Implementación de un laboratorio mediante contenedores

Implementation of a laboratory using containers

Mario Román Dono

UNIVERSIDAD
COMPLUTENSE
MADRID

Trabajo Fin de Grado

Grado en Ingeniería Informática
Facultad de Informática

Dirigido por
Inmaculada Pardines Lence y Marcos Sánchez-Élez Martín

Madrid, 2021–2022

Implementación de un laboratorio mediante contenedores

Memoria para el Trabajo de Fin de Grado

Mario Román Dono

Dirigido por

Inmaculada Pardines Lence y Marcos Sánchez-Élez Martín

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2021 – 2022

Agradecimientos

Normalmente este espacio suele estar reservado para agradecer a todas aquellas personas importantes en la vida del autor que le han acompañado y ayudado a lo largo de la elaboración del TFG, y si lo hacemos más extenso, a lo largo de todo el grado: padres, hermanos, otros familiares, amigos del instituto, compañeros de facultad... Y, obviamente, en mi caso no es para menos, y sería egoísta por mi parte no acordarme de ellos cuando me han corregido trabajos, me han dado sugerencias cuando lo necesitaba o, simplemente, me han apoyado durante estos cuatro años.

Pero pienso que, como este trabajo significa el fin de mi etapa como estudiante —al menos por el momento—, lo que realmente tiene sentido es que este espacio esté dedicado especialmente a las siguientes personas:

Lola, Javier, Luis, Julia, Sergio, Ana, Juan, Julio, Clara, Gonzalo, Raquel, Juan Carlos y, cómo no, Marcos e Inma.

Todos ellos, profesores que he tenido a lo largo de toda mi vida, desde primaria hasta este último año de carrera. Ya sea por haberse esforzado para facilitarme las cosas en momentos complicados, por haber logrado trasmitir sus conocimientos para que me pudiera convertir en una persona más completa o, sencillamente, por haber hecho que vaya a guardar innumerables buenos recuerdos de mi paso por el colegio, el instituto y la universidad.

Por estos motivos, muchas gracias por todo, porque sin vosotros no habría podido llegar hasta donde estoy hoy, terminando este trabajo.

Resumen

Los laboratorios docentes son una parte esencial de la formación de los estudiantes en las titulaciones de Informática, ya que permiten poner en práctica los conocimientos aprendidos en las clases teóricas. En muchas de las asignaturas de estas titulaciones se requiere el uso de virtualización para poder trabajar en entornos aislados, con el fin de poder probar programas o comandos peligrosos sin poner en riesgo la integridad del laboratorio, así como para poder recrear infraestructuras de red completas. Habitualmente, la virtualización se efectúa a través de máquinas virtuales, lo que supone una gran sobrecarga y un bajo rendimiento durante la realización de las prácticas. Por ello, en este Trabajo Fin de Grado se pretende evaluar la puesta en funcionamiento de un laboratorio docente mediante contenedores. Se realizará un estudio del funcionamiento de los contenedores, así como de las necesidades que poseen los laboratorios docentes, para así diseñar una propuesta, utilizando Docker como programa de gestión de los contenedores, que se adecúe a los requisitos exigidos y que permita realizar las prácticas de forma satisfactoria. Esta propuesta será evaluada en una prueba de concepto consistente en realizar una práctica de la asignatura Redes y Seguridad II del Grado en Ingeniería Informática de la Universidad Complutense de Madrid. Además, se comparará esta propuesta con el despliegue actual para determinar si efectivamente se produce una mejora de rendimiento, y se realizará un análisis de seguridad que verifique que no existen riesgos derivados de la ejecución de contenedores.

Palabras clave: contenedores, Docker, laboratorios docentes, virtualización.

Abstract

Teaching laboratories are an essential part of the education of students in Computer Science degrees, since they allow them to put into practice the knowledge learned in theoretical classes. Many of the subjects of these degrees require the use of virtualization to be able to work in isolated environments, in order to test dangerous programs or commands without compromising the integrity of the laboratory, as well as to be able to recreate complete network infrastructures. Usually, virtualization is done through virtual machines, which means a great overload and a low performance during the realization of the practices. For this reason, the aim of this Final Degree Project is to evaluate the implementation of a teaching laboratory using containers. A study of the functioning of the containers will be carried out, as well as of the needs of the teaching laboratories, in order to design a proposal, using Docker as the container management program, that meets the requirements demanded and that allows to perform the practices in a satisfactory way. This proposal will be evaluated in a proof of concept consisting of carrying out a practice of the subject Computer Networks Security II of the Degree in Computer Science Engineering of the Complutense University of Madrid. In addition, this proposal will be compared with the current deployment to determine if there is indeed a performance improvement, and a security analysis will be performed to verify that there are no risks derived from the execution of containers.

Keywords: containers, Docker, teaching laboratories, virtualization.

Sobre TEFLONX

Teflon X(cc0 1.0(documentación) MIT(código))es una plantilla de LaTeX creada por David Pacios Izquierdo con fecha de Enero de 2018. Con atribuciones de uso CC0.

Esta plantilla fue desarrollada para facilitar la creación de documentación profesional para Trabajos de Fin de Grado, Trabajos de Fin de Máster o Doctorados. La versión usada es la X

V:X Overleaf V2 with XeLaTeX, margin 1in, bib

Contacto
Autor: David Pacios Izquierdo
Correo: dpacios@ucm.es
ASCII: ascii@ucm.es
Despacho 110 - Facultad de Informática

Índice general

Página

1. Introducción **1**
1.1. Motivación 1
1.2. Objetivos del trabajo 3
1.3. Estado del arte 3
1.4. Planificación de las tareas 4
1.5. Organización de la memoria 4

1. Introduction **6**
1.1. Motivation 6
1.2. Objective of the project 8
1.3. State-of-the-art 8
1.4. Plannning of the tasks 9
1.5. Organization of the report 9

2. Virtualización **11**
2.1. Orígenes 11
2.2. Tipos de virtualización 12
2.2.1. Virtualización completa 12
2.2.2. Paravirtualización 13
2.2.3. Virtualización a nivel de sistema operativo 14
2.3. Virtualización a nivel de sistema operativo y contenedores 14
2.3.1. Repaso histórico 15
2.3.2. *Software* de contenedores disponible 17
2.3.3. Ejemplos de casos de uso 19
2.3.4. Rendimiento 21
2.3.5. Seguridad 22

3. Implementación basada en las necesidades de los laboratorios docentes **25**
3.1. Situación actual 25
3.2. Características de los contenedores de reemplazo 27
3.2.1. Diseño de los contenedores 27
3.2.2. Imagen base de los contenedores 28
3.2.3. Conexión de red entre los contenedores 29
3.2.4. Uso de aplicaciones de interfaz gráfica 30
3.2.5. Compartición de archivos entre contenedores 31
3.3. Elección del *software* de contenedores 31

3.4. Despliegue de los contenedores para las prácticas 32
3.5. Imágenes disponibles para las prácticas 34
3.6. Ejecución de los contenedores: ¿nativa o sobre una máquina virtual? . . . 34
3.6.1. Elección del hipervisor 35
3.6.2. Elección del sistema operativo de la máquina virtual 36

4. Creación del entorno de contenedores 41
4.1. Instalación y configuración de Alpine 41
4.1.1. Creación de la máquina virtual 41
4.1.2. Instalación básica de Alpine 41
4.1.3. Configuración gráfica 42
4.1.4. Instalación de Docker y ajustes adicionales 42
4.2. Desarrollo de la herramienta de despliegue de contenedores 45
4.2.1. Versión preliminar de la herramienta 45
4.2.2. Versión final de la herramienta 45

5. Resultados 60
5.1. Entorno de pruebas 60
5.2. Prueba de concepto 60
5.3. Evaluación del rendimiento 66
5.3.1. Espacio de disco utilizado 67
5.3.2. Tiempo de creación 68
5.3.3. Uso de CPU, memoria y entrada/salida 70
5.3.4. Discusión 71
5.4. Análisis de seguridad 71

6. Conclusiones y trabajo futuro 74
6.1. Conclusiones 74
6.2. Trabajo futuro 75

6. Conclusions and future work 76
6.1. Conclusions 76
6.2. Future work 77

7. Bibliografía y enlaces de referencia 78

Apéndice A. Código de la versión preliminar de la herramienta de despliegue 89

Apéndice B. Manual de uso de la herramienta de despliegue de contenedores 99

Apéndice C. Resultado de Docker Bench for Security 101

Capítulo 1

Introducción

1.1. Motivación

Los laboratorios docentes son una parte esencial de la formación de los estudiantes en las titulaciones de Informática, ya que permiten poner en práctica los conocimientos aprendidos en las clases teóricas. Desde el punto de vista más básico, un laboratorio docente no es más que una sala de ordenadores situada en una facultad de Informática, cuyos sistemas pueden ser utilizados por los estudiantes para realizar las prácticas de las asignaturas que estén cursando. Sin embargo, dependiendo de cada asignatura, las características de los laboratorios pueden variar significativamente.

Por ejemplo, para realizar prácticas en asignaturas de programación, los estudiantes pueden trabajar directamente con los entornos de desarrollo y los compiladores o intérpretes que se encuentren instalados en los ordenadores del laboratorio. Por el contrario, en aquellas asignaturas relacionadas con la seguridad de sistemas y redes, es necesario emplear algún tipo de aislamiento que permita que se puedan probar programas o comandos peligrosos sin poner en riesgo la integridad del laboratorio.

El método más comúnmente utilizado en la actualidad para atajar este problema es recurrir a la virtualización, en concreto, al uso de máquinas virtuales, que permiten la creación de subsistemas aislados entre sí y del propio ordenador anfitrión. Esto abre un amplio abanico de oportunidades: desde la posibilidad de probar un sistema operativo diferente hasta la ejecución de *software* malicioso o la reproducción de ataques informáticos con el fin de estudiar su comportamiento, pasando por la instalación de nuevos programas en caso de no tener permisos para hacerlo sobre la máquina anfitriona. Por otra parte, dado que es posible ejecutar varias máquinas virtuales dentro de un mismo sistema, se pueden llegar a construir infraestructuras de red completas, recreando clientes, *routers* y servidores de manera fiel a como se comportarían en un entorno real. Todo ello mientras se garantiza que la infraestructura subyacente no se llegará a ver afectada en ningún caso.

No obstante, las máquinas virtuales tienen una notable desventaja, y es que su utilización conlleva una grave penalización en el rendimiento del sistema debido a que es necesario virtualizar completamente el sistema operativo de cada entorno. La sobrecarga es todavía mayor en aquellos casos donde sea necesario ejecutar más de una máquina virtual al mismo

tiempo, como pueden ser aquellas prácticas que modelen una infraestructura de red. Este problema se vio gravemente reflejado durante la crisis del COVID-19 ya que, tras el cambio a una modalidad de docencia no presencial, los estudiantes se vieron obligados a llevar a cabo las prácticas en sus ordenadores personales, que en muchos casos no disponían de una potencia similar a la de los ordenadores disponibles en las facultades, por lo que su experiencia de uso fue bastante pobre o incluso no pudieron llegar a completar las prácticas.

Por este motivo, sería muy conveniente encontrar una alternativa que mantenga un aislamiento similar al ofrecido por las máquinas virtuales pero que, al mismo tiempo, no suponga una bajada de rendimiento tan acusada. Una posibilidad sería recurrir al *cloud computing*, ya sea a través de una nube pública como la de AWS [1] o Azure [2], o construir una nube privada situada en las instalaciones de cada universidad. Posteriormente, los estudiantes se conectarían a dicha nube a través de acceso remoto, tanto desde los ordenadores del propio laboratorio como desde sus portátiles personales. Con ello se conseguiría un aumento del rendimiento debido a que ya no se dependería de los recursos *hardware* de la máquina que utilice cada estudiante sino solamente de las características de la nube, e incluso se podrían escalar tanto hacia arriba como hacia abajo los recursos asignados a cada entorno según el uso que se esté realizando en cada momento.

Existen ya algunos trabajos que exploran esta posibilidad [3] [4]. Sin embargo, su mayor inconveniente es el coste que supondría. Si se optase por una nube pública, se pasaría a un modelo donde todos los meses se tendrían que realizar pagos por los recursos utilizados en las prácticas. Por otro lado, en el caso de la nube privada, esto se evitaría al convertirse en el propietario del *hardware*, pero el desembolso inicial para crear el sistema podría ser muy elevado, amén de los gastos de mantenimiento en los sucesivos años.

La otra alternativa podría ser la virtualización a nivel de sistema operativo, que consiste en que los entornos virtualizados —denominados contenedores [5] en este caso— comparten el núcleo del sistema operativo con el ordenador anfitrión, de forma que no es necesario virtualizar un sistema operativo completo por cada entorno, a diferencia de lo que sucede con las máquinas virtuales. De esta forma, tanto el rendimiento del sistema como los tiempos de creación y encendido de los entornos son mejorados considerablemente, sin incurrir en ningún coste adicional, ni siquiera de mantenimiento.

No obstante, las ventajas del uso de contenedores no solo se reducen a aquellas asignaturas donde sea necesario utilizar la virtualización por necesidad:

- En aquellas asignaturas donde se realicen proyectos de programación podría ser interesante cambiar a una metodología basada en contenedores con el fin de prevenir posibles errores originados por el uso de versiones distintas de lenguajes o bibliotecas entre los ordenadores de los estudiantes y los de los profesores.
- Debido al propio diseño de los contenedores, el proceso de creación y despliegue suele ser más ágil que en el caso de las máquinas virtuales, lo cual sería más beneficioso de cara a su mantenimiento por parte de los técnicos de laboratorio.
- Estos entornos permiten una mayor portabilidad, lo que abre la posibilidad de que los estudiantes podrían empezar las prácticas en los ordenadores del laboratorio y finalizarlas posteriormente en sus ordenadores personales de una forma sencilla.

- Finalmente, es necesario destacar que el uso de contenedores es cada vez más habitual en la industria informática, por lo que podría ser interesante aprovechar esta situación para que los estudiantes aprendan cómo utilizar una herramienta que puede ser requerida cuando comiencen a desarrollar su actividad profesional.

Por todo ello, en este trabajo se pretende evaluar la viabilidad de la implementación de un laboratorio docente utilizando contenedores, analizando sus ventajas y desventajas.

1.2. Objetivos del trabajo

El objetivo principal de este trabajo es evaluar la puesta en funcionamiento de un laboratorio docente mediante contenedores en lugar de máquinas virtuales, comprobando si es posible llevar a cabo las prácticas de las asignaturas de sistemas y redes de las titulaciones de la Facultad de Informática de la Universidad Complutense de Madrid y analizando su comportamiento en términos de rendimiento y seguridad.

Específicamente, se pretende analizar qué características deben tener los contenedores para servir como reemplazo de las máquinas virtuales, estudiar qué *software* de contenedores es el más apropiado para llevar a cabo este proyecto, determinar qué entorno resulta más beneficioso para la ejecución de los contenedores —ejecución nativa o sobre una máquina virtual— y, finalmente, comparar la propuesta diseñada con el despliegue actual, tanto a nivel de rendimiento como de seguridad.

1.3. Estado del arte

La problemática descrita en la sección 1.1 es común a casi todos aquellos laboratorios donde se emplea virtualización clásica, por lo que existen bastantes trabajos relacionados que exploran las distintas posibilidades que permitan reemplazar el uso de máquinas virtuales, ya sea a través del *cloud computing* o mediante el uso de contenedores. Centrándonos en esta segunda alternativa, en [6] se expone el proceso de migración de máquinas virtuales a contenedores en Tele-Lab, una plataforma de aprendizaje en línea especializada en ciberseguridad. La implementación de los contenedores se llevó a cabo de forma satisfactoria y se pudo aumentar la escalabilidad y el rendimiento del entorno sin recurrir a otras alternativas más costosas como la nube pública. Sin embargo, en dicho trabajo era necesario evaluar más a fondo y corregir la seguridad de los contenedores, ya que estos se ejecutaban en modo privilegiado.

