

Escuela Técnica
Superior
de Ingeniería de
Telecomunicación

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

15 de marzo de 2022

TRABAJO FIN DE GRADO
Grado en Ingeniería Telemática

Autor: Enrique Fernández Sánchez

Tutor: José María Malgosa Sanahuja



Universidad
Politécnica
de Cartagena

Índice general

Índice de figuras	4
Listado de ejemplos	6
Glosario de términos	7
Agradecimientos	8
1 Introducción	9
1.1 Contexto del trabajo	9
1.2 Objetivos	10
1.3 Resumen capítulos de la memoria	10
2 Virtualización de Funciones de Red	12
2.1 Tecnologías implicadas	14
2.2 Virtualización ligera	15
3 Interfaces de red virtuales en Linux	16
3.1 Nombrado predecible de dispositivos	16
3.2 MAC compartida: <code>enp2s0:{0,1,2...}</code>	18
3.3 VLAN 802.1q: <code>enp2s0.{0,1,2...}</code>	18
3.4 VLAN 802.1ad: <code>enp2s0.{0,1,2...}.{0,1,2...}</code>	19
3.5 Pares ethernet virtuales	20
3.6 TUN/TAP	22
4 Espacios de nombres en Linux	29
4.1 ¿Qué es un <i>espacio de nombres</i> ?	29
4.2 ¿Cuántos namespaces hay?	29
4.3 ¿Cómo crear/acceder a un namespace?	30
4.4 UTS namespace	30
4.5 Mount namespace	32
4.5.1 Casos de uso del montaje tipo bind	35
4.6 Process ID namespace	40
4.7 Network namespace	41
4.8 User ID namespace	44
4.9 Interprocess Communication namespace	44
4.10 Control groups namespace	45
4.11 Time namespace	48
4.12 Ejemplo: topología de red con <code>ip</code>	49
4.13 Ejemplo: topología de red con <code>unshare</code>	51

5	Virtualización ligera y contenedores	52
5.1	Creando nuestro propio “contenedor”	53
5.2	Contenedores LXC	62
5.3	Contenedores Docker	69
6	Caso práctico: Virtualización para la simulación de redes	80
6.1	Mininet: Simulador de redes	80
6.1.1	Primeros pasos	81
6.1.2	Ejemplos	82
6.2	Containernet: Simulador de redes	92
6.2.1	Primeros pasos	93
6.2.2	Implementación topología simple	94
7	Conclusiones y propuestas futuras	98
8	Bibliografía	100
	Enlaces y referencias	100
	Imágenes	102
A	Anexos	103
A.1	Código asociado a topologías de red	103

Índice de figuras

2.1	Comparativa enfoque clásico de las redes contra el enfoque virtualizado.	13
2.2	Esquema y ámbito de aplicación de las APIs Southbound y Northbound . .	14
2.3	Comparativa entre virtualización dura y virtualización ligera.	15
3.1	Regla udev definida por el usuario	17
3.2	Diagrama conexión VLANs	18
3.3	Trama ethernet utilizando VLAN 802.1ad	19
3.4	Ejemplo básico de utilización de pares virtuales ethernet	20
3.5	Ejemplo avanzado de utilización de pares virtuales ethernet, utilizando bridge	21
3.6	Comparativa en capa OSI de las interfaces TUN/TAP.	23
3.7	Uso de la aplicación sock con interfaces tuntap.	28
3.8	Tabla de encamiamiento para una interfaz tuntap.	28
3.9	Captura de un paquete emitido por la interfaz tap0.	28
4.1	Archivos asociados al grupo test_cg una vez lo creamos	45
4.2	Salida tras ejecutar el programa de Python sin limitar	46
4.3	Salida tras ejecutar el programa de Python, una vez limitado	46
4.4	Diferentes namespaces en Linux y su API de acceso.	48
5.1	Ejecución del script setup-image (1/2)	59
5.2	Ejecución del script setup-image (2/2)	60
5.3	Ejecución del script runtime-exec (1/2)	60
5.4	Ejecución del script runtime-exec (2/2)	61
5.5	Ejecución del script runtime-exec con comando httpd	61
5.6	Ejecución del script runtime-exec con comando httpd	61
5.7	Logotipo del proyecto linuxcontainers.org	62
5.8	Comprobación instalación de LXC usando lxc-checkconfig	66
5.9	Instalación de contenedor LXC (1/2)	67
5.10	Instalación de contenedor LXC (2/2)	67
5.11	Comprobación de la versión de Ubuntu instalada en el contenedor LXC	68
5.12	Namespaces asociados a un contenedor LXC ejecutado en el host	68
5.13	Arquitectura Docker	69
5.14	Logotipo de Docker	70
5.15	Ejecución comando docker version sin “demonio” en funcionamiento	70
5.16	Ejecución comando docker run hello-world para comprobar instalación .	71
5.17	Ejecución comando docker run -it ubuntu bash para levantar un con- tenedor ubuntu	73
5.18	Comprobación de la versión de ubuntu del contenedor desplegado	73
5.19	Comprobación de la versión de Ubuntu del contenedor desplegado	74

5.20	Búsqueda en DockerHub de una imagen base para nuestro contenedor.	75
5.21	Creación de imagen Docker para aplicación Python de pruebas	76
5.22	Ejecución de la imagen Docker creada, utilizando Dockerfile	76
5.23	Ejecución contenedor Docker para ver sus namespaces asociados.	77
5.24	Resultado de comando <code>ip netns list</code> con contenedores Docker.	77
5.25	Network namespaces de un Docker container	78
5.26	Comprobación de si un network namespaces pertenece a un contenedor Docker. .	79
5.27	Namespaces asociados a un contenedor Docker.	79
6.1	Diagrama de la topología simple a implementar.	82
6.2	Captura de la configuración del switch usando Open vSwitch.	83
6.3	Ejecución ping entre <code>host1</code> y <code>host2</code>	84
6.4	Ejecución topología simple en <code>mininet</code>	84
6.5	Ping entre hosts de topología simple en <code>mininet</code>	85
6.6	Comprobación de namespaces utilizados por topología en <code>mininet</code>	85
6.7	Ejecución de <code>iperf</code> sobre topología simple en <code>mininet</code>	86
6.8	Ejecución topología simple limitada en <code>mininet</code> . (1/2)	87
6.9	Ejecución topología simple limitada en <code>mininet</code> . (2/2)	87
6.10	Ejecución topología <code>mininet</code> sin controlador asignado (terminal A)	89
6.11	Ejecución del controlador Ryu (terminal inferior, B) y posterior <code>pingall</code> en <code>mininet</code> (terminal superior, A)	90
6.12	Herramienta <i>Topology Viewer</i> del controlador Ryu	91
6.13	Logotipo del proyecto <code>containernet</code>	92
6.14	Instalación <code>containernet</code> en Ubuntu 20.04 LTS (1/2)	93
6.15	Instalación <code>containernet</code> en Ubuntu 20.04 LTS (2/2)	94
6.16	Diagrama topología simple utilizando <code>containernet</code>	94
6.17	Ejecución de topología simple en <code>containernet</code> (1/5)	95
6.18	Ejecución de topología simple en <code>containernet</code> (2/5)	96
6.19	Ejecución de topología simple en <code>containernet</code> (3/5)	96
6.20	Ejecución de topología simple en <code>containernet</code> (4/5)	97
6.21	Ejecución de topología simple en <code>containernet</code> (5/5)	97

Listado de ejemplos

3.1	Ejemplo de uso de <code>tunctl</code> para controlar interfaces TUN/TAP	24
3.2	Ejemplo de uso de <code>ip</code> para controlar interfaces TUN/TAP	24
3.3	Aplicación de ejemplo para crear tun/tap (<code>tuntap.c</code>)	25
3.4	Instrucciones para realizar el ejemplo de crear tun/tap	26
3.5	Compilar e instalar programa <code>sock</code>	27
3.6	Creación interfaz TAP	27
3.7	Uso de aplicacion <code>sock</code> para crear cliente servidor asociado a un puerto	27
4.1	Ejemplo de uso de UTS namespace	30
4.2	Ejemplo de un persistencia namespace	31
4.3	Uso de <code>mount namespace</code> con dispositivo físico	32
4.4	Uso de <code>mount namespaces</code> con “ <code>tmpfs</code> ”	37
4.5	Caso de uso de <code>shared subtree</code>	39
4.6	Caso de uso de <code>slave mount</code>	39
4.7	Caso de uso de <code>bind</code> y <code>shared subtree</code>	39
4.8	Uso de <code>process id namespace</code>	40
4.9	Creation persistent network namespace	41
4.10	Comprobar network namespaces existentes	41
4.11	Asignar interfaz loopback a un namespace	41
4.12	Ejecutar cualquier programa con un network namespace	41
4.13	Ejemplo configuración de NAT entre <code>eth0</code> y <code>veth</code>	42
4.14	Configuración de NAT para dar Internet a un network namespace	43
4.15	Ejemplo de uso UID namespace	44
4.16	Programa en Python que consume 4 GB de RAM	46
5.1	Creación de imagen para contenedor utilizando Alpine y union mounts	53
5.2	Ejecución de runtime basado en imagen de Alpine	55
5.3	Configuración interfaz NAT bridge en LXC	63
5.4	Configuración contenedor LXC para usar NAT bridge	64
5.5	Crear un contenedor con privilegios en LXC	64
5.6	Crear un contenedor con privilegios en LXC, modo no interactivo	64
5.7	Configuración interfaz NAT bridge y aplicaciones X a un contenedor LXC	65
5.8	Configuración para mapear UID y GID para un contenedor sin privilegios en LXC	66
5.9	Código Python de ejemplo para crear un Dockerfile	74
5.10	Contenido del archivo Dockerfile para crear contenedor con código Python	75
A.1	Código topología simple utilizando mininet (<code>basic.py</code>)	103
A.2	Código topología simple con limitación de «recursos» en mininet (<code>limit.py</code>)	103
A.3	Código topología simple en containernet (<code>containernet_example.py</code>)	104

Glosario de términos

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 interprete 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 se refiere al identificador de grupo dentro del sistema de Linux.

root. Cuenta superusuario del sistema operativo Linux.

NAT (*Network Address Translation*). Traducción de direcciones de red, mecanismo utilizado por router para cambiar paquetes entre redes con direcciones IP diferentes.

DHCP (*Dynamic Host Configuration Protocol*). Protocolo de red que permite a un servidor asignar dinámicamente direcciones IP y otros parámetros, a cada dispositivo de la red en la que se encuentre.

API (*Application Programming Interface*). Conjunto de subrutinas, funciones y procedimientos que ofrecen cierto software, con el objetivo de ser utilizado por otro software externo.

SNMP (*Simple Network Management Protocol*). Protocolo de la capa de aplicación que permite el intercambio de información de administración entre los dispositivos de una red habilitados con SNMP.

Agradecimientos

En primer lugar, agradecer a mi tutor, José María Malgosa Sanahuja, por presentarme este proyecto y animarme en todo momento para su realización. Agradezco mucho su tiempo, su dedicación y su ayuda, que han sido una parte clave para la realización de este trabajo.

Por otro lado, me gustaría agradecer a mi familia, en especial a mis padres, Encarnación y Miguel Ángel, y a mi hermano y a mi cuñada, Miguel y Yolanda; ellos han supuesto un pilar fundamental en este camino que ha sido para mi la ingeniería, ellos han sido mi principal apoyo, y sin él, no hubiera llegado hasta donde estoy hoy. También agradecer a mis padrinos, Ana y Anastasio, ya que siempre han tenido palabras de ánimo para mi, incluso en este último año, en el que han sabido transmitir esa fortaleza, tan propia de ellos. Especial mención a mis abuelos, estoy seguro que estarían muy orgullosos de mi camino y de cómo su nieto es ahora ingeniero. Agradecer también a mis suegros, María Dolores y Francisco, ya que suponen otra fuente importante de apoyo y confianza.

Especial mención a Lucía Francoso Fernández, compañera del grado en sistemas de telecomunicación, con la cual he tenido la suerte de emprender codo con codo muchos de los proyectos que he llevado entre manos estos años. Fuente de apoyo incondicional, una mina de creatividad y una brillante ingeniera. Sin ella, mi paso por la universidad no hubiera el mismo.

Por último, agradecer a mis compañeros del *Free Open Source Club* (<https://fosc.space>), ya que gracias a ellos he puesto en valor el movimiento *Open Source*. Me han servido de trampolín para entrar a un mundo muy bonito que es el conocimiento libre y poder vivir experiencias únicas. Con ellos, he pasado muchas horas de trastear y aprender sobre cualquier cosa que se nos ocurría, lo que nosotros amistosamente llamamos “*hackear*”.

Capítulo 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 muchos de los conceptos que ha ido aprendiendo durante su paso por el grado, pudiendo añadir puntos de innovación, y aportar soluciones nuevas a un proyecto específico.

En este documento recojo lo que sería mi memoria en relación al TFE. En él se detallarán las diferentes investigaciones realizadas sobre el concepto de virtualización en sistemas Linux, el funcionamiento de los contenedores y la utilización de la virtualización ligera para la evaluación de redes de comunicaciones, en nuestro caso, de conmutación de paquetes.

Además, toda documentación y códigos utilizados en este proyectos se ha liberado en un repositorio de *GitHub* (<https://github.com/>). En dicho repositorio, encontraremos toda la información recopilada para este trabajo, además de los códigos fuente de la memoria, y las pruebas realizadas. La dirección del repositorio es la siguiente: <https://github.com/Raniita/lightweight-virtualization-nfv/>

1.1 Contexto del trabajo

Este proyecto nace con el objetivo de profundizar en conceptos novedosos para el ámbito de la ingeniería telemática, como virtualización de hardware físico de red y los contenedores, con el fin de solucionar problemas de escalabilidad y de optimización.

Por otra parte, muchas de las herramientas de evaluación y simulación de redes de comunicaciones son costosas en términos de medios a utilizar (tanto hardware como software), lo que limita seriamente su escalabilidad a la hora de evaluar redes de cierta envergadura. La virtualización de redes (en especial la virtualización ligera) nos permitirá desarrollar simuladores de redes más complejas sin consumir excesivos «recursos» computacionales. Por ello, en este proyecto 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.

1.2 Objetivos

Como bien hemos adelantado en el apartado anterior, el objetivo principal de este proyecto es el de estudiar propuestas en el ámbito de simulación y evaluación de redes de comunicación, 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 los espacios de nombres (*namespaces*) y cómo podemos aplicarlo para virtualizar redes.
- Profundizar en el concepto de interfaces virtuales en Linux.
- Desgranar el concepto amplio de contenedor, relacionándolo con los espacios de nombres.
- Aportar una serie de ejemplos que puedan servir de guía para la evaluación de redes de comunicaciones, utilizando virtualización ligera.

1.3 Descripción de los capítulos 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*. Comienzo del proyecto. Se exponen introducción, objetivos y el contexto del trabajo.
- **Capítulo 2:** *Virtualización de funciones de red*. Se pretende contextualizar el motivo por el que la virtualización de las redes es importante. Además, se comentan las tecnologías involucradas para dar lugar a la virtualización.
- **Capítulo 3:** *Interfaces de red virtuales en Linux*. Se describirán las diferentes interfaces de red virtuales que nos ofrece Linux. Se evaluarán sus ventajas y cómo nos pueden ser útiles para implementar topologías de red virtualizadas.
- **Capítulo 4:** *Espacio de nombres en Linux*. Se estudiará el concepto de espacio de nombres. Se aportará una explicación de cada espacio de nombres disponibles, además de ejemplos de aplicación de los más importantes.
- **Capítulo 5:** *Virtualización ligera y contenedores*. Se profundiza en diferentes técnicas de virtualización ligera, además de proponer ejemplos y primeros pasos para cada una de las soluciones propuestas.

- **Capítulo 6:** *Caso práctico: Virtualización para la simulación de redes.* Se proponen varios ejemplos de *software* en el que se utiliza virtualización ligera para la simulación de redes.
- **Capítulo 7:** *Conclusiones.* Se muestran las conclusiones obtenidas, y se proponen líneas de investigación futuras.

Capítulo 2

Virtualización de Funciones de Red

El punto de partida de la “virtualización de funciones de red” (NFV) surge a partir de las necesidades de las operadoras de red para solucionar los problemas de escalado de la red, generando una red poco flexible. Estos problemas se pueden resumir en los siguientes:

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

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 [1] en el que se detallaba de forma teórica la solución de NFV.

En esta arquitectura de red basada en virtualización, distinguimos varios roles:

- VNF (*Virtual Network Function*). Aplicaciones software que ofrecen ciertas funciones (enrutadores, firewall, etc) y que se implementan como máquinas virtuales, y que no necesitan un sistema hardware específico.
- NFVi (*NFV infrastructure*). Se trata de la infraestructura disponible para una arquitectura basada en NFV. Supone la infraestructura donde se desplegarán los VNF, que incluye «recursos» de computo, almacenamiento y de red.
- VIM (*Virtualized Infrastructure Manager*). Es el elemento responsable de controlar y gestionar la infraestructura NFV (NFVi). Se encarga de mantener el inventario de «recursos» y de ir “colocando” las diferentes funciones de red en los «recursos» disponibles de la red. Esto nos permite realizar tareas de orquestación.
- MANO (*Management and orchestration*). Elemento clave para la implementación de NFV. Es el encargado de coordinar los «recursos» de la red y gestionar las funcionalidades de red. Esta compuesto por los elementos: orquestador NFV (NFVO), gestor de VNF (VNFM) y gestor de infraestructura virtualizada (VIM), con los que se comunicará para coordinar los «recursos» de una arquitectura NFV.

En el modelo tradicional, cada dispositivo/funcionalidad se corresponde con un hardware concreto. Éstos son dispositivos embebidos y solo cumplen una función específica (firewall, balanceador de carga, router, etc). Utilizando la virtualización de red (NFV), podemos llegar al punto de tenerlo todo virtualizado. Teniendo una “imagen” de router, 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).

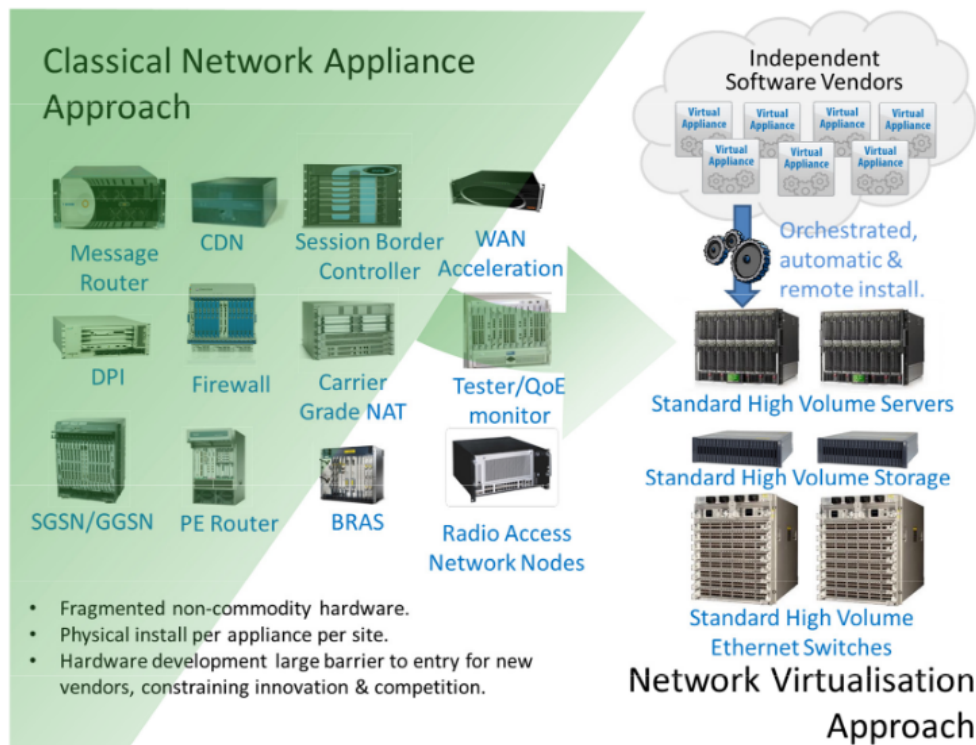


Figura 2.1: Comparativa enfoque clásico de las redes contra el enfoque virtualizado.

En el enfoque clásico, las funciones de red están basadas en tener un software y hardware específico para cada dispositivo. Sin embargo, NFV nos permite desplegar esos mismos «recursos» software y hardware en servidores físicos de propósito general, dando lugar a una mejor gestión de los «recursos» y un mayor aprovechamiento. Por lo tanto, un mismo nodo físico, puede ser DHCP, router o Firewall. [2]

La virtualización de funciones de red supone una oportunidad para reducir costes y acelerar el desarrollo de servicios para los operadores de red.

Tal y como podemos ver en la comparativa de la Figura 2.1, 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

Aunque la virtualización ya es una realidad en muchos entornos reales, para que este cambio de paradigma en las redes se materialice y pase a ser una solución viable, tienen que realizarse antes una serie de desarrollos de calado, especialmente en el campo de las Redes Definidas por Software (SDN, *Software Defined Networking*).

A modo de resumen, *Software Defined Networks* (SDN) desacopla el plano de control y el plano de datos de los routers y switches de una red. El controlador SDN es el encargado de programar los dispositivos de la red vía software. En entornos SDN, el switch/router solo sabe encaminar paquetes de datos de la forma en la que el controlador SDN lo ha programado. Hasta el momento no hay nada nuevo, ya que esto mismo se consigue -por ejemplo- mediante el protocolo SNMP. La novedad que introduce SDN consiste en que la programación de los routers/switches es dinámica, permitiendo que se vaya adaptando automáticamente a las condiciones de la red.

