ciclos de CPU, como espacio en RAM.

Por otro lado, tenemos el concepto de **contenedor**. Entendemos contenedor, en el ámbito de la virtualización, como una abstracción a alto nivel de un sistema aislado creado utilizando namespaces y cgroups. Por lo tanto, si el kernel nos da la posibilidad de trabajar a bajo nivel utilizando dichos namespaces, en este caso, buscamos ir más allá y encapsular dicho espacio aislado en unas APIs de alto nivel, que sean mucho más fáciles de entender y de implementar. Tenemos muchos ejemplos de sistemas que trabajan con contenedores, y muchos de ellos implementados en una amplia variedad de lenguajes [4]. Algunos de los más importantes son:

- LXC, LinuX Containers, escrito en C: <https://linuxcontainers.org/>
- Docker, escrito en Go: <https://www.docker.com/>
- Podman, escrito en Go: <https://podman.io/>
- systemd-nspawn, escrito en C, implementado dentro de systemd: <https://github.com/systemd/systemd>
- Vagga, escrito en Rust: <https://github.com/tailhook/vagga>

Por último, es interesante comentar el concepto de chroot, ya que muchas de estas abstracciones de los namespaces hacen uso de esta técnica para su funcionamiento. Un chroot es una operación Unix que permite cambiar la ruta aparente de un directorio para un usuario en específico. Un proceso ejecutado después de realizar un chroot solo tendrá acceso al nuevo directorio definido y a sus subdirectorios, esta operación recibe el nombre de chroot jail, ya que los procesos no pueden leer ni escribir fuera del nuevo directorio. Este método suele ser de gran utilidad en virtualizaciones a nivel de kernel, es decir, virtualización ligera o contenedores. [38]

5.1 Creando nuestro propio “contenedor”

5.2 Contenedores LXC

En este apartado vamos a comentar la aplicación de contenedores LXC, LinuX Containers. Como bien hemos comentado en el apartado anterior, LXC consiste en una virtualización a nivel de sistema operativo (virtualización ligera), creada por el proyecto `linuxcontainers.org`, con el objetivo de poder ejecutar diferentes espacios aislados (contenedores) utilizando un único host, lo llamaremos LXC host. Aunque pueda parecer que estamos ante una técnica de virtualización basada en máquinas virtuales, se trata de espacios virtuales, en los que cada uno dispone de su propia CPU, memoria, redes, etc. Esto lo consigue gracias al uso de los namespaces y los cgroups en el host LXC. [39]. Algunas de las características más destacables son:

- Utiliza mount namespace para conseguir la estructura de directorios propia de una distribución de Linux.
- El proceso asignado a cada namespaces es el `init`, por lo tanto tendremos un proceso de arranque del sistema.
- Tiene sus propios usuarios, incluido un usuario `root`.
- Una vez dentro del contenedor, podemos instalar aplicaciones utilizando el gestor de paquetes de la distribución (`apt`, `zipper`, `pacman`, etc...)

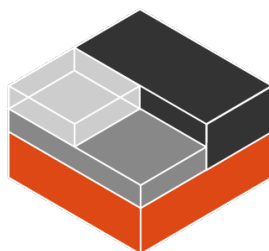


Figura 18: Logotipo del proyecto `linuxcontainers.org` [8]

Contenedores sin privilegios

Una característica muy importante de este tipo de contenedores es que permiten la posibilidad de configurar contenedores de dos tipos: contenedores con privilegios y contenedores sin privilegios. Esto es importante ya que si definimos un contenedor con privilegios, hay ciertas acciones que nos permitirían realizar comandos en el host. En el caso de que nuestro contenedor sea distribuido por terceros, puede suponer un problema grave de seguridad, ya que un atacante podría tomar el control de nuestro contenedor, y por consiguiente, también podría acceder a la información del host. Es por esto por lo que surge el concepto de contenedores sin privilegios (*unprivileged containers*), considerados como una técnica mucho más segura ya que disponen de un nivel añadido de aislamiento respecto al host. La clave reside en “mapear” el UID del usuario `root` de nuestro contenedor a un UID del host que no tenga permisos de administrador. Por lo tanto, si un atacante consigue acceder a nuestro contenedor, y pudiera acceder al host, se vería con que no tiene permisos para realizar ninguna acción. [39]

Tipos de configuración de red en el host

LXC soporta dos tipos de conexiones virtuales de red. Estas son las siguientes:

- NAT bridge. En este modo, LXC tiene su propio bridge (lxcbr0) que funciona en conjunto con las aplicaciones dnsmasq e iptables del host, dando lugar a que se puedan utilizar servicios red como DNS, DHCP y NAT dentro del propio contenedor.
- Host bridge. En este modo, es necesario que el host configure su propio bridge para dar servicio a las aplicaciones que creamos pertinentes. Esta opción nos permite mucha flexibilidad a la hora de interconectar nuestros contenedores para una funcionalidad específica.