No obstante, el proyecto más avanzado en este campo es, sin duda, Labtainers [7] [8]. Este proyecto permite el despliegue automatizado de contenedores distribuidos en redes virtuales para la práctica de ejercicios de ciberseguridad. Pero, a diferencia del trabajo anterior, en el que los usuarios debían conectarse a una platafoma para acceder a los entornos virtualizados, en este caso los contenedores se ejecutan en la máquina del usuario, ya sea sobre el propio sistema operativo del ordenador o encapsulados en una máquina virtual. Este *framework* incluye más de 50 ejercicios ya preparados, entre los que destacan prácticas de vulnerabilidades de *buffer overflow*, inyección SQL o configuración de redes privadas virtuales (VPN). Además, los ejercicios pueden ser personalizados para cada estudiante y corregidos automáticamente en base a las acciones que se hayan realizado en los contenedores. También es posible crear nuevos ejercicios, pero no se pueden desplegar

contenedores que no estén asociados a un ejercicio en concreto, lo que limita en cierta medida los casos en los que se puede usar.

Por último, CvLabs [9] también permite realizar ejercicios de ciberseguridad sobre contenedores, pero vuelve a optar por el uso de un servidor al que los usuarios se deben conectar por Internet. Los contenedores se crean a través de una interfaz web, y a nivel interno se gestionan mediante un orquestador de contenedores para controlar el escalado en todo momento. CvLabs puede ser desplegado sobre una máquina física, en una máquina virtual, en una nube privada o en una nube pública.

En este trabajo se va a seguir un enfoque similar al de Labtainers, por el que los contenedores se ejecutarán en las máquinas de los laboratorios o en los ordenadores de los estudiantes. De esta forma, se evita el tener que disponer de una plataforma adicional, lo que supone un beneficio, tanto en gastos de infraestructura, como en evitar que los estudiantes deban estar siempre conectados al servidor correspondiente.

No obstante, no se continuará con el enfoque rígido de dicho proyecto, en el que los profesores de cada asignatura deben crear la estructura de contenedores previamente. Se forzará a que sean los propios estudiantes quienes definan la topología de red correspondiente, realicen los cambios necesarios en la configuración de los contenedores e instalen el *software* requerido en cada práctica. Así, los estudiantes podrán fortalecer sus conocimientos sobre sistemas operativos y redes, además de posibilitar el uso de estos entornos en un rango de asignaturas más variado.

1.4. Planificación de las tareas

En primer lugar, se realizó un estudio del funcionamiento de los contenedores y se analizaron las características que tenían los laboratorios docentes, para poder diseñar una propuesta que se adecuase a sus requisitos.

Una vez se establecieron estas exigencias, se llevó a cabo una primera aproximación que tuvo como objetivo determinar si era posible realizar una práctica de una asignatura de sistemas y redes usando contenedores de manera satisfactoria. Se estudió con qué opciones debían crearse los contenedores, así como las propiedades con las que se debía configurar el entorno de ejecución.

Tras efectuar esta primera prueba, se revisó la implementación de la solución con el objetivo de ver si era posible aplicar mejoras funcionales, de seguridad o de rendimiento.

Finalmente, se comparó la versión final de la propuesta con respecto al despliegue actual de máquinas virtuales para evaluar si efectivamente se producía una ganancia de rendimiento. Asimismo, también se realizó un análisis de seguridad que pretendía determinar si existían riesgos derivados del uso de contenedores y que verificase que la solución diseñada cumplía con los requisitos deseados.

1.5. Organización de la memoria

Esta memoria se organiza en seis capítulos. El primero incluye la motivación de este trabajo, su objetivo, el estado del arte y la planificación de las tareas llevadas a cabo a lo largo del proyecto.

El segundo capítulo contiene una introducción teórica sobre la virtualización y sobre los contenedores. Se describen las tecnologías de virtualización más conocidas en la actualidad, con especial énfasis en la virtualización a nivel de sistema operativo. Además, se explican los posibles usos de esta técnica y ciertas consideraciones con respecto a su rendimiento y seguridad.

El tercer capítulo muestra en detalle cuáles son las características que poseen los laboratorios docentes en la Facultad de Informática de la Universidad Complutense de Madrid. En concreto, se explican qué propiedades tiene el *software* de virtualización y las máquinas virtuales que se utilizan en la actualidad para la realización de las prácticas. A continuación, se especifican cómo deben ser los contenedores que servirán de reemplazo a las máquinas virtuales, el programa de gestión de contenedores que se utilizará y las características del entorno donde se desplegarán los contenedores.

En el cuarto capítulo se detalla el proceso de creación del entorno de contenedores. En primer lugar, se expone cómo se llevó a cabo la instalación y configuración de la máquina virtual de Alpine para poder ejecutar Docker correctamente. Posteriormente, se explican las características de la herramienta desarrollada en este proyecto para el despliegue y gestión automatizada de los contenedores.

En el quinto capítulo se detallan los resultados de la prueba de concepto llevada a cabo, consistente en la realización de una práctica de la asignatura Redes y Seguridad II del Grado en Ingeniería Informática de la Universidad Complutense de Madrid, usando varios contenedores. También se incluye una comparativa de rendimiento entre el uso de contenedores y el entorno basado en máquinas virtuales, así como un análisis de seguridad de la solución diseñada.

Finalmente, en el sexto capítulo se exponen las conclusiones de este trabajo y el posible trabajo futuro que se podría realizar a partir de los avances conseguidos en este proyecto.

Chapter 1

Introduction

1.1. Motivation

Teaching laboratories are an essential part of the education of students in Computer Science degrees, since they allow them to put into practice the knowledge learned in theoretical classes. From the most basic point of view, a teaching laboratory is nothing more than a computer room located in a faculty of Computer Science, whose systems can be used by students to perform the practices of the subjects they are taking. However, depending on each subject, the characteristics of the laboratories can vary significantly.

For example, for programming courses, students can work directly with the development environments and compilers or interpreters that are installed on the laboratory computers. On the other hand, in those subjects related to system and network security, it is necessary to employ some form of isolation that allows dangerous programs or commands to be tested without compromising the integrity of the laboratory.

The most common method currently used to tackle this problem is to resort to virtualization, specifically using virtual machines, which allow the creation of subsystems isolated from each other and from the host computer itself. This opens up a wide range of opportunities: from the possibility of testing a different operating system to the execution of malicious software or the reproduction of computer attacks in order to study their behavior, as well as the installation of new programs in the event of not having permission to do so on the host machine. Moreover, since it is possible to run several virtual machines within the same system, complete network infrastructures can be built, recreating clients, routers and servers in a way that is faithful to how they would behave in a real environment. All this while guaranteeing that the underlying infrastructure will not be affected in any way.

Nevertheless, virtual machines have a notable disadvantage, and that is that their use entails a serious penalty in system performance due to the need to completely virtualize the operating system of each environment. The overhead is even greater in those cases where it is necessary to run more than one virtual machine at the same time, such as those practices that model a network infrastructure. This problem was seriously reflected during the COVID-19 crisis since, after the change to a remote teaching modality, students were forced to carry out the practices on their personal computers, which in many cases did not

have a similar power to that of the computers available in the faculties, so their experience of use was quite poor or they were not even able to complete the practices.

For this reason, it would be highly desirable to find an alternative that maintains a similar isolation to that offered by virtual machines but, at the same time, does not cause such a sharp drop in performance. One possibility would be to use cloud computing, either through a public cloud such as AWS [1] or Azure [2], or to build a private cloud located on the premises of each university. Students would then connect to that cloud via remote access, either from the lab's own computers or from their personal laptops. This would increase performance because it would no longer depend on the hardware resources of the machine used by each student but only on the characteristics of the cloud, and the resources assigned to each environment could even be scaled up and down according to the use that is being made at any given time.

There is already some work exploring this possibility [3] [4]. However, the biggest drawback is the cost involved. If the public cloud is chosen, it would mean moving to a model where payments would have to be made every month for the resources used in the practices. On the other hand, in the case of the private cloud, this would be avoided by becoming the owner of the hardware, but the initial outlay to create the system could be very high, in addition to the maintenance costs in subsequent years.

The other alternative could be OS-level virtualization, which consists of virtualized environments—called containers [5] in this case—sharing the operating system kernel with the host computer, so that it is not necessary to virtualize a complete operating system for each environment, unlike what happens with virtual machines. This way, both the performance of the system and the creation and start-up times of the environments are considerably improved, without incurring in any additional cost, not even maintenance costs.

However, the advantages of using containers are not only limited to those subjects where virtualization is necessary:

- In those subjects where programming projects are developed, it could be interesting to change to a container-based methodology in order to prevent possible errors caused by the use of different versions of languages or libraries between the students' and teachers' computers.
- Due to the design of the containers themselves, the creation and deployment process is usually more agile than in the case of virtual machines, which would be more beneficial in terms of maintenance by lab technicians.
- These environments provide greater portability, which opens up the possibility that students could easily start the practices on the laboratory computers and finish them later on their personal computers.
- Finally, it should be noted that the use of containers is becoming more and more common in the IT industry, so it could be interesting to take advantage of this situation so that students could learn how to use a tool that may be required when they begin to develop their professional activity.

For all these reasons, this project aims to evaluate the feasibility of implementing a teaching laboratory using containers, analyzing their advantages and disadvantages.

1.2. Objective of the project

The objective of this project is to evaluate the implementation of a teaching laboratory using containers instead of virtual machines, checking if it is possible to carry out the practices of the systems and networks subjects of the degrees of the Faculty of Computer Science of the Complutense University of Madrid and analyzing its behavior in terms of performance and security.

Specifically, we intend to analyze what characteristics the containers must have to serve as a replacement for virtual machines, study which container software is the most appropriate to implement this project, determine which environment is more beneficial for the execution of the containers —native execution or on a virtual machine— and finally, compare the designed proposal with the current deployment, both at performance and security level.

1.3. State-of-the-art

The problem described in section 1.1 is common to almost all laboratories where classical virtualization is used, therefore there are many related works that explore the different possibilities to replace the use of virtual machines, either through cloud computing or through the use of containers. Focusing on this second alternative, [6] presents the migration process from virtual machines to containers in Tele-Lab, an e-learning platform specialized in cybersecurity. The implementation of containers was carried out successfully and it was possible to increase the scalability and performance of the environment without resorting to other more expensive alternatives such as the public cloud. However, in that work it was necessary to further evaluate and correct the security of the containers, as they were running in privileged mode.

However, the most advanced project in this field is undoubtedly Labtainers [7] [8]. This project allows the automated deployment of distributed containers in virtual networks for the practice of cybersecurity exercises. But unlike the previous work, in which users had to connect to a platform to access the virtualized environments, in this case the containers run on the user's machine, either on the computer's own operating system or encapsulated in a virtual machine. This framework includes more than 50 ready-made exercises, among which are practices on buffer overflow vulnerabilities, SQL injection or virtual private network (VPN) configuration. In addition, the exercises can be customized for each student and automatically corrected based on the actions performed in the containers. It is also possible to create new exercises, but it is not possible to deploy containers that are not associated with a particular exercise, which limits to some extent the cases in which it can be used.

Finally, CvLabs [9] also allows to perform cybersecurity exercises on containers, but again opts for the use of a server to which users must connect via the Internet. Containers are created through a web interface, and internally are managed by a container orchestrator to control scaling at all times. CvLabs can be deployed on a physical machine, in a virtual machine, in a private cloud or in a public cloud.

In this project we will follow a similar approach to Labtainers, whereby the containers will be run on the laboratory machines or on the students' computers. This avoids the

requirement for an additional platform, which is a benefit both in terms of infrastructure costs and in terms of avoiding the need for students to always be connected to the corresponding server.

However, the rigid approach of this project, in which the teachers of each subject must create the container structure beforehand, will not be maintained. The students themselves will be forced to define the corresponding network topology, make the necessary changes in the containers' configuration and install the software required in each practice. Thus, students will be able to strengthen their knowledge of operating systems and networks, in addition to being able to use these environments in a more varied range of subjects.

1.4. Plannning of the tasks

First of all, a study of the functioning of the containers was conducted and the characteristics of the teaching laboratories were analyzed in order to design a proposal that would meet their requirements.

Once these requirements were established, a first approach was carried out with the objective of determining whether it was possible to perform a practice of a systems and networks course using containers in a satisfactory way. The options with which the containers should be created were studied, as well as the properties with which the execution environment should be configured.

After this first test, the implementation of the solution was reviewed to see if it was possible to apply functional, security or performance improvements.

Finally, the final version of the proposal was compared with the current deployment of virtual machines to evaluate whether there was indeed a performance gain. A security analysis was also performed to determine if there were any risks derived from the use of containers and to verify that the designed solution complied with the desired requirements.

1.5. Organization of the report

This report is organized in six chapters. The first chapter includes the motivation for this work, its objective, the state-of-the-art and the planning of the tasks carried out throughout the project.

The second chapter contains a theoretical introduction about virtualization and containers. The most popular virtualization technologies currently available are described, with special emphasis on OS-level virtualization. In addition, the possible uses of this technique and certain considerations regarding its performance and security are explained.

The third chapter shows in detail the characteristics of the teaching laboratories in the Faculty of Computer Science of the Complutense University of Madrid. Specifically, the properties of the virtualization software and the virtual machines that are currently used for the realization of the practices are explained. Then, it is specified how the containers that will serve as replacement for the virtual machines should be designed, the container

management program that will be used and the characteristics of the environment where the containers will be deployed.

The fourth chapter details the process of creating the containers environment. First of all, it is explained how the installation and configuration of the Alpine virtual machine was performed in order to be able to run Docker correctly. Subsequently, the features of the tool developed in this project for the automated deployment and management of containers are explained.

The fifth chapter details the results of the proof of concept carried out, which consisted in the implementation of a practice of the Computer Networks Security II subject of the Degree in Computer Science Engineering of the Complutense University of Madrid, using several containers. A performance comparison between the use of containers and the environment based on virtual machines is also included, as well as a security analysis of the designed solution.

Finally, the sixth chapter presents the conclusions of this project and the possible future work that could be done based on the progress achieved in this project.

Capítulo 2

Virtualización

2.1. Orígenes

Antes de explicar qué son los contenedores y por qué pueden resultar atractivos como alternativa para implementar laboratorios docentes, es interesante realizar un breve repaso de la tecnología en la que están basados, la virtualización, y cómo ha ido evolucionando a lo largo de la historia hasta llegar al punto en el que se encuentra en la actualidad.

La virtualización consiste en el reparto de los recursos *hardware* de un sistema informático mediante *software* con el objetivo de crear uno o varios subsistemas aislados entre sí y del sistema anfitrión. Esto permite que dentro de una única máquina se puedan generar varios entornos virtuales diferentes que se comportan de idéntica manera a como si se ejecutasen en una máquina física.

Dependiendo del tipo de virtualización, el reparto y particionado de *hardware* se realiza de formas distintas, pero en general todas coinciden en que, gracias a una capa de *software* conocida comúnmente como hipervisor, se consigue que todos los dispositivos del sistema sean vistos por el sistema virtual como si estuvieran disponibles únicamente para él solo, con independencia del uso adicional que realice el sistema anfitrión u otros posibles sistemas virtualizados que existan en la máquina.