Para programar bajo el paradigma de SDN, se utilizan APIs llamadas respectivamente Southbound y Northbound. La primera es una API de muy bajo nivel que permite comunicar el controlador con el equipo de red. La segunda es de alto nivel y permite que el administrador de la red pueda programar los equipos de forma fácil y amigable.

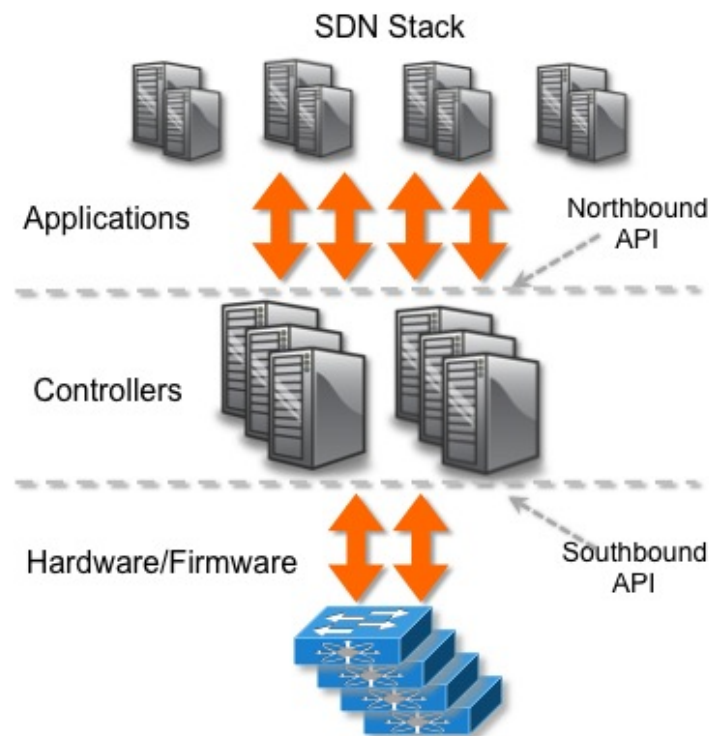


Figura 2.2: Esquema y ámbito de aplicación de las APIs Southbound y Northbound

2.2 Virtualización ligera

Entendemos la virtualización ligera (también conocida como *containers*) como la virtualización de los «recursos» propios de un sistema operativo, tales como memoria, CPU, interfaces de red, etc. Dicha virtualización la hace el propio *kernel* del sistema operativo, proporcionando diferentes espacios de usuarios aislados entre sí. En la virtualización ligera todos los sistemas virtualizados dependen del *kernel* de la máquina *host*, mientras que en el caso de una máquina virtual (virtualización dura), cada instancia tendría su propio *kernel* (ver Figura 2.3). [3]

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

- 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”.
- Disponer de la posibilidad de iniciar, mover, parar o borrar la instancia virtualizada.

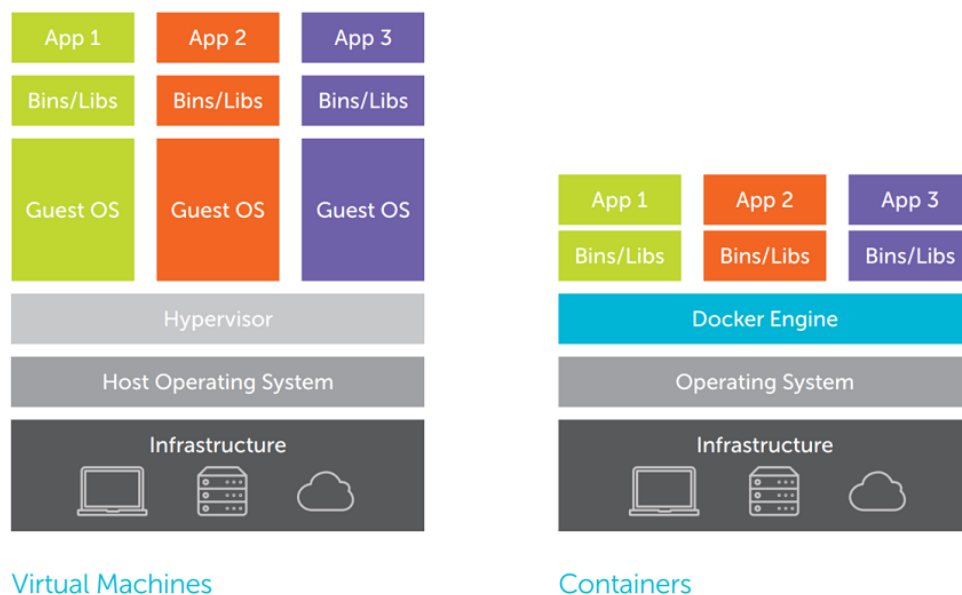


Figura 2.3: Comparativa entre virtualización dura y virtualización ligera.

A 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.

Capítulo 3

Interfaces de red virtuales en Linux

En este capítulo, vamos a profundizar en el concepto de interfaces de red virtuales, específicas al sistema operativo basados en el kernel 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 comando `ip link help`.

En este trabajo vamos a comentar la siguientes interfaces [4]:

- MAC compartida: `enp2s0:{0,1,2...}`
- VLAN 802.1q: `enp2s0.{0,1,2...}`
- VLAN 802.1ad: `enp2s0.{0,1,2...}.{0,1,2...}`
- Pares de ethernet virtuales (*veth pairs*)
- TUN/TAP

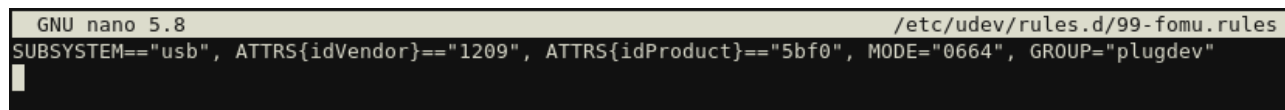
3.1 Nombres predecibles en dispositivos físicos.

Con el fin de comprender la forma en que 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, y su función principal es permitir al administrador del sistema operativo manejar dinámicamente los dispositivos conectados al ordenador mediante la gestión de eventos del sistema. Estos eventos que recibe el servicio `udev` son específicamente 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 las acciones necesarias que devuelven el control al kernel, como por ejemplo, cargar módulos específicos o un firmware de un dispositivo. [5]

Además, `udev` también se encarga de gestionar los nodos de dispositivos ubicados en el directorio `/dev` añadiéndolos, encadenándolos 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. [5]

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 3.1) 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 3.1: Regla udev definida por el usuario

En el caso de las interfaces físicas de red, vamos a suponer que estamos utilizando el etiquetado de interfaces de red tradicional, tales como `eth0`, `eth1`, etc. 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` [6]). Es por este motivo por lo que antes podíamos tener interfaces del tipo `eth0` y ahora nos encontramos con la siguiente nomenclatura `enps30`. Este cambio surge ya que anteriormente se nombraban las diferentes interfaces durante el *boot* del sistema, por lo que podría pasar que lo que en un primer momento se nombró como `eth1`, en el próximo arranque fuera `eth0`, dando lugar a incontables errores en configuración del sistema. Es por esto por lo que se empezó a trabajar en soluciones alternativas. Por ejemplo, la que acabamos de comentar (udev), 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 hot-plug, 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 *alias*. La funcionalidad principal que tienen este tipo de interfaces es la de asignar varias direcciones 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
```

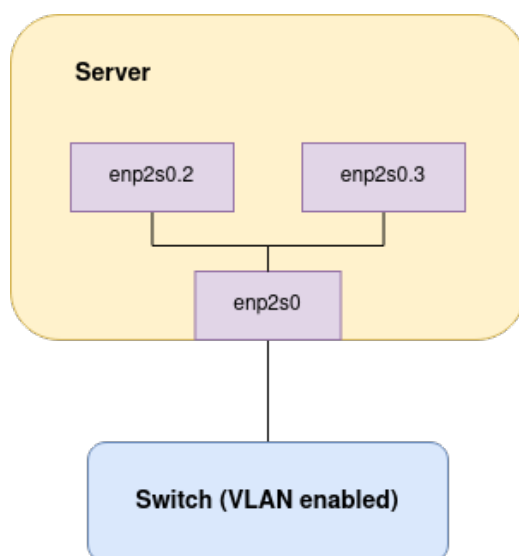


Figura 3.2: Diagrama conexión VLANs

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

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 se creó para que los ISP (*Internet Service Provider*) pudieran utilizar *tagging* VLAN en sus propias redes sin interferir con las VLANs creadas por los propios usuarios. Además que amplía el límite de 4094 VLANs diferentes permitidas por el estándar 802.1Q [7][8]. 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:

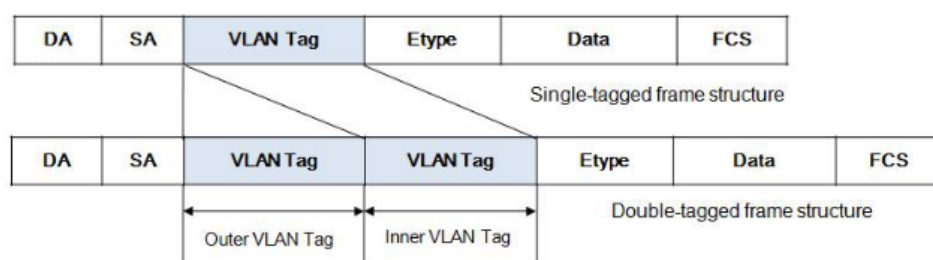


Figura 3.3: Trama ethernet utilizando VLAN 802.1ad

Para configurar un enlace utilizando este estándar, tenemos que ejecutar los siguientes comandos [9]:

```
$ 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 ethernet virtuales

Los `veth` (Ethernet virtuales) son un dispositivo virtual que forman un túnel local ethernet. El dispositivo se crea en parejas. [4]

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 está también apagado. A modo de ejemplo, nos fijamos en una estructura básica en la que dos aplicaciones se comunican utilizando `veth`, tal y como vemos en la figura (3.4)

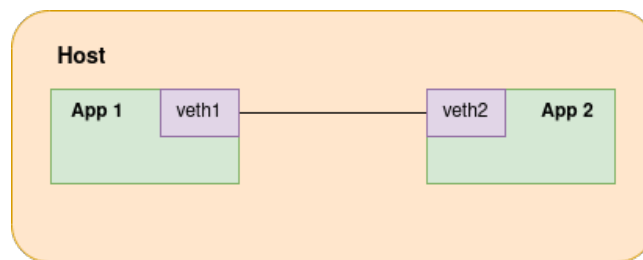


Figura 3.4: Ejemplo básico de utilización de pares virtuales ethernet

Las interfaces virtuales ethernet se inventaron con el fin de comunicar diferentes *network namespaces*. Aunque profundizaremos en ello más adelante, los namespaces 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 3.4 sería el siguiente [10]:

```
$ 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:

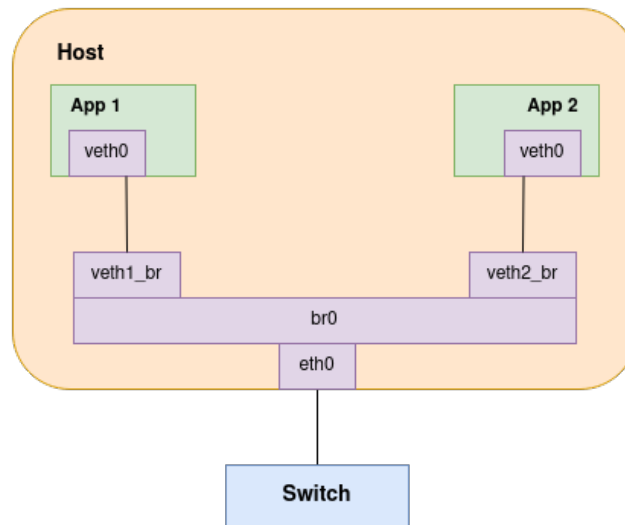


Figura 3.5: Ejemplo avanzado de utilización de pares virtuales ethernet, utilizando bridge

Como podemos comprobar en la figura 3.5, es necesario que utilizemos 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 [11][12].

```
$ 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 a través de un *socket* de red. 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 [13]:

- 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 [13]:

- 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.

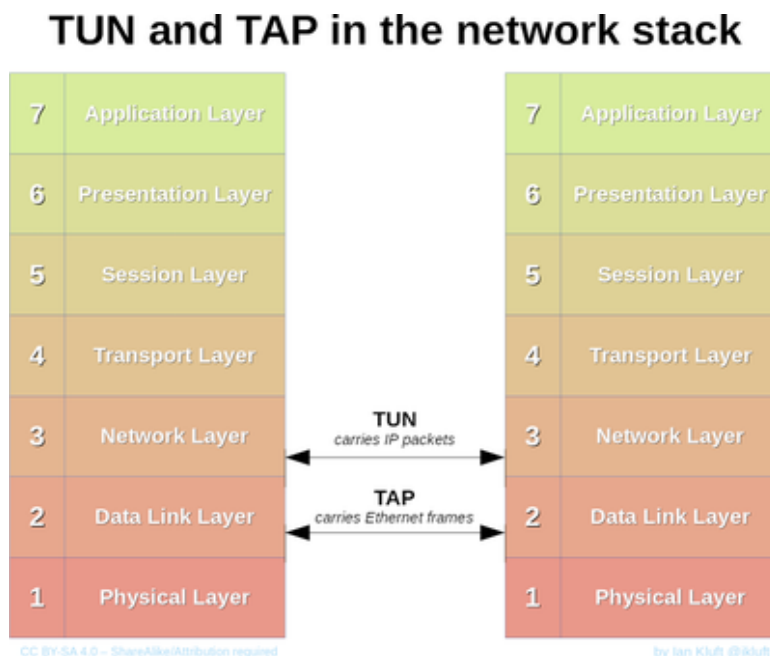


Figura 3.6: Comparativa en capa OSI de las interfaces TUN/TAP.

Si nos fijamos en la figura 3.6, 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 ampliamente utilizadas ya que nos permiten añadir a nuestro sistema tantas tarjetas ethernet virtuales (software) como se necesiten. Posteriormente -utilizando los comandos Linux tradicionales para configurar redes- se le asigna una IP y ya pueden ser utilizadas por cualquier aplicación -incluso de forma compartida- como si fuera una interfaz de red hardware.

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 3.1 y 3.2 donde se comentan algunos de los comandos más importantes para trabajar con estas interfaces en cada uno de los programas mencionados. [14]

Ejemplo 3.1: Ejemplo de uso de `tunctl` para controlar interfaces TUN/TAP

```
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 3.2: Ejemplo de uso de `ip` para controlar interfaces TUN/TAP

```
1 # Show help
2 ip tuntap help
3
4 # Create tun/tap devices
5 ip tuntap add dev tap0 mod tap      # create tap
6 ip tuntap add dev tun0 mod tun      # create tun
7
8 # Delete tun/tap devices
9 ip tuntap del dev tap0 mod tap      # delete tap
10 ip tuntap del dev tun0 mod tun      # delete tun
11
```


Ejemplo: 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 su estructura.

El código utilizado sería el que vemos en [3.3]. 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. [15]

Ejemplo 3.3: Aplicación de ejemplo para crear tun/tap (`tuntap.c`)

```
1 /**
2 Receive incoming packages over tun/tap device.
3 stdout -> size of received package
4 **/
5
6 #include <net/if.h>
7 #include <sys/ioctl.h>
8 #include <sys/stat.h>
9 #include <fcntl.h>
10 #include <string.h>
11 #include <sys/types.h>
12 #include <linux/if_tun.h>
13 #include <stdlib.h>
14 #include <stdio.h>
15
16 int tun_alloc(int flags)
17 {
18
19     struct ifreq ifr;
20     int fd, err;
21     char *clonedev = "/dev/net/tun";
22
23     if ((fd = open(clonedev, O_RDWR)) < 0) {
24         return fd;
25     }
26
27     memset(&ifr, 0, sizeof(ifr));
28     ifr.ifr_flags = flags;
29
30     if ((err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0) {
31         close(fd);
32         return err;
33     }
34
35     printf("Open tun/tap device: %s for reading...\n", ifr.ifr_name);
36
37     return fd;
38 }
39
```

```

40 int main()
41 {
42
43     int tun_fd, nread;
44     char buffer[1500];
45
46     /* Flags: IFF_TUN   - TUN device (no Ethernet headers)
47      *         IFF_TAP   - TAP device
48      *         IFF_NO_PI - Do not provide packet information
49      */
50     tun_fd = tun_alloc(IFF_TAP | IFF_NO_PI);
51
52     if (tun_fd < 0) {
53         perror("Allocating interface");
54         exit(1);
55     }
56
57     while (1) {
58         nread = read(tun_fd, buffer, sizeof(buffer));
59         if (nread < 0) {
60             perror("Reading from interface");
61             close(tun_fd);
62             exit(1);
63         }
64
65         printf("Read %d bytes from tun/tap device\n", nread);
66     }
67     return 0;
68 }
69

```

Los pasos para realizar este ejemplo serían los siguientes:

Ejemplo 3.4: Instrucciones para realizar el ejemplo de crear tun/tap

```

1 # Guardamos el código anterior en tuntap.c
2 # Compilamos el programa
3 gcc tuntap.c -o tun
4
5 ### Terminal 1
6 # Ejecutamos el binario, este se quedara a la escucha en la interfaz creada
7 ./tun
8
9 ### Terminal 2
10 # Asignamos una IP a la interfaz recién creada
11 ip addr add 192.168.209.138/24 dev tun0
12 ip link set dev tun0 up
13
14 # Capturamos el tráfico que viaja por la interfaz
15 tcpdump -i tun0
16
17 ### Terminal 3
18 # Mandamos tráfico a la interfaz creada, un ping por ejemplo:
19 ping -c 4 192.168.209.139 -I tun0
20

```

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` [16], que nos servirá para crear un socket IP en un puerto específico.

Ejemplo 3.5: Compilar e instalar programa `sock`

```
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 3.6: 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 3.7: 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 (3.7) podemos ver el ejemplo realizado.

Si comprobamos la tabla de enrutamiento de la interfaz creada, `tap0`, podemos ver como aparece asociado el tráfico de la IP que le dimos a dicha interfaz, con que utilice la interfaz `tap0`. (ver figura [3.8])

```

rani@raniita ~ % sock -s 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 1025
Hola puerto 1025
Bienvenido puerto 1025
❏

rani@raniita ~ % sock 192.168.3.1 1026
Hola puerto 1026
Bienvenido puerto 1026
❏

```

Figura 3.7: Uso de la aplicación sock con interfaces tuntap.

```

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 3.8: 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, los kernels modernos los redirigen todos por loopback. En la figura 3.9 podemos ver un ejemplo de captura de un paquete enviado por el tap0.

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

▶ 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
 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 3.9: Captura de un paquete emitido por la interfaz tap0.

Capítulo 4

Espacios 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 uno (o varios) procesos. Constituyen la base tecnológica en la que se asientan las técnicas de virtualización más modernas (como puede ser Docker, Kubernetes, etc) y de ahí la importancia de conocer en profundidad su funcionamiento. 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 ¿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. [17]. Actualmente, tenemos 8 tipos diferentes de namespaces, siendo el último añadido en la versión 5.8 (lanzada el 2 de Agosto de 2020) [18].

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

4.3 ¿Cómo crear/acceder a un namespace?

Los namespaces deben asociarse a procesos. Dichos procesos pueden estar programados de antemano para ejecutarse en un namespace específico, utilizando la función *close()*. Sin embargo, lo habitual es utilizar los comandos *unshare* y *nsenter* para asignar un proceso cualquiera a un namespace.

- *unshare*. Permite asociar un proceso cualquiera a uno o varios namespaces. Si no se indica ningún proceso, utilizará el *bash* (o cualquier otro shell que tenga configurado el sistema) como proceso por defecto.
- *nsenter*. Permite añadir un proceso a un namespace existente.

4.4 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 variable 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 namespaces podría ser el siguiente [19]:

Ejemplo 4.1: 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 (proceso) en los namespaces que se especifiquen (uno o varios).
- Tenemos que especificar la ruta del ejecutable que queremos aislar. Si no se especifica, se asume *bash* como proceso por defecto.
- La sintaxis sería tal que: *unshare [options] [program] [<argument>...]*

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 4.2: 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
9 # Volvemos a entrar al namespace
10 nsenter --uts=/root/ns-uts /bin/bash
11 hostname # Nos devuelve 'FooBar'
12
13 # Salimos del namespace
14 exit
15
16 umount /root/ns-uts # Eliminamos el namespace definitivamente
17
```

Tal y como vemos en el ejemplo (4.2), utilizamos los comandos `unshare` y `nsenter` para manipular un namespace de tipo UTS (`hostname`). Lo importante es comprobar que si asociamos un namespace a un archivo, podemos recuperar dicho namespace si utilizamos el comando `nsenter`. Además, si quisiéramos eliminar permanentemente dicho namespace, tendríamos que hacer uso del comando `umount`.