Utilizar NAT bridge en un contenedor

A modo de ejemplo, si quisiéramos utilizar redes tipo NAT bridge en un contenedor [39], primero tendríamos que crear el archivo /etc/default/lxc-net con el siguiente contenido:

Ejemplo 21: Configuración interfaz NAT bridge en LXC

```
1 # Leave USE_LXC_BRIDGE as "true" if you want to use lxcbr0 for your
2 # containers. Set to "false" if you'll use virbr0 or another existing
3 # bridge, or mavlan to your host's NIC.
4 USE_LXC_BRIDGE="true"
5
6 # If you change the LXC_BRIDGE to something other than lxcbr0, then
7 # you will also need to update your /etc/lxc/default.conf as well as the
8 # configuration (/var/lib/lxc/<container>/config) for any containers
9 # already created using the default config to reflect the new bridge
10 # name.
11 # If you have the dnsmasq daemon installed, you'll also have to update
12 # /etc/dnsmasq.d/lxc and restart the system wide dnsmasq daemon.
13 LXC_BRIDGE="lxcbr0"
14 LXC_ADDR="10.0.3.1"
15 LXC_NETMASK="255.255.255.0"
16 LXC_NETWORK="10.0.3.0/24"
17 LXC_DHCP_RANGE="10.0.3.2,10.0.3.254"
18 LXC_DHCP_MAX="253"
19 # Uncomment the next line if you'd like to use a conf-file for the lxcbr0
20 # dnsmasq. For instance, you can use 'dhcp-host=maill1,10.0.3.100' to have
21 # container 'maill1' always get ip address 10.0.3.100.
22 #LXC_DHCP_CONFILE=/etc/lxc/dnsmasq.conf
23
24 # Uncomment the next line if you want lxcbr0's dnsmasq to resolve the .lxc
25 # domain. You can then add "server=/lxc/10.0.3.1' (or your actual $LXC_ADDR)
26 # to your system dnsmasq configuration file (normally /etc/dnsmasq.conf,
27 # or /etc/NetworkManager/dnsmasq.d/lxc.conf on systems that use NetworkManager).
28 # Once these changes are made, restart the lxc-net and network-manager services.
29 # 'container1.lxc' will then resolve on your host.
30 #LXC_DOMAIN="lxc"
31
```

Ahora, necesitamos modificar la *template* del contenedor LXC para que utilice la interfaz que hemos configurado. Modificamos la plantilla con ruta `/etc/lxc/default.conf` tal que:

Ejemplo 22: Configuración contenedor LXC para usar NAT bridge

```
1 lxc.net.0.type = veth
2 lxc.net.0.link = lxcbr0
3 lxc.net.0.flags = up
4 lxc.net.0.hwaddr = 00:16:3e:xx:xx:xx
5
```

Para que todos estos cambios se realicen, es necesario que tengamos activado el servicio `lxc-net.service`.

```
> sudo systemctl enable lxc-net.service
> sudo systemctl start lxc-net.service
```

Crear un contenedor con privilegios utilizando LXC

Si decidimos crear un contenedor con privilegios, tendríamos que seguir los siguientes pasos [40]:

Ejemplo 23: Crear un contenedor con privilegios en LXC

```
1 sudo lxc-create --template download --name <NombreContenedor>
2
```

Con este comando, LXC nos preguntará de manera interactiva por un `root filesystem` para el contenedor a descargar, además de la distribución elegida, la versión o la arquitectura. Si queremos hacerlo de manera que no sea interactivo, podemos crear el contenedor tal que:

Ejemplo 24: Crear un contenedor con privilegios en LXC, modo no interactivo

```
1 sudo lxc-create --template download --name <NombreContenedor> -- --dist debian --
  release stretch --arch amd64
2
```

De esta manera, ya tendríamos creado nuestro contenedor. Para poder manejar dichos contenedores, es conveniente conocer los comandos disponibles por LXC. Algunos de los más importantes son los siguientes [40]:

- `sudo lxc-ls --fancy`. Lista los contenedores disponibles por el host.
- `sudo lxc-info --name <NombreContenedor>`. Permite conocer la información de un contenedor específico.
- `sudo lxc-attach --name <NombreContenedor>`. Nos conecta directamente con la shell de nuestro contenedor.
- `sudo lxc-start --name <NombreContenedor>--daemon`. Permite arrancar el contenedor.
- `sudo lxc-stop --name <NombreContenedor>`. Permite parar un contenedor que esté en ejecución.
- `sudo lxc-destroy --name <NombreContenedor>`. Elimina un contenedor, incluido su directorio `root`.

Una vez creamos el contenedor, es interesante modificar su configuración para asignarle una interfaz de red. Esto lo haremos modificando el archivo config con ruta:

```
/var/lib/lxc/<NombreContenedor>/config
```

Añadiremos la siguiente configuración para que utilice el NAT bridge que definimos con anterioridad, además de permitir la ejecución de aplicaciones tipo X11, utilizando xorg (aplicaciones con interfaz gráfica) [41].

Ejemplo 25: Configuración interfaz NAT bridge y aplicaciones X a un contenedor LXC

```
1 # Network configuration (bridge)
2 lxc.net.0.type = veth
3 lxc.net.0.veth.pair = veth_containerLXC
4 lxc.net.0.flags = up
5 lxc.net.0.ipv4.address = 172.16.27.2/24
6
7 ## for xorg
8 lxc.mount.entry = /dev/dri dev/dri none bind,optional,create=dir
9 lxc.mount.entry = /dev/snd dev/snd none bind,optional,create=dir
10 lxc.mount.entry = /tmp/.X11-unix tmp/.X11-unix none bind,optional,create=dir,ro
11 lxc.mount.entry = /dev/video0 dev/video0 none bind,optional,create=file
12
```

Ahora ya tenemos listo el contenedor para arrancarlo, solo nos quedaría asignar una IP válida al host, que también está dentro del bridge de LXC. Para ello, ejecutaremos:

Ejemplo 26: Asignar IP al host LXC y arrancar un contenedor

```
1 ip link add 172.16.27.1/24 dev veth_containerLXC
2 sudo lxc-start --name <NombreContenedor>
3 sudo lxc-attach --name <NombreContenedor>
4
5 # Para ejecutar interfaz grafica
6 startx
7
```