Esta tecnología tuvo sus orígenes en el año 1968 de la mano de IBM con la creación del sistema operativo CP/CMS [10]. Estaba fundamentado en la idea de tiempo compartido, que permitía que múltiples usuarios pudieran llevar a cabo sus cargas de trabajo de forma aislada sobre una misma máquina mediante la simulación de un sistema operativo completo para cada uno de ellos. Este concepto proporcionaba dos importantes ventajas: para los propietarios de los *mainframes* fabricados por IBM, implicaba que sus sistemas pudieran ser aprovechados en aquellos momentos en los que no estuvieran siendo utilizados; por otra parte, abría la posibilidad de que aquellos usuarios que no podían realizar el enorme desembolso que suponía la compra de uno de estos ordenadores pudieran acceder de una forma más económica a estos.

Aunque la idea del tiempo compartido siguió madurando a lo largo de la década de los 70, llegando a ser soportada por más ordenadores y por sistemas operativos como Multics y su sucesor Unix, a partir de los años 80 empezó a caer en desuso debido a la aparición

del ordenador personal, promoviendo que cada usuario pudiera realizar su trabajo en su máquina individual en lugar de que todos ellos se tuvieran que conectar a un único *mainframe* [10].

A finales de los años 90 y principios de los 2000 la virtualización experimentó un resurgimiento, siendo destacable el lanzamiento en 1999 de VMWare Workstation, el primer software capaz de virtualizar la arquitectura x86 [11]. A partir de entonces los avances realizados en este campo se sucedieron rápidamente, como el soporte *hardware* a la virtualización por parte de Intel y AMD con Intel VT-X y AMD-V en 2005 [12], [13] o el desarrollo de nuevos programas de virtualización ampliamente utilizados hoy en día, como por ejemplo Xen en 2003, creado en la Universidad de Cambridge [14]; KVM —siglas de Kernel-based Virtual Machine— en 2006, que incorporaba esta tecnología como un módulo del *kernel* de Linux [15]; o VirtualBox en 2007, que sería adquirido por Oracle poco tiempo después [16].

Gracias a estos progresos, la virtualización pudo empezar a ser más ampliamente utilizada y a ofrecer más posibilidades. Ya no servía únicamente para compartir el *hardware* de un sistema como en el caso del CP/CMS, sino que permitía utilizar sistemas operativos diferentes sin necesidad de modificar el ya existente, instalar aplicaciones no compatibles con el sistema anfitrión, probar *software* sin miedo a que el ordenador se vea dañado...

Uno de los casos más notables es el del *cloud computing*, fundamentado en gran medida en la virtualización. Por ejemplo, la mayoría de los servicios disponibles en el mayor proveedor de computación en la nube, Amazon Web Services, se basan en la creación de entornos virtualizados altamente elásticos y escalables dentro de sus servidores. A su vez, dos de sus servicios de computación sin servidor, AWS Lambda y AWS Fargate, utilizan Firecracker como creador y gestor de estos entornos virtualizados, empleando para ello KVM [17].

2.2. Tipos de virtualización

Una vez definida qué es la virtualización, podemos analizar los distintos tipos que existen, en qué se diferencian unos de otros y en qué situaciones puede ser más ventajoso optar por cada uno de ellos [18], [19]:

2.2.1. Virtualización completa

Esta técnica de virtualización simula completamente el *hardware* subyacente, de manera que los entornos virtualizados, también conocidos como máquinas virtuales, no tienen constancia de que están siendo ejecutados sobre una máquina anfitrión. En sus comienzos, esta tecnología utilizaba una doble táctica para ejecutar las instrucciones del sistema operativo invitado: las instrucciones no virtualizables eran traducidas por el hipervisor mediante un procedimiento conocido como traducción bi-

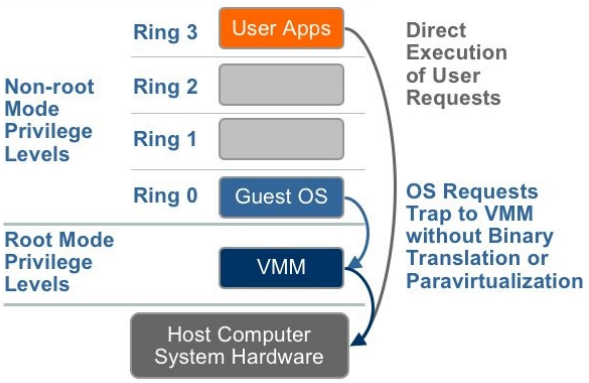

Figura 2.1: Funcionamiento de la virtualización completa con soporte de hardware. Obtenido de [18]

naria, mientras que las instrucciones del espacio de usuario se ejecutaban directamente sobre el procesador. Sin embargo, gracias a la aparición de las adaptaciones *hardware* de Intel y AMD mencionadas anteriormente —y obviando el caso del IBM System/370, que en 1972 ya se encontraba adaptado para la virtualización del CP/CMS [20]—, la traducción binaria fue reemplazada por un nuevo modo de ejecución en la CPU, de tal forma que aquellas instrucciones privilegiadas son capturadas directamente por el hipervisor, mejorando así el rendimiento. En la figura 2.1 podemos ver un esquema que muestra el funcionamiento de la virtualización completa asistida por *hardware*. Esta técnica de virtualización es empleada por la mayoría de aplicaciones de VMWare o por KVM, por ejemplo.

2.2.2. Paravirtualización

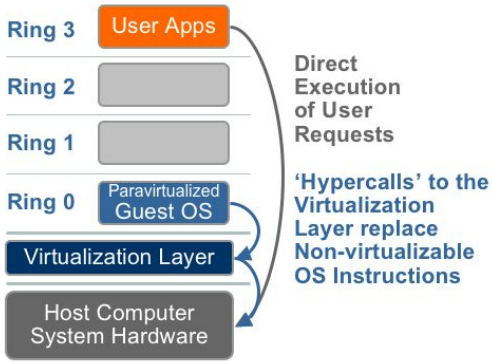

Figura 2.2: Funcionamiento de la paravirtualización. Obtenido de [18]

En este caso, las instrucciones no virtualizables se reemplazan por *hypercalls*, permitiendo la comunicación directa del sistema operativo invitado con el hipervisor, lo que mejora el rendimiento en algunas ocasiones. La figura 2.2 presenta cómo funciona la paravirtualización. Para poder emplear esta técnica, el núcleo del sistema operativo debe estar adaptado para la paravirtualización. En sus primeras etapas, esto suponía una importante limitación debido a que exigía que existiera una versión del *kernel* para cada hipervisor; no obstante, esto fue solventado en la versión 2.6.23 del núcleo de Linux gracias a la aparición de `paravirt-ops`, posibilitando que el mismo *kernel* pudiera ser ejecutado en cualquier hipervisor que admitiese paravirtualización así como en *bare metal* sin realizar ninguna modificación adicional [21]. En el caso de Windows, no es posible utilizarlo como sistema operativo invitado debido a que no soporta la paravirtualización [22]. Xen es el hipervisor más popular que utiliza exclusivamente esta tecnología. Además, algunos hipervisores de virtualización completa como VirtualBox, Hyper-V o VMWare Workstation habilitan la paravirtualización parcialmente para la realización de tareas específicas, obteniendo las ventajas de ambos enfoques.

Antes de pasar al tercer y último tipo de virtualización, es necesario especificar que, tanto en el caso de la paravirtualización como en el de la virtualización completa, los hipervisores se dividen en dos categorías [19]:

- Tipo 1: También denominados como hipervisores nativos, se caracterizan por que se ejecutan directamente sobre el *hardware* del sistema, y los recursos utilizados por las máquinas virtuales son asignados directamente por el hipervisor, que actúa como el sistema operativo anfitrión. Por este motivo, su eficiencia es superior debido a que no existen intermediarios. Algunos ejemplos son KVM [23], Xen [24], Microsoft Hyper-V [25] o VMWare ESXi [26].
- Tipo 2: En esta clase de hipervisores, conocidos como hipervisores alojados, la gestión del *hardware* es llevada a cabo por el sistema operativo de la máquina anfitrión, que actúa como intermediario entre el *hardware* y el hipervisor. Suelen ser

usados en entornos donde el rendimiento no es crítico, ya que la capa del sistema operativo anfitrión supone una sobrecarga adicional. Sin embargo, tienen como ventaja que suelen ser más fáciles de instalar y gestionar para usuarios menos experimentados que los de tipo 1. Las alternativas más populares actualmente son Oracle VM VirtualBox [16], VMWare Workstation Pro [27] y Parallels Desktop [28].

2.2.3. Virtualización a nivel de sistema operativo

Esta técnica es radicalmente diferente a las anteriores debido a que en este caso no existe un hipervisor ni máquinas virtuales, sino que los entornos virtualizados son instancias creadas a nivel de usuario que no incluyen un sistema operativo completo, ya que hacen uso del *kernel* del ordenador anfitrión para sus operaciones.

Los contenedores se basan en la virtualización a nivel de sistema operativo, siendo esta la tecnología principal de este trabajo. Es por ello que se dedica la siguiente sección a realizar una explicación más detallada, necesaria para entender mejor el alcance del trabajo desarrollado en este proyecto.

2.3. Virtualización a nivel de sistema operativo y contenedores

Como ya hemos expuesto, la virtualización a nivel de sistema operativo se basa en crear entornos aislados de la máquina anfitrión en el nivel de usuario, sin necesidad de virtualizar completamente el sistema operativo invitado ya que todos los entornos comparten el *kernel* anfitrión. Por otra parte, en esta clase de virtualización los hipervisores son sustituidos por aplicaciones denominadas motores que gestionan la creación y vida de los entornos. Dependiendo de la tecnología concreta, estos entornos reciben un nombre diferente —zonas, servidores privados virtuales, jaulas...—, pero actualmente el nombre más común es el de contenedores. En la figura 2.3 se puede ver la diferencia de diseño entre las máquinas virtuales y los contenedores, que reemplazan el hipervisor por un motor y prescinden del sistema operativo invitado.

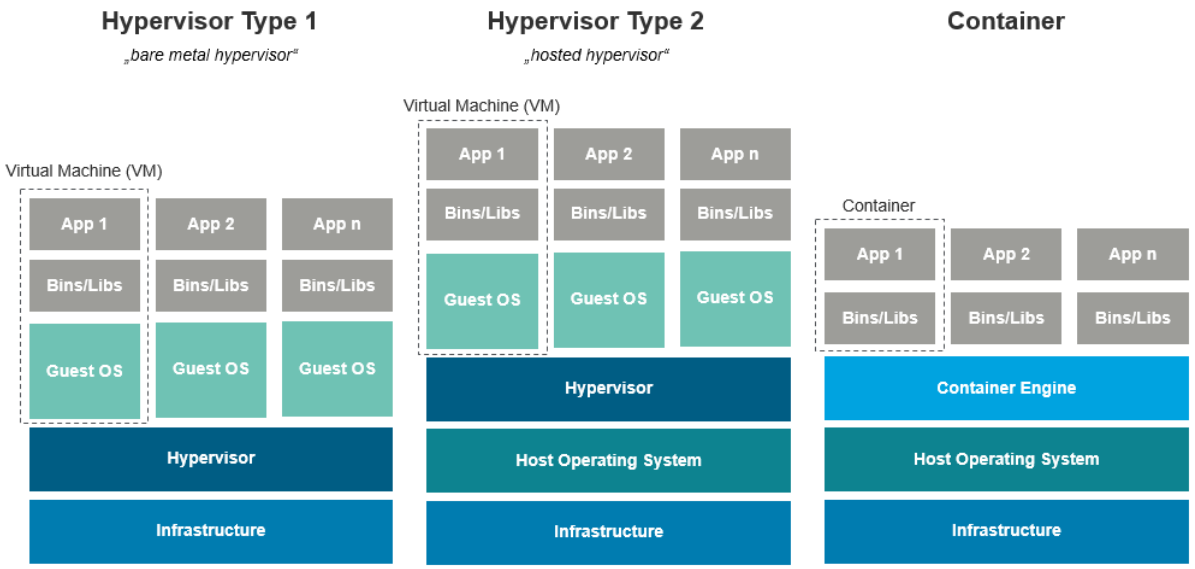

Figura 2.3: Comparación entre hipervisores y contenedores. Obtenido de [29]

Al no existir hipervisores ni tener que emplear un sistema operativo invitado completo, la sobrecarga que implica virtualizar un sistema utilizando esta técnica es notablemente inferior a la que resultaría de utilizar virtualización completa o paravirtualización. Además, no solamente repercute en un menor uso de los recursos del sistema, sino que el tiempo de creación y puesta en marcha de estos entornos es también muy inferior al de las máquinas virtuales, debido a que el núcleo del sistema operativo ya está cargado en memoria y se ejecutan menos tareas durante la etapa de arranque. Estos aspectos serán vistos con más detalle en la sección 2.3.4.

Estas razones son algunas de las causas por las que esta clase de virtualización ha ganado especial relevancia en los últimos años, pero no las únicas. Sin embargo, antes de continuar explicando cuáles son los factores adicionales existentes, es relevante hacer un repaso histórico, mostrando desde sus primeros pasos hasta el estado actual de esta tecnología [30].

2.3.1. Repaso histórico

Aunque no se trata de virtualización como tal, el primer avance considerado como precursor de esta tecnología fue la incorporación de la llamada de sistema `chroot` al núcleo de UNIX en 1979 [31]. Esta llamada cambia aparentemente el directorio raíz de un proceso y de sus hijos, de tal forma que se consigue un aislamiento parcial del sistema de archivos, imposibilitando el acceso a otros archivos fuera del directorio especificado en la llamada.

FreeBSD introdujo en su versión 4.0, publicada en el año 2000, las *jails* o jaulas. Extendiendo la funcionalidad de `chroot`, se virtualizan además los sistemas de usuarios y de red, de tal forma que las jaulas sí se consideran como virtualización a nivel de sistema operativo [32]. En 2001 este concepto llegó a Linux de la mano de Linux-VServer, cambiando en este caso la denominación de jaulas a servidores privados virtuales [33]. No obstante, para su funcionamiento se requería que el *kernel* estuviera parcheado previamente, lo cual provocó que su adopción fuese bastante limitada. Otras alternativas

similares aparecieron en años sucesivos, como Solaris Containers en 2004 para Solaris u OpenVZ en 2005 para Linux —en el caso de este último, también requería de un núcleo modificado tal y como sucedía con Linux-VServer—.

En 2006 se produjo un avance sustancial en la virtualización a nivel de sistema operativo con la creación de los *Process Containers*, que más tarde serían renombrados a *control groups*, más comúnmente conocidos como `cgroups`. Desarrollados por Paul Menage, se trataban de modificaciones añadidas a la versión 2.6 del *kernel* de Linux que posibilitaban la creación de grupos de procesos que recibieran un uso equitativo del *hardware* del sistema, de tal forma que ninguno de ellos pudiera acaparar todo el uso de los recursos disponibles [34].

Por otro lado, entre las versiones 2.4 y 3.8 del *kernel* —y principalmente en la versión 2.6— se añadieron los *namespaces*, una característica esencial para mejorar el aislamiento de los contenedores. Cada *namespace* permite «envolver un recurso global del sistema en una abstracción que permite que los procesos dentro del *namespace* lo vean como si tuvieran su propia instancia aislada del recurso» [35]. De esta manera, cada entorno virtualizado podía disponer de su propio sistema de ficheros, pila de red o árbol de procesos —entre otros—, sin que fuera posible que los procesos dentro de un *namespace* pudiesen ver los recursos fuera de este.