4.5 Mount namespace

Este namespace fue el primero en aparecer en el Kernel de Linux (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). [20][21]

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 la llamada al sistema `clone()`, para crear un nuevo proceso asociado en el nuevo namespace; o bien utilizando el comando `unshare`, para mover un proceso dentro del nuevo namespace. Por defecto, cuando creamos un nuevo `mount namespace`, éste recibe una copia del punto de montaje inicial, replicado por el namespace que lo ha llamado. [21]

Un concepto importante es que los `mount namespaces` 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 propaguen 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. [22]

Por lo tanto, un namespace de tipo *mount* nos permite modificar un sistema de archivos en concreto, sin que otros namespaces puedan 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 4.3: Uso de `mount namespace` con dispositivo físico

```
1 $ sudo su                                # run a shell in a new mount namespace
2 $ unshare -m /bin/sh
3 $ mount /dev/sda2 /mnt/                  # hard drive
4 $ ls /mnt/cp
5 > /mnt/cp                               # File cp exists on hard drive
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 namespace lo que hacemos es realizar un *mount* de un disco duro, dentro de este encontramos un archivo `cp`. Sin embargo, una vez salimos del namespace, podemos comprobar como no podemos acceder al montaje que realizamos dentro del namespace. Por lo tanto, solo desde el namespace que realizó el montaje tendremos acceso a la información del disco duro.

¿Qué es un montaje tipo **bind**?

Para comprender correctamente como funciona un mount namespaces, es interesante entender que es exactamente un montaje tipo bind y sus diferencias respecto chroot o mount namespace. [23]

En primer lugar, es importante repasar el funcionamiento básico de mount, es decir, podemos montar un dispositivo en un directorio. Esto nos permite acceder al sistema de archivos de ese dispositivo. Por ejemplo, supongamos que queremos montar un disco duro, el cual representamos como /dev/sdb2, y que montaremos sobre el directorio /mnt/HDD1: [24]

```
$ mkdir /mnt/HDD1
$ mount /dev/sdb2 /mnt/HDD1
$ ls /mnt/HDD1
main.pdf images/ storage_shared/
```

Podemos ver como dentro del sistema de archivos del disco duro, encontramos un archivo y un par de carpetas. Si verificamos la lista de todos los dispositivos que tenemos montados, utilizando el comando findmnt --real, obtenemos la siguiente salida:

TARGET	SOURCE	FSTYPE	OPTIONS
/	/dev/sda2	ext4	rw,relatime
--/home	/dev/sdc1	ext4	rw,relatime
--/mnt/HDD1	/dev/sdb2	ext4	
rw,relatime,fmask=0022,dmask=0022,codepage=437,errors=remount-ro			
--/boot	/dev/sda1	vfat	
rw,relatime,fmask=0022,dmask=0022,codepage=437,errors=remount-ro			

Como podemos ver, en este momento hay cuatro dispositivos montados en nuestro sistema, incluido el disco duro (el que montamos en /mnt/HDD1). Cada dispositivo tiene asociado un directorio, por el cual podremos acceder a sus archivos. Procedemos a desmontar el disco duro:

```
$ umount /dev/HDD1
```

Sin embargo, también podemos realizar un montaje de un directorio sobre otro directorio, para ello utilizaremos el comando mount --bind. Podemos entender el montaje bind como un *alias*, por ejemplo, si realizamos un montaje bind del directorio /tmp/dir1 sobre /tmp/disk, ambos tendrán el mismo contenido. Podremos acceder a los archivos de /tmp/dir1 desde /tmp/disk, y viceversa.

Cualquier directorio puede ser utilizado en un montaje bind. Al igual que si el directorio origen es un dispositivo, el dispositivo formará un montaje bind en la ruta de destino. Además, es importante tener en cuenta que cuando utilizamos el parámetro bind, los puntos de montaje que estaban dentro de la carpeta origen no son remontados, por lo que si queremos que el directorio origen y todos sus montajes de dentro de ese directorio, se monten con el bind, tendremos que utilizar el parámetro --rbind (*recursive bind*). Por último, recalcar que después de realizar un montaje tipo bind, no tendremos acceso al contenido original del directorio destino.

A modo de ejemplo, realizamos los siguientes pasos:

```
$ mkdir /tmp/dir1
$ touch /tmp/dir1/bind_example
$ touch /tmp/dir1/mount_example
```

Creamos un directorio `/tmp/disk` y realizamos un montaje tipo `bind` sobre el:

```
$ mkdir /tmp/disk
$ mount --bind /tmp/dir1 /tmp/disk
```

Observamos el contenido de ambos directorios:

```
$ ls -l /tmp/dir1
total 40K
-rw-r--r-- 1 rani rani 0 Mar 10 18:46 bind_example
-rw-r--r-- 1 rani rani 0 Mar 10 18:47 mount_example
$ ls -l /tmp/disk
total 40K
-rw-r--r-- 1 rani rani 0 Mar 10 18:46 bind_example
-rw-r--r-- 1 rani rani 0 Mar 10 18:47 mount_example
```

Ambos directorios contienen los mismos archivos, podemos comprobarlo creando un archivo en uno de ellos:

```
$ echo "file" > /tmp/dir1/file.txt
$ cat /tmp/disk/file.txt
file
```

Al igual que hicimos en el ejemplo anterior, podemos utilizar el comando `findmnt --real` para verificar los montajes del sistema.

TARGET	SOURCE	FSTYPE	OPTIONS
/	/dev/sda2	ext4	rw,relatime
--/home	/dev/sdc1	ext4	rw,relatime
--/boot	/dev/sda1	vfat	
rw,relatime,fmask=0022,dmask=0022,codepage=437,errors=remount-ro			
--/tmp/disk	/dev/sda2[/tmp/dir1]	ext4	rw,relatime

Por último, desmontamos `/tmp/disk`

```
$ umount /tmp/disk
```

4.5.1 Casos de uso del montaje tipo **bind**

Ejemplo: acceder a archivos escondidos por un punto de montaje

Cuando montamos un dispositivo en un directorio, el contenido que había en el directorio destino antes del montaje se oculta por el punto de montaje. Por lo tanto, perdemos el acceso al contenido original. Sin embargo, tal y como vimos en la explicación anterior, si utilizamos `--bind`, el montaje no montará los puntos de montaje que estén asociados a ese directorio, por lo cual, podemos aprovechar esta funcionalidad para acceder a los archivos que están ocultos. A modo de ejemplo, supongamos que tenemos un directorio `/mnt/storage`, pero no hemos montado el disco duro todavía.

```
$ echo test_hidden > /mnt/storage/hidden_file
$ ls /mnt/storage
hidden_file
```

Montamos el disco duro en la ruta que hemos comentado:

```
$ mount /dev/sdb2 /mnt/storage
$ ls /mnt/storage
main.pdf images/ storage_shared/
```

Para acceder al archivo oculto, podemos utilizar un montaje tipo `bind`. Para ello, creamos una carpeta `/tmp/storage` y realizamos el montaje:

```
$ mkdir /tmp/storage
$ mount --bind /mnt /tmp/storage
```

Ahora, podemos comprobar como hemos recuperado el archivo oculto:

```
$ ls /tmp/storage/storage
$ cat /tmp/storage/hidden_file
test_hidden
```

Como podemos comprobar, se ha utilizado el directorio `/mnt` ya que son los *submounts* los que no se montan usando el `bind`. Si hubiéramos realizado el montaje `bind` sobre el directorio `/mnt/storage`, no hubiéramos podido acceder al archivo oculto. Para terminar, desmontamos los respectivos puntos de montaje:

```
$ umount /mnt/storage /tmp/storage
```

Ejemplo: acceder a archivos fuera de un entorno **chroot**

Otro ejemplo de uso de los montajes tipo `bind` es el de montar directorios dentro de un entorno `chroot` (un programa que permite un aislamiento limitado de procesos). Utilizando `chroot`, podemos elegir el directorio raíz en donde ejecutaremos nuestro programa. Por ejemplo, podemos utilizar `chroot` para ejecutar `httpd` en el directorio raíz `/home/apache`. Esto lo que hará es convertir `/home/apache` en el directorio raíz de ese programa, es decir, directorio `/`. Si quisiéramos acceder al directorio `/home/apache/www` sería equivalente a `/www`. Esto permite

a que el proceso `httpd` no pueda acceder a ningún archivo fuera del directorio `/home/apache`.

Sin embargo, si el proceso necesita acceder a archivos de fuera del `chroot`, por ejemplo, ciertas librerías del sistema, podemos utilizar un montaje tipo `bind` para hacer accesibles los directorios necesarios dentro del directorio al que hemos hecho `chroot`.

A modo de ejemplo, creamos el directorio `/tmp/program` sobre el que haremos un `chroot`. Después, le añadiremos ciertas carpetas del sistema.

```
$ mkdir /tmp/program
$ mkdir /tmp/program/bin
$ mkdir /tmp/program/lib64
```

Realizamos los montajes tipo `bind` antes de ejecutar el `chroot`.

```
$ mount --bind /bin /tmp/program/bin
$ mount --bind /lib64 /tmp/program/lib64
$ chroot /tmp/program
```

Con la terminal dentro del `chroot`, podemos comprobar como podemos acceder a los puntos de montaje que previamente habíamos configurado.

```
$ ls -l /
total 508K
drwxr-xr-x   6 root root 172K Mar  8 11:38 bin
drwxr-xr-x 305 root root 288K Mar  7 20:14 lib64
```

Como podemos comprobar, dentro del entorno `chroot` solo tenemos acceso a los directorios `/bin` y `/lib64`.

Ejemplo: convertir un directorio en solo lectura

Un montaje `bind` es utilizado para aplicar una modificación sobre los permisos, o opciones del montaje, de una parte del `tree` en específico. Esto lo podemos ver con un ejemplo: queremos que ciertas carpetas de nuestra aplicación tengan permisos de solo lectura, para ello podemos ejecutar los siguientes comandos:

```
$ mount --bind /mnt/MI_USB /home/user/MI_USB
$ mount -o remount,ro,bind /home/user/MI_USB
```

Ejemplo: montaje `bind` sobre un mismo directorio

Por otro lado, otro uso del montaje tipo `bind`, es la posibilidad realizar un montaje sobre un mismo directorio. Por ejemplo, lo realizamos sobre el directorio `/mnt/MI_USB`:

```
$ mount --bind /mnt/MI_USB /mnt/MI_USB
```

Al ejecutar dicho comando, lo que estamos haciendo es convertir un directorio del `tree` en un `device` o dispositivo de nuestro sistema, es decir, a partir de ahora (mientras esté montado) la ruta `/mnt/MI_USB` será tratado como un dispositivo montado sobre ese directorio. Esto mismo nos aporta cierta ventajas, ya que podemos aplicar las opciones que hemos comentado anteriormente, tales como modificar los permisos de escritura o lectura, o cambiar las opciones de montaje.

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 sólido. 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. [25]

Este tipo de archivos es comúnmente utilizado en los siguientes directorios: `/tmp`, `/var/lock`, `/var/run`, entre otros muchos. Desde el punto de vista de los namespaces, tiene la ventaja de que -si el sistema de archivo no es muy grande- permite crear un punto de montaje de forma fácil y rápida.

Un ejemplo de esto podría ser el siguiente: [26]

Ejemplo 4.4: 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
```

shared subtrees

Supongamos que un proceso quiere clonar su propio namespace, pero quiere mantener el acceso a un USB que previamente hemos montado en nuestro sistema. Para satisfacer esta situación, podemos utilizar la funcionalidad `shared subtrees`, ya que nos permite la propagación automática y controlada de eventos de montaje y desmontaje entre diferentes namespaces [27]

`shared subtrees` nos aporta cuatro maneras diferentes de realizar un montaje. Dichas opciones son las siguientes:

- **shared mount.** Un montaje de este tipo puede ser replicado en tantos puntos de montaje como se quieran y todas las réplicas seguirán siendo exactamente iguales. Es decir, una vez enlazados los directorios origen y destino a través de la opción `bind`, cualquier montaje nuevo que se haga en el directorio origen se reflejará en el directorio destino (y viceversa).
- **slave mount.** Un montaje de tipo *esclavo* funciona como un montaje compartido, menos para los eventos asociados al montaje y desmontaje, que solo se propagarán hacia él. Es decir, una vez enlazados los directorios origen y destino a través de la opción `bind`, cualquier montaje nuevo que se haga en el directorio origen se reflejará en el directorio destino pero no a la inversa.
- **private mount.** Un montaje de tipo *privado* no permite realizar ni recibir ningún tipo de propagación.
- **unbindeable mount.** Un montaje de tipo *privado* que no permite ser asociado/vinculable.

Si quisiéramos asignar alguno de estos tipos de montajes a nuestro dispositivo, tendríamos que hacer uso del comando `mount` con los siguientes parámetros de estado:

- **shared mount** → `mount --make-shared /mnt`
- **slave mount** → `mount --make-slave /mnt`
- **private mount** → `mount --make-private /mnt`
- **unbindeable mount** → `mount --make-unbindeable /mnt`

Por otro lado, si quisiéramos aplicar los cambios anteriores de manera recursiva para que se cambie el tipo de montaje para todos los montajes por debajo de la jerarquía (*recursivo*) del directorio que determinemos, tendríamos que utilizar los comandos siguientes: [28]

- **shared mount** recursivo → `mount --make-rshared /mnt`
- **slave mount** recursivo → `mount --make-rslave /mnt`
- **private mount** recursivo → `mount --make-rprivate /mnt`
- **unbindeable mount** recursivo → `mount --make-runbindeable /mnt`

Casos de uso de **shared subtree**

1. Supongamos un proceso que quiere crear un namespace, pero quiere mantener el acceso a un disco duro extraíble que previamente estaba montado en el sistema. [27]

Ejemplo 4.5: Caso de uso de **shared subtree**

```
1 # El administrador del sistema realiza el montaje del usb con
2 # permisos de 'shared'
3 # mount sobre mismo directorio: modificar opciones de montaje
4 $ mount --bind /mnt/usb /mnt/usb
5 $ mount --make-shared /mnt/usb
6
7 # Cualquier namespace que se cree tendra acceso a /mnt/usb
8 # Por lo tanto, cuando un USB se conecte y se monte, dicho
9 # montaje se propagara a todos los namespaces creados
10
```

2. Supongamos que un proceso quiere que su montaje sea invisible para otros procesos, pero que a la vez pueda ver el resto de montajes del sistema. [27]

Ejemplo 4.6: Caso de uso de **slave mount**

```
1 # El administrador permite que todo el sistema sea un subtree de tipo
   shared
2 $ mount --make-rshared /
3
4 # Un proceso puede clonar el sistema de archivos. En un momento,
5 # marca parte de sus archivos como subtree tipo slave
6 $ mount --make-rslave /mitreeprivado
7
8 # Por lo tanto, cualquier montaje realizado en /mitreeprivado no aparecera
   en
9 # el resto de namespaces. Sin embargo, los montajes que se realicen en
10 # ese directorio pero desde el namespaces default si que se programan
11 # al proceso asociado
12
```

3. Supongamos que queremos realizar un montaje en el que podamos “vincular” dos directorios, de modo que todo lo que hagamos en un directorio se refleje en el otro. [27]

Ejemplo 4.7: Caso de uso de **bind** y **shared subtree**

```
1 # El administrador permite que un directorio pueda ser replicado utilizando
   el tipo shared
2 $ mount --make-shared /mnt
3
4 # Asignamos un montaje tipo bind a dicho montaje, y lo dirigimos a otro
   directorio
5 $ mount --bind /mnt /tmp
6
7 # Los montajes /mnt y /tmp se compartiran mutuamente. Cualquier montaje
8 # o desmontaje que realicemos dentro de esos directorios se veran
9 # reflejados en el resto de montajes
10
```

4.6 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* representa a un número entero que utiliza el Kernel para identificar los procesos de manera unívoca. [29]

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. [30]

Ejemplo 4.8: 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, además de un tratamiento especial ya que supone un `init process` dentro de ese namespace [19].

4.7 Network namespace

Este namespaces nos permite aislar la parte de 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 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 4.9: 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 4.10: 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 4.11: 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 "levante" 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 4.12: Ejecutar cualquier programa con un network namespace

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

Ejemplo: topología de red usando **network namespaces** y 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, eso conllevaría que no podríamos tener conexión a Internet en nuestros namespaces. La solución para esto reside en los veth-pair.

Como ya hemos visto anteriormente, 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 4.13: 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 4.14: 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.

4.8 User ID namespace

Cada sistema dispone de una manera de monitorizar el 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.

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 4.15: 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. Por ejemplo, si a la vez que creamos un user namespace, también creamos un mount namespace, este mount namespace si que tendría privilegios de root [19]

4.9 Interprocess Communication namespace

Este namespace supone uno de los más técnicos, complicados de entender y explicar. *Inter-process communication* (IPC) 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.

4.10 Control groups namespace

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 [31]:

- **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 ésta 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 4.1: Archivos asociados al grupo `test_cg` una vez lo creamos

Ejemplo: limitar uso de memoria RAM

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 50 MB. Esto se haría tal que:

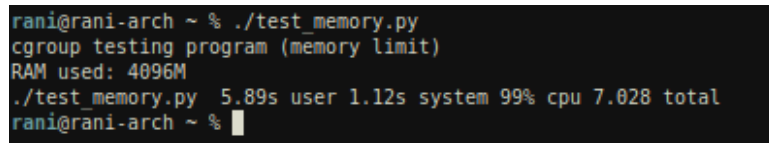
```
$ echo 50000000 > /sys/fs/cgroup/memory/<NombreGrupo>/  
memory.limit_in_bytes
```

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 4.16: 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:



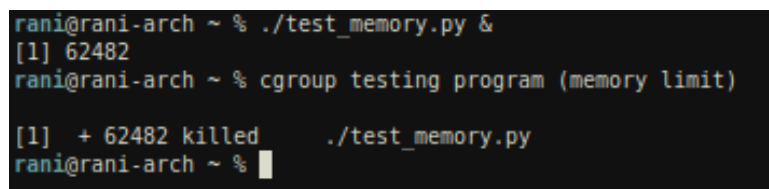
```
rani@rani-arch ~ % ./test_memory.py  
cgroup testing program (memory limit)  
RAM used: 4096M  
./test_memory.py 5.89s user 1.12s system 99% cpu 7.028 total  
rani@rani-arch ~ %
```

Figura 4.2: 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/<NombreGrupo>/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 4.3: Salida tras ejecutar el programa de Python, una vez limitado

Como podemos comprobar en las imágenes [4.2] y [4.3], la salida del programa no coincide. Esto es debido a que como el programa Python ha superado el límite establecido para su grupo, `cgroups` cerrado 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*”.

Ejemplo: limitar uso de CPU

Por otro lado, si quisiéramos limitar el uso de CPU de un programa, tendríamos que escribir en los archivos `cpu.cfs_quota_us` o `cpu.cfs_period_us`, dentro de la rama de `cpu`. Estos parámetros representan el tiempo de ejecución asociado a un proceso. En este ejemplo, vamos a limitar el proceso a 1 ms (1000 us), esto sería tal que:

```
$ echo 1000 > /sys/fs/cgroup/cpu/<NombreGrupo>/cpu.cfs_quota_us
```

Esto significa que cada 100ms de tiempo, el proceso está limitado por `cgroups` a utilizar solo 1ms del tiempo de CPU, dando lugar a utilizar solo el 1 % de la CPU.

Para comprobar el funcionamiento, podemos utilizar el siguiente *script* en *background* que nos pone un núcleo de la CPU al 100 %:

```
$ while : ; do : ; done &
```

Al ejecutar dicho comando, la tarea se quedará funcionando en *background*. Podemos comprobarlo si utilizamos herramientas para ver el consumo de los procesos, como pueden ser `top` o `htop`. Una vez tenemos el PID asociado a la tarea de prueba, podemos proceder a añadirlo a los procesos que son limitados con el `cgroup` creado, en este caso únicamente para limitar la CPU.

```
$ echo <PID> > /sys/fs/cgroup/cpu/<NombreGrupo>/tasks
```

Una vez ejecutado, si volvemos a ejecutar `top` o `htop`, podemos ver como el tiempo asignado al «recurso» será mucho mejor, y estrictamente igual o inferior al límite que establecimos en la creación de la regla para nuestro `cgroup`.

Ejemplo: monitorización de un «recurso»

A modo de comentario, otra de las tareas importante que desempeña `cgroups` es la monitorización del consumo de «recursos» de ciertas aplicaciones. Por ejemplo, para el caso del *runtime* de Docker (hablaremos más en profundidad en [5.3]) para ejecutar contenedores basados en namespaces, utiliza `cgroups` para asignar el consumo (CPU, RAM, uso de disco, red...) que tiene un contenedor en concreto. De este modo, podemos discernir y aplicar limitaciones a un contenedor en específico, sin tener que afectar a otros contenedores que estén funcionando en ese momento en el host.

Para estas tareas, limitación y monitorización de «recursos» en contenedores de Docker, éste nos facilita una serie de comandos para facilitar la creación y asignación de contenedores a un `cgroup` en específico, facilitando en consecuencia el uso de `cgroups` para limitar los «recursos» de los contenedores que estemos utilizando.

4.11 Time namespace

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. [32]

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. [33]

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.