Crear un contenedor sin privilegios utilizando LXC

En el caso de optar por la opción más segura, que es la de crear un contenedor sin privilegios en el host. Tendremos que realizar una serie de configuraciones previas [42]. La primera consistirá en crear un usuario sin privilegios para LXC:

```
sudo useradd -s /bin/bash -c 'unprivileged lxc user' -m lxc_user
sudo passwd lxc_user
```

Ahora, necesitamos buscar los valores de grupo (subgid) e identificación (subuid) del usuario creado, para ello ejecutamos lo siguiente:

```
sudo grep lxc_user /etc/sub{gid,uid}
```

Obteniendo por consola una salida similar a la siguiente:

```
/etc/subgid:lxc_user:100000:65536
/etc/subuid:lxc_user:100000:65536
```

Utilizando las mismas configuraciones de red que en el apartado de contenedor con privilegios, procedemos a acceder al usuario asignado a LXC. Ejecutaremos el comando `id` para conocer los diferentes `uid` y `gid` asignados.

```
su lxc_user
id
```

Tendremos una salida similar a:

```
uid=1002(lxc_user) gid=1002(lxc_user) groups=1002(lxc_user)
```

El siguiente paso sería crear los directorios de configuración de LXC, y copiar la configuración *default* a dichos directorios.

```
mkdir -p /home/lxc_user/.config/lxc
cp /etc/lxc/default.conf /home/lxc_user/.config/lxc/default.conf
```

Por último, tendríamos que modificar dicho archivo para realizar un “mapeo de permisos”. Con el fin de asignar la ejecución del contenedor al usuario LXC que acabamos de crear. Al final del archivo `default.conf` que acabamos de copiar, añadimos lo siguiente:

Ejemplo 27: Configuración para mapear UID y GID para un contenedor sin privilegios en LXC

```
1 lxc.id_map = u 0 100000 65536
2 lxc.id_map = g 0 100000 65536
3
```

Una vez realizados todos estos pasos, ya podríamos crear nuestro contenedor sin privilegios. Lo haríamos de manera similar que en el apartado anterior, es decir utilizando el comando `lxc-create`. Además, también será importante modificar el archivo `config` de nuestro contenedor para asignar la interfaz de red y permitir la ejecución de aplicaciones con interfaz gráfica.

5.3 Contenedores Docker

En este apartado vamos a profundizar en la herramienta Docker. Al igual que en el caso de LXC, Docker permite la creación y la gestiones de aplicaciones aisladas entre sí, utilizando virtualización ligera; es decir, nos permite gestionar facilmente un entorno de contenedores.

Tal y como lo definen sus creadores [43], Docker es una plataforma de desarrollo, despliegue y ejecución de aplicaciones. Docker permite separar entre aplicaciones de tu infraestructura, permitiendo desplegar el software mucho más rápido. Uno de los objetivos que persigue esta plataforma es el de minimizar el tiempo entre programar y tener ejecutando la aplicación en producción. Algunas de las ventajas que nos aporta esta plataforma son:

- Ejecutar aplicaciones de manera aislada, uso de contenedores.
- Desplegar un número elevado de contenedores en un mismo host.
- Contenedores ligeros y que además, contienen todo lo necesario para ejecutar la aplicación. Independientemente del host.
- Facilidad a la hora de compartir contenedores con otros compañeros.
- Equivalencia entre despliegue local y despliegue en producción.

Arquitectura de Docker

Docker utiliza una arquitectura de cliente-servidor. El cliente de Docker se comunica con un “demonio” de la máquina host (servicio en ejecución en segundo plano), que tiene como funciones: crear, ejecutar y distribuir los contenedores Docker. Gracias a esta arquitectura, el cliente Docker puede estar en la misma máquina, o bien se puede conectar a un “demonio” remoto, esto se puede realizar utilizando una REST API (conectar a máquina remota) o bien UNIX sockets (conectar a máquina local)[43].

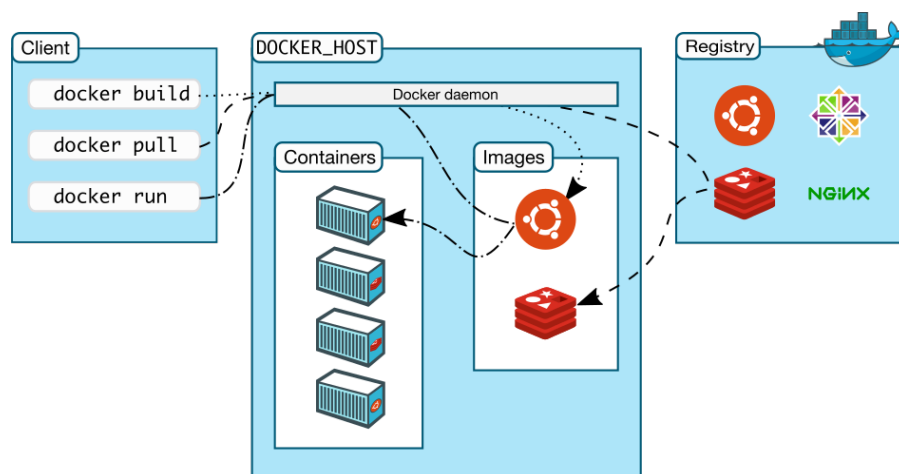


Figura 19: Arquitectura Docker [9]

Tecnologías utilizadas por Docker

Docker está desarrollado en el lenguaje de programación Go (<https://golang.org/>), y utiliza las ventajas aportadas por el kernel de Linux para desarrollar su funcionalidad. Estas tecnologías son, los namespaces, explicados en el apartado 4, que nos permiten aislar los diferentes espacios de trabajo de nuestras aplicaciones; especialmente utilizarán cgroups para aplicar reglas a la hora de ejecutar cada contenedor. [43]



Figura 20: Logotipo de Docker [10]

Primeros pasos en Docker

Lo primero que tendremos que hacer para utilizar Docker, es proceder a instalar dicha herramienta. Para ello, tendríamos que seguir los pasos que se nos especifican en <https://docs.docker.com/get-docker/>.