Estas dos funciones fueron aprovechadas por LXC —siglas de Linux Containers—, el primer *software* de gestión de contenedores disponible en Linux sin necesidad de previamente aplicar cambios al núcleo [36]. Su desarrollo comenzó en 2008 por parte de ingenieros de IBM, pero actualmente se trata de un proyecto de *software* libre mantenido principalmente por Canonical, empresa responsable de la distribución de Linux Ubuntu.

Sin embargo, no fue hasta 2013 cuando los contenedores empezaron a tomar la amplia dimensión que tienen en la actualidad, lo cual se debe en gran medida a la aparición de Docker [37]. Docker empezó como una herramienta basada en LXC pero contando con una interfaz de usuario más simplificada, de manera que el proceso de creación y gestión de contenedores se hacía mucho más sencillo. Por otra parte, otra de las claves de su éxito fue la incorporación de contenedores con un sistema de archivos por capas, lo que permitía que los contenedores pudieran ser creados a partir de imágenes. Estas imágenes, creadas a partir de archivos de texto denominados Dockerfiles o descargadas de registros como Docker Hub, posibilitaban que, a partir de una imagen base, se pudiera añadir *software* u otros archivos de una forma sencilla. La gran ventaja de este enfoque es que así se podían crear y distribuir contenedores exactamente idénticos en entornos muy diferentes entre sí, todo ello de forma rápida y sencilla [38].

En marzo de 2014 Docker dejó de utilizar LXC como biblioteca para interuactuar con el *kernel* de Linux por defecto, sustituyéndose por `libcontainer`, desarrollada por los propios ingenieros de Docker [39]. Al mes siguiente, Amazon Web Services se convertiría en uno de los primeros proveedores de computación en la nube en dar soporte a Docker, concretamente en su servicio AWS Elastic Beanstalk [40], lo que propiciaría que la adopción de los contenedores fuese cada vez mayor. Ese mismo año Google anunció Kubernetes [41], una aplicación de orquestación de contenedores que permitía la gestión y escalado de contenedores a gran escala.

En 2015 Docker donó su formato de contenedor y su *runtime* a la Open Container Initiative (OCI), un proyecto con el respaldo de la Linux Foundation y formado por Docker,

CoreOS, Red Hat, Amazon y otras muchas empresas relacionadas con el *cloud computing* y la virtualización [42]. Esto generó que en 2017 fuera posible la publicación de la OCI Image Format Specification [43] y la OCI Runtime Specification [44], dos estándares abiertos que definían las características que debían cumplir los contenedores y las herramientas relacionadas para su creación y ejecución. La adopción de estos estándares posibilitaba que, en caso de que cualquier usuario desease dejar de usar algún *software* de contenedores y sustituirlo por otro, la transición se realizase de forma fluida y sin dificultades.

Por último, podemos cerrar este repaso histórico con la llegada de los contenedores a Windows de la mano de Docker en 2016, con la posibilidad de ejecutar entornos basados tanto en Linux como en Windows [45].

2.3.2. *Software* de contenedores disponible

Docker

Creado por Solomon Hykes en 2013, Docker [37] es el *software* de contenedores más utilizado y conocido en la actualidad. Para diferenciarlo de otros proyectos de la compañía, su nombre completo es Docker Engine, pero ambos nombres se suelen utilizar indistintamente. El motor de Docker está formado por tres elementos [46]: un servidor o demonio —`dockerd`— que se encarga de la creación y gestión de los contenedores, un cliente que interactúa con el servidor y una API que sirve de puente entre ambos elementos. Está disponible tanto para Linux como para Windows y macOS, a través de Docker Desktop para los dos últimos.

Docker define dos objetos principales en su ciclo de funcionamiento: las imágenes, que son plantillas de solo lectura que sirven para definir la creación de un contenedor y que pueden ser construidas a partir de un archivo de texto denominado Dockerfile o descargadas directamente de registros como Docker Hub, y los contenedores, que son las instancias ejecutables creadas a partir de una imagen a la que se le ha añadido una capa de lectura/escritura. El formato de las imágenes empleadas por Docker cumple con la OCI Image Format Specification.

El demonio de Docker utiliza containerd como entorno de ejecución, el cual «gestiona el ciclo de vida completo de los contenedores de un sistema, desde la obtención y almacenamiento de imágenes hasta la ejecución y supervisión de los contenedores, pasando por el almacenamiento a bajo nivel, la gestión de red y mucho más» [47]. A más bajo nivel, containerd emplea runc como sistema para la creación de los contenedores, sirviéndose de `libcontainer` —mencionado anteriormente— para la interacción con el *kernel* de Linux [48]. runc cumple con lo descrito en la OCI Runtime Specification y forma parte de la Open Container Iniciative, mientras que containerd está mantenido por la Linux Foundation.

Desde abril de 2017 gran parte de los componentes del servidor de Docker pasaron a formar parte de Moby Project, un proyecto que actúa como *upstream* de Docker y que permite que se puedan crear nuevas herramientas de gestión de contenedores distintas a Docker a partir de su tecnología [49].

Podman

Podman [50] es una alternativa a Docker cuyo desarrollo comenzó a finales de 2017. Aunque se trata de un *software* libre perteneciente a la organización containers [51], el mayor contribuidor de este proyecto es Red Hat.

Podman utiliza la misma sintaxis en sus comandos que Docker, por lo que es posible reemplazar uno por otro de una manera muy sencilla. Sin embargo, su principal diferencia con respecto a dicho programa reside en que prescinde del modelo cliente-servidor, ejecutando directamente las tareas de gestión de contenedores mediante el enfoque clásico *fork/exec* [52]. Por otro lado, Podman fue pionero en el desarrollo de los contenedores *rootless*, permitiendo que los contenedores pudieran ser creados por usuarios que no dispusieran de permisos de superusuario, mejorando la seguridad. Esta opción fue añadida a Docker como experimental en su versión 19.03 y pasó a un estado estable en la 20.10 [53]. Por último, Podman incorpora el concepto de `pods`, originario de Kubernetes y que facilita la gestión de varios contenedores al mismo tiempo —de ahí el nombre de la herramienta, abreviación de Pod Manager—.

En lugar de emplear un motor de contenedores como containerd, Podman utiliza su propia biblioteca, libpod. No obstante, para la creación a bajo nivel de los contenedores sí que utiliza runc por defecto, aunque es previsible que en el futuro sea reemplazado por crun, otra implementación de la OCI Runtime Specification escrita en C en lugar de Go creada por la misma organización [54].

Además, Podman también cumple con el OCI Image Format Specification, por lo que es posible utilizar las mismas imágenes que emplea Docker o descargarlas desde el propio Docker Hub. Sin embargo, no dispone nativamente de la capacidad de construir imágenes o de enviarlas a un registro, sino que para ello utiliza dos proyectos hermanos, buildah y skopeo respectivamente [55].

Con la salida de la versión 8 en 2019, Red Hat decidió eliminar Docker de Red Hat Enterprise Linux y dejar de darle soporte, promoviendo en su lugar el uso de Podman [56].

CoreOS rkt

rkt [57] fue un proyecto lanzado por CoreOS Container Linux en diciembre de 2014, aunque desde la adquisición de CoreOS por parte de Red Hat en 2018 su actividad empezó a ser menor, hasta el cese definitivo de su desarrollo en febrero de 2020 [58]. No obstante, es relevante hacer una pequeña mención a este programa porque durante su ciclo de vida fue uno de los mayores competidores de Docker. Al igual que Podman, optaba por un modelo sin servidor, y también contaba con el concepto de `pods`. Además, fueron los creadores de appc [59], un estándar que definía los formatos de imagen y las características de sus contenedores y que, tal como hiciera Docker en 2015, fue donado a la Open Container Iniciative para servir como base de la OCI Image Format Specification y la OCI Runtime Specification.

Linux Containers

Como se había explicado en la sección 2.3.1, Linux Containers —más comúnmente conocido como LXC— es un *software* de gestión de contenedores mantenido principalmente

por Canonical cuyos orígenes se remontan hasta 2008. A diferencia del resto de programas vistos, LXC aboga por lo que se denominan contenedores de sistema, pretendiendo ofrecer una experiencia más parecida a la de las máquinas virtuales (la diferencia será explicada detalladamente más adelante). Se puede interactuar con LXC a través de LXD —siglas de Linux Containers Daemon—, un *frontend* que permite la gestión de los contenedores de una manera más sencilla. Está basado en imágenes, por lo que es posible descargar distribuciones de Linux completas para su uso en lugar de tener que instalar todo el software deseado a mano [60].

2.3.3. Ejemplos de casos de uso

Como se ha visto a lo largo de toda esta sección, la virtualización a nivel de sistema operativo y los contenedores permiten ejecutar cargas de trabajo aisladas de una manera más ligera a si se optase por utilizar máquinas virtuales. Para ello, el enfoque más tradicional y simple es el de LXC y sus contenedores de sistema, en el que dentro de un contenedor se ejecutan varios procesos a la vez, simulando un sistema operativo completo. Además, en este caso los contenedores se conciben con estado, es decir, que pueden ser apagados y reiniciados conservando toda la información de ejecuciones previas.

Sin embargo, esta no es la única utilidad de los contenedores, ya que lo que realmente hizo que esta tecnología despegara como una de las herramientas más utilizadas en la actualidad fue la creación de los contenedores de aplicaciones. Este concepto, impulsado gracias al propio diseño de Docker, consiste en que dentro de cada contenedor solo se ejecuta un único proceso, almacenando exclusivamente el código o los binarios de la aplicación y sus dependencias. De esta forma, se facilita enormemente la creación de entornos idénticos y portables, lo que genera que sea más fácil ejecutar la aplicación siempre en las mismas condiciones, reduciendo así la posibilidad de que aparezcan fallos derivados de la configuración del entorno.

Por ejemplo, supongamos que tenemos el siguiente código, un simple *Hola Mundo* escrito en Python:

```
print('Hola mundo')
```

La forma tradicional de ejecutar esta aplicación sería instalar Python en el ordenador de cada usuario que fuera a utilizar esta aplicación. Al ser un programa tan simple, en este caso no habría demasiados inconvenientes. Pero a medida que fuera creciendo el código escrito, es posible que el programa se quedase «anclado» a versiones concretas del lenguaje de programación o bibliotecas adicionales. De esta manera, la complejidad para mantener entornos idénticos crece cada vez más, ya que es posible que cada desarrollador tenga instalado una versión diferente del *software* o que los entornos de desarrollo y producción difieran enormemente, lo que hace que sea más difícil ejecutar siempre el programa en las mismas condiciones y puede provocar la aparición de *bugs*.

Con los contenedores de aplicaciones se resuelven estos problemas en gran medida. Nuestra aplicación de ejemplo se podría «contenerizar» usando el siguiente Dockerfile —compatible tanto con Docker como con Podman—:

```
FROM python:3
WORKDIR /usr/src/app
COPY holaMundo.py .
CMD ["python", "./holaMundo.py"]
```

Lo que hace este Dockerfile es crear una imagen nueva a partir de la imagen de Python disponible en Docker Hub, publicada por Docker [61]. Además, se especifica que se va a usar la versión 3 de Python, aunque es posible concretar más —por ejemplo, cambiando el 3 por la versión 3.8.12—. A continuación, se crea y se cambia a un nuevo directorio, `/usr/src/app`, que será la ubicación donde guardaremos nuestra aplicación (este paso es opcional). Luego, se copia el código desde el ordenador a la imagen, y finalmente se establece el comando por defecto que se ejecutará cuando el contenedor sea creado.

Así, cada usuario que quiera utilizar este programa lo único que tendrá que hacer es instalar un programa como Docker o Podman, obtener el Dockerfile —la práctica habitual es incluirlo junto a la propia aplicación—, construir la imagen y crear un contenedor nuevo a partir de dicha imagen, olvidándose de tener que verificar cuál es la versión de Python que tiene instalada en su equipo y cambiarla si es necesario. Además, dependiendo del caso de uso es posible añadir más instrucciones al Dockerfile, pudiendo instalar dependencias con `pip` o con el gestor de paquetes del contenedor, crear nuevos usuarios, establecer variables de entorno...

En conclusión, el uso de contenedores de aplicación permite la generación de entornos reproducibles y con las mismas características en todos los sistemas donde sean ejecutados.

A diferencia de los contenedores de sistema, donde se ejecutan varios procesos en cada contenedor, lo habitual en los contenedores de aplicaciones es ejecutar un único proceso por contenedor. Esto permite que los contenedores puedan ser concebidos como entidades efímeras y sin estado, es decir, que no almacenen información dentro del contenedor y que puedan ser creados y destruidos de una forma rápida sin que esto suponga un inconveniente. Además, gracias a este planteamiento el tamaño de las imágenes y los contenedores se mantiene más reducido. Debido a esto, el proceso en funcionamiento dentro de cada contenedor suele recibir el PID 1, por lo que por defecto no se ejecutan sistemas `init`.

Para ejemplificar esta característica, supongamos que se desea crear una pila LAMP sobre un sistema determinado. LAMP son las siglas de Linux, Apache, MySQL y PHP, y suele ser una de las configuraciones más habituales a la hora de desplegar un servidor web. En lugar de ejecutar un único contenedor con los tres procesos, se crearían tres contenedores independientes para cada proceso pero conectados entre sí. Gracias a ello, se reducen las dependencias entre aplicaciones, porque si por ejemplo se desease actualizar la versión de MySQL, esto no afectaría en ningún caso al funcionamiento de Apache. Por otra parte, en caso de que cualquiera de los contenedores se detuviera, el resto seguirían funcionando correctamente —y si se sigue la directriz de no almacenar información directamente en el contenedor, al volver a crear un contenedor nuevo la situación será exactamente igual a la de antes del fallo—. Finalmente, otra de las posibilidades que abre esta estrategia es la capacidad de escalado rápido para cada componente, lo que permite añadir contenedores adicionales en caso necesario como en el caso de que se produzca algún fallo o haya un

aumento de la carga de trabajo —tarea que se puede realizar fácilmente gracias a los orquestadores de contenedores como Kubernetes—.

No obstante, la ejecución de un único proceso por sistema no es una restricción estricta. Además, en determinados casos puede ser necesario el uso de `init` que actúe como padre del proceso ejecutado, por ejemplo para el manejo de procesos *zombie* generados por dicho proceso. Para estos dos escenarios, tanto Docker como Podman disponen de la opción de generar un proceso `init` que adquiera el PID 1.

Finalmente, es importante resaltar que una de las desventajas más destacadas de la virtualización a nivel de sistema operativo es que es imposible probar versiones diferentes del *kernel* del sistema operativo anfitrión dentro de los contenedores. En caso de que se desease utilizar una versión anterior o modificada, sigue siendo necesario recurrir al uso de máquinas virtuales.

2.3.4. Rendimiento

Como ya se ha visto a lo largo de este capítulo, los contenedores ofrecen un rendimiento superior al de las máquinas virtuales gracias a su propio diseño, tanto en el tiempo que se necesita para su creación, como en los recursos que utilizan a lo largo de su ciclo de vida. En esta sección se enumeran algunos trabajos que confirman este planteamiento.

En [62] los autores realizaron una comparación entre Docker y KVM, evaluando su rendimiento y uso de CPU, memoria, almacenamiento y red con respecto al rendimiento nativo del sistema. Además de utilizar *benchmarks* para ello, también compararon su funcionamiento con dos aplicaciones reales, Redis y MySQL. Concluyeron que «Docker iguala o supera el rendimiento de KVM en todos los casos probados», llegando incluso a ofrecer un rendimiento casi idéntico al nativo, salvo en las pruebas de entrada/salida y de red. Esta pérdida de rendimiento puede ser justificada por el uso del sistema de archivos por capas y por la NAT que utilizan los contenedores por defecto, pero ambos problemas pueden ser solventados mediante el uso de volúmenes y el modo de conexión `host` de Docker —aunque en este último caso se perdería el aislamiento entre el contenedor y el anfitrión en la capa de red—.