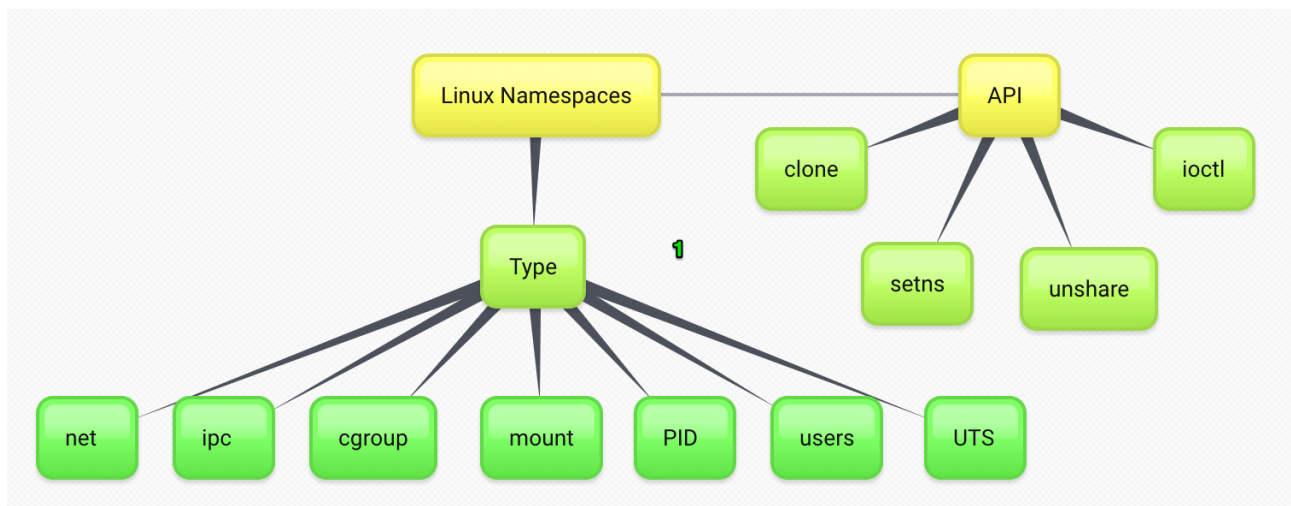


Figura 4.4: Diferentes namespaces en Linux y su API de acceso.

4.12 Ejemplo: topología de red 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 network namespaces 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 automáticamente 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
```

4.13 Ejemplo: topología de red usando comando `unshare`

En el ejemplo anterior, utilizábamos el comando `ip` para manejar los network namespaces, sin embargo, eso nos limitaba los tipos namespaces que queríamos 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, en la mayoría de los casos tendremos que utilizar los comandos `unshare` y `nsenter`.

Capítulo 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 cómo supone una de las piezas más importantes para la virtualización tal y como hoy en 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 capítulo 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 [34]. 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. [35]

5.1 Creando nuestro propio “contenedor”

En este apartado vamos a comentar las diferentes fases que deberíamos seguir si quisiéramos implementar nuestro propio contenedor. En este caso, las fases son muy similares a las que utiliza la tecnología de Docker, la cual comentaremos más en detalle en los apartados siguientes.

Lo primero, es interesante comentar que aunque nos referimos a contenedores, muchas personas entienden contenedores como únicamente los `Docker Containers`. Sin embargo existen otros runtimes para ejecutar contenedores en el host. En general, todos implementan las especificaciones realizadas por *Linux Foundation’s Container Initiative* (OCI) [36], esta fundación recoge directivas tanto para la creación de imágenes de contenedores como de sus runtimes. Por lo tanto, existen diferentes runtimes en función de la solución que queramos realizar, y cada runtime trabaja a un nivel de profundidad diferente. Por ejemplo, los contenedores `LXC` [37], que entraremos en detalle en el siguiente apartado, se entienden como contenedores de bajo nivel, mientras que un ejemplo de contenedores de alto nivel podría ser `containerd` [38].

Imágenes

Una imagen de un contenedor consiste en un sistema de archivos root, que nos aporta todas las dependencias necesarias por la aplicación a encapsular en un contenedor. Dichas imágenes consisten en capas, que se montan por el runtime utilizando un tipo de montaje de unión (manera de combinar múltiples directorios en uno único y que este contenga los contenidos de ambos directorios combinados) [39], normalmente se utiliza `overlayfs` [40]. Esto nos permite dividir nuestro sistema de archivos en capas, siendo el sistema de archivos más bajo inmutable (los directorios referidos al sistema y que no tienen nada que ver con la aplicación a encapsular), y otra capa superior en la que encontraremos todo lo referido a nuestra aplicación y que sí podremos modificar.

A modo de ejemplo, comentamos el siguiente script de bash creado por el usuario [Github: Nuvalence](https://github.com/Nuvalence), en donde detalla los diferentes pasos para implementar un contenedor utilizando namespaces, y además utilizando montajes de unión.

Ejemplo 5.1: Creación de imagen para contenedor utilizando Alpine y union mounts

```
1 #!/bin/bash
2
3 # Copyright 2020 Nuvalence <https://github.com/Nuvalence/diy-container.git>
4 #
5 # Licensed under the Apache License, Version 2.0 (the "License");
6 # you may not use this file except in compliance with the License.
7 # You may obtain a copy of the License at
```

```
8 # http://www.apache.org/licenses/LICENSE-2.0
9 # https://github.com/Nuvalence/diy-container/blob/master/LICENSE
10 #
11 # Modified on Jan 2022 by: Enrique Ranii <https://github.com/Raniita/container-
    alpine.git>
12
13 # Deploy a image of Alpine Linux inside a "container". Using rootfs as Union
    Filesystem
14 # Installing apache2 for httpd test, and check on host that network link is
    enabled
15 # Usage:
16 #   - sudo ./setup-image
17 # Enable verbose output using:
18 #   - sudo VERBOSE=1 ./setup-image
19
20 set -e
21
22 [[ -n "$VERBOSE" ]] && set -x
23
24 ROOTFS_URL="http://dl-cdn.alpinelinux.org/alpine/v3.15/releases/x86_64/alpine-
    minirootfs-3.15.0-x86_64.tar.gz"
25
26 # Remove existing layers
27 rm -rf layers *.tar.gz
28
29 #
30 # Layer 1 setup {immutable}
31 #
32
33 echo "*** Configuring layer 1..."
34 mkdir -p layers/1
35 wget "$ROOTFS_URL"
36 tar -zxvf *.tar.gz -C layers/1
37 rm -rf *.tar.gz
38
39 # Assign permissions to devices on layer1
40 mknod -m 622 layers/1/dev/console c 5 1
41 mknod -m 666 layers/1/dev/null c 1 3
42 mknod -m 666 layers/1/dev/zero c 1 5
43 mknod -m 666 layers/1/dev/ptmx c 5 2
44 mknod -m 666 layers/1/dev/tty c 5 0
45 mknod -m 444 layers/1/dev/random c 1 8
46 mknod -m 444 layers/1/dev/urandom c 1 9
47 chown -v root:tty layers/1/dev/{console,ptmx,tty}
48
49 #
50 # Layer 2 setup {application layer}
51 #
52
53 echo "*** Configuring layer 2..."
54 mkdir -p layers/2/etc
55 echo "nameserver 8.8.8.8" >> layers/2/etc/resolv.conf
56
57 echo "*** Instal and configure apache2 {for httpd test}..."
58 LOWER_DIR=layers/1 UPPER_DIR=layers/2 ./runtime-exec /bin/sh -c "apk update &&
    apk add apache2"
59 echo "ServerName localhost" >> layers/2/etc/apache2/httpd.conf
60
```

En dicho script identificamos las siguientes fases:

1. Descarga del `rootfs` de la distribución de Linux Alpine.
2. Eliminar instalaciones previas de la imagen a crear.
3. Extraer y modificar los permisos de lo que será la capa 1 inmutable de nuestro contenedor.
4. Configurar servidor DNS de dentro del contenedor e instalar dependencias del programa a ejecutar, en nuestro caso haremos las pruebas con el programa `httpd`

Runtime

Una vez tenemos la imagen creada, el siguiente paso es el `runtime` encargado de ejecutarla. Para ello, creamos un “contenedor” utilizando los siguientes namespaces:

1. Network. Utilizado para dotar de acceso a internet y acceso al host al contenedor.
2. PID. Asignar un nuevo PID y GID al contenedor.
3. IPC. Aislar ciertos «recursos» del host al contenedor.
4. UTS. Utilizado para asignar un hostname específico al contenedor
5. cgroups. Utilizado para la gestión de límites de «recursos» y monitorizar el contenedor.
6. mount. Sistema de archivos del contenedor.

Al igual que en el caso anterior, comentamos el siguiente script de ejemplo creado por el usuario [Github: Nuvalence](#), en donde se nos detallan los diferentes pasos a realizar para tener el `runtime` de nuestro contenedor ejecutando la imagen que previamente hemos creado.

Ejemplo 5.2: Ejecución de runtime basado en imagen de Alpine

```
1 #!/bin/bash
2
3 # Copyright 2020 Nuvalence <https://github.com/Nuvalence/diy-container.git>
4 #
5 # Licensed under the Apache License, Version 2.0 (the "License");
6 # you may not use this file except in compliance with the License.
7 # You may obtain a copy of the License at
8 #   http://www.apache.org/licenses/LICENSE-2.0
9 #   https://github.com/Nuvalence/diy-container/blob/master/LICENSE
10 #
11 # Modified on Jan 2022 by: Enrique Ranii <https://github.com/Raniita/container-
    alpine.git>
12
13 # Requirements: Must run before -> sudo ./setup-image
14 # Deploy a container that executes a application inside apline
15 # Usage:
16 #   - sudo ./runtime-exec <command>
17 # Example:
18 #   - sudo ./runtime-exec httpd -D FOREGROUND
19 # Enable verbose output using:
20 #   - sudo VERBOSE=1 ./runtime-exec <command>
21
22 set -e
```

```
23
24 [[ -n "$VERBOSE" ]] && set -x
25
26 # Container IP
27 CONTAINER_SUBNET_CIDR=10.178.61.1/24
28 CONTAINER_IP=10.178.61.2
29
30 #
31 # Host network setup
32 #
33
34 echo "*** Configure Host network setup {iptables, bridge, network namespace}..."
35
36 # Enable packet forwarding
37 IP_FORWARD=$(cat /proc/sys/net/ipv4/ip_forward)
38 echo 1 > /proc/sys/net/ipv4/ip_forward
39
40 # Allow forwarding of packets to and from the container subnet
41 iptables -A FORWARD -s $CONTAINER_SUBNET_CIDR -j ACCEPT
42 iptables -A FORWARD -d $CONTAINER_SUBNET_CIDR -m conntrack --ctstate RELATED,
    ESTABLISHED -j ACCEPT
43
44 # Re-write the source address for packets originating from the container subnet
45 iptables -t nat -A POSTROUTING -s $CONTAINER_SUBNET_CIDR -j MASQUERADE
46
47 # Create a bridge device to act as a gateway for the container subnet
48 ip link add dev br-container type bridge
49
50 # Set the bridges's IP
51 ip addr add $CONTAINER_SUBNET_CIDR dev br-container
52
53 # Create two peered virtual ethernet adapters
54 ip link add dev veth-host type veth peer name veth-container
55
56 # Assign the host's virtual adapter to the bridge
57 ip link set dev veth-host master br-container
58
59 # Bring up devices
60 ip link set dev veth-host up
61 ip link set dev br-container up
62
63 #
64 # Network namespace setup
65 #
66
67 # Create a network namespace
68 ip netns add container
69
70 # Assign the container's virtual adapter to the namespace
71 ip link set dev veth-container netns container
72
73 # Set the container's IP
74 ip netns exec container \
75 ip addr add dev veth-container $CONTAINER_IP/$(basename $CONTAINER_SUBNET_CIDR)
76
77 # Bring up the container's device
78 ip netns exec container \
79 ip link set dev veth-container up
```



```
80
81 # Set the container's default route
82 ip netns exec container \
83 ip route add default via $(dirname $CONTAINER_SUBNET_CIDR)
84
85 #
86 # Container filesystem setup
87 #
88
89 echo "*** Mount container filesystem using rootfs..."
90
91 # Create the upper, work, and rootfs directories
92 mkdir upperdir workdir rootfs
93
94 # Set directory defaults
95 [[ -z "$LOWER_DIR" ]] && LOWER_DIR=layers/2:layers/1
96 [[ -z "$UPPER_DIR" ]] && UPPER_DIR=upperdir
97
98 # Create an overlay filesystem
99 mount -t overlay overlay -o lowerdir=$LOWER_DIR,upperdir=$UPPER_DIR,workdir=
   workdir rootfs
100
101 # Setup /sys
102 mount --bind /sys rootfs/sys
103
104 #
105 # Define cleanup routine
106 #
107
108 function cleanup()
109 {
110     echo "** Revert all configuration and shutdown..."
111
112     # Revert host network setup
113     ip link del veth-host
114     ip link del br-container
115     iptables -t nat -D POSTROUTING -s $CONTAINER_SUBNET_CIDR -j MASQUERADE
116     iptables -D FORWARD -d $CONTAINER_SUBNET_CIDR -m conntrack --ctstate RELATED,
   ESTABLISHED -j ACCEPT
117     iptables -D FORWARD -s $CONTAINER_SUBNET_CIDR -j ACCEPT
118     echo $IP_FORWARD > /proc/sys/net/ipv4/ip_forward
119
120     # Cleanup the network namespace
121     ip netns del container
122
123     # Cleanup the container filesystem
124     umount rootfs/sys rootfs
125     rm -rf upperdir workdir rootfs
126
127     echo "*** Done."
128 }
129
130 # Call cleanup on exit
131 trap cleanup EXIT
132
133 #
134 # Launch the container
135 #
```

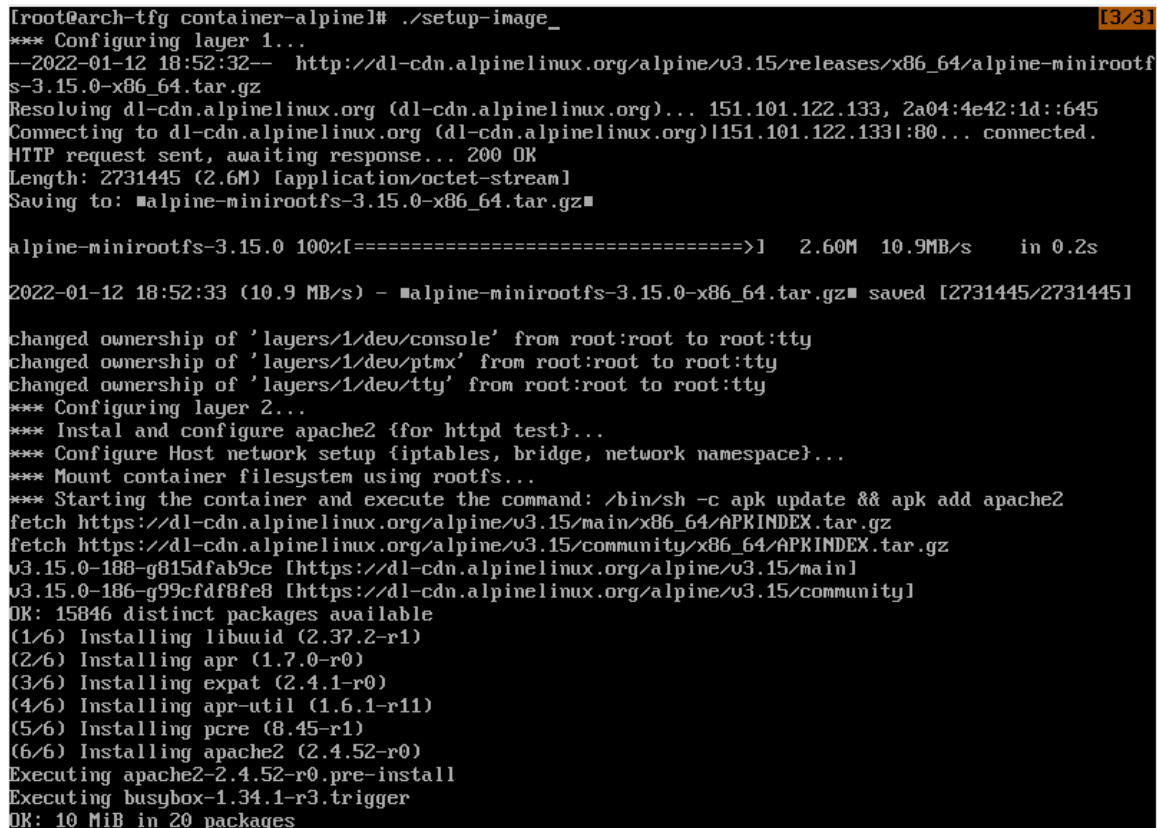
```
136
137 echo "*** Starting the container and execute the command: $@"
138 # Enter namespaces and exec the provided command
139 ip netns exec container \
140 unshare --pid --ipc --uts --cgroup --mount --root=rootfs --mount-proc --fork "$@"
141
```

Para comprender mejor el código, vamos a detallar las siguientes fases por las que pasa la ejecución del runtime.

1. Configuración de la red. En el host se permite la redirección de paquetes y se activa la NAT para el contenedor
2. Se crea una interfaz virtual tipo bridge, se le asigna una dirección IP
3. Creamos una pareja de ethernet virtuales, uno será asignado al host y la otra al contenedor.
4. Se crea el network namespace utilizando el comando `ip netns add`, se le asigna el ethernet virtual y se le da una dirección IP. Además, configura la ruta *default* de los paquetes IP.
5. Se realiza el montaje del sistema de archivos, diferenciando en dos capas. Una capa, llamada capa inferior, en la que se almacena todo lo necesario de la distribución a ejecutar y que será inmutable para la aplicación a encapsular, y otra capa en la que encontramos los directorios que si tendrá acceso dicha aplicación. En este caso, se utiliza un montaje de tipo *overlay*, a modo de ejemplo de los montajes de unión.
6. Se realiza un montaje de tipo *bind* para todos los «recursos» del sistema, que se encuentran en el directorio `/sys`.
7. Se define una función para revertir todos los cambios realizados al sistema.
8. Por último, se ejecuta el contenedor con el comando `unshare` y especificando los namespaces que vamos a utilizar, además le ponemos como punto de entrada al `unshare` la aplicación que el usuario haya especificado (por ejemplo: `sh` o `httpd`).

Ejemplo: ejecución de un contenedor basado en imagen Linux Alpine

1. Primero ejecutamos el script llamado `setup-image` para configurar la imagen de nuestro contenedor, para ello ejecutamos dicho script con permisos de root.



```
[root@arch-tfg container-alpine]# ./setup-image_
*** Configuring layer 1...
--2022-01-12 18:52:32-- http://dl-cdn.alpinelinux.org/alpine/v3.15/releases/x86_64/alpine-minirootfs-3.15.0-x86_64.tar.gz
Resolving dl-cdn.alpinelinux.org (dl-cdn.alpinelinux.org)... 151.101.122.133, 2a04:4e42:1d::645
Connecting to dl-cdn.alpinelinux.org (dl-cdn.alpinelinux.org)|151.101.122.133|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2731445 (2.6M) [application/octet-stream]
Saving to: alpine-minirootfs-3.15.0-x86_64.tar.gz

alpine-minirootfs-3.15.0 100%[=====>] 2.60M 10.9MB/s in 0.2s

2022-01-12 18:52:33 (10.9 MB/s) - alpine-minirootfs-3.15.0-x86_64.tar.gz saved [2731445/2731445]

changed ownership of 'layers/1/dev/console' from root:root to root:tty
changed ownership of 'layers/1/dev/ptmx' from root:root to root:tty
changed ownership of 'layers/1/dev/tty' from root:root to root:tty
*** Configuring layer 2...
*** Instal and configure apache2 {for httpd test}...
*** Configure Host network setup {iptables, bridge, network namespace}...
*** Mount container filesystem using rootfs...
*** Starting the container and execute the command: /bin/sh -c apk update && apk add apache2
fetch https://dl-cdn.alpinelinux.org/alpine/v3.15/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.15/community/x86_64/APKINDEX.tar.gz
v3.15.0-188-g815dfab9ce [https://dl-cdn.alpinelinux.org/alpine/v3.15/main]
v3.15.0-186-g99cfd8fe8 [https://dl-cdn.alpinelinux.org/alpine/v3.15/community]
OK: 15846 distinct packages available
(1/6) Installing libuuid (2.37.2-r1)
(2/6) Installing apr (1.7.0-r0)
(3/6) Installing expat (2.4.1-r0)
(4/6) Installing apr-util (1.6.1-r11)
(5/6) Installing pcre (8.45-r1)
(6/6) Installing apache2 (2.4.52-r0)
Executing apache2-2.4.52-r0.pre-install
Executing busybox-1.34.1-r3.trigger
OK: 10 MiB in 20 packages
```

Figura 5.1: Ejecución del script `setup-image` (1/2)

```

changed ownership of 'layers/1/dev/console' from root:root to root:tty
changed ownership of 'layers/1/dev/ptmx' from root:root to root:tty
changed ownership of 'layers/1/dev/tty' from root:root to root:tty
*** Configuring layer 2...
*** Instal and configure apache2 {for httpd test}...
*** Configure Host network setup {iptables, bridge, network namespace}...
*** Mount container filesystem using rootfs...
*** Starting the container and execute the command: /bin/sh -c apk update && apk add apache2
fetch https://dl-cdn.alpinelinux.org/alpine/v3.15/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.15/community/x86_64/APKINDEX.tar.gz
v3.15.0-188-g815dfab9ce [https://dl-cdn.alpinelinux.org/alpine/v3.15/main]
v3.15.0-186-g99cfd8fe8 [https://dl-cdn.alpinelinux.org/alpine/v3.15/community]
OK: 15846 distinct packages available
(1/6) Installing libuuid (2.37.2-r1)
(2/6) Installing apr (1.7.0-r0)
(3/6) Installing expat (2.4.1-r0)
(4/6) Installing apr-util (1.6.1-r11)
(5/6) Installing pcre (8.45-r1)
(6/6) Installing apache2 (2.4.52-r0)
Executing apache2-2.4.52-r0.pre-install
Executing busybox-1.34.1-r3.trigger
OK: 10 MiB in 20 packages
** Revert all configuration and shutdown...
*** Done.
[root@arch-tfg container-alpine]#

```

Figura 5.2: Ejecución del script setup-image (2/2)

- Una vez tenemos la imagen creada, ya podemos ejecutar el runtime con el comando que nosotros queramos. Para ello, tendremos que ejecutar el comando `./runtime-exec <commando>`

```

[root@arch-tfg container-alpine]# ./runtime-exec sh
*** Configure Host network setup {iptables, bridge, network namespace}...
*** Mount container filesystem using rootfs...
*** Starting the container and execute the command: sh
/ #

```

Figura 5.3: Ejecución del script runtime-exec (1/2)

Dado que ya estamos dentro de una shell del contenedor, podemos comprobar como solo podemos ver los procesos del contenedor, y solo tenemos la interfaz de red asignada al contenedor.

- Ejecutamos el comando `httpd` y comprobamos que el host puede acceder al servicio.

Como podemos comprobar en la siguiente imagen, el comando `httpd` se ejecuta correctamente en el contenedor. Por otro lado, utilizamos la herramienta `wget` para comprobar que tenemos acceso al servicio web que expone el comando `httpd`, el contenido del archivo `index.html` lo podemos ver también en las siguientes figuras.