Una vez instalado, podemos comprobar que el “demonio” está funcionando si ejecutamos los siguientes comandos desde el cliente de docker.

```
sudo docker version
```

El comando nos devolverá información sobre el cliente de docker e intentará conectar con el “demonio”. En el caso de que nos responda como en la imagen (21), tendremos que ejecutar el comando:

```
sudo systemctl start docker.service
```

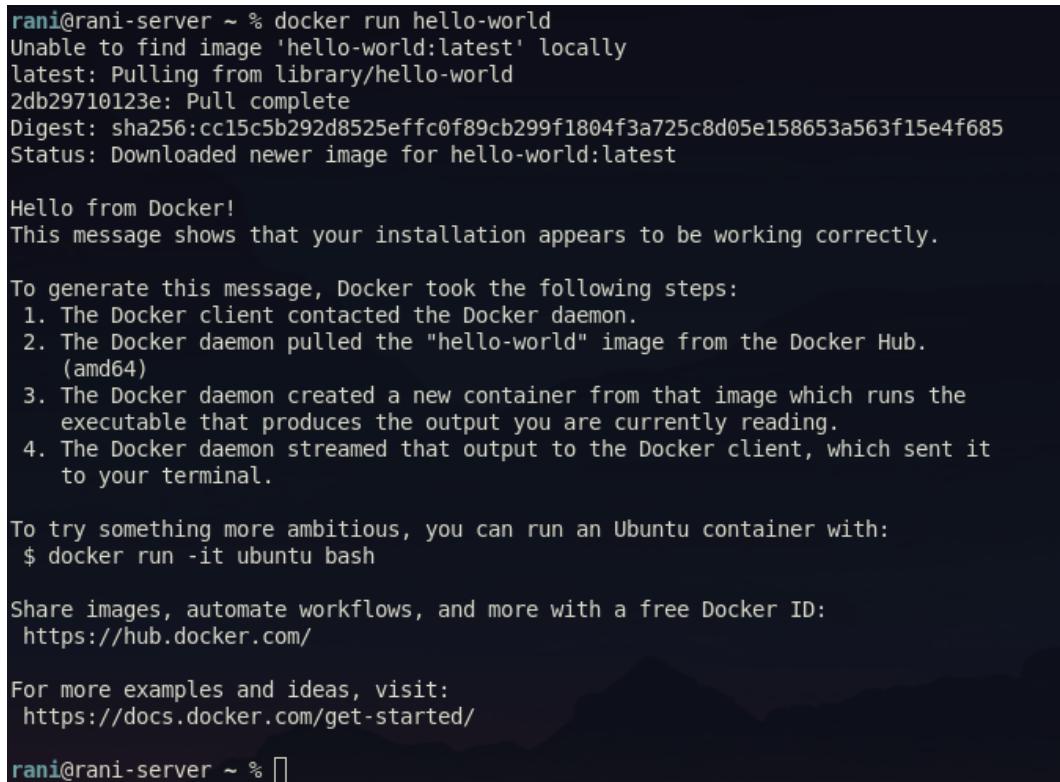
```
rani@raniita ~ % docker version
Client:
 Version:           20.10.10
 API version:       1.41
 Go version:        go1.17.2
 Git commit:        b485636f4b
 Built:             Tue Oct 26 03:44:01 2021
 OS/Arch:           linux/amd64
 Context:           default
 Experimental:      true
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?
1 rani@raniita ~ %
```

Figura 21: Ejecución comando `docker version` sin “demonio” en funcionamiento

Una vez comprobado que tenemos conexión con el “daemon”. Lo siguiente, será ejecutar nuestro primer contenedor. En este caso, haremos uso de un contenedor “especial” que nos verifica la instalación de docker en nuestros host. Para ello, ejecutamos el siguiente comando:

```
docker run hello-world
```

Una vez termina de ejecutarse el comando, la salida por consola que obtendremos sería equivalente a la siguiente figura:



```
rani@rani-server ~ % docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:cc15c5b292d8525effc0f89cb299f1804f3a725c8d05e158653a563f15e4f685
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/

rani@rani-server ~ %
```

Figura 22: Ejecución comando `docker run hello-world` para comprobar instalación

Como podemos ver en la figura 22, el contenedor nos muestra por consola los pasos que ha seguido para completar su ejecución. Antes de comentar los pasos seguidos, es importante aclarar el concepto de “imagen”.

Llamamos imagen de Docker a un archivo de sistema, compuesto por diferentes capas de información, que es utilizado para desplegar un contenedor de Docker. Entendemos estas imágenes como la plantilla base desde la que partimos para crear nuevos contenedores para ejecutar aplicaciones, o bien para crear una nueva imagen.

Una vez estamos familiarizados con el concepto de imagen, podemos proceder a comentar los pasos realizados por Docker para mostrarnos el mensaje de la figura 22. Dichos pasos serían los siguientes:

1. El cliente de docker contacta con el “daemon” de docker.
2. El “daemon” descarga la imagen “hello-world” del repositorio público de imágenes de docker (Docker Hub).
3. El “daemon” crea un nuevo contenedor a partir de esa imagen, que correrá un ejecutable que producen la salida que hemos obtenido.
4. El “daemon” envía la información de salida del contenedor al cliente de docker, permitiendo al usuario ver la salida en su terminal.