Otros trabajos obtuvieron resultados en la misma línea, como el de Morabito et al. [63]. En él, se comparó el funcionamiento de KVM, LXC y Docker con respecto a la ejecución sin virtualizar, verificando que los contenedores generaban menor sobrecarga que las máquinas virtuales. Es relevante destacar que en este estudio LXC obtuvo un mejor rendimiento en las operaciones de entrada/salida que Docker, si bien se puede achacar a la misma razón mencionada en el párrafo anterior.

Li et al. [64] volvieron a comparar el rendimiento de Docker con el de las máquinas virtuales, pero a diferencia de los anteriores trabajos, utilizan VMWare Workstation Pro, un hipervisor de tipo 2, para la creación de las máquinas virtuales. Las conclusiones no variaron, ya que afirmaron que «el rendimiento medio de los contenedores es generalmente superior que el de las máquinas virtuales y es incluso comparable al de la máquina física en múltiples casos».

En lo relativo al tiempo de creación y arranque de los contenedores, varios estudios [65] [66] confirman que este tiempo suele ser bastante inferior al que necesitan las máquinas virtuales para ser arrancadas, debido al hecho de que los contenedores no necesitan cargar

un sistema operativo completo.

Uno de los casos de uso habituales de la virtualización es la construcción de sistemas de computación en la nube, ya sean públicas o privadas. En [67] se comparó el uso de contenedores, máquinas virtuales y *bare-metal* en OpenStack, un *software* para crear nubes privadas. Las máquinas virtuales obtuvieron el peor rendimiento en cuanto a memoria y entrada/salida, mientras que los contenedores tenían el tiempo de arranque más rápido de las tres alternativas y un rendimiento similar al del *bare-metal* salvo en las operaciones de red.

En cuanto al uso de los contenedores en aplicaciones reales, en [68] se evaluó el rendimiento de Docker frente al de VMWare ESXi al ejecutar Cassandra, un sistema de gestión de bases de datos NoSQL. Los resultados obtenidos fueron que los contenedores de Docker consumían menos recursos y generaban menor sobrecarga que las máquinas virtuales de VMWare. Además, en el caso de Docker se llegó a alcanzar un número de transacciones por segundo similar al caso no virtualizado. Por último, cabe destacar que en este estudio se comprobó el funcionamiento tanto de un solo contenedor como de varios sobre la misma máquina, sin que el rendimiento se resintiera en ningun momento.

En [69] se comparó el funcionamiento de los contenedores y de las máquinas virtuales sobre OpenStack para la ejecución de aplicaciones de computación de alto rendimiento (HPC). Obtuvieron que «los sistemas basados en contenedores son más eficientes a la hora de reducir el tiempo total de ejecución de las aplicaciones de HPC y tienen un mejor sistema de gestión de memoria al ejecutar varios contenedores en paralelo». De manera similar, en [70] se compararon contenedores de Docker y máquinas virtuales de KVM al usar aplicaciones intensivas en computación y aplicaciones intensivas en datos, obteniendo Docker un mejor rendimiento en ambos casos, especialmente cuando se aumenta el número de instancias.

Finalmente, en [71] se realizó un estudio exhaustivo de los distintos elementos que podían afectar al rendimiento de los contenedores, como son el gestor de contenedores —Docker, Podman o LXC—, el *runtime* utilizado —`runc`, `crun` o Kata—, el sistema de archivos de los contenedores o la imagen de los contenedores. Se concluyó que los aspectos que provocaban un mayor impacto en el rendimiento eran el sistema de archivos, siendo `overlay2` —utilizado por Docker por defecto— el que proporcionaba mejor rendimiento, y el entorno de ejecución, con `crun` otorgando los mejores tiempos de ejecución gracias a su implementación en C. Por otro lado, es interesante mencionar que entre Docker y Podman no hubo grandes diferencias de rendimiento, a pesar de que el primero utiliza un demonio para su funcionamiento mientras que el segundo no.

2.3.5. Seguridad

A pesar de que los contenedores compartan el *kernel* con el sistema operativo anfitrión, no existen motivos para afirmar que, por diseño, los contenedores sean menos seguros u ofrezcan menos aislamiento que las máquinas virtuales. Sin embargo, es necesario tener presente cuáles son los mecanismos de seguridad que ofrecen los contenedores y qué problemas pueden surgir a causa de una mala configuración. En [72] se realizó un amplio estudio de todas las posibles amenazas que pueden existir a la hora de ejecutar contenedores y qué contramedidas se pueden aplicar para mitigarlas.

El principal sistema que aplican los contenedores para que haya aislamiento entre ellos

y la máquina anfitriona son los *namespaces*. Como ya se vio en el apartado 2.3.1, los *namespaces* provocan que cada contenedor tenga una visión única de subsistemas como la lista de procesos, la pila de red, el sistema de archivos o la comunicación inter-procesos. De esta forma, es imposible que un contenedor pueda acceder a la información del sistema anfitrión o al resto de contenedores.

Además, gracias a los `cgroups`, que se encargan de limitar la utilización de los recursos *hardware* por parte de los contenedores, se evitan posibles ataques de denegación de servicio (DoS) desde el contenedor hacia el *host* [73].

Otra de las características que posee el núcleo de Linux y que sirve para mejorar la seguridad de los contenedores son las *capabilities*. Las *capabilities* permiten la gestión granular de los privilegios del sistema. De esta forma, incluso si un proceso se está ejecutando como `root`, es posible limitar las tareas que puede realizar. En el caso de Docker, los contenedores se ejecutan por defecto con una lista reducida de *capabilities*, lo que impide realizar algunas acciones peligrosas dentro del contenedor como pueden ser montar directorios del sistema anfitrión, cargar módulos del *kernel* o acceder a ficheros de dispositivos [74].

Para aumentar aún más la seguridad, una opción disponible es utilizar módulos de seguridad de Linux como SELinux, AppArmor o Seccomp. SELinux es un módulo de seguridad que permite limitar las acciones que puede llevar a cabo un usuario o un proceso a través de etiquetas. Gracias a ello, se pueden ajustar las acciones que se pueden ejecutar dentro del contenedor [75]. AppArmor funciona de forma similar, ya que controla los comandos disponibles y el acceso a ficheros mediante el uso de perfiles. Finalmente, Seccomp permite restringir las llamadas al sistema que se pueden realizar dentro del contenedor. Por defecto, tanto Docker como Podman utilizan AppArmor y Seccomp para reducir los privilegios de los contenedores, y ambos permiten especificar perfiles personalizados para cada uno de los módulos [76] [77].

Sin embargo, aunque los programas de gestión de contenedores posean sus propias medidas para garantizar la seguridad dentro de los contenedores, también es importante tener en consideración las acciones que puede realizar un usuario malicioso si llega a tener disponible la posibilidad de crear contenedores en un sistema. Por ejemplo, es posible que decida montar el directorio raíz del anfitrión dentro de un contenedor o utilice la opción `--privileged` para saltarse las limitaciones impuestas por las *capabilities*, lo que pondría en grave riesgo al sistema. Para atajar esta amenaza, Docker y Podman solo permiten al usuario `root` la gestión de los contenedores. No obstante, en el caso de Docker es posible añadir a un usuario al grupo `docker` para que pueda ejecutar comandos sin necesidad de ser superusuario. Esto significaría que el administrador del sistema debe tener suficiente confianza en dicho usuario para añadirle al grupo, ya que a efectos prácticos se comportaría como si fuese `root`.

Ambos programas también permiten la ejecución de contenedores en modo *rootless* gracias a los *user namespaces*, que realizan un mapeo entre la lista de usuarios del contenedor y del sistema anfitrión. De esta forma, incluso si se crease un contenedor privilegiado y dentro de él se llevasen a cabo acciones con el usuario `root`, el *host* en ningún caso se llegaría a ver afectado porque desde fuera del contenedor el usuario `root` se vería como un usuario sin privilegios [53].

Las imágenes ejecutadas por los contenedores también son otro factor de riesgo debido a

que pueden incluir vulnerabilidades o puertas traseras. Esta amenaza se puede enfrentar mediante el análisis de posibles vulnerabilidades presentes en las imágenes —por ejemplo con el comando `docker scan` [78]— y mediante el uso de imágenes confiables. Docker Content Trust [79] es una funcionalidad de Docker que permite firmar imágenes y enviarlas a registros como Docker Hub. Gracias a esta firma, el usuario final puede verificar que la imagen no ha sido modificada por un atacante durante el proceso de descarga.

Finalmente, cabe destacar la posibilidad de que exista una vulnerabilidad en el propio *software* de contenedores o en el *kernel* de Linux. Esta eventualidad sí puede suponer un riesgo mayor en el caso de los contenedores frente a las máquinas virtuales, ya que si la vulnerabilidad fuese explotada desde dentro del contenedor, el atacante tendría acceso al sistema anfitrión, mientras que en el caso de la máquina virtual todavía tendría que superar la barrera del hipervisor. No obstante, existen maneras de reducir esta amenaza: por un lado, mediante los *user namespaces* para que, en caso de que el atacante escapase del contenedor, no tuviera privilegios en el sistema anfitrión; por otro lado, aplicando los parches de seguridad necesarios de manera inmediata en caso de que se descubra una vulnerabilidad de esta clase.

Capítulo 3

Implementación basada en las necesidades de los laboratorios docentes

3.1. Situación actual

En el ámbito de las titulaciones de la Facultad de Informática de la Universidad Complutense de Madrid, los laboratorios desempeñan un papel preeminente en gran parte de las asignaturas impartidas. Los estudiantes realizan prácticas de programación, fundamentos de *hardware* o gestión de bases de datos a lo largo de su etapa formativa, entre otras áreas; sin embargo, en este apartado nos vamos a centrar especialmente en la situación actual en las asignaturas de sistemas y redes de dichas titulaciones, debido a que sus prácticas suelen requerir del uso de virtualización de forma obligatoria tal y como se ha explicado en el apartado 1.1.

Dentro de las asignaturas de sistemas y redes se realizan prácticas que abarcan desde la programación de sistemas de ficheros, la configuración de infraestructuras de red o la puesta en marcha de servidores hasta la recreación de ataques informáticos, tanto locales como remotos. Con el objetivo de acotar aún más la descripción de la situación actual, vamos a tomar como ejemplo la problemática actual de la asignatura Redes y Seguridad II, optativa de itinerario del Grado en Ingeniería Informática de la Universidad Complutense de Madrid, ya que sus prácticas normalmente exigen la utilización de varias máquinas virtuales al mismo tiempo. No obstante, esta problemática se puede hacer extensible al resto de asignaturas de sistemas y redes impartidas.

Las prácticas de Redes y Seguridad II están concebidas con el propósito de afianzar los conocimientos vistos en las sesiones teóricas sobre seguridad en entornos con varios sistemas conectados a través de una o más redes. De esta manera, en las sesiones prácticas se tratan conceptos como la configuración correcta de *firewalls*, el despliegue de *proxys*, la puesta en marcha de redes privadas virtuales (VPN) o la realización de ataques informáticos a servicios vulnerables dentro de una red.

Para llevar a cabo estas prácticas, en la actualidad se sigue un modelo basado en virtua-

lización con máquinas virtuales. La máquina virtual que se usa principalmente en esta asignatura es una máquina con Debian 9 [80] como sistema operativo base, en la que se encuentran instalados diversos programas de ciberseguridad, como pueden ser Wireshark [81] para monitorear el estado de la red, iptables [82] para la configuración de *firewalls*, OpenVPN [83] para la creación de redes privadas virtuales o Apache HTTP Server [84] para trabajar la fortificación de servidores web. Además, suele ser habitual el uso de otra máquina virtual con Kali Linux [85], una distribución de Linux enfocada a la ciberseguridad y al *pentesting*, que cuenta con aplicaciones como nmap [86] o ZAP [87]. Por último, en algunas prácticas también se emplea Metasploitable 2 [88], una máquina virtual que incluye de forma intencionada vulnerabilidades para poder estudiar su explotación. Todas estas máquinas virtuales se ejecutan sobre Oracle VM VirtualBox, aunque también son compatibles con otros hipervisores de tipo 2 como VMWare Workstation Pro.

La dinámica de estas prácticas suele consistir en crear una infraestructura de red con varias instancias de cada máquina virtual, de tal manera que, dependiendo del caso, unas instancias actuarán como clientes o atacantes, otras como *routers* que conectarán redes diferentes y, finalmente, otras que desempeñarán el papel de servidor o de máquina atacada. Por ello, lo más normal es que a lo largo de una práctica sea necesario desplegar más de una máquina virtual sobre el sistema anfitrión, llegando incluso a utilizar cinco máquinas virtuales al mismo tiempo como en el caso de la primera práctica de Redes y Seguridad II.

Asimismo, es relevante señalar que, en cada una de las prácticas, el proceso de creación y despliegue de las máquinas virtuales se realiza de forma interactiva por parte de los estudiantes. Es decir, este proceso no se encuentra automatizado a través de una herramienta o *script*, sino que son los estudiantes quienes deben clonar las máquinas y configurar sus interfaces de red una a una. Si bien es cierto que este hecho puede ser aprovechado para que los estudiantes aprendan nociones de administración de máquinas virtuales y redes, el proceso puede ser largo y propenso a que se produzcan fallos humanos, lo que reduce el tiempo disponible para realizar la práctica en sí.

Como ya se había anticipado en el capítulo anterior, el uso de máquinas virtuales puede suponer una grave penalización al rendimiento debido a la sobrecarga que ejerce el hipervisor y a la necesidad de ejecutar un sistema operativo completo en cada una de las máquinas virtuales. Este hecho resulta especialmente notable en las prácticas de Redes y Seguridad II, en las que el rendimiento dista mucho de ser óptimo a causa de tener que emplear varias máquinas virtuales al mismo tiempo, aumentando aún más la ralentización del sistema. Además, cabe destacar que, en la actualidad, la mayoría de los ordenadores disponibles en los laboratorios de la Facultad de Informática no cuentan con discos de estado sólido (SSD), un hecho que contribuye a que el rendimiento sea todavía peor debido a que la entrada/salida suele ser uno de los mayores cuellos de botella en la virtualización [63] [89]. Todo ello ha supuesto que durante este curso la experiencia de uso haya sido muy negativa para los estudiantes, con unos tiempos de encendido de las máquinas virtuales muy elevados y un rendimiento por debajo de lo aceptable.

Estos hechos hacen patente la necesidad de estudiar la viabilidad de implementar las prácticas de las asignaturas de sistemas y redes reemplazando las máquinas virtuales por contenedores, tal y como se había expuesto en el apartado 1.1, sin olvidar el resto de ventajas derivadas de su implementación mencionadas en dicha sección y que podrían ser aprovechadas en las prácticas de otras asignaturas.

3.2. Características de los contenedores de reemplazo

En esta sección se describen qué características deben tener los contenedores para que puedan servir de reemplazo a las máquinas virtuales usadas en las prácticas de las asignaturas mencionadas en el apartado anterior.

3.2.1. Diseño de los contenedores

En primer lugar, se debe decidir cómo se van a diseñar los contenedores. El enfoque que proponen Docker y Podman para sus contenedores de aplicación es ejecutar un único proceso en cada uno de los contenedores. De este modo, en una práctica habitual de Redes y Seguridad II se deberían tener, por ejemplo, unos contenedores que actúen exclusivamente como *routers*, otros que sirvan únicamente para ejecutar programas como Wireshark, otros en los que se encuentre desplegado *software* como servidores web y, finalmente, otros que dispongan de una *shell* para ejecutar comandos que interactúen con el resto de contenedores.