```
[root@arch-tfg container-alpine]# ./runtime-exec sh
*** Configure Host network setup {iptables, bridge, network namespace}...
*** Mount container filesystem using rootfs...
*** Starting the container and execute the command: sh
/ # ps
PID   USER     TIME  COMMAND
  1  root      0:00  sh
  2  root      0:00  ps
/ # ifconfig
veth-container Link encap:Ethernet HWaddr 52:D4:30:94:B1:CC
    inet addr:10.178.61.2 Bcast:0.0.0.0 Mask:255.255.255.0
    inet6 addr: fe80::50d4:30ff:fe94:b1cc/64 Scope:Link
    UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
    RX packets:27 errors:0 dropped:0 overruns:0 frame:0
    TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:1000
    RX bytes:2222 (2.1 KiB) TX bytes:656 (656.0 B)

/ #
```

Figura 5.4: Ejecución del script runtime-exec (2/2)

```
[root@arch-tfg container-alpine]# ./runtime-exec httpd -D FOREGROUND
*** Configure Host network setup {iptables, bridge, network namespace}...
*** Mount container filesystem using rootfs...
*** Starting the container and execute the command: httpd -D FOREGROUND

[... output omitted ...]

[root@arch-tfg container-alpine]# wget http://10.178.61.2
--2022-01-12 19:08:45-- http://10.178.61.2/
Connecting to 10.178.61.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 45 [text/html]
Saving to: 'index.html.1'

index.html.1          100%[=====]          45  --.-KB/s    in 0s
2022-01-12 19:08:45 (8.29 MB/s) - 'index.html.1' saved [45/45]

[root@arch-tfg container-alpine]#
```

Figura 5.5: Ejecución del script runtime-exec con comando httpd

```
[root@arch-tfg container-alpine]# cat index.html
<html><body><h1>It works!</h1></body></html>
[root@arch-tfg container-alpine]# _
```

Figura 5.6: Ejecución del script runtime-exec con comando httpd

Por lo tanto, podemos concluir que este ejemplo es válido para demostrar el funcionamiento de un contenedor, basado en namespaces y en sistemas de montaje de capas.

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. [37]. 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...)

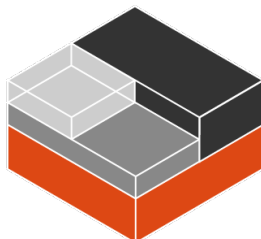


Figura 5.7: Logotipo del proyecto linuxcontainers.org

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. [37]

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 de 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 [37], primero tendríamos que crear el archivo `/etc/default/lxc-net` con el siguiente contenido:

Ejemplo 5.3: 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=maill,10.0.3.100' to have
21 # container 'maill' 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 5.4: 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 [41]:

Ejemplo 5.5: 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 5.6: 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 [41]:

- `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) [42].

Ejemplo 5.7: Configuración interfaz NAT bridge y aplicaciones X a un contenedor LXC

```
1 # Network configuration (NAT bridge)
2 lxc.net.0.type = veth
3 lxc.net.0.veth.pair = lxcbr0
4 lxc.net.0.flags = up
5
6 ## for xorg
7 lxc.mount.entry = /dev/dri dev/dri none bind,optional,create=dir
8 lxc.mount.entry = /dev/snd dev/snd none bind,optional,create=dir
9 lxc.mount.entry = /tmp/.X11-unix tmp/.X11-unix none bind,optional,create=dir,ro
10 lxc.mount.entry = /dev/video0 dev/video0 none bind,optional,create=file
11
```

Para iniciar la interfaz gráfica, una vez tengamos iniciado el contenedor y estemos dentro de una shell, ejecutamos el comando `startx`.

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 [43]. 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 aparatado 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 5.8: 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
```

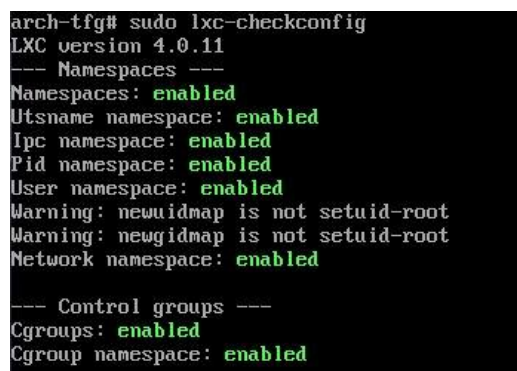
Una vez realizados todos estos pasos, ya podríamos crear nuestro contenedor sin privilegios. Lo haríamos de manera similar que en el apartado anterior, es decir utilizando el comando `lxc-create`. Además, también será importante modificar el archivo `config` de nuestro contenedor para asignar la interfaz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

Ejemplo: namespaces en un contenedor LXC de Ubuntu

A modo de ejemplo, vamos a crear un contenedor (con privilegios) paso a paso, y por último, vamos a verificar que namespaces está utilizando dicho contenedor. Para esto, vamos a crear un contenedor de Ubuntu 20.04.03 (Focal) utilizando el catálogo de imágenes de LXC.

1. Lo primero que tenemos que hacer es verificar que LXC está instalado correctamente en nuestro sistema, una manera fácil de comprobarlo es con el siguiente comando:

```
$ sudo lxc-checkconfig
```



```
arch-tfg# sudo lxc-checkconfig
LXC version 4.0.11
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: enabled
Warning: newuidmap is not setuid-root
Warning: newgidmap is not setuid-root
Network namespace: enabled

--- Control groups ---
Cgroups: enabled
Cgroup namespace: enabled
```

Figura 5.8: Comprobación instalación de LXC usando `lxc-checkconfig`

2. Creamos el contenedor de Ubuntu siguiendo los pasos de la creación guiada de contenedores que hemos comentado en apartados anteriores.

```
$ sudo lxc-create --template download --name <NombreContenedor>
```

```
arch-tfg# sudo lxc-create --template download --name lxc-contenedor
Setting up the GPG keyring
Downloading the image index

----
DIST      RELEASE ARCH      VARIANT BUILD
----
almalinux      8      amd64   default 20220119_23:08
almalinux      8      arm64   default 20220119_23:08
alpine 3.12      amd64   default 20220119_13:00
alpine 3.12      arm64   default 20220119_13:00
alpine 3.12      armhf   default 20220119_13:00
alpine 3.12      i386    default 20220119_13:00
alpine 3.12      ppc64el default 20220119_13:00
alpine 3.12      s390x   default 20220119_13:00
alpine 3.13      amd64   default 20220119_13:00
alpine 3.13      arm64   default 20220119_13:00
```

Figura 5.9: Instalación de contenedor LXC (1/2)

De manera interactiva, el programa nos preguntará por diferentes opciones para nuestro contenedor. Para elegir una distribución, podemos acceder a la siguiente web: [imágenes LXC](#), en las que aparecen todas las imágenes disponibles. En este caso, escribimos lo siguiente para cada una de ellas:

- *Distribution*: ubuntu
- *Release*: focal
- *Architecture*: amd64

```
Distribution:
ubuntu
Release:
focal
Architecture:
amd64

Using image from local cache
Unpacking the rootfs

----
You just created an Ubuntu focal amd64 (20220113_07:42) container.

To enable SSH, run: apt install openssh-server
No default root or user password are set by LXC.
arch-tfg#
```

Figura 5.10: Instalación de contenedor LXC (2/2)

3. Iniciamos el contenedor que acabamos de crear.

```
$ sudo lxc-start -n <NombreContenedor> -d
```

4. Accedemos a la terminal del contenedor que acabamos de iniciar.

```
$ sudo lxc-attach -n <NombreContenedor>
```

5. Una vez dentro del contenedor de Ubuntu, podemos comprobar con el siguiente comando que efectivamente se ha instalado la distribución que hemos elegido.

```
$ cat /etc/os-release
```

```
arch-tfg# sudo lxc-attach -n lxc-contenedor
# 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
#
```

Figura 5.11: Comprobación de la versión de Ubuntu instalada en el contenedor LXC

6. Por último, en otra terminal, procedemos a obtener el PID asociado a dicho contenedor. Para ello, podemos hacer uso del siguiente comando:

```
$ sudo lxc-info -p -n <NombreContenedor>
```

Una vez tenemos el PID, podemos utilizar el comando `lsns` los namespaces que están asociados a dicho PID. Para ello, ejecutamos lo siguiente:

```
$ sudo lsns | grep <PID_contenedor>
```

```
arch-tfg# sudo lxc-info -p -n lxc-contenedor
PID:          537
arch-tfg#
arch-tfg# sudo lsns | grep 537
4026532246 mnt      11    537 root      /sbin/init
4026532247 uts      10    537 root      /sbin/init
4026532248 ipc      11    537 root      /sbin/init
4026532249 pid      11    537 root      /sbin/init
4026532250 cgroup   11    537 root      /sbin/init
4026532252 net      11    537 root      /sbin/init
arch-tfg#
```

Figura 5.12: Namespaces asociados a un contenedor LXC ejecutado en el host

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 fácilmente un entorno de contenedores.

Tal y como lo definen sus creadores [44], `Docker` es una plataforma de desarrollo, despliegue y ejecución de aplicaciones. `Docker` permite separar entre aplicaciones de la 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.
- 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 entre usuarios.
- Equivalencia entre despliegue local y despliegue en producción.

Arquitectura de Docker

`Docker` utiliza una arquitectura 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)[44].

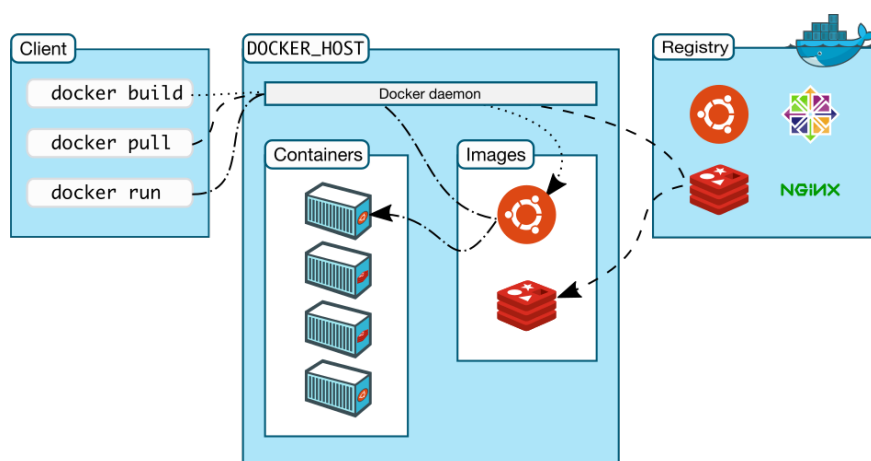


Figura 5.13: Arquitectura Docker

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. [44]



Figura 5.14: Logotipo de Docker

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 (5.15), tendremos que ejecutar el comando:

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

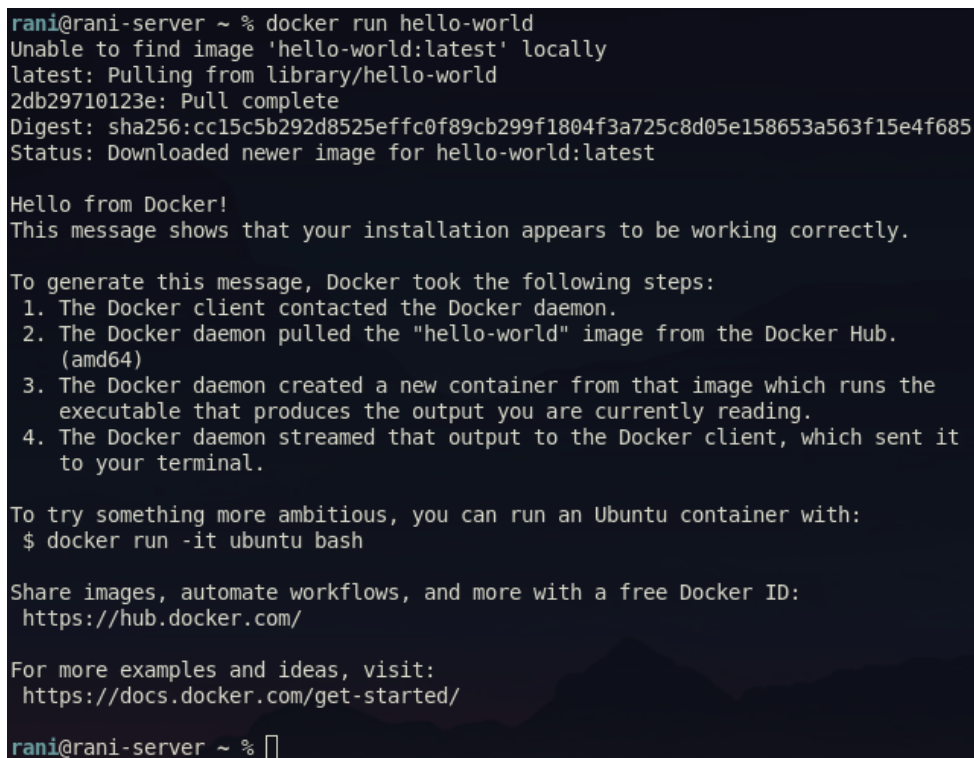
```
rani@raniita ~ % docker version
Client:
 Version:           20.10.10
 API version:       1.41
 Go version:        go1.17.2
 Git commit:        b485636f4b
 Built:             Tue Oct 26 03:44:01 2021
 OS/Arch:           linux/amd64
 Context:           default
 Experimental:      true
Cannot connect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

Figura 5.15: 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 5.16: Ejecución comando `docker run hello-world` para comprobar instalación

Como podemos ver en la figura 5.16, 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 5.16. 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 5.17 y 5.18).


```
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 5.17: 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 5.18: Comprobación de la versión de ubuntu del contenedor desplegado

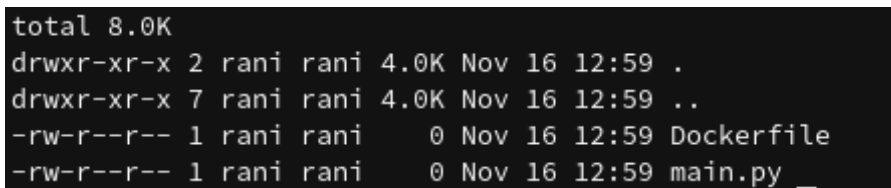
Las imágenes que utilizamos, serán almacenadas en el directorio `/var/lib/docker`. Por lo tanto, toda la configuración asociada a un contenedor Docker (configuración de red, volúmenes, imágenes, etc) la podremos encontrar en dicho directorio. Las imágenes que tengamos disponibles para utilizar en el hosts, están almacenadas en `/var/lib/docker/overlay2`

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 [45]. 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 así (ver Figura [5.19]):



```
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 5.19: 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 5.9: Código Python de ejemplo para crear un `Dockerfile`

```
1 #!/usr/bin/env python3
2
3 print("Saludos desde tu contendor!!")
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 5.20).

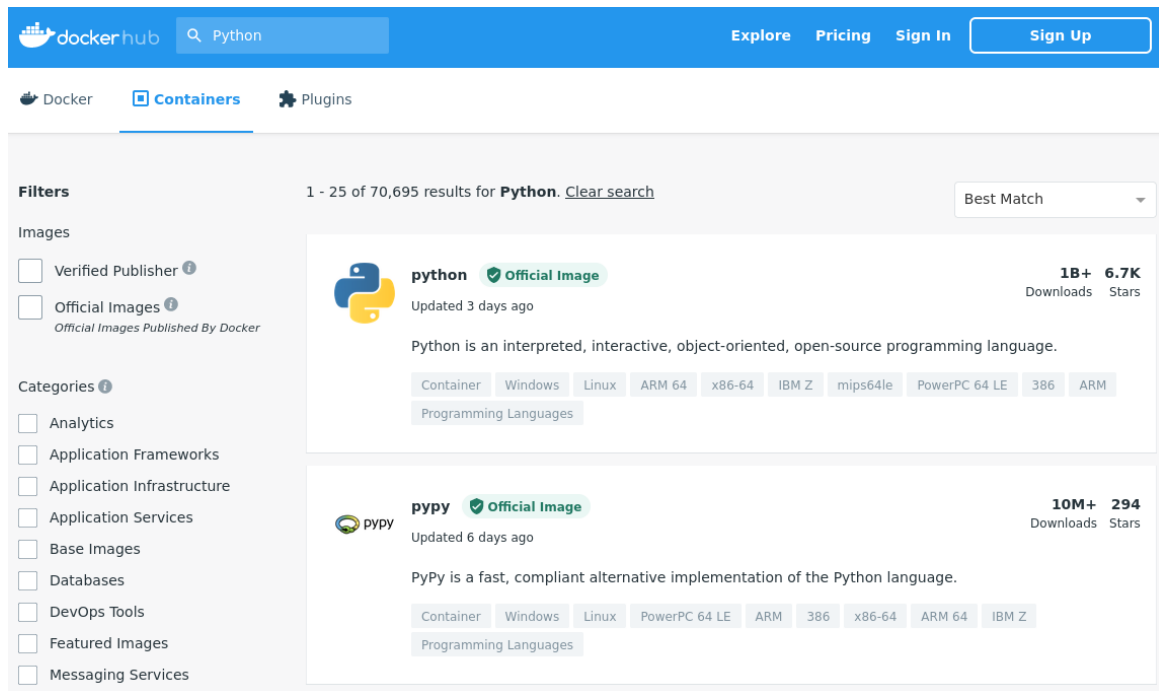


Figura 5.20: Búsqueda en DockerHub de una imagen base para nuestro contenedor.

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 la 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`:

Ejemplo 5.10: 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 5.21). Para ello, tenemos que ejecutar el siguiente comando:

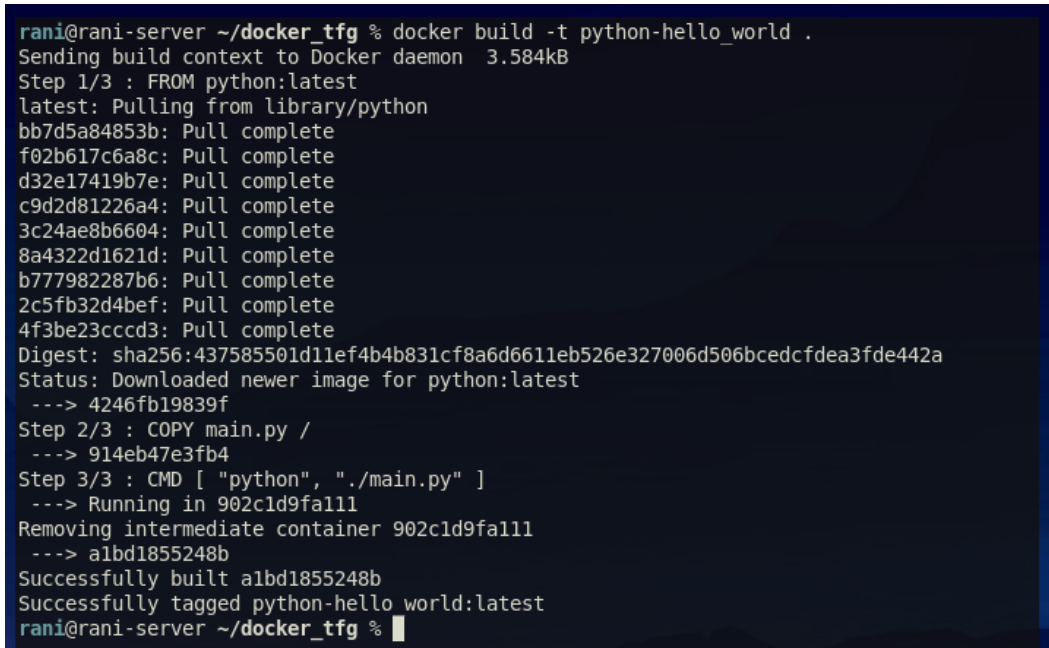
```
$ docker build -t python-hello_world .
```

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:

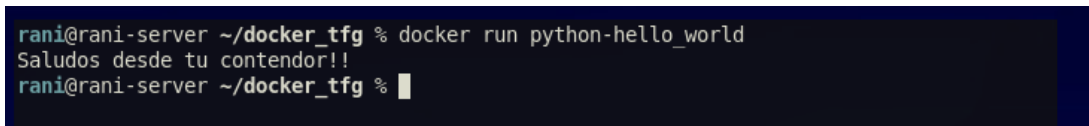
```
$ docker run python-hello_world
```

En la terminal deberíamos comprobar el código se ha ejecutado correctamente (ver Figura 5.22).



```
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
--> a1bd1855248b
Successfully built a1bd1855248b
Successfully tagged python-hello_world:latest
rani@rani-server ~/docker_tfg %
```

Figura 5.21: 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 5.22: Ejecución de la imagen Docker creada, utilizando Dockerfile

Ejemplo: **network namespace** asociado a un contenedor Docker

A modo de comprobación de que en efecto Docker utiliza namespaces para la creación de sus contenedores, vamos a profundizar en como obtener los namespaces asociados a un contenedor en específico. [46]

Primero, vamos a iniciar un contenedor con la imagen de Docker Alpine:

```
$ docker pull alpine
$ docker run -d --name docker-ns alpine sleep 3600d
$ docker ps | grep alpine
```



```
arch-tfg# docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
59bf1c3509f3: Pull complete
Digest: sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
arch-tfg# docker run -d --name docker-ns alpine sleep 3600d
8383ce131583495105720f2943ace10ec1ad0215f309138b5bffa2b58088286f
arch-tfg# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS        NAMES
8383ce131583   alpine    "sleep 3600d"           5 seconds ago Up 3 seconds             docker-ns
arch-tfg#
arch-tfg#
```

Figura 5.23: Ejecución contenedor Docker para ver sus namespaces asociados.