Por otro lado, en esa misma ejecución del contenedor “hello-world”, se nos invita a ejecutar lo siguiente:

```
docker run -it ubuntu bash
```

Si analizamos la sintaxis del comando anterior, podemos diferenciar entre cinco elementos diferentes:

- `docker`. Binario de la máquina linux, corresponde con el cliente de docker.
- `run`. Argumento del cliente de docker, permite crear y ejecutar un contenedor.
- `-it`. Opciones del argumento `run`. En este caso, tenemos configurado que sea de tipo interactivo (`-i`), es decir que nos muestre por consola la salida del comando; y además, hemos seleccionado que configure el terminal como un terminal dentro del contenedor que vamos a crear (`-t`).
- `ubuntu`. Imagen que hemos elegido para nuestro contenedor. Primero la buscará localmente, y si no la encuentra, comprobará en el repositorio público de imágenes (Docker Hub).
- `bash`. Binario a ejecutar dentro del contenedor, en este caso corresponde con una consola shell.

Una vez ejecutamos dicho comando, automáticamente se crea un contenedor con una imagen de `ubuntu` y se nos ejecuta una consola, en la que podemos navegar y trabajar de manera aislada a nuestra máquina host. (ver Figuras 23 e 24).


```
rani@rani-server ~ % docker run -it ubuntu bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
7b1a6ab2e44d: Already exists
Digest: sha256:626ffe58f6e7566e00254b638eb7e0f3b11d4da9675088f4781a50ae288f3322
Status: Downloaded newer image for ubuntu:latest
root@374e417f9fb2:/#
```

Figura 23: Ejecución comando `docker run -it ubuntu bash` para levantar un contenedor ubuntu

Para comprobar que estamos en una distribución ubuntu, y además ver la versión que el docker “daemon” ha descargado, ejecutamos el comando siguiente. Como podemos comprobar, el contenedor está utilizando la última versión LTS (Long Term Support, versiones más estables y probadas que tendrán soporte durante un largo periodo de tiempo) disponible hasta la fecha.

```
rani@rani-server ~ % docker run -it ubuntu bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
7b1a6ab2e44d: Already exists
Digest: sha256:626ffe58f6e7566e00254b638eb7e0f3b11d4da9675088f4781a50ae288f3322
Status: Downloaded newer image for ubuntu:latest
root@33960b103434:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.3 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.3 LTS"
VERSION_ID="20.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
VERSION_CODENAME=focal
UBUNTU_CODENAME=focal
root@33960b103434:/#
```

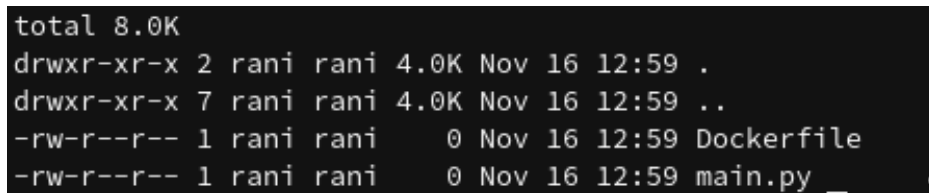
Figura 24: Comprobación de la versión de ubuntu del contenedor desplegado

Concepto de **Dockerfile**. Primer contenedor

En este apartado, a modo de ejemplo, vamos a crear nuestro propio contenedor en el que correremos una aplicación de Python muy sencilla [44]. Para ello, primero vamos a crear una carpeta en nuestro ordenador, en ella crearemos dos archivos:

- Uno se llamará `main.py` y contendrá el código que será ejecutado en el contenedor.
- Otro llamado `Dockerfile`, en el que detallaremos las instrucciones que tiene que seguir docker para crear nuestro contenedor.

Por lo tanto, la carpeta nos quedaría tal que así (ver Figura [25]):



```
total 8.0K
drwxr-xr-x 2 rani rani 4.0K Nov 16 12:59 .
drwxr-xr-x 7 rani rani 4.0K Nov 16 12:59 ..
-rw-r--r-- 1 rani rani  0 Nov 16 12:59 Dockerfile
-rw-r--r-- 1 rani rani  0 Nov 16 12:59 main.py
```

Figura 25: Comprobación de la versión de ubuntu del contenedor desplegado

1. Primero procedemos a editar el archivo Python `main.py`, para ello utilizamos el siguiente código:

Ejemplo 28: Código Python de ejemplo para crear un `Dockerfile`

```
1 #!/usr/bin/env python3
2
3 print("Saludos desde tu contenedor!!")
4
```

2. Ahora procederemos a la parte de modificar el `Dockerfile`.

Lo primero que tenemos que tener claro es lo que queremos que haga nuestro contenedor. En este caso sería que ejecutase el código Python de `main.py`, en otro caso podría ser que quisiéramos desplegar otro tipo de aplicación. Para hacer esto, primero tenemos que buscar una imagen que nos sirva de base para construir nuestro contenedor. En el caso de Python, nos podría servir un `ubuntu`, ya que viene con Python preinstalado, sin embargo, vamos a buscar una imagen mucho más específica. Para ello, nos dirigimos a la página de DockerHub <https://hub.docker.com/> y buscamos Python en el buscador (ver Figura 26).