Esta restricción de ejecutar un único proceso por contenedor, que recordemos que no es estricta tal y como se había explicado en la sección 2.3.3, tiene sentido cuando lo que se desea es «contenerizar» una aplicación, es decir, que la aplicación se ejecute en un contenedor con el objetivo de garantizar que el entorno es siempre el mismo independientemente de dónde se realice el despliegue. Sin embargo, creemos que en este proyecto resulta más beneficioso aplicar un enfoque más flexible y que sea más parecido a las máquinas virtuales actuales, ya que se simplificaría el proceso de despliegue y el diseño de las prácticas, además de que para los estudiantes resultaría mucho más sencillo utilizar un único contenedor polivalente que crear varios que estén especializados en una única tarea.

Para ello, una opción podría ser utilizar contenedores de sistema a través de LXC. No obstante, los contenedores de LXC suelen ser más pesados que sus homólogos de Docker o Podman, ya que intentan emular la utilización de un sistema operativo completo, por lo que incluyen y ejecutan una cantidad de servicios y aplicaciones bastante superior a lo que suele ser habitual en los contenedores de aplicación, como por ejemplo demonios `init`, editores de texto o incluso clientes SSH [90].

Otra alternativa sería utilizar contenedores de aplicación creando una imagen especializada para cada asignatura. Por ejemplo, para crear un contenedor similar a la máquina de Debian utilizada en Redes y Seguridad II, se podría partir de la imagen de Debian 9 disponible en Docker Hub [91], instalar algunos programas habituales en todas las prácticas como Wireshark, configurar un usuario diferente a `root` pero que pueda emplear `sudo`, y establecer `bash` como el programa por defecto al iniciar el contenedor. El Dockerfile resultante sería el siguiente:

```
FROM debian:9
# Para evitar que salgan mensajes de la configuración de Wireshark
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y wireshark
RUN groupadd -g 1000 usuario && useradd -m -u 1000 -g usuario usuario \
```

```
&& echo usuario:usuario | chpasswd && echo "usuario ALL=(ALL) ALL" \
>> /etc/sudoers && echo "Defaults lecture = never" >> /etc/sudoers
USER usuario
WORKDIR /home/usuario
CMD bash
```

De esta manera, todos los contenedores emplearían esta imagen como base, y los estudiantes podrían ejecutar todas las tareas requeridas en cada práctica utilizando `bash`. Si se deseara añadir *software* adicional como, por ejemplo, iptables para la realización de la práctica de configuración de *firewalls*, se podría instalar a través de `apt-get` dentro del contenedor o, aprovechando el sistema por capas de las imágenes OCI, crear una nueva imagen partiendo de esa imagen base. Si suponemos que dicha imagen se ha construido con el nombre `rys`, el Dockerfile de la imagen nueva podría ser:

```
FROM rys:latest
# Necesitamos volver a ser root para instalar el paquete
USER root
RUN apt-get update && apt-get install -y iptables
USER usuario
```

Así, se puede disponer de unos contenedores polivalentes que sirven para realizar todas las tareas requeridas en las prácticas, con un peso más reducido que si se utilizasen contenedores de sistema y con el añadido de la modularidad por capas para adaptar esta imagen base dependiendo de la práctica o de la asignatura. Por ello, se optará por esta opción en este trabajo.

Para que los contenedores funcionen correctamente, deberán ser creados con la opción `--init`, que insertará dentro de los contenedores un sistema `init` reducido, encargado de la gestión de los procesos *zombie* y el reenvío de señales al resto de programas en ejecución.

3.2.2. Imagen base de los contenedores

La imagen base que se utilizará en este proyecto, denominada `rys`, está diseñada para servir como reemplazo de la máquina virtual de Debian de la asignatura de Redes y Seguridad II, por lo que es muy similar a la descrita en el anterior apartado, pero incluyendo además los siguientes programas:

- `firefox-esr`: Durante el desarrollo de las prácticas suele ser necesario usar un navegador web para descargar archivos desde el Campus Virtual de la asignatura o para acceder a los servidores web desplegados en otras instancias. Por ello, se ha optado por Firefox ESR debido a que es el navegador que se utiliza actualmente en la máquina virtual de Redes y Seguridad II.
- `wireshark`: Wireshark es utilizado en la mayoría de las prácticas de Redes y Seguridad II para monitorear el estado de la red y estudiar los paquetes enviados y recibidos en cada máquina.

- `nano`: La imagen de Debian no incluye ningún editor de texto, por lo que resulta necesario instalar uno para poder modificar archivos a lo largo de la práctica.
- `iputils-ping`: Una de las tareas habituales en todas las prácticas de Redes y Seguridad II es usar `ping` para comprobar la conectividad de red, para lo que se debe instalar este programa.
- `iproute2`: Este paquete es imprescindible para poder crear las infraestructuras de red requeridas en las prácticas, ya que para llevar a cabo esta tarea se deben modificar las direcciones IP y las tablas de enrutamiento de los contenedores.
- `sudo`: Para emular la configuración de la máquina virtual de Debian, se debe crear un usuario diferente a `root` pero que pueda obtener permisos de superusuario a través de `sudo`.

El Dockerfile final es el siguiente:

```
FROM debian:9
# Para evitar que salgan mensajes de la configuración de Wireshark
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y firefox-esr wireshark \
iputils-ping iproute2 sudo nano
RUN groupadd -g 1000 usuario && useradd -m -u 1000 -g usuario usuario \
&& echo usuario:usuario | chpasswd && echo "usuario ALL=(ALL) ALL" \
>> /etc/sudoers && echo "Defaults lecture = never" >> /etc/sudoers
USER usuario
WORKDIR /home/usuario
CMD bash
```

3.2.3. Conexión de red entre los contenedores

Para poder recrear las topologías de red de las prácticas, se necesita que los contenedores estén situados en redes aisladas de Internet e independientes entre sí, y que estas redes se interconecten mediante otros contenedores que actúen como *routers*.

La creación de estas redes es posible tanto en Docker como en Podman, a través del comando `<docker|podman> network create <nombreRed>`. Luego, cada contenedor puede ser conectado a una o varias redes ejecutando `<docker|podman> network connect <nombreRed> <nombreContenedor>`. Estas redes son idénticas a la red `bridge` a la que se conectan los contenedores por defecto durante su creación, por lo que se puede acceder a Internet desde dentro del propio contenedor incluso si solamente está conectado a esta red personalizada. Para interrumpir la conexión a Internet, tan solo es necesario eliminar la ruta por defecto a la puerta de enlace de la red en cada uno de los contenedores.

Los contenedores pueden desempeñar el papel de `routers` con el mismo procedimiento que en las máquinas virtuales, es decir, activando el reenvío de paquetes con `sysctl -w net.ipv4.conf.all.forwarding=1` y cambiando las tablas de enrutamiento de los contenedores para que los paquetes de red se dirijan a los *routers*. Este atributo del *kernel* suele estar ya activado en los sistemas con Docker o Podman instalado, puesto que es necesario para que los contenedores puedan disponer de conexión a Internet. Si no estuviese habilitado, se tendría que modificar desde el sistema anfitrión, ya que la

modificación del núcleo del sistema operativo desde los contenedores está restringida, a menos que se habilite la opción insegura `--privileged`.

De forma predeterminada, no se permite cambiar la configuración de red desde los contenedores, por lo que no se podrían llevar a cabo las tareas propias de la prácticas como establecer nuevas direcciones IP o cambiar las tablas de enrutamiento. No obstante, esta limitación puede ser eliminada si los contenedores se crean agregando la *capability* adicional `NET_ADMIN`.

3.2.4. Uso de aplicaciones de interfaz gráfica

La mayoría de las máquinas virtuales utilizadas en las asignaturas de sistemas y redes cuentan con un sistema de ventanas y un entorno de escritorio completo, con el fin de que la interacción con el sistema por parte del estudiante sea más sencilla. Sin embargo, esto no es así en el caso de los contenedores, ya que, debido a su carácter reducido, no suelen incluir ningún sistema de ventanas. Este hecho hace que, a priori, no sea posible ejecutar aplicaciones gráficas como Wireshark o Firefox dentro de un contenedor. Sin embargo, existen varias posibilidades para solventar este problema.

Una opción podría ser utilizar un sistema de ventanas X como X.Org [92] dentro del contenedor, al igual que sucede en el caso de las máquinas virtuales. De forma similar, también se podría instalar un servidor SSH o VNC para poder acceder remotamente a las aplicaciones gráficas desde el sistema anfitrión. Si bien estas opciones podrían funcionar sin complicaciones, tienen el inconveniente de que el peso de los contenedores aumentaría notablemente, además de que es posible que el rendimiento disminuyese debido a que se ejecutarían más procesos dentro del contenedor.

Existe una posibilidad que no implica aumentar el tamaño de los contenedores, que consiste en compartir el *socket* de X11 de la máquina anfitriona con los contenedores [93]. Gracias a los *bind mounts* de Docker y Podman, que permiten compartir archivos entre el *host* y los contenedores [94], se puede conseguir que las instancias virtuales puedan hacer uso del sistema de ventanas X del anfitrión de forma directa. Además de compartir el *socket*, también es necesario establecer la variable de entorno `DISPLAY` dentro de los contenedores.

Esta alternativa cuenta con un inconveniente, y es que, si no se configura correctamente la seguridad del sistema X, es posible que un atacante pudiera acceder al resto de ventanas de la máquina, incluyendo las del anfitrión. Para evitar esta amenaza, la opción más segura es crear una *untrusted cookie* para cada contenedor, compartida con ellos a través de *bind mounts* [95]. De esta manera, se garantiza que el aislamiento de los contenedores sigue siendo pleno, además de evitar ataques por parte de otros usuarios. Para que esta alternativa funcione, aparte de compartir la *cookie* también se debe configurar la variable de entorno `XAUTHORITY` en cada contenedor, que debe ser igual a la ruta donde se encuentra la *cookie* compartida. La creación de la *cookie* se realiza ejecutando los siguientes comandos, siendo necesario que `xauth` se encuentre instalado en el sistema:

```
Cookiefile=~/containercookie
:> $Cookiefile
xauth -f $Cookiefile generate $DISPLAY . untrusted timeout 3600
Cookie="$(xauth -f $Cookiefile nlist $DISPLAY | sed -e 's/^..../ffff/')"
```

```
echo "$Cookie" | xauth -f "$Cookiefile" nmerge -
```

3.2.5. Compartición de archivos entre contenedores

En muchas prácticas resulta conveniente que exista una manera de poder enviar y recibir archivos entre las distintas instancias. Un caso real puede ser las prácticas de configuración de un túnel VPN con clave estática previamente compartida, donde se debe realizar el intercambio de la clave entre los extremos del túnel. En la actualidad, esta tarea se realiza mediante SCP, una utilidad de copia de archivos a través de SSH, pero si se optase por esta misma opción, se debería contar con un cliente y un servidor SSH en cada contenedor, aumentando su peso.

La opción más conveniente vuelve a ser hacer uso de los *bind mounts*. Por ejemplo, se podría montar la carpeta de usuario del anfitrión en todos los contenedores de la práctica en una ruta como `/mnt/shared`. Así, cada vez que se quisiera compartir un archivo entre los contenedores, o entre un contenedor y el sistema anfitrión, solo habría que copiar el archivo a este directorio.

3.3. Elección del *software* de contenedores

Una vez que se han determinado las características que deben tener los contenedores para poder realizar las prácticas de una manera idéntica a como se realizan con máquinas virtuales, se debe elegir el *software* de gestión de contenedores que se va a utilizar en este proyecto.

Como ya se había expuesto en la sección 2.3.2, en la actualidad las aplicaciones de contenedores más populares son Docker, Podman y LXC. Las dos primeras están centradas en la creación de contenedores de aplicaciones, mientras que LXC utiliza contenedores de sistema. Tal y como se ha descrito en la sección 3.2.1, se va a optar por utilizar contenedores de aplicaciones, por lo que LXC queda descartado.

Entre Docker y Podman no existen muchas diferencias, ya que Podman cuenta con una interfaz de línea de comandos compatible con Docker que le permite realizar las mismas acciones [50]. Un aspecto que diferenciaba a Podman de Docker hasta hace pocos años era que el primero permitía la creación de contenedores sin necesidad de ser `root`, pero esta opción también se encuentra disponible en Docker desde la versión 19.03, como se ha explicado en el apartado 2.3.2. Por otra parte, Podman cuenta con la capacidad de crear `pods`, mientras que Docker no, pero esta herramienta no resulta de utilidad en este proyecto. En cuanto al rendimiento, tanto en los estudios mencionados en el apartado 2.3.4 como en las pruebas realizadas a lo largo de este trabajo no se han detectado diferencias significativas entre ambos programas.

El único motivo que puede hacer que Docker sea una mejor opción es que su uso está mucho más extendido que Podman. Además, Docker lleva más años disponible, ya que su desarrollo comenzó casi cinco años antes. Estos dos hechos suponen que exista una mayor cantidad de documentación y de recursos disponibles para su consulta en el caso de Docker, así como más herramientas compatibles con este programa y que pueden resultar de utilidad en este proyecto, como analizadores de seguridad y vulnerabilidades.

Por esta razón, se elige Docker como *software* para la creación y gestión de los contenedores.

3.4. Despliegue de los contenedores para las prácticas

En Docker, la forma más común de crear y ejecutar nuevos contenedores es a través del comando `docker container run`. Un ejemplo básico de su funcionamiento puede ser el siguiente comando: `docker container run -i -t ubuntu`. Si se consulta la documentación de este comando [96], vemos que se realiza una descarga de la imagen de Ubuntu desde Docker Hub [97] y a continuación se crea un contenedor que ejecutará la aplicación por defecto de la imagen —en este caso, Bash— con las opciones `-i` y `-t`, consistentes en mantener la entrada estándar abierta y asociar un pseudo-terminal respectivamente, de forma que se permita la interacción con el contenedor.

Si bien es cierto que este comando es sencillo y fácil de recordar, puede volverse bastante extenso si se añaden opciones adicionales como el tipo de red del contenedor, la definición de variables de entorno o la creación de volúmenes para compartir ficheros, y si se desea crear más de un solo contenedor a la vez, la tarea se convertirá en muy compleja y pesada para el usuario. Por ello, es necesario automatizar este proceso para que el despliegue de varios contenedores sea rápido y fácil.

Además, la automatización de este proceso supondría una ventaja con respecto a la implementación con máquinas virtuales, puesto que, si recordamos lo expuesto sobre la situación actual al comienzo de este capítulo, actualmente el procedimiento en cada práctica consiste en clonar las máquinas virtuales base tantas veces como sea necesario, configurar las redes a las que debe estar conectada cada instancia y encender las máquinas. Si este proceso se automatiza, se puede obtener una ganancia de tiempo que puede ser aprovechada por los estudiantes para centrarse en los contenidos de la práctica en cuestión.