Si ejecutamos el comando `ip netns list`, podemos comprobar como nos devuelve una lista vacía. De primeras, podríamos entender esto como que Docker no utiliza network namespaces, sin embargo, esto pasa porque Docker “esconde” estos namespaces por defecto.



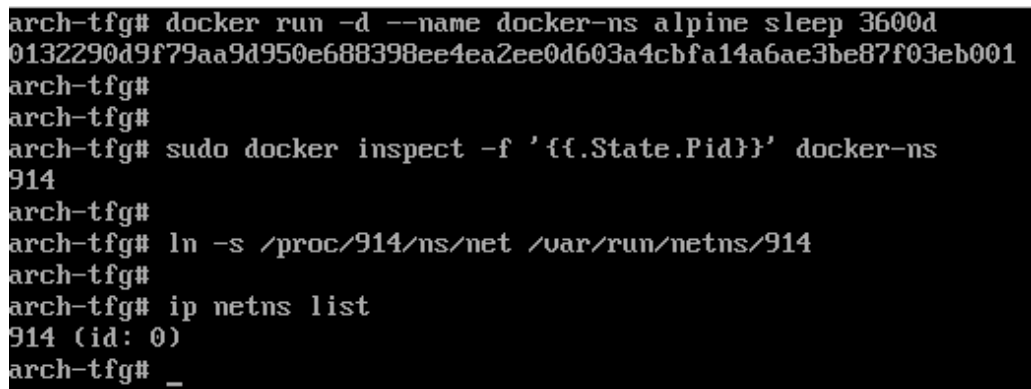
```
arch-tfg# docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
59bf1c3509f3: Pull complete
Digest: sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
arch-tfg# docker run -d --name docker-ns alpine sleep 3600d
8383ce131583495105720f2943ace10ec1ad0215f309138b5bffa2b58088286f
arch-tfg# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS        NAMES
8383ce131583   alpine    "sleep 3600d"           5 seconds ago Up 3 seconds             docker-ns
arch-tfg#
arch-tfg#
arch-tfg# ip netns list
arch-tfg#
```

Figura 5.24: Resultado de comando `ip netns list` con contenedores Docker.

Si queremos exponer los namespaces de un contenedor, tenemos que hacerlo manualmente. Para ello, utilizamos los siguientes comandos:

```
$ sudo docker inspect -f '{{.State.Pid}}' docker-ns
```

```
$ ln -s /proc/<Pid>/ns/net /var/run/netns/<Pid>
```



```
arch-tfg# docker run -d --name docker-ns alpine sleep 3600d
0132290d9f79aa9d950e688398ee4ea2ee0d603a4cbfa14a6ae3be87f03eb001
arch-tfg#
arch-tfg#
arch-tfg# sudo docker inspect -f '{{.State.Pid}}' docker-ns
914
arch-tfg#
arch-tfg# ln -s /proc/914/ns/net /var/run/netns/914
arch-tfg#
arch-tfg# ip netns list
914 (id: 0)
arch-tfg# _
```

Figura 5.25: Network namespaces de un Docker container

Una vez ya nos aparece el network namespace dentro del comando `ip netns list`, podemos comprobar que en efecto es el namespace asociado a nuestro contenedor, para ello podemos ejecutar los siguientes comandos:

```
$ ip netns exec <ip netns output> ip address
```

Que tiene que dar una salida equivalente a los siguientes comandos:

```
$ docker exec -it docker-ns sh
/ ip address
```

Tal y como podemos ver en la figura 5.26, ambos comandos dan el mismo resultado, por lo que quedaría comprobado que Docker utiliza network namespaces para dar conectividad a sus contenedores. Además, es interesante como “esconde” los identificadores de los namespaces que crea para dichos contenedores. Una vez conocemos el identificador, podríamos generar arquitecturas de red mucho más complejas, como por ejemplo añadir un ethernet virtual para hacer *port mirroring* al tráfico del contenedor, este ejemplo se ve más detallado en el siguiente enlace [Traffic mirror with OVS and Docker](#)

```

arch-tfg# ip netns exec 914 ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
arch-tfg#
arch-tfg#
arch-tfg# docker exec -it docker-ns sh
/ # ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
/ #

```

Figura 5.26: Comprobación de si un network namespaces pertenece a un contenedor Docker.

```

arch-tfg# lsns | grep 914
4026532248 mnt      1    914 root      sleep 3600d
4026532249 uts      1    914 root      sleep 3600d
4026532250 ipc      1    914 root      sleep 3600d
4026532251 pid      1    914 root      sleep 3600d
4026532253 net      1    914 root      sleep 3600d
4026532310 cgroup   1    914 root      sleep 3600d
arch-tfg#

```

Figura 5.27: Namespaces asociados a un contenedor Docker.

Capítulo 6

Caso práctico: Virtualización para la simulación de redes

En este apartado vamos a detallar los conceptos de virtualización de topologías de red para la evaluación y/o simulación de situaciones concretas de nuestras redes. Para ello, podemos modelar nuestros nodos de la red utilizando contenedores, siguiendo de esta manera el concepto de “*Virtual Network Function*” que introdujimos en el segundo apartado de este documento. [2]

Como bien hemos visto en el apartado anterior [5], un contenedor es una abstracción de alto nivel de un sistema aislado, utilizando “*namespaces*”. Por lo tanto, tenemos varias maneras de enfocar el camino hacia el objetivo de modelar los nodos de nuestra red. Dichos caminos radican en la metodología de contenedores vamos a utilizar. Por ejemplo, algunos de los caminos a seguir podrían ser los siguientes:

- Crear un *shell script* para desplegar los namespaces necesarios de nuestra topología, utilizando los comandos `nsenter` y `unshare`.
- Desplegar e interconectar contenedores LXC, en el que cada contenedor cumpla una función dentro de nuestra topología (host, switch, controller...)
- Utilizar un simulador de red basado en “namespaces”, dicho simulador se llama Mininet [47] y está creado en el lenguaje Python. El objetivo de este programa es el de crear una red virtual realista, utilizando un mismo kernel para todos los nodos de la red. Además, aporta las herramientas necesarias para interactuar con dicha topología.

6.1 Mininet: Simulador de redes

Profundizando en la última solución aportada, mininet [47], destacamos sus ventajas en la customización de topologías, además de la facilidad de compartir y desplegar dichas topologías en un hardware de uso general utilizando un único comando. Es por esto por lo que mininet es una herramienta muy útil para el desarrollo, la enseñanza y la investigación, siendo una herramienta muy determinante para la experimentación en el campo del *Software Defined Networking* (SDN) ya que permite utilizar controladores basados en OpenFlow.

6.1.1 Primeros pasos

Lo primero que tenemos que hacer para empezar a trabajar con este simulador, es proceder a su instalación. En la página web oficial, se nos detallan diferentes métodos para la instalación [Mininet: Download/Get Started with Mininet](#), dichos métodos son:

- Máquina virtual Mininet, todo previamente instalado.
- Instalación directamente del código fuente
- Instalación utilizando paquetes de una distribución.

La opción de utilizar una máquina virtual preconfigurada es mucho más rápida, ya que solo tendremos que descargar la última versión del enlace que encontramos en la web oficial del proyecto [Github: mininet releases](#). Una vez descargada, importamos dicha imagen en nuestro hipervisor, por ejemplo: *VirtualBox*. Ejecutamos la máquina virtual recién importada y accedemos al sistema con el usuario: `mininet/mininet`.

Sin embargo, si deseamos realizar una instalación de `mininet` sobre nuestra distribución Linux. Podemos instalar el programa en función de la distribución que estemos usando:

- Debian/Ubuntu: `sudo apt-get install mininet`
- Fedora: `sudo dnf install mininet`
- openSUSE: `zypper in mininet`
- Arch Linux: `sudo pacman -S mininet`

Una vez instalado, podemos comprobar la versión ejecutando el siguiente comando en una terminal:

```
$ mn --version
```

Por otro lado, Mininet permite utilizar diferentes switches y controladores para sus topologías. Para esta explicación, vamos a utilizar Open vSwitch [11] como controlador, en el modo *bridge/standalone*. Si queremos comprobar que Open vSwitch está instalado correctamente para mininet, podemos ejecutar el siguiente comando en una terminal:

```
$ mn --switch ovsbr --test pingall
```

En el caso de que el comando anterior nos de un error, tendremos que instalar Open vSwitch para nuestra distribución y asegurarnos de que el servicio esté funcionando. Para iniciar el servicio de Open vSwitch, tendremos que hacer lo siguiente:

```
$ sudo systemctl start openvswitch.service
```

En el caso de Arch Linux, sería tal que así:

```
$ sudo systemctl start ovs-vswitchd.service
```

6.1.2 Ejemplos

Topología simple

En este apartado, vamos a trabajar la dualidad entre implementar una topología utilizando únicamente namespaces, a implementar esa misma topología utilizando mininet. Para ello, vamos a trabajar con una red formada por dos hosts, conectados entre sí mediante un conmutador de paquetes (*switch*), que será manejado por un controlador.

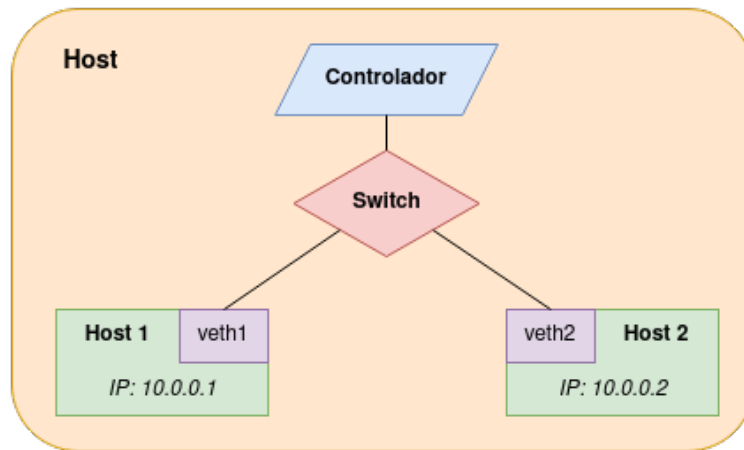


Figura 6.1: Diagrama de la topología simple a implementar.

Ejemplo: topología simple sin usar mininet

Primero, accedemos a nuestra máquina host, en la que desplegaremos los namespaces asociados a la topología. Una vez estamos dentro de la cuenta `root`, procedemos a crear los `network namespaces` de cada host.

```
$ ip netns add host-1
$ ip netns add host-2
```

Con los `network namespaces` ya creados, lo siguiente es crear el conmutador de paquetes. En este caso, vamos a utilizar *Open vSwitch* [11], que previamente hemos instalado en conjunto con mininet. Para crear dicho switch, ejecutamos el siguiente comando:

```
$ ovs-vsctl add-br switch-1
```

Para interconectar los hosts con el conmutador, utilizaremos interfaces de red virtuales de tipo pares de ethernet (`veth pairs`) (explicados en [3.5]). Por eso mismo, utilizamos los siguientes comando para establecer los enlaces punto a punto:

```
$ ip link add host-1-veth1 type veth peer name switch-1-veth1
$ ip link add host-2-veth2 type veth peer name switch-1-veth2
```

En este punto, tendremos que asociar las interfaces a los `network namespaces`.

```
$ ip link set host-1-veth1 netns host-1
$ ip link set host-2-veth2 netns host-2
```

Por otro lado, también será necesario que añadamos el otro extremo de las interfaces al switch.

```
$ ovs-vsctl add-port switch-1 switch-1-veth1
$ ovs-vsctl add-port switch-1 switch-1-veth2
```

Ahora es el momento de configurar el switch para que funcione como un conmutador. Para ello, utilizamos el controlador Open vSwitch (tal y como pudimos ver en los ejemplos asociados a pares ethernet [3.5]). Por lo tanto, para hacerlo funcionar en nuestra topología, tendremos que ejecutar los siguientes comandos:

```
$ ovs-vsctl set-controller switch-1 tcp:127.0.0.1
$ ovs-testcontroller ptcp: &
```

Si queremos comprobar que hemos creado correctamente el switch virtual, podemos utilizar el siguiente comando:

```
$ ovs-vsctl show
```



```
arch-tfg# ovs-vsctl show
3586ca30-226c-47e9-aa4d-36bc214f4b33
    Bridge switch-1
        Controller "tcp:127.0.0.1"
            is_connected: true
        Port switch-1
            Interface switch-1
                type: internal
        Port switch-1-veth1
            Interface switch-1-veth1
        Port switch-1-veth2
            Interface switch-1-veth2
arch-tfg# _
```

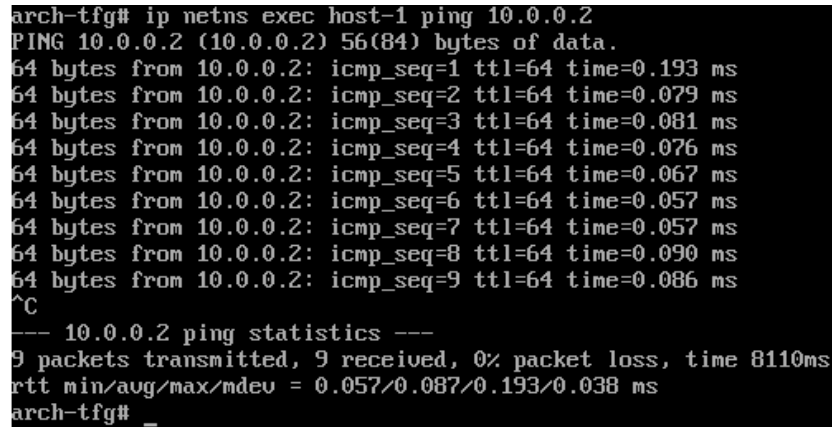
Figura 6.2: Captura de la configuración del switch usando Open vSwitch.

Una vez tenemos todos los dispositivos configurados, lo que tendremos que hacer es asignar direcciones IP a cada host, y pondremos en funcionamiento las interfaces asociadas.

```
$ ip netns exec host-1 ip addr add 10.0.0.1/24 dev host-1-veth1
$ ip netns exec host-2 ip addr add 10.0.0.2/24 dev host-2-veth2
$ ip netns exec host-1 ip link set host-1-veth1 up
$ ip netns exec host-1 ip link set lo up
$ ip netns exec host-2 ip link set host-2-veth2 up
$ ip netns exec host-2 ip link set lo up
$ ip link set switch-1-veth1 up
$ ip link set switch-1-veth2 up
```

En este punto, ya deberíamos tener conectividad entre ambos hosts. Para ello, lo que podemos hacer es realizar un ping entre `host-1` (IP: 10.0.0.1) y `host-2` (IP: 10.0.0.2).

```
$ ip netns exec host-1 ping 10.0.0.2
```



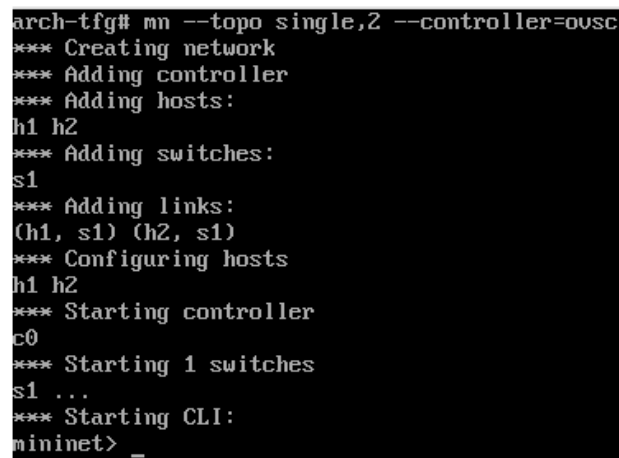
```
arch-tfg# ip netns exec host-1 ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.193 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.079 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.081 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.067 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.057 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.057 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.090 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.086 ms
^C
--- 10.0.0.2 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8110ms
rtt min/avg/max/mdev = 0.057/0.087/0.193/0.038 ms
arch-tfg# _
```

Figura 6.3: Ejecución ping entre `host1` y `host2`.

Ejemplo: topología simple utilizando **mininet**

En este caso, vamos a replicar la misma topología del ejemplo anterior, pero esta vez utilizando `mininet`. Dado que es una topología sencilla, podemos ejecutarla con un único comando.

```
$ mn --topo single,2 --controller=ovsc
```



```
arch-tfg# mn --topo single,2 --controller=ovsc
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> _
```

Figura 6.4: Ejecución topología simple en `mininet`.

Como podemos comprobar en la captura anterior, una vez ejecutamos el comando, la topología se crea automáticamente. Si quisiéramos comprobar la conectividad entre los diferentes hosts, solo tendríamos que ejecutar el siguiente comando dentro de la terminal de mininet.

```
mininet> h1 ping h2
```

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.95 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.218 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.059 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.056 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.083 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.059 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.059 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.060 ms
^C
--- 10.0.0.2 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7083ms
rtt min/avg/max/mdev = 0.056/0.317/1.949/0.618 ms
mininet>
```

Figura 6.5: Ping entre hosts de topología simple en mininet.

Mientras tenemos activa la topología de mininet, podemos comprobar qué namespaces ha creado para dicha topología.

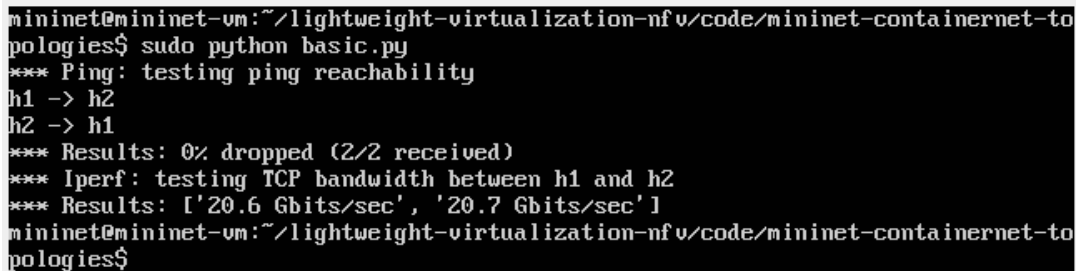
```
arch-tfg# lsns | grep mininet
4026532246 mnt      1   688 root      bash --norc --noediting -is mininet:h1
4026532248 net      1   688 root      bash --norc --noediting -is mininet:h1
4026532305 mnt      1   690 root      bash --norc --noediting -is mininet:h2
4026532307 net      1   690 root      bash --norc --noediting -is mininet:h2
arch-tfg# _
```

Figura 6.6: Comprobación de namespaces utilizados por topología en mininet.

Como bien podemos ver en la imagen (6.6), mininet solo está utilizando los namespaces mount y network para el despliegue de la topología.

Si quisiéramos implementar esta topología simple utilizando la API Python que está incluida en mininet, podemos ejecutar el código del Anexo 4 [A.1] (ver código A.1)

Ejecutando el código implementado, podemos comprobar como al iniciar la topología se realizan comprobaciones tanto de conectividad, utilizando ping, como de velocidades, utilizando iperf. En la figura 6.7 podemos comprobar cual es el rendimiento de los enlaces al ejecutar dicha topología.