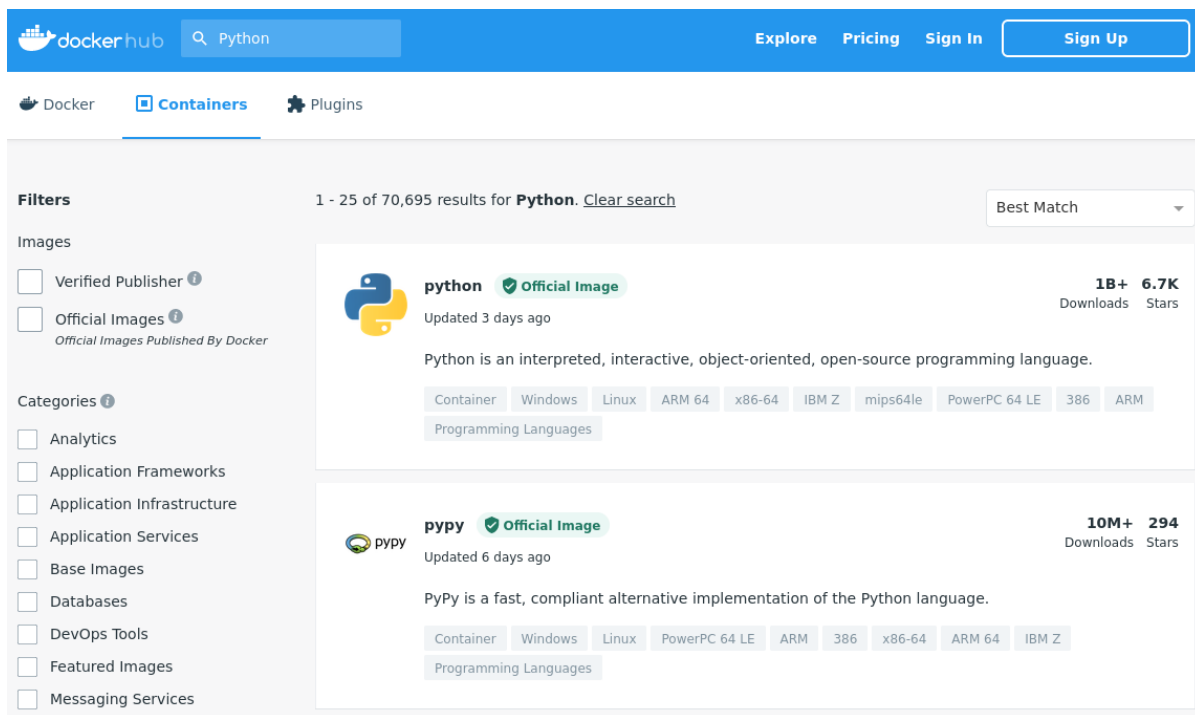


Figura 26: Búsqueda en DockerHub de una imagen base para nuestro contenedor.

El primer resultado que obtenemos es el de una imagen de contenedor que nos permite ejecutar código Python. Además, podemos ver como es muy apoyada por la comunidad (dato importante de cara a al estabilidad y seguridad de nuestro contenedor). Por lo tanto, usaremos la imagen python para utilizarla como base de nuestro contenedor. Ahora, procedemos a editar el archivo Dockerfile tal que así:

Ejemplo 29: Contenido del archivo Dockerfile para crear contenedor con código Python

```

1  # Un Dockerfile siempre necesita importar una imagen como base
2  # Para ello utilizamos 'FROM'
3  # Elegimos 'python' para la imagen y 'latest' como version de esa imagen
4  FROM python:latest
5
6  # Para ejecutar nuestro codigo Python, lo copiamos dentro del contenedor
7  # Para ello utilizamos 'COPY'
8  # El primer parametro 'main.py' es la ruta origen del archivo en el host
9  # El segundo parametro '/' es la ruta destino del archivo dentro del
   contenedor
10 # En este caso, ponemos el archivo en el root del sistema
11 COPY main.py /
12
13 # Definimos el comando a ejecutar cuando iniciemos el contenedor
14 # Para ello utilizamos 'CMD'
15 # Para ejecutar la aplicacion utilizariamos "python ./main.py".
16 CMD [ "python", "./main.py" ]
17

```

3. Una vez tenemos los dos archivos, ya podemos crear la imagen de nuestro contenedor (ver Figura 27). Para ello, tenemos que ejecutar el siguiente comando:

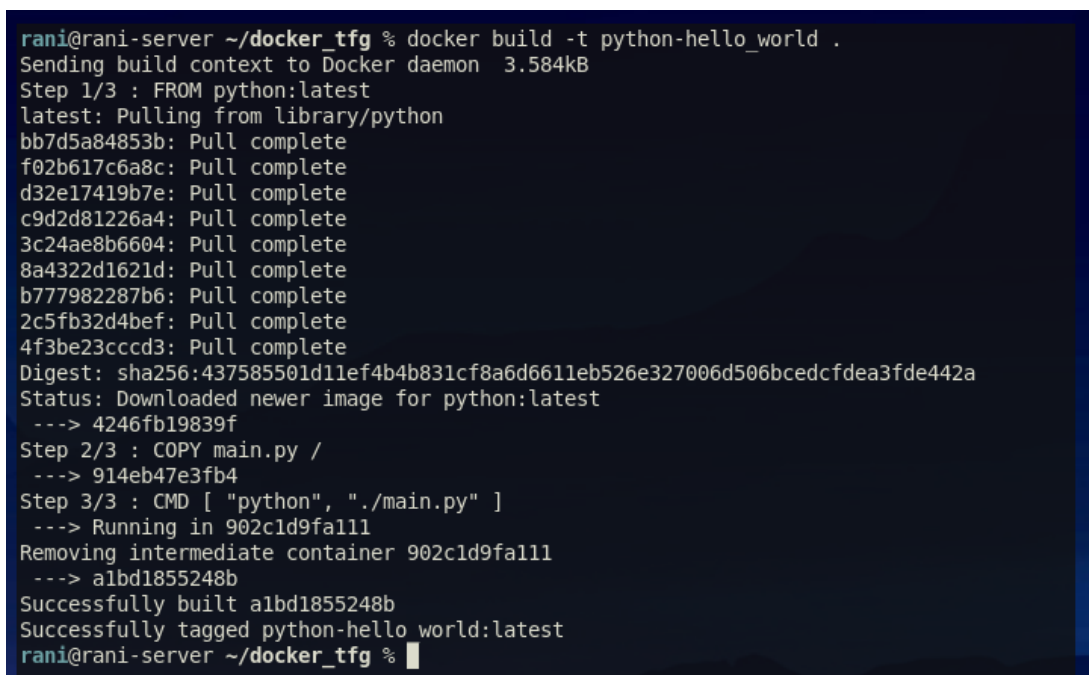
```
$ docker build -t python-hello_world .
```