Docker ya proporciona una herramienta para llevar a cabo este cometido: Docker Compose [98]. Su funcionamiento consiste en definir los contenedores deseados en un archivo con formato YAML denominado `docker-compose.yml`, para posteriormente ejecutar el comando `docker compose up` que creará los contenedores de forma automática según lo especificado en el archivo. Por ejemplo, si se quisiera crear tres contenedores idénticos al que se había indicado al comienzo de la sección, el formato del fichero `docker-compose.yml` sería el siguiente:

```
services:
  primer-contenedor:
    image: ubuntu
    stdin_open: true
    tty: true
  segundo-contenedor:
    image: ubuntu
    stdin_open: true
    tty: true
```

```
  tercer-contenedor:
    image: ubuntu
    stdin_open: true
    tty: true
```

Como se puede ver, Docker Compose es una herramienta muy útil que facilita enormemente la creación de varios contenedores al mismo tiempo. Sin embargo, no se termina de ajustar bien a nuestro proyecto por una razón, y es que su uso requeriría que los estudiantes tuvieran que aprender la sintaxis de los comandos de Docker y las opciones necesarias para crear los contenedores con todo lo necesario para realizar las prácticas.

Debido a este motivo, la automatización del despliegue se llevará a cabo mediante una nueva herramienta creada específicamente para este propósito, que permitirá que los estudiantes puedan proporcionar ciertos parámetros específicos para cada práctica pero abstrayendo el uso de Docker.

Esta herramienta funcionará de una forma similar a Docker Compose, por lo que para cada práctica se tendrá que preparar un archivo de texto en el que se describan las características de los contenedores requeridos. Estos archivos deberán estar escritos en formato YAML —seleccionado por su legibilidad en comparación a otros como JSON—, y tendrán una estructura similar a la siguiente:

```
nombre_practica: ejemplo
contenedores:
  - nombre: maquina1
    imagen: rys
    redes:
    - nombre: 1
      ip: 192.168.1.10/24
    background: true
  - nombre: router
    imagen: rys
    redes:
    - nombre: 1
    - nombre: 2
      ip: 192.168.2.10/24
  - nombre: maquina2
    imagen: rys
    redes:
    - nombre: 2
```

En primer lugar, se debe establecer el nombre de la práctica para poder agrupar posteriormente todos los contenedores según este nombre. A continuación, en el campo `contenedores` se debe escribir la lista de instancias requeridas, y en cada uno de los elementos de la lista se deben establecer como mínimo los campos `nombre`, `imagen` y `redes`. Este último campo vuelve a ser una lista, en la que se debe especificar el nombre de las redes a las que se desea que cada contenedor esté conectado y, opcionalmente, la dirección IP que

se desea asignar en cada interfaz. Si no se proporciona la dirección IP, Docker realizará la asignación de forma automática. Por último, el campo `background` también es opcional, y en caso de que tenga un valor igual a verdadero, la ventana de terminal para interactuar con el contenedor no se creará. Este campo es de utilidad en el caso de que se utilicen contenedores con los que no haga falta interactuar de forma directa, como por ejemplo servidores web, ya que podrían funcionar en segundo plano sin inconvenientes.

De esta forma, la herramienta leerá este archivo y creará los contenedores y redes descritos. El nombre final en ambos componentes seguirá este formato, `<nombre_practica>_<nombre>`, lo que posibilitará que posteriormente se puedan realizar acciones sobre todos los contenedores o redes de la práctica al mismo tiempo, como detenerlos, volverlos a iniciar o destruirlos.

Esta herramienta será la única que tendrán disponible los estudiantes para crear contenedores en el sistema, ya que no se les dará permisos de `sudo` ni se les agregará al grupo `docker` con el fin de evitar que puedan ejecutar contenedores con opciones peligrosas de forma manual. El archivo binario de la herramienta deberá pertenecer al usuario `root` y disponer del permiso `setuid`, para que pueda ser utilizada por usuarios sin privilegios.

3.5. Imágenes disponibles para las prácticas

Más allá de la imagen base descrita en la sección 3.2.2, en muchas prácticas puede resultar útil emplear imágenes adicionales, por ejemplo, que incluyan sistemas de gestión de bases de datos o servidores web, para evitar así tener que instalar manualmente el *software*. Para ello, la opción más sencilla sería especificar en el fichero YAML la imagen deseada, por ejemplo `mysql` [99] o `httpd` [100], y que la herramienta de despliegue se encargase de descargarlas desde el registro correspondiente —habitualmente, Docker Hub—.

Sin embargo, esta opción supondría una amenaza para el sistema porque se podría llegar a ejecutar imágenes que incluyan vulnerabilidades u opciones peligrosas, riesgo que ya se ha evaluado en la sección 2.3.5. Por ello, la herramienta de despliegue no realizará ninguna descarga, sino que estará limitada para utilizar las imágenes que se encuentren de manera local, que tendrán que haber sido construidas o descargadas previamente por el administrador del sistema.

3.6. Ejecución de los contenedores: ¿nativa o sobre una máquina virtual?

Para poder crear los entornos de las prácticas, existen dos alternativas: o bien instalar el motor de Docker directamente en el ordenador que vaya a utilizarse, o bien emplear una máquina virtual exclusivamente para ejecutar los contenedores.

En el caso de que se optase por la primera opción, la ventaja más importante es que se evitaría la sobrecarga propia de ejecutar una máquina virtual. Además, los recursos empleados por los contenedores no se encontrarían limitados por las características de la máquina virtual, sino que podrían disponer de todas las capacidades del ordenador anfitrión sin restricciones —a menos que se establezcan estas restricciones en el momento

de crear los contenedores mediante las opciones `--cpu`, `--memory` y sus derivados [101]—. Todo ello permitiría que el rendimiento teórico fuera superior, lo cual sería más beneficioso de cara a alcanzar el objetivo de este trabajo.

Sin embargo, a pesar de que los contenedores proporcionan un aislamiento confiable gracias a los *namespaces* y a que no se establecerán opciones peligrosas durante su creación, hay que tener en cuenta que la intención de este proyecto es implementarlo en un laboratorio docente real, por lo que se debe ser especialmente cuidadoso y aplicar un punto de precaución extra con respecto a las características de seguridad del entorno.

Una opción para mejorar la seguridad del sistema podría ser emplear la funcionalidad `rootless` de Docker, en la que el demonio de Docker se ejecuta sin permisos de `root` y los usuarios del contenedor están mapeados a otros distintos fuera del contenedor. De esta forma, si el usuario `root` del contenedor lograse escapar del mismo, en el sistema anfitrión sería visto como un usuario sin privilegios. Sin embargo, esta posibilidad ha sido evaluada y se han encontrado problemas con los permisos de los ficheros compartidos con los contenedores a través de los *bind mounts*, por lo que no se ha podido aplicar esta medida. Por otra parte, cabe destacar que esta funcionalidad también ha sido comprobada con Podman, pero igualmente ha sido descartada porque en este caso no se le asignarían direcciones IP a los contenedores, lo que dificultaría enormemente la posibilidad de establecer comunicaciones entre las distintas instancias durante el desarrollo de las prácticas [102].

Viendo que actualmente no es posible utilizar contenedores `rootless`, la posibilidad más razonable y sencilla de implementar para aumentar la seguridad del sistema es ejecutar los contenedores dentro de una máquina virtual. Gracias a ella, se obtiene otra capa de aislamiento entre el sistema operativo anfitrión y el entorno de prácticas, reduciéndose aún más las posibilidades de que el laboratorio docente se pueda llegar a ver afectado. Sin embargo, la sobrecarga será teóricamente mayor que si se ejecutasen los contenedores de forma nativa.

No obstante, consideramos que esta posible pérdida de rendimiento puede ser asumible si, a cambio, se puede obtener una ganancia sustancial en la seguridad del sistema. Por ello, finalmente se optará por instalar el motor de Docker y crear los contenedores dentro de una máquina virtual.

Aun así, el desarrollo de los *scripts* y herramientas utilizadas para crear y lanzar los contenedores se intentará hacer de forma que sean independientes del sistema operativo de la máquina virtual, permitiendo incluso que, si el usuario lo deseara, se puedan ejecutar fuera de una máquina virtual. Asimismo, el rendimiento de los contenedores de las prácticas será evaluado tanto dentro como fuera de la máquina virtual en la sección 5.3.

3.6.1. Elección del hipervisor

En primer lugar, es necesario decidir qué hipervisor va a ser el encargado de crear y gestionar la máquina virtual del proyecto. Como ya vimos en la sección 2.2, existen dos categorías de hipervisores: de tipo 1 y de tipo 2. Los hipervisores de tipo 1 destacan por su menor latencia, al no requerir de un sistema operativo que regule los recursos asociados a las máquinas virtuales, por lo que suelen ser los elegidos en servidores y entornos de producción. Por el contrario, los hipervisores de tipo 2 son más adecuados para

usuarios finales y entornos de desarrollo, ya que es más sencillo y rápido administrar las máquinas virtuales con esta clase de hipervisores. Por ello, debido a que en este proyecto es más importante la facilidad de uso para los estudiantes que la rapidez, escogeremos un hipervisor de tipo 2.

En esta tabla se comparan las características principales de los hipervisores de tipo 2 más utilizados en la actualidad:

	Oracle VM VirtualBox	**VMWare Workstation Player**	**Parallels Desktop**
Plataformas compatibles	Windows, macOS, Linux, Solaris	Windows, Linux	macOS
Sistemas operativos virtualizables	Windows, Linux, OS/2, macOS (experimental), Solaris, FreeBSD, DOS	Windows, Linux, OS/2, Solaris, FreeBSD, DOS	Windows, Linux, OS/2, macOS, Solaris, FreeBSD
Precio	Gratuito	Gratuito para uso personal y educativo	99,99€ / ordenador
Licencia	GPL v2	Propietaria	Propietaria
Soporte formato OVF	Sí	Sí	No
Soporte para instantáneas	Sí	No	No

Tabla 3.1: Comparativa entre hipervisores de tipo 2 (los colores indican cómo de favorable resulta cada alternativa, siendo de menos a más favorable rojo, amarillo y verde en este orden. Azul significa neutro). Información obtenida de [16], [27], [28]

Parallels Desktop es la opción menos recomendable debido a su alto precio y a que es solamente compatible con sistemas macOS. Entre Oracle VM Virtualbox y VMWare Workstation Player, las diferencias más notables —más allá de su licencia— son la imposibilidad de instalar VMWare Workstation Player en macOS y Solaris y la capacidad de Oracle VM Virtualbox de poder generar instantáneas de la máquina virtual. Las instantáneas permiten guardar el estado de una máquina virtual en un momento determinado para poder volver a él si es necesario, con lo que se puede «experimentar» de una forma más cómoda al saber que los cambios realizados son fácilmente reversibles.

Por ello, teniendo en cuenta estas dos ventajas, finalmente nos decantamos por Oracle VM VirtualBox. No obstante, si en el futuro se quisiera utilizar VMWare Workstation Player, el proceso de migración resultaría muy sencillo debido al soporte del formato Open Virtualization Format (OVF) por parte de ambos hipervisores.

3.6.2. Elección del sistema operativo de la máquina virtual

Para elegir el sistema operativo que se va a ejecutar en la máquina virtual del entorno de contenedores, se deben tener en cuenta una serie de aspectos. En primer lugar, esta elección estará restringida a aquellos sistemas operativos en los que se pueda ejecutar Docker.

Actualmente Docker puede ser utilizado en sistemas operativos Windows, macOS y Linux [103]. En los dos primeros casos, la forma de realizar la instalación es a través de la

aplicación Docker Desktop [104], la cual incluye el motor de Docker, el cliente de línea de comandos de Docker, Docker Compose, Docker Content Trust, Kubernetes, y Credential Helper. Por otro lado, si se desean utilizar estas herramientas en Linux, se debe instalar cada una de ellas por separado.

Utilizar macOS como sistema operativo anfitrión supondría muchos problemas, debido a que el soporte por parte de VirtualBox todavía se encuentra en fase experimental [105]. Además, existen restricciones tanto técnicas como de licencia que limitan el uso de este sistema operativo a hardware específico. Por estos motivos, no resulta viable la elección de macOS como sistema base.

En cuanto a Windows, existirían menos problemas técnicos debido a que está ampliamente soportado para su uso por parte de VirtualBox. Pero optar por Windows tiene dos claras desventajas:

1. Se necesitaría obtener una licencia de uso por cada máquina virtual empleada, lo que supondría un mayor coste en el caso de que se utilizase en laboratorios docentes reales.
2. El uso de Docker Desktop en una máquina virtual de Windows no está soportado por Docker debido a la forma de virtualizar el hardware por parte de los hipervisores [106].

Por tanto, Windows también quedaría descartado.

De este modo, la única opción que queda disponible es utilizar una distribución de Linux. A diferencia de Windows, en este caso no existen problemas con respecto a instalar el motor de Docker y ejecutar contenedores dentro de una máquina virtual, e incluso algunos expertos en Docker aseguran que «dependiendo de lo que necesites hacer, una máquina virtual puede ser el mejor lugar para alojar tus contenedores» [107].

Dentro de Linux, actualmente existe una cantidad inabarcable de distribuciones disponibles para ser instaladas, diferenciándose principalmente en el uso para el que están diseñadas: ciberseguridad, servidores, educación, investigación... En nuestro caso, nos interesa centrarnos únicamente en tres clases de distribuciones:

De propósito general

Como su nombre indica, se caracterizan por estar destinadas a poder ser utilizadas en prácticamente cualquier ámbito. Suelen incluir una gran cantidad de software como editores de texto, navegadores, compiladores o reproductores de vídeo con el objetivo de que no sea el usuario el encargado de instalar estos programas, permitiendo que la experiencia de uso sea más satisfactoria. Pueden utilizarse tanto con interfaz gráfica como a través de línea de comandos. Algunas de las distribuciones más conocidas son Ubuntu, Debian, Fedora o Linux Mint.

Enfocadas a flujos de trabajo con contenedores

Desde hace algunos años, se han popularizado algunas distribuciones diseñadas especialmente para la ejecución de contenedores. Dependiendo de la distribución, este enfoque se implementa de distinta manera. Las más populares en la actualidad son:

- Fedora CoreOS [108]: Esta distribución nació a partir de la fusión entre CoreOS Container Linux y Fedora Atomic Host. Sigue una filosofía de sistema operativo

inmutable, lo que significa que el directorio raíz se monta en modo solo lectura, permitiendo la escritura únicamente en los directorios /etc y /var. De esta forma, si se quiere modificar o actualizar el sistema operativo, en lugar de alterar el sistema en tiempo de ejecución, se crea una nueva capa o imagen, cambiando a esta en el momento en que se reinicie el sistema. De ahí el concepto «inmutable», porque todas las máquinas que estén ejecutando la misma imagen del sistema operativo mantendrán siempre el mismo estado. Por otra parte, las actualizaciones se ejecutan de forma automática, pero gracias a la estructura por capas, es posible volver al estado previo a la actualización si es necesario. Este diseño hace que sea una buena elección para la ejecución de aplicaciones dentro de contenedores en entornos de producción, especialmente en clústers o entornos alojados en la nube debido a que, de esta manera, todas las máquinas del clúster serán idénticas entre sí.

- Fedora Silverblue [109]: Esta versión de Fedora toma algunas de las ventajas de CoreOS, como la inmutabilidad del sistema operativo o la posibilidad de poder volver a una versión anterior tras una actualización. Sin embargo, a diferencia de CoreOS, esta distribución está enfocada al uso en las etapas de desarrollo. Por ello, más allá de estas diferencias se comporta de una forma muy similar a Fedora Workstation. Incluye GNOME como entorno de escritorio y Flatpak, una utilidad para instalar aplicaciones de forma aislada del resto del sistema, incluyendo todas las dependencias necesarias.
- Flatcar Container Linux [110]: Se trata de un derivado de CoreOS Container Linux, por lo que comparte muchas similitudes con Fedora CoreOS: inmutabilidad del sistema operativo, actualizaciones automáticas, ausencia de gestor de paquetes...
- RancherOS [111]: Apuesta por un enfoque minimalista, en el que todos los procesos se ejecutan como contenedores controlados por Docker, incluidos los servicios del sistema. Además, solo incluye los servicios necesarios para ejecutar Docker, con lo que su tiempo de arranque y su uso de disco duro son muy reducidos. Sin embargo, en octubre de 2021 dejó de estar en mantenimiento.