```
mininet@mininet-vm:~/lightweight-virtualization-nfv/code/mininet-container-net-topologies$ sudo python basic.py
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['20.6 Gbits/sec', '20.7 Gbits/sec']
mininet@mininet-vm:~/lightweight-virtualization-nfv/code/mininet-container-net-topologies$
```

Figura 6.7: Ejecución de iperf sobre topología simple en mininet.

Ejemplo: limitación de «recursos» usando mininet

En este ejemplo, vamos a profundizar sobre como aplicar limitaciones de «recursos» a una topología desplegada con mininet. Algunos de los «recursos» más importantes a tener en cuenta son los siguientes:

- *CPU*. Nos permite asignar el máximo uso de CPU de un dispositivo. Esto nos permitirá ajustar nuestra simulación a un entorno más real, además de permitirnos gestionar más los «recursos» disponibles.
- *Latencia*. Permite establecer una latencia máxima a un enlace entre dispositivos.
- *Ancho de banda*. Permite establecer un máximo de ancho de banda para un enlace entre dispositivos.

Por lo tanto, a modo de ejemplo, vamos a utilizar la topología anterior, pero aplicando una limitación de ancho de banda, latencia y CPU a uno de los enlaces que conectan un host con el switch. El código utilizado para este ejemplo lo podemos encontrar en el Anexo 4 [A.1] (ver código A.2).

Para ejecutar la topología, utilizamos el siguiente comando:

```
sudo python limit.py
```

Como podemos comprobar en la figura 6.8, al ejecutar la topología, vemos como en consola nos aparece un aviso de que se ha limitado un enlace a 5 Mbit de ancho de banda y 80 ms de latencia. Por otro lado, una vez la topología esta funcionando, se procede a una comprobación de conectividad entre ambos host, seguidamente, se ejecuta la herramienta iperf para comprobar cual es el ancho de banda máximo entre ambos hosts.

```

mininet@mininet-vm:~/lightweight-virtualization-nfv/code/mininet-container-net-topologies$ sudo python limit.py
(5.00Mbit 80ms delay) (5.00Mbit 80ms delay) *** Configuring hosts
h1 (cfs 10000/100000us) h2
*** Starting controller
c0
*** Starting 1 switches
s1 (5.00Mbit 80ms delay) ... (5.00Mbit 80ms delay)
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['4.58 Mbits/sec', '6.50 Mbits/sec']
*** Starting CLI:
mininet>

```

Figura 6.8: Ejecución topología simple limitada en mininet. (1/2)

Comparando los resultados obtenidos entre la topología sin limitar (ver figura 6.7) y la topología limitada tanto en CPU como en los enlaces (ver figura 6.8), podemos comprobar que efectivamente el ancho de banda que obtenemos de la herramienta *iperf* se ve claramente reducido, pasando de 20 Gbits/sec a 4.58 Mbits/sec. Por lo tanto, podemos concluir que efectivamente se están aplicando correctamente la limitación de ancho de banda.

Por otro lado, otra medida interesante a comparar es el tiempo que tarda un paquete entre que es enviado a un host y la respuesta vuelve al host emisor. Esto lo podemos comprobar con la herramienta *ping*. Dado que hemos fijado la latencia de uno de los enlaces (fijado a 80 ms), deberíamos obtener un valor del campo *time* mayor que al que obtenemos en la topología simple sin limitar. Ver figura 6.9.

```

mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=161 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=161 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=160 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=163 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=160 ms
^C
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4003ms
rtt min/avg/max/mdev = 160.353/161.052/162.867/0.932 ms
mininet>

```

Figura 6.9: Ejecución topología simple limitada en mininet. (2/2)

Al igual que para el ancho de banda, los resultados de la topología sin limitar (ver figura 6.5) y la topología limitada (ver figura 6.9), son totalmente diferentes. Estas limitaciones son muy interesantes para modelar una topología con características lo más cercanas a un despliegue real. Por lo tanto, para simular una red, deberemos configurar estos valores de ancho de banda y latencia para cada uno de los enlaces de la topología.

Ejemplo: controlador externo

En los ejemplos anteriores de `mininet`, hemos utilizado `Open vSwitch` actuando como un conmutador *learning switch*. Dicho modo de funcionamiento tiene como tarea distribuir, de manera dinámica y según ciertas reglas, la información asociada al encaminamiento de los paquetes de la red, rellenando automáticamente las `flow-table` de los switches.

Sin embargo, `mininet` nos permite la utilización de otros controladores SDN externos, que además pueden estar basados en las diferentes versiones del protocolo OpenFlow que están disponibles actualmente, permitiendo así compatibilidad con más dispositivos físicos disponibles en el mercado.

Algunos de los controladores SDN que más se utilizan actualmente son los siguientes:

- `OpenDayLight` [48]. Controlador SDN más utilizado. Desarrollado en el lenguaje Java. Filosofía modular, permite desarrollar aplicaciones, de modo que se pueda parametrizar y automatizar las redes a la escala que queramos.
- `POX/NOX` [49][50]. Controlador SDN escrito en Python/C++. Soporta OpenFlow 1.0. Permite un desarrollo rápido y asíncrono.
- `Ryu` [51]. Controlador SDN escrito en Python. Filosofía basada en componentes, por lo que permite una fácil integración con nuevas aplicaciones. Aporta una `northbound API` bien definida. Soporta OpenFlow 1.0, 1.2, 1.3, 1.4 y 1.5, además de otros protocolos como `Netconf`.
- `ONOS` [52]. Controlador SDN muy utilizado. Desarrollado en el lenguaje Java y que tiene una arquitectura modular.

Controlador Ryu

A modo de ejemplo, vamos a instalar y ejecutar una topología en `mininet` utilizando el controlador SDN `Ryu`.

Lo primero, para instalar `Ryu` tenemos que ejecutar el comando siguiente. Para este ejemplo, vamos a utilizar la misma máquina virtual que estamos utilizando para `mininet`.

```
$ git clone https://github.com/faucetsdn/ryu.git
$ pip install ryu/.
```

En una terminal (la llamaremos terminal A), ejecutaremos una topología de `mininet`. Por ejemplo, utilizando el siguiente comando:

```
$ sudo mn --topo tree,depth=3 --mac --switch ovsk --controller remote
```

Con dicha topología, estaremos simulando una red de tipo árbol con 8 hosts, 7 switches `Open vSwitch` y un controlador remoto (escuchando en la dirección `127.0.0.1:6653`).

Una vez ejecutada la topología, podemos comprobar que no tenemos conectividad entre los host utilizando el comando `pingall`, esto es debido a que no tenemos ejecutando nuestro controlador. (ver Figura 6.10)

```
mininet@mininet-vm:~$ sudo mn --topo tree,depth=3 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7
*** Adding links:
(s1, s2) (s1, s5) (s2, s3) (s2, s4) (s3, h1) (s3, h2) (s4, h3) (s4, h4) (s5, s6) (s5, s7) (s6, h5) (s6, h6) (s7, h7) (s7, h8)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8
*** Starting controller
c0
*** Starting 7 switches
s1 s2 s3 s4 s5 s6 s7 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X X X X X
h2 -> X X X X X X X
h3 -> X X X X X X X
h4 -> X X X X X X X
h5 -> X X X X X X X
h6 -> X X X X X X X
h7 -> X X X X X X X
h8 -> X X X X X X X
*** Results: 100% dropped (0/56 received)
```

Figura 6.10: Ejecución topologia mininet sin controlador asignado (terminal A)

En este caso, vamos a ejecutar nuestro controlador con una configuración de switch que soporta OpenFlow 1.3 (alojado en `simple_switch_13.py`). Para ello, ejecutamos el siguiente comando en una terminal diferente a la terminal A (la llamaremos terminal B).

```
$ ryu run ryu.app.simple_switch_13
```

Una vez ejecutado, nos aparecerá en la terminal B la información acerca de los módulos que Ryu ha cargado para dar la funcionalidad que deseamos. Además, en dicha terminal se irán mostrando los diferentes paquetes que conmute el switch.

Si ahora comprobamos la conectividad en la topología de mininet (escribiendo en la terminal A el comando `pingall`), podemos ver como efectivamente ahora si que los paquetes llegan a su destino. Por lo tanto, podemos determinar que efectivamente el controlador SDN está rellenando correctamente la `flow-table` del switch. (ver Figura 6.11)

```

mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8
h2 -> h1 h3 h4 h5 h6 h7 h8
h3 -> h1 h2 h4 h5 h6 h7 h8
h4 -> h1 h2 h3 h5 h6 h7 h8
h5 -> h1 h2 h3 h4 h6 h7 h8
h6 -> h1 h2 h3 h4 h5 h7 h8
h7 -> h1 h2 h3 h4 h5 h6 h8
h8 -> h1 h2 h3 h4 h5 h6 h7
*** Results: 0% dropped (56/56 received)
mininet>

mininet@mininet-vm:~/ryu$ ryu run ryu.app.simple_switch_13
loading app ryu.app.simple_switch_13
loading app ryu.controller.ofp_handler
instantiating app ryu.app.simple_switch_13 of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
packet in 0000000000000003 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000002 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000003 00:00:00:00:00:02 00:00:00:00:00:01 2
packet in 0000000000000004 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 3
packet in 0000000000000003 00:00:00:00:00:01 00:00:00:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000005 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 3
packet in 0000000000000007 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 3
packet in 0000000000000006 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 3
packet in 0000000000000003 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000002 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000004 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 3

```

Figura 6.11: Ejecución del controlador Ryu (terminal inferior, B) y posterior pingall en mininet (terminal superior, A)

Por otro lado, otra funcionalidad interesante de este controlador es el visualizador de topologías. Nos permite ver en una página web la topología de nuestra red, es decir, como se interconectan los diferentes switches de la red. Para ello, solo tenemos que ejecutar el siguiente comando en la terminal B (si tenemos activo el ejemplo de `simple_switch_13.py`, pararemos su ejecución con Ctrl+C):

```
$ ryu run --observe-links ryu.app.simple_switch_13 ryu.app.gui_
topology.gui_topology
```

Ahora, una vez ejecutado el comando anterior, tenemos que utilizar un navegador web y dirigirnos a la siguiente dirección: <http://localhost:8080/>. Después de acceder a dicha página, podemos ver como se nos presenta la topología de la red que estamos controlando (ver Figura 6.12). [53]

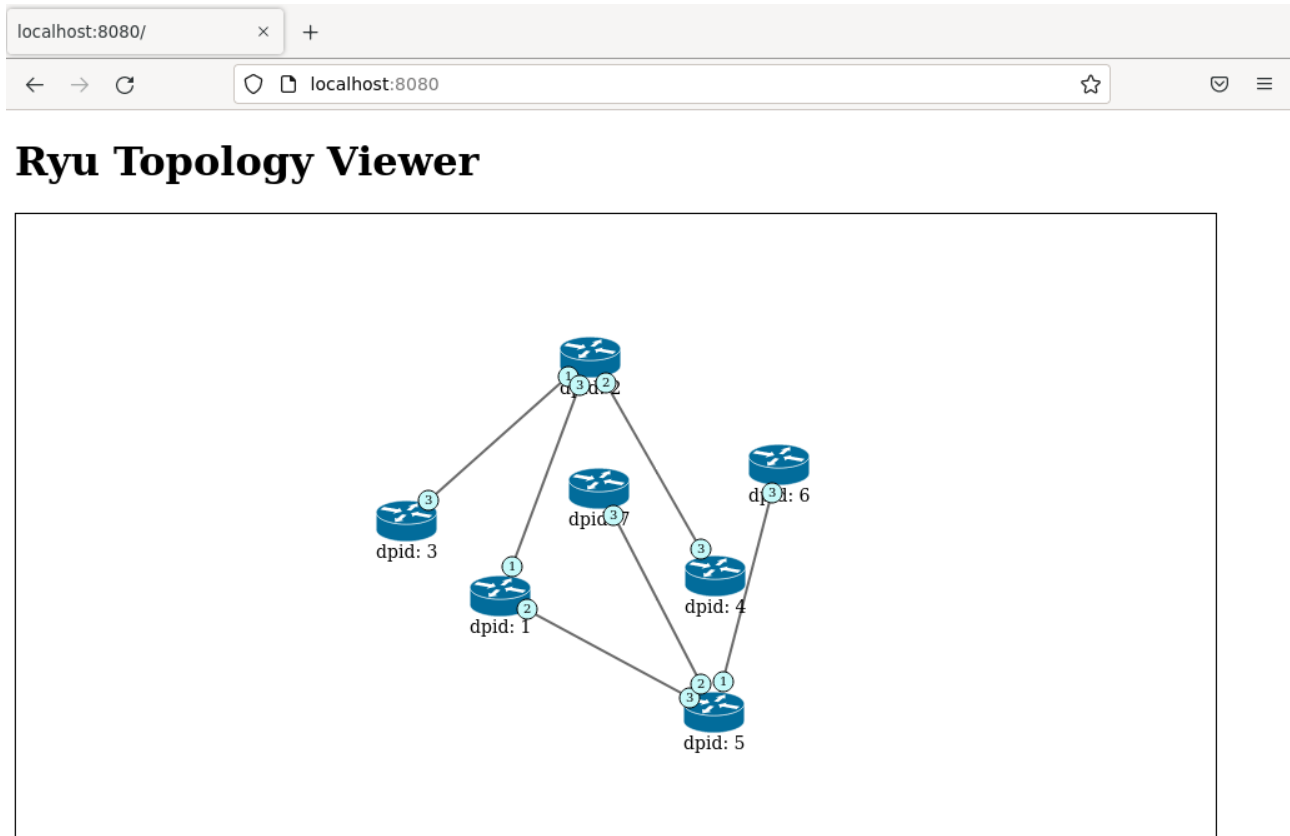


Figura 6.12: Herramienta *Topology Viewer* del controlador Ryu

6.2 Containernet: Simulador de redes

Por otro lado, existen otras implementaciones de simuladores que extienden la funcionalidad de `mininet`, permitiendo abarcar muchas más situaciones a simular. A modo de ejemplo, destacamos `containernet` [54], siendo un simulador de red basado en `mininet` que permite utilizar contenedores de `Docker` como hosts en una red emulada. Esta funcionalidad permite construir entornos de simulación para redes de tipo *cloud*.

Además, es un proyecto muy activo por la comunidad investigadora, ya que permite evaluar topologías de *cloud computing*, *network function virtualization* (NFV) o *multi-access edge computing* (MEC). Un ejemplo de esto es el emulador [vim-emu: A NFV multi-PoP emulation Platform](#), desarrollado por [SONATA project](#), y que está basado en `containernet`.

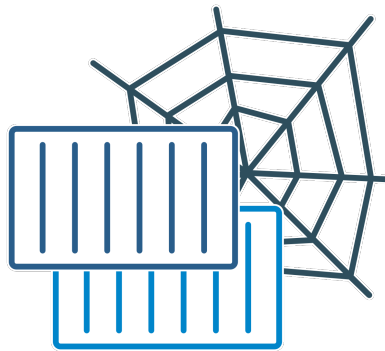


Figura 6.13: Logotipo del proyecto `containernet`

Características

Algunas de las características más importantes que nos aporta `containernet`, son las siguientes:

- Añadir y eliminar contenedores `Docker` a topologías de `mininet`. Interconectar dichos contenedores con la topología.
- Ejecutar comandos dentro de los contenedores, utilizando `mininet CLI`.
- Cambios dinámicos en las topologías. Permite añadir o eliminar contenedores a topologías que estén en ejecución.
- Limitar «recursos» de los contenedores `Docker`, de forma individual.
- Permite configurar los enlaces en base a: *delay*, *bandwidth*, *loss* o *jitter*.

6.2.1 Primeros pasos

Lo primero que tenemos que hacer para trabajar con este simulador es proceder a su instalación. Para ello, nos vamos a guiar por los pasos que están detallados en la documentación del proyecto en [Github: containernet](#). Lo primero que podemos comprobar es que se nos aportan dos opciones de instalación:

- **Bare-metal installation.** Se refiere a una instalación sobre la distribución Ubuntu 20.04 LTS con Python 3.
- **Nested Docker deployment.** Utiliza un contenedor Docker con privilegios para desplegar en su interior otro entorno Docker en el que ejecutar las topologías de containernet. Esta opción tiene deshabilitado la característica de limitación de «recursos».

Para nuestro entorno de pruebas, hemos elegido instalar containernet en una máquina virtual en *VirtualBox*, utilizando Ubuntu 20.04 LTS, tal y como leemos en su documentación. Para ello, seguimos las instrucciones tal que:

1. Instalación de Ansible

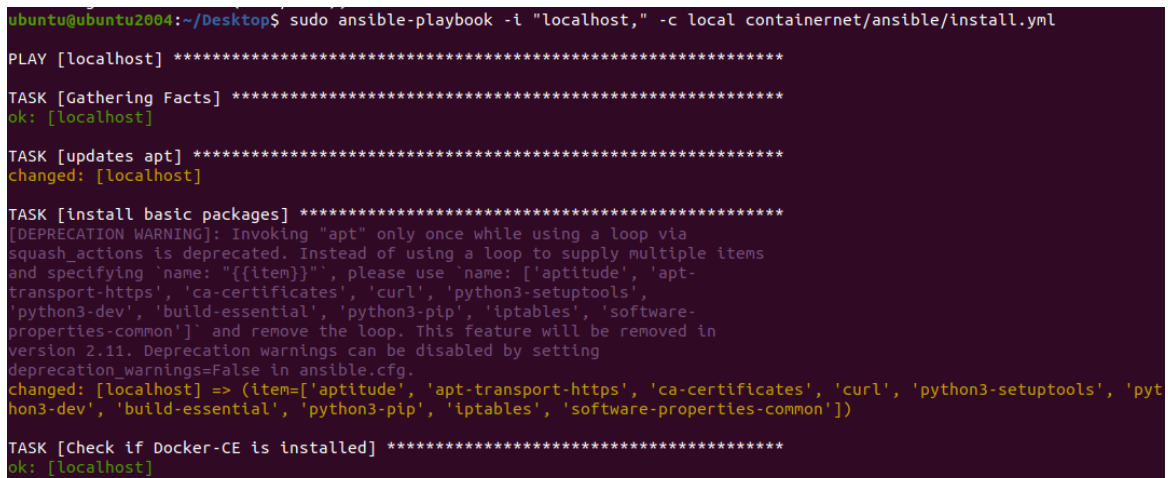
```
$ sudo apt-get install ansible
```

2. Clonamos el repositorio de Github:

```
$ git clone https://github.com/containernet/containernet.git
```

3. Ejecutamos el “playbook” de Ansible, el cual instalará todas las dependencias y posteriormente containernet.

```
$ sudo ansible-playbook -i "localhost," -c local containernet/ansible/install.yml
```



```
ubuntu@ubuntu2004:~/Desktop$ sudo ansible-playbook -i "localhost," -c local containernet/ansible/install.yml
PLAY [localhost] *****
TASK [Gathering Facts] *****
ok: [localhost]
TASK [updates apt] *****
changed: [localhost]
TASK [install basic packages] *****
[DEPRECATION WARNING]: Invoking "apt" only once while using a loop via
squash_actions is deprecated. Instead of using a loop to supply multiple items
and specifying 'name: "{{item}}"', please use 'name: ['aptitude', 'apt-
transport-https', 'ca-certificates', 'curl', 'python3-setuptools',
'python3-dev', 'build-essential', 'python3-pip', 'iptables', 'software-
properties-common']' and remove the loop. This feature will be removed in
version 2.11. Deprecation warnings can be disabled by setting
deprecation_warnings=False in ansible.cfg.
changed: [localhost] => (item=['aptitude', 'apt-transport-https', 'ca-certificates', 'curl', 'python3-setuptools', 'pyt
hon3-dev', 'build-essential', 'python3-pip', 'iptables', 'software-properties-common'])
TASK [Check if Docker-CE is installed] *****
ok: [localhost]
```

Figura 6.14: Instalación containernet en Ubuntu 20.04 LTS (1/2)

```

TASK [install pytest] *****
changed: [localhost]

TASK [install docker py] *****
changed: [localhost]

TASK [install python-iptables] *****
changed: [localhost]

TASK [install pexpect] *****
ok: [localhost]

TASK [build and install Containernet (using Mininet installer)] *****
changed: [localhost]

TASK [install Containernet Python egg etc.] *****
changed: [localhost]

TASK [download 'ubuntu' docker image for Containernet example] *****
changed: [localhost]