La opción `-t` nos permite asignar un nombre a nuestra imagen, en nuestro caso hemos elegido `python-hello_world`.

4. Ya tenemos nuestra imagen creada, por lo que podemos ejecutar nuestro contenedor y comprobar que nuestro código Python se ejecuta correctamente. Para lanzar el contenedor ejecutamos el siguiente comando:

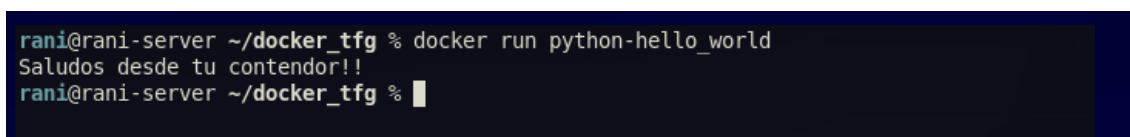
```
$ docker run python-hello_world
```

En la terminal deberíamos comprobar el código se ha ejecutado correctamente (ver Figura 28).



```
rani@rani-server ~/docker_tfg % docker build -t python-hello_world .
Sending build context to Docker daemon 3.584kB
Step 1/3 : FROM python:latest
latest: Pulling from library/python
bb7d5a84853b: Pull complete
f02b617c6a8c: Pull complete
d32e17419b7e: Pull complete
c9d2d81226a4: Pull complete
3c24ae8b6604: Pull complete
8a4322d1621d: Pull complete
b777982287b6: Pull complete
2c5fb32d4bef: Pull complete
4f3be23cccd3: Pull complete
Digest: sha256:437585501d11ef4b4b831cf8a6d6611eb526e327006d506bcedcfdea3fde442a
Status: Downloaded newer image for python:latest
--> 4246fb19839f
Step 2/3 : COPY main.py /
--> 914eb47e3fb4
Step 3/3 : CMD [ "python", "./main.py" ]
--> Running in 902c1d9fa111
Removing intermediate container 902c1d9fa111
--> albd1855248b
Successfully built albd1855248b
Successfully tagged python-hello_world:latest
rani@rani-server ~/docker_tfg %
```

Figura 27: Creación de imagen Docker para aplicación Python de pruebas



```
rani@rani-server ~/docker_tfg % docker run python-hello_world
Saludos desde tu contendor!!
rani@rani-server ~/docker_tfg %
```

Figura 28: Ejecución de la imagen Docker creada, utilizando Dockerfile

6 Caso práctico: Virtualización para simulación de redes

6.1 Interconexión física de diferentes red virtuales

6.2 Evaluación de prestaciones

7 Conclusiones

7.1 Propuestas futuras

Bibliografía

Enlaces y referencias

1. *Namespaces*
2. Tutorial: Espacio de nombres en Linux
3. *Time namespaces coming to linux*
4. *Container is a lie. Namespaces*
5. *Namespaces. Uso de cgroups.*
6. *Introduction to Network Namespaces*
7. *Build a container by hand: the mount namespace*
8. Identificador de procesos (*process id*)
9. *Linux PID namespaces work with containers*
10. *Network Namespaces*
11. *Introduction to Linux interfaces for virtual networking*
12. *Introducción a los grupos de control (cgroups) de Linux*
13. *Time namespaces*
14. *Network namespaces. Assign and configure*
15. How to configure Network Between Guest VM and Host in Virtualbox
16. How to install ansible on fedora for it and server automation
17. *Herramientas de virtualización libres para sistemas GNU/Linux*
18. *What is Network Function Virtualization(NFV)?*
19. NFV white paper
20. Youtube: NFV y SDN: las redes del futuro y del presente - Cristina Santana — T3chFest 2018
21. Percentage of server that run in Linux
22. ¿Qué son las redes definidas por software (SDN)?
23. ¿Qué son las redes virtuales?
24. *Arch Linux Wiki: udev*

25. Predictable Network Interface Names
26. Open vSwitch
27. Veth Devices, Network Namespaces and Open vSwitch
28. IEEE 802.1ad
29. FS.COM QinQ Operation
30. OpenWrt: Linux Network Interfaces
31. SDN & NFV
32. TUN/TAP interface (on Linux)
33. Creating tap/tun devices with IP tuntap and tuncctl as detailed in Linux network tools
34. Principio de diseño del controlador TUN/TAP de la tarjeta virtual
35. What are namespaces cgroups how do they work
36. TFG. Evaluación de prestaciones mediante NVF
37. Aplicación sock
38. What is chroot jail and How to Use it?
39. ArchLinux Wiki: Linux Containers
40. Ubuntu LXC
41. ArchLinux Wiki: Xorg
42. How to create unprivileged LXC container
43. Docker overview
44. A beginner's guide to Docker - how to create your first Docker application
45. Linux containers primitives: mount namespaces and information leaks
46. Mount namespaces and shared subtrees
47. Arch Linux Wiki: Tmpfs
48. CloudNativeLab: Mount namespaces
49. Youtube LiveOverflow: How Docker Works - Intro to Namespaces

Imágenes

1. NFV white paper
2. *What is Network Function Virtualization(NFV)?*
3. SDN & NFV
4. FS.COM QinQ Operation
5. Namespaces y API de acceso.
6. Wikipedia: TUN/TAP
7. Docker - container and lightweight virtualization
8. Linux Containers Logotipo
9. Imagen Docker Architecture
10. Logotipo Docker

Anexos

Anexo 1. Instalación de **ansible** para automatizar una VM

En el caso de que queramos utilizar la herramienta ansible para configurar una VM, tendremos que seguir los siguientes pasos.