En el ámbito de este trabajo se deben descartar algunas de estas distribuciones por varios motivos. En primer lugar, RancherOS no es una opción viable debido a la falta de soporte y actualizaciones. En segundo lugar, Fedora CoreOS y Flatcar Container Linux no están diseñadas para su uso en equipos de escritorio, sino que su caso de uso más común es en servidores de producción. Por esta razón, no incluyen sistemas de ventanas ni entornos de escritorio, característica imprescindible en nuestro proyecto para poder ejecutar aplicaciones gráficas dentro de los contenedores. Debido a estos motivos, Fedora Silverblue es la única alternativa dentro de esta clase de distribuciones que se ajusta a nuestros requerimientos.

Distribuciones mínimas

Estas distribuciones tienen unos requisitos muy poco exigentes en lo referente a frecuencia de CPU o cantidad de memoria RAM. Además, suelen ocupar muy poco espacio en disco y ofrecen un tiempo de arranque más rápido en comparación con otros sistemas operativos más pesados. Se utilizan en entornos y máquinas especializadas donde estos aspectos son cruciales, como *routers*, sistemas embebidos, ordenadores antiguos... En nuestro caso, resultan de interés porque se busca un sistema operativo ligero con el fin de destinar la mayor cantidad de recursos posible a la ejecución de contenedores.

El ejemplo más representativo es Alpine Linux [112]. Su eslogan, «Small. Simple. Secure.», resume adecuadamente la filosofía de esta distribución. Ofrece las utilidades habituales de Linux a través de BusyBox, y emplea musl como biblioteca estándar de C en lugar de glibc. Además, incluye una cantidad mínima de software ya instalado, aunque es posible instalar programas adicionales a través de apk, su gestor de paquetes. Todo ello la convierte en una distribución muy ligera, ocupando tan solo 130 MB en su instalación básica. Debido a esta razón, se utiliza habitualmente como base para imágenes de contenedores, bajando su peso en este caso hasta los 8 megabytes. Por otra parte, todos los binarios son compilados como *Position Independent Executables* (PIE) e incorporan protección contra *stack-smashing*, lo que previene algunos ataques que aprovechen vulnerabilidades de *stack buffer overflow*.

A continuación se muestra una tabla en la que se comparan las características principales de tres de las distribuciones mencionadas, una por cada ámbito de uso. En el caso de las distribuciones de propósito general, se ha escogido Debian por ser la distribución que se emplea actualmente en la máquina virtual principal de la asignatura Redes y Seguridad II. De esta forma, se puede ver claramente si existen mejoras con respecto al estado actual si se escoge un nuevo sistema operativo o si, por el contrario, no resulta necesario cambiar de elección.

	Debian	**Fedora Silverblue**	**Alpine**
Soporte para Docker	Sí	Sí	Sí
Gestor de paquetes	Sí (dkpg)	Sistema diferente al tradicional de Fedora (rpm-ostree y Flatpak en lugar de rpm/yum)	Sí (apk)
Incluye entorno de escritorio por defecto	Sí (GNOME)	Sí (GNOME)	No, pero es posible instalar XFCE, GNOME, KDE o MATE mediante apk
Requisito de espacio mínimo / recomendado	2 GB / 10 GB	15 GB / 20 GB	130 MB / 1 GB
Requisito de memoria RAM mínima / recomendada	1 GB / 2 GB	2 GB / 4 GB	512 MB / 2 GB
Inmutabilidad del sistema operativo	No	Sí	No
Optimizadas para el uso en máquinas virtuales	No	No	Sí (disponible una versión del kernel optimizada)

Tabla 3.2: Comparativa entre distribuciones Linux (los colores indican cómo de favorable resulta cada alternativa, siendo de menos a más favorable rojo, amarillo y verde en este orden). Información obtenida de [113], [114], [115]

Analizando esta tabla, podemos observar que la mayor ventaja de Debian es que se trata de un sistema operativo sencillo y confiable, que cuenta con un gestor de paquetes clásico y un entorno de escritorio por defecto. No obstante, no existe un motivo especial que la convierta en la mejor elección en este caso de uso específico, ya que ni está optimizada para el uso en máquinas virtuales ni para flujos de trabajo con contenedores.

En el caso de Fedora Silverblue, su mayor virtud es la inmutabilidad del sistema y la posibilidad de poder hacer *rollback* fácilmente tras una actualización. Estas características son muy útiles en el caso de entornos de desarrollo con el objetivo de mantener sistemas idénticos y minimizar la aparición de posibles *bugs*. Sin embargo, en nuestro proyecto no se van a requerir entornos exactamente iguales para ejecutar los contenedores de las prácticas ni tampoco es necesario contar con un sistema siempre actualizado, por lo que no está claro que se les vaya a sacar partido a estas funciones. Por el contrario, Silverblue es la distribución más exigente de las tres en cuanto a requerimientos de disco y memoria RAM, aspecto que se debe valorar negativamente debido a que, como el sistema operativo se va a ejecutar de forma virtualizada, supone una mayor sobrecarga, dejando menos recursos disponibles para los contenedores. Por último, es interesante destacar que, si bien no supone una desventaja como tal, el utilizar rpm-ostree y Flatpak en lugar de un gestor de paquetes estándar supone una diferencia a la que los administradores del sistema se tendrían que adaptar.

Por último, Alpine destaca por ser la distribución más ligera de la comparativa, algo muy relevante por las razones ya mencionadas en el párrafo anterior. Además, es posible realizar la instalación usando un *kernel* optimizado para ser ejecutado en máquinas virtuales, lo que reduce el espacio ocupado en disco. El mayor inconveniente puede ser que, debido a su propio diseño, cuenta con muy poco software preinstalado —ni siquiera tiene un entorno de escritorio—, pero es posible instalar todo lo necesario para la creación del a través de `apk`, su gestor de paquetes.

En conclusión, considerando que las propiedades específicas de Fedora Silverblue no serán de utilidad en el ámbito de este trabajo, Alpine se muestra como la mejor elección posible, especialmente por su ligereza que le hace diferenciarse de las distribuciones de propósito general como Debian. Por ello, finalmente será Alpine el sistema operativo que servirá como base del entorno para el uso de los contenedores.

Capítulo 4

Creación del entorno de contenedores

4.1. Instalación y configuración de Alpine

4.1.1. Creación de la máquina virtual

En primer lugar, es necesario descargar la imagen de instalación de Alpine desde su página web [112]. Escogeremos la versión `virt` debido a que, tal y como se ha visto en el anterior capítulo, está optimizada para el uso en máquinas virtuales.

Una vez descargada, creamos una máquina virtual en VirtualBox. En cuanto a la configuración de la máquina, el único aspecto reseñable es el tipo y versión, porque Alpine no está incluido en la lista de versiones de VirtualBox, a diferencia de otras distribuciones más populares como Ubuntu o Red Hat. Debido a esto, se tendrá que seleccionar *Other Linux (64 bits)*.

Sobre la memoria RAM y el número de procesadores asignados, de momento se establecerán 1024 *megabytes* y un solo procesador, lo que es suficiente para llevar a cabo la instalación y configuración de Alpine, incluyendo entorno gráfico. Estas especificaciones pueden ser cambiadas en un futuro, y de hecho se deberían de ajustar al *hardware* de las máquinas en donde vaya a ser ejecutado este entorno. No obstante, estos valores se mantendrán a lo largo de todo el proyecto debido a que coinciden con las características de la máquina virtual de Debian de Redes y Seguridad II. Así, se podrá realizar una comparativa más equitativa entre ambas opciones.

4.1.2. Instalación básica de Alpine

Al encender la máquina virtual por primera vez, Alpine se ejecuta en modo LiveCD, permitiendo al usuario probar el sistema operativo sin necesidad de llevar a cabo la instalación. Tras iniciar sesión con el usuario `root` sin contraseña, se debe ejecutar `setup-alpine` para comenzar el proceso de instalación. Este comando configura los elementos esenciales del sistema operativo, como la zona horaria, la distribución de teclado o el servidor que utilizará `apk` para descargar los paquetes. El aspecto más relevante en este

apartado es el modo de instalación, pudiendo escoger entre `none` —el sistema operativo se ejecuta enteramente desde la memoria RAM—, `data` —igual que `none` pero la partición de *swap* y el directorio `/var` se montan desde almacenamiento persistente— o `sys` — instalación tradicional en un disco duro—. Esta última será la opción seleccionada en nuestro caso.

4.1.3. Configuración gráfica

Una vez finalizada la instalación de Alpine, se configurará el entorno gráfico. En primer lugar, es necesario activar los repositorios `edge`, `community` y `testing` de `apk`, para lo que se tendrán que descomentar las líneas que apuntan a dichos repositorios en el archivo `/etc/apk/repositories`. A continuación, se debe ejecutar el comando `setup-xorg-base`, que instalará Xorg como servidor gráfico, así como otras utilidades necesarias. Una vez finalice este proceso, es necesario instalar el controlador de vídeo correspondiente a VirtualBox y las *Guest Additions*, con el objetivo de obtener un mejor rendimiento visual. Para ello, se debe ejecutar el siguiente comando:

```
apk add xf86-video-vboxvideo virtualbox-guest-additions
virtualbox-guest-additions-openrc virtualbox-guest-additions-x11
```

Por último, se debe instalar el entorno de escritorio. En este caso, hemos optado por XFCE, un entorno ligero y fácil de utilizar. Para instalarlo, hay que ejecutar:

```
apk add xfce4 xfce4-terminal xfce4-screensaver lightdm-gtk-greeter
elogind polkit-elogind
```

Después de que se haya completado la instalación, hay que añadir los servicios del bus de escritorio y del inicio de sesión gráfico al arranque del sistema con los comandos `rc-update add dbus && rc-update add lightdm`. Finalmente, se puede arrancar el entorno gráfico reiniciando el sistema.

4.1.4. Instalación de Docker y ajustes adicionales

El primer ajuste que se debe realizar tras la instalación gráfica es configurar X11 para poder utilizar la distribución de teclado española. Para ello, hay que agregar el siguiente fragmento al archivo `/etc/X11/xorg.conf` y reiniciar el sistema:

```
Section "InputClass"
    Identifier         "Keyboard Default"
    MatchIsKeyboard    "yes"
    Option             "XkbLayout" "es"
EndSection
```

Para instalar el motor de Docker en Alpine solamente hay que descargar e instalar el paquete correspondiente a través de `apk`. Tras su instalación, se debe iniciar el *daemon* y establecer que se active durante el proceso de arranque:

```
apk add docker && service docker start && rc-update add docker boot
```

Si se desea ejecutar contenedores que incluyan aplicaciones gráficas como Wireshark, también es necesario instalar `xauth` mediante `apk`. Este programa gestiona los permisos del servidor X para que las aplicaciones ejecutadas en los contenedores puedan acceder al servidor gráfico de la máquina virtual.

En segundo lugar, es necesario crear un usuario sin permisos de superusuario que será utilizado por los estudiantes para realizar las prácticas. Además, este usuario debe pertenecer al grupo adicional `vboxsf` si se desea utilizar la función de carpetas compartidas de VirtualBox. El usuario puede ser creado ejecutando el siguiente comando:

```
useradd -m -G vboxsf usuario
```

Además, es necesario establecer la contraseña del nuevo usuario para que pueda ser utilizado. Para llevar a cabo esta tarea, se debe ejecutar `passwd usuario`.

A continuación, se debe proceder a la activación de las *Guest Additions* relacionadas con X11 cuando `usuario` inicie sesión. Estos añadidos activan distintas opciones que hacen que la experiencia de uso de VirtualBox sea más satisfactoria. Para ello, se puede ejecutar `/usr/sbin/VBoxClient` con estos argumentos:

- `--checkhostversion`: Inicia el servicio de notificación de versión del anfitrión.
- `--clipboard`: Permite compartir el portapapeles entre la máquina anfitriona y la invitada.
- `--draganddrop`: Habilita el intercambio de archivos entre ambas máquinas mediante *drag and drop*.
- `--seamless`: Permite usar el modo de escritorio fluido.
- `--vmsvga`: Reescala la resolución de pantalla de la máquina invitada si se utiliza el controlador de vídeo VMSVGA (activado por defecto).

Estos comandos deben ser lanzados por el usuario `root` para su correcto funcionamiento. Sin embargo, las *Guest Additions* solo pueden ser activadas después de haber iniciado sesión en el entorno gráfico, por lo que no puede ser automatizado a través de un *script* de `init`. Para solucionar este problema, hay que seguir los siguientes pasos:

1. Se debe instalar `sudo` con el comando `apk add sudo` y, mediante `visudo`, añadir la siguiente línea al archivo `/etc/sudoers`:

   ```
   usuario ALL = (root) NOPASSWD: /usr/sbin/VBoxClient
   ```

   Esto permite a `usuario` ejecutar `VBoxClient` como superusuario sin que se le solicite una contraseña.

2. Por cada opción de `VBoxClient` que se quiera habilitar, se debe crear un archivo con la extensión `.desktop` en la carpeta `.config/autostart` del usuario. Este archivo no requiere ningún nombre específico, pero debe tener el siguiente formato —cambiando `exec` y `name` acordemente a la opción deseada—:

```
[Desktop Entry]

Type=Application
Exec=sudo /usr/sbin/VBoxClient --checkhostversion
Name=VBox Check host version
```

Tras instalar todos los elementos necesarios y configurar correctamente el sistema, el espacio de disco ocupado es de aproximadamente 730 megabytes, muy por debajo de lo que ocuparía una instalación similar de Debian o Fedora Silverblue. Si se comprueba el tamaño de la carpeta de la máquina virtual desde fuera, este tamaño sube hasta los 987 MB. Ambas mediciones se pueden consultar en las figuras 4.1 y 4.2 respectivamente.

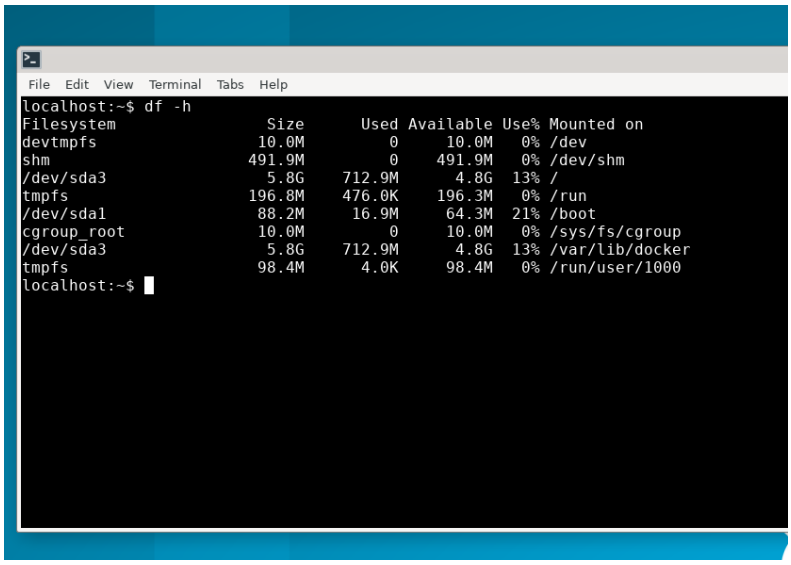

Figura 4.1: Espacio de disco ocupado por Alpine medido desde dentro de la máquina virtual