PLAY RECAP *****
localhost : ok=17  changed=14  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0

ubuntu@ubuntu2004:~/Desktop$

```

Figura 6.15: Instalación containernet en Ubuntu 20.04 LTS (2/2)

Si la instalación es satisfactoria, la salida de la consola debería ser muy similar a la que vemos en la imagen 6.15. Al final de la ejecución del “playbook” de Ansible, se nos muestra un resumen de las tareas ejecutadas, y cuales de ellas se han realizado correctamente, o bien, cuales han dado error al ejecutarse. En nuestro caso, se ha instalado correctamente.

6.2.2 Implementación topología simple

En este apartado vamos a comentar como desplegar una topología simple dentro de containernet. Dado que la filosofía de este programa es extender la funcionalidad de mininet hacia la posibilidad de que los hosts sean contenedores de Docker, lo que tendremos como topología son varios contenedores Docker, interconectados cada uno con un switch. Como podemos ver, el diagrama de la red es muy parecido al que utilizamos como topología simple en mininet.

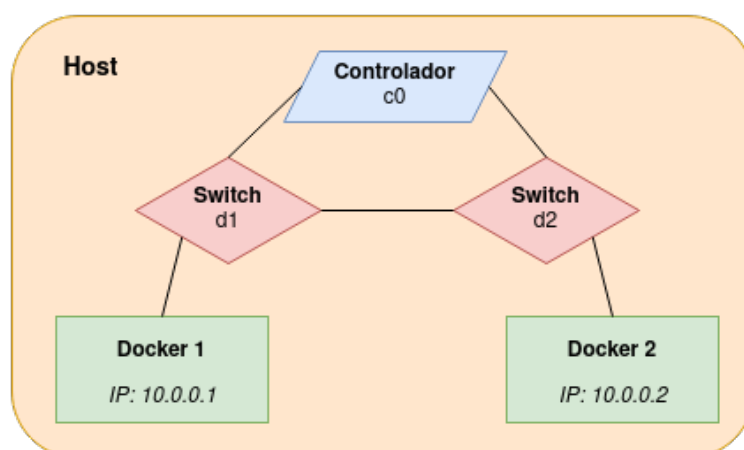


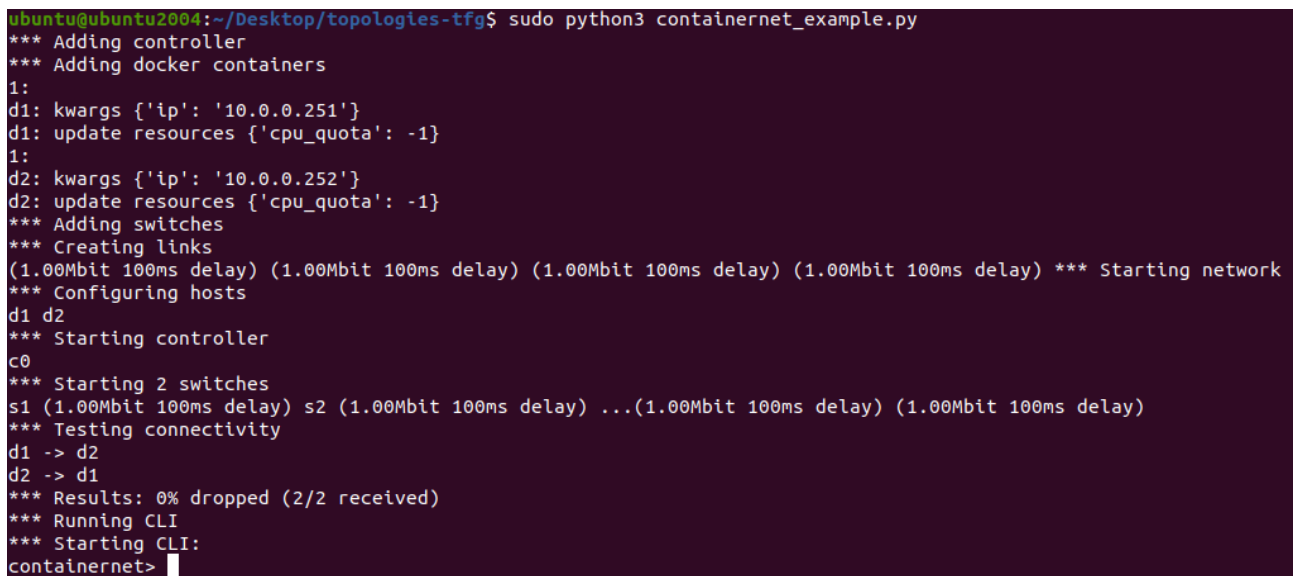
Figura 6.16: Diagrama topología simple utilizando containernet

Esta topología que podemos ver en el diagrama, está ubicada en la carpeta de `examples` dentro de `containernet`. Con esta topología podemos ver las bases del simulador, y comprobar como puede trabajar con las características de `mininet`, a la vez de implementar los hosts como contenedores Docker. El código Python utilizado para ejecutar el ejemplo lo encontramos en el Anexo 4 [A.1] (ver código A.3).

Para ejecutar el ejemplo, tendremos que lanzar el siguiente comando:

```
$ sudo python3 containernet_example.py
```

La salida por consola será muy similar a la que obtendríamos en `mininet`, sin embargo, podemos comprobar como aparece la información respectiva de nuestros contenedores Docker que van a tener la función de host en la topología. Todo esto lo podemos ver en la figura 6.17.



```
ubuntu@ubuntu2004:~/Desktop/topologies-tfg$ sudo python3 containernet_example.py
*** Adding controller
*** Adding docker containers
1:
d1: kwargs {'ip': '10.0.0.251'}
d1: update resources {'cpu_quota': -1}
1:
d2: kwargs {'ip': '10.0.0.252'}
d2: update resources {'cpu_quota': -1}
*** Adding switches
*** Creating links
(1.00Mbit 100ms delay) (1.00Mbit 100ms delay) (1.00Mbit 100ms delay) (1.00Mbit 100ms delay) *** Starting network
*** Configuring hosts
d1 d2
*** Starting controller
c0
*** Starting 2 switches
s1 (1.00Mbit 100ms delay) s2 (1.00Mbit 100ms delay) ...(1.00Mbit 100ms delay) (1.00Mbit 100ms delay)
*** Testing connectivity
d1 -> d2
d2 -> d1
*** Results: 0% dropped (2/2 received)
*** Running CLI
*** Starting CLI:
containernet> █
```

Figura 6.17: Ejecución de topología simple en containernet (1/5)

Una vez se termina de ejecutar la topología, nos deja activa la CLI para que podamos interactuar con la misma. En nuestro caso, vamos a comprobar cual es la configuración de red asociada al host 1, para ver como ha gestionado las interfaces del contenedor de Docker. Para ello, ejecutamos el siguiente comando:

```
containernet> d1 ifconfig
```

Como podemos comprobar en la figura 6.18, el host 1 tiene una interfaz (`d1-eth0`) que es la que estará en uso para nuestra topología de `containernet`, además dispone de otra interfaz (`eth0`) que será la que utilice para comunicarse con el controlador de Docker. Por lo tanto, podemos ver como `containernet` no manda su tráfico por el bridge que crea Docker, sino que añade una interfaz específica (un extremo de un par ethernet virtual, *veth*) por el que encaminará el tráfico de la simulación.

```

containernet> d1 ifconfig
d1-eth0  Link encap:Ethernet  HWaddr 8a:33:82:c2:97:9c
         inet addr:10.0.0.251  Bcast:0.0.0.0  Mask:255.0.0.0
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:48 errors:0 dropped:0 overruns:0 frame:0
         TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:5976 (5.9 KB)  TX bytes:280 (280.0 B)

eth0     Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
         inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:50 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
         RX bytes:6984 (6.9 KB)  TX bytes:0 (0.0 B)

lo       Link encap:Local Loopback
         inet addr:127.0.0.1  Mask:255.0.0.0
         UP LOOPBACK RUNNING  MTU:65536  Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

containernet>

```

Figura 6.18: Ejecución de topología simple en containernet (2/5)

Por otro lado, mientras tenemos la topología ejecutándose, podemos comprobar como aparecen los contenedores Docker en nuestro sistema. Además, también podemos acceder a una *shell* de uno de los contenedores (hosts de containernet) y comprobar que efectivamente el resultado es el mismo que si accedemos desde la CLI de containernet. Para ello, hemos comprobado la configuración de red del mismo host que vimos en la figura 6.18, esto lo podemos ver en la figura 6.19

```

root@ubuntu2004:/home/ubuntu# docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS          NAMES
afd4b7b42f7c   ubuntu:trusty  "/bin/bash"             About a minute ago    Up About a minute           mn.d2
fe568ecf16d0   ubuntu:trusty  "/bin/bash"             About a minute ago    Up About a minute           mn.d1
root@ubuntu2004:/home/ubuntu# docker attach fe568ecf16d0
root@d1:/# ifconfig
d1-eth0  Link encap:Ethernet  HWaddr 8a:33:82:c2:97:9c
         inet addr:10.0.0.251  Bcast:0.0.0.0  Mask:255.0.0.0
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:52 errors:0 dropped:0 overruns:0 frame:0
         TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:6366 (6.3 KB)  TX bytes:280 (280.0 B)

eth0     Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
         inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:53 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
         RX bytes:7304 (7.3 KB)  TX bytes:0 (0.0 B)

lo       Link encap:Local Loopback
         inet addr:127.0.0.1  Mask:255.0.0.0
         UP LOOPBACK RUNNING  MTU:65536  Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@d1:/#

```

Figura 6.19: Ejecución de topología simple en containernet (3/5)

Al igual que hacíamos en mininet, podemos comprobar conectividad entre dos hosts utilizando el comando ping, para ello podemos ejecutar en la CLI el siguiente comando:

```
containernet> d1 ping -c4 d2
```

Una vez ejecutado, observamos como la salida nos muestra el ping que se ha ejecutado, y como efectivamente hay conectividad entre los dos hosts. (ver figura 6.20)

```
ubuntu@ubuntu2004:~/Desktop/topologies-tfg$ sudo python3 containernet_example.py
*** Adding controller
*** Adding docker containers
1:
d1: kwargs {'ip': '10.0.0.251'}
d1: update resources {'cpu_quota': -1}
1:
d2: kwargs {'ip': '10.0.0.252'}
d2: update resources {'cpu_quota': -1}
*** Adding switches
*** Creating links
(1.00Mbit 100ms delay) (1.00Mbit 100ms delay) (1.00Mbit 100ms delay) *** Starting network
*** Configuring hosts
d1 d2
*** Starting controller
c0
*** Starting 2 switches
s1 (1.00Mbit 100ms delay) s2 (1.00Mbit 100ms delay) ...(1.00Mbit 100ms delay) (1.00Mbit 100ms delay)
*** Testing connectivity
d1 -> d2
d2 -> d1
*** Results: 0% dropped (2/2 received)
*** Running CLI
*** Starting CLI:
containernet> d1 ping -c4 d2
PING 10.0.0.252 (10.0.0.252) 56(84) bytes of data.
64 bytes from 10.0.0.252: icmp_seq=1 ttl=64 time=234 ms
64 bytes from 10.0.0.252: icmp_seq=2 ttl=64 time=217 ms
64 bytes from 10.0.0.252: icmp_seq=3 ttl=64 time=203 ms
^C
--- 10.0.0.252 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2010ms
rtt min/avg/max/mdev = 203.942/218.741/234.650/12.566 ms
containernet>
```

Figura 6.20: Ejecución de topología simple en containernet (4/5)

Por último, si ya no queremos seguir ejecutando la topología, tendremos que ejecutar el comando `exit`. Este comando nos permitirá revertir la configuración asociada a la topología, además de apagar y eliminar los contenedores Docker que estaban ejerciendo como hosts. El resultado de la ejecución de este comando lo podemos ver en la figura 6.21.

```
containernet> exit
*** Stopping network*** Stopping 1 controllers
c0
*** Stopping 3 links
...
*** Stopping 2 switches
s1 s2
*** Stopping 2 hosts
d1 d2
*** Done
*** Removing NAT rules of 0 SAPs
ubuntu@ubuntu2004:~/Desktop/topologies-tfg$
```

Figura 6.21: Ejecución de topología simple en containernet (5/5)

Capítulo 7

Conclusiones y propuestas futuras

En este capítulo vamos a comentar las conclusiones obtenidas al realizar el presente trabajo.

En primer lugar, hemos presentado la problemática actual de la escalabilidad de las redes y como se está trabajando para un futuro basado en el software y en la descentralización. Esto supone una búsqueda para poder alejarnos de hardware específico/embebido, para acercarnos a un hardware de carácter general en el que podamos desplegar nuestros servicios, todo esto, potenciado con la virtualización de los sistemas. A raíz de esto, hemos definido NFV y los conceptos más importantes para entender que és y por qué es un desarrollo muy importante para las telecomunicaciones. Además, lo hemos “diferenciado” respecto al concepto de SDN, y comprobado como ambas son tecnologías que van a ir de la mano, ya que una se apoya en la otra.

Por otro lado, se ha explicado las diferentes interfaces de red virtuales de Linux, ya que son realmente importantes para las soluciones de virtualización. Para completar la explicación, se ha seguido una metodología basada en ejemplos, por lo que hemos podido comprobar la funcionalidad y el objetivo de cada una de las interfaces expuestas. Hemos comprobado la importancia de interfaces como pueden ser los pares ethernet o bien los tup/tap.

En relación a los namespaces, se han definido los namespaces presentes dentro del Kernel de Linux. Se ha verificado el potencial que posee esta tecnología, y como está muy presente en diferentes tecnologías que están fuertemente arraigadas por la comunidad, por ejemplo LXC o Docker. Gracias a los ejemplos, se ha podido comprender fácilmente el funcionamiento de cada uno de los namespaces.

En el punto de virtualización ligera, hemos resumido las opciones disponibles para realizar virtualización ligera. Además, hemos recreado un contenedor paso a paso, de modo que hemos podido comprobar la importancia de los namespaces. Por otro lado, con las definiciones de LXC y de Docker, hemos iniciado al uso de ambas herramientas, facilitando su posterior uso.

Por último, se han propuesto ciertas herramientas que utilizan la virtualización ligera para simular redes de comunicaciones. En este caso hemos hablado de mininet, como una implementación basada totalmente en namespaces, y de containernet como un “fork” de mininet que está extendido para trabajar con hosts de Docker. Ambas herramientas son muy importantes ya que permiten crear entornos de desarrollo e investigación para el área de SDN. Además, nos permite realizar pruebas con topologías de gran tamaño de manera sencilla y rápida.

En resumen, podemos concluir que el presente trabajo ha cumplido todos los objetivos propuestos, ya que se ha profundizado sobre la virtualización ligera, y se ha ido construyendo una “guía” a base de explicaciones y ejemplos, que nos permite comprender los entresijos de simuladores de redes basados en dicha virtualización, como puede ser `mininet`.

Propuestas futuras

Algunas de las líneas de investigación que pueden resultar interesantes para continuar este proyecto, son las siguientes:

- Profundizar en las diferencias de rendimiento entre utilizar una virtualización “dura”, usando un hipervisor (por ejemplo `VirtualBox`), respecto a una solución basada en contenedores, como las que hemos comentado en este trabajo. Por otro lado, en temas de rendimiento, también se podría estudiar cual podría ser el límite en la simulación de redes utilizando virtualización ligera.
- Investigar sobre modificaciones del protocolo `OpenFlow`, como puede ser <https://github.com/Orange-OpenSource/oko>, que nos permite aplicar programas con los que filtrar paquetes de la red utilizando `BPF` (*Berkley Packet Filter*). Por otro lado, sería realmente importante investigar la tecnología `DPDK`, ya que nos permite acceder directamente a la información de nuestras interfaces de red, además de mejorar considerablemente el *throughput* en comparación de si utilizamos un *Linux bridge*.
- Estudiar el lenguaje de programación `P4` (<https://p4.org/>), ya que ahora mismo está siendo muy importante para el desarrollo de soluciones software en routers y switches. Se está observando gran aceptación por la comunidad investigadora, y hay un creciente interés para su implantación. El objetivo de este lenguaje de programación es el de controlar el flujo de los paquetes de datos entre los diferentes planos de información que maneja el dispositivo. `P4` está definido teniendo en mente que tiene que ser específico para las redes, por lo que presenta ciertas construcciones específicas para optimizar el flujo de la información.

Capítulo 8

Bibliografía

Enlaces y referencias

1. [NFV white paper](#)
2. [What is Network Function Virtualization \(NFV\)?](#)
3. [TFG. Evaluación de prestaciones mediante NVF](#)
4. [Introduction to Linux interfaces for virtual networking](#)
5. [Arch Linux Wiki: udev](#)
6. [Predictable Network Interface Names](#)
7. [IEEE 802.1ad](#)
8. [FS.COM QinQ Operation](#)
9. [OpenWrt: Linux Network Interfaces](#)
10. [Network namespaces. Assign and configure](#)
11. [Open vSwitch](#)
12. [Veth Devices, Network Namespaces and Open vSwitch](#)
13. [TUN/TAP interface \(on Linux\)](#)
14. [Creating tap/tun devices with IP tuntap and tuncctl as detailed in Linux network tools](#)
15. [Principio de diseño del controlador TUN/TAP de la tarjeta virtual](#)
16. [Aplicación sock](#)
17. [Namespaces](#)
18. [Namespaces. Uso de cgroups.](#)
19. [Tutorial: Espacio de nombres en Linux](#)
20. [Linux containers primitives: mount namespaces and information leaks](#)

21. [Mount namespaces and shared subtrees](#)
22. [*Build a container by hand: the mount namespace*](#)
23. [Stackexchange: what is a bind mount?](#)
24. [Understanding Bind Mounts](#)
25. [Arch Linux Wiki: Tmpfs](#)
26. [CloudNativeLab: Mount namespaces](#)
27. [Kernel.org: shared subtrees](#)
28. [Linux manual page: mount](#)
29. [Identificador de procesos \(*process id*\)](#)
30. [*Linux PID namespaces work with containers*](#)
31. [What are namespaces cgroups how do they work](#)
32. [*Time namespaces coming to linux*](#)
33. [*Time namespaces*](#)
34. [*Container is a lie. Namespaces*](#)
35. [What is chroot jail and How to Use it?](#)
36. [Open Container Initiative \(OCI\)](#)
37. [ArchLinux Wiki: Linux Containers](#)
38. [containerd](#)
39. [Wikipedia: Union Mount](#)
40. [Wikipedia: Overlayfs](#)
41. [Ubuntu LXC](#)
42. [ArchLinux Wiki: Xorg](#)
43. [How to create unprivileged LXC container](#)
44. [Docker overview](#)
45. [A beginner's guide to Docker - how to create your first Docker application](#)
46. [Play with Container Network Interface](#)
47. [Mininet: An Instant Virtual Network on your Laptop \(or other PC\)](#)
48. [OpenDaylight \(ODL\)](#)
49. [Github: POX](#)

50. [Github: NOX](#)
51. [Github: Ryu](#)
52. [Open Networking Operating System \(ONOS\)](#)
53. [Ryu Doc: Topology Viewer](#)
54. [Github: containernet](#)

Imágenes

- Figura 2.1: [NFV white paper](#)
- Figura 2.2: [The Northbound API- A Big Little Problem](#)
- Figura 2.3: [Introducción a contenedores Kubernetes y Openshift](#)
- Figura 3.3: [FS.COM QinQ Operation](#)
- Figura 3.6: [Wikipedia: TUN/TAP](#)
- Figura 4.4: [Namespaces y API de acceso.](#)
- Figura 5.7: [Linux Containers Logotipo \(LXC\)](#)
- Figura 5.13: [Imagen Docker Architecture](#)
- Figura 5.14: [Logotipo Docker](#)
- Figura 6.13: [Logotipo containernet](#)

Apéndice A

Anexos

A.1 Código asociado a topologías de red

Topología simple en **mininet**

Ejemplo A.1: Código topología simple utilizando mininet (`basic.py`)

```
1 from mininet.net import Mininet
2 from mininet.topo import Topo
3
4 topo = Topo()           # Empty topology
5 topo.addSwitch("s1")    # Adding switch
6 topo.addHost("h1")      # Add Host
7 topo.addHost("h2")      # Add Host
8 topo.addLink("h1", "s1") # Link host to switch
9 topo.addLink("h2", "s1")
10
11 net = Mininet(topo)     # Start Mininet
12 net.start()
13 net.pingAll()
14 net.iperf()
15 net.stop()
16
```

Topología simple con limitación de «recursos» en **mininet**

Ejemplo A.2: Código topología simple con limitación de «recursos» en mininet (`limit.py`)

```
1 from mininet.net import Mininet
2 from mininet.log import setLogLevel
3 from mininet.cli import CLI
4 from mininet.node import CPULimitedHost
5 from mininet.link import TCLink
6
7 def limit_perfTest():
8     net = Mininet()
9
10    # Host 1: CPU at 10%
11    h1 = net.addHost('h1', cls=CPULimitedHost, cpu=0.1)
12    h2 = net.addHost('h2')
```

```

13
14 s1 = net.addSwitch('s1')
15 c0 = net.addController('c0')
16
17 # Link h1<->s1: BW 5mbps & delay 80ms
18 net.addLink(h1,s1, cls=TCLink, bw=5, delay='80ms')
19 net.addLink(h2,s1)
20
21 net.start()
22 net.pingAll()
23 net.iperf()
24 CLI(net)
25 net.stop()
26
27 if __name__ == '__main__':
28     setLogLevel('info')
29     limit_perftest()
30

```

Topología simple en containernet

Ejemplo A.3: Código topología simple en containernet (containernet_example.py)

```

1 #!/usr/bin/python
2 """
3 This is the most simple example to showcase Containernet.
4 """
5 from mininet.net import Containernet
6 from mininet.node import Controller
7 from mininet.cli import CLI
8 from mininet.link import TCLink
9 from mininet.log import info, setLogLevel
10 setLogLevel('info')
11
12 net = Containernet(controller=Controller)
13 info('*** Adding controller\n')
14 net.addController('c0')
15 info('*** Adding docker containers\n')
16 d1 = net.addDocker('d1', ip='10.0.0.251', dimage="ubuntu:trusty")
17 d2 = net.addDocker('d2', ip='10.0.0.252', dimage="ubuntu:trusty")
18 info('*** Adding switches\n')
19 s1 = net.addSwitch('s1')
20 s2 = net.addSwitch('s2')
21 info('*** Creating links\n')
22 net.addLink(d1, s1)
23 net.addLink(s1, s2, cls=TCLink, delay='100ms', bw=1)
24 net.addLink(s2, d2)
25 info('*** Starting network\n')
26 net.start()
27 info('*** Testing connectivity\n')
28 net.ping([d1, d2])
29 info('*** Running CLI\n')
30 CLI(net)
31 info('*** Stopping network\n')
32 net.stop()
33

```