1. Asegurarnos de que tenemos instalado el programa en el host. La principal dependencia de ansible es Python.

Para instalar ansible en diferentes distribuciones sería tal que:

- Ubuntu 21.04:

Ejemplo 30: Instalación ansible en Ubuntu

```
1 sudo apt install -y software-properties-common
2 sudo add-apt-repository --yes --update ppa:ansible/ansible
3 sudo apt update
4 sudo apt install -y ansible
5
```

- Fedora 32:

Ejemplo 31: Instalación ansible en Fedora

```
1 sudo dnf update
2 sudo dnf install ansible
3
```

- OpenSUSE:

Ejemplo 32: Instalación ansible en OpenSUSE

```
1 zypper in ansible
2
```

- Arch Linux:

Ejemplo 33: Instalación ansible en Arch Linux

```
1 sudo pacman -S ansible
2
```

2. Creamos una clave SSH para utilizarla como método de autenticación con la maquina virtual.

Ejemplo 34: Crear clave SSH para autenticación en VM

```
1 (host)$ ssh-keygen -t ed25519 -C "Host Ansible ssh key"
2
```

3. Utilizando nuestro gestor de máquinas virtuales, encendemos la VM. Es importante que nos aseguremos que tiene internet, por lo que configuramos que la interfaz de red sea de tipo *NAT*. Dentro de la máquina, buscaremos que ip tiene asignada con el comando `ip address`. Desde este momento, dejaremos la máquina virtual encendida.
4. Procedemos a copiar la clave pública SSH que hemos generado en el paso dos. Utilizamos el siguiente comando, sustituyendo con el usuario e IP específico de la máquina. Tendremos que configurar una red tipo *Host-only network* para poder comunicarnos correctamente con nuestra máquina virtual, para ello podemos seguir los pasos detallados en [15]

Ejemplo 35: Copiar clave pública de ansible a VM

```
1 (host)$ ssh-copy-id -i $HOME/.ssh/id_ed25519.pub <user>@<IP VM>
2
```

5. En este momento ya podríamos conectar con la máquina virtual utilizando `ansible`. Sin embargo, por comodidad, lo que vamos a hacer es crear un archivo que funcione con inventario de servidores. Nos servirá para guardar las direcciones IP de los servidores en los que queremos ejecutar comandos remotos con `ansible`. Para ello, creamos un archivo `inventory`. Como ejemplo, dicho archivo puede ser tal que:

Ejemplo 36: Contenido del archivo inventory de ansible

```
1 ## VMs locals
2 [virtualbox]
3 10.0.100.1
4 10.0.100.2
5
6 ## Server SSH
7 [server]
8 192.168.1.200
9
```

6. Para probar la conectividad. Nos aseguramos de que tenemos la máquina virtual encendida y que además, hemos copiado la clave ssh y tenemos su IP añadida a nuestro inventario. Después, procedemos a ejecutar el siguiente comando en la máquina host:

```
(host)$ ansible -i inventory -m ping
```

En la consola nos aparecerá la información de cada uno de las IP que habrá probado para ejecutar el comando remoto `ping`.

Anexo 2. Configuración de *Guest Network* para comunicar con la VM

Para comunicar nuestra máquina host [15], con una *virtual machine* es necesario que estén en una misma red, es por esto por lo que vamos a crear una red en específico para ello. En el programa *Virtualbox*, esta funcionalidad recibe el nombre de *Host-only*.

Es importante que cuando queramos configurar una máquina virtual con esta funcionalidad que siempre pongamos el adaptador de red 1 como el que tendrá la comunicación *guest*, mientras que el adaptador 2 será el que tendrá la conexión a internet via *NAT*.

Procedemos a enumerar los pasos a seguir para configurar una máquina virtual con conectividad con el host, utilizando *Virtualbox*.

1. Abrimos el programa *Virtualbox*. Navegamos a la sección: **File** → **Preferences**.
2. Seleccionamos la tabla de ***Host-only Networks***. Procedemos a pulsar el botón + para añadir una nueva red.
3. Asignamos las diferentes IP que consideremos. Es importante que dejemos activado el servidor DHCP, así la máquina virtual tendrá una IP válida dentro de la red.
4. Ahora, lo que tenemos que hacer es añadir una interfaz de red a nuestra máquina virtual. Para ello, vamos a las **Settings** de la máquina virtual. En el apartado de Network, cambiamos el **Adapter 1** a **Host-only Adapter**, y asignamos una segunda interfaz de red con NAT (Adapter 2 attached to NAT).
5. Para comprobar que lo hemos hecho correctamente. Arrancamos la máquina virtual, ejecutamos el comando `ip address`. Nos deberían aparacer las diferentes interfaces que previamente hemos configurado, es decir, una interfaz de loopback, otra que corresponde al Host-only network, y por último, una que corresponda con la interfaz de NAT.

Anexo 3. Playbooks **ansible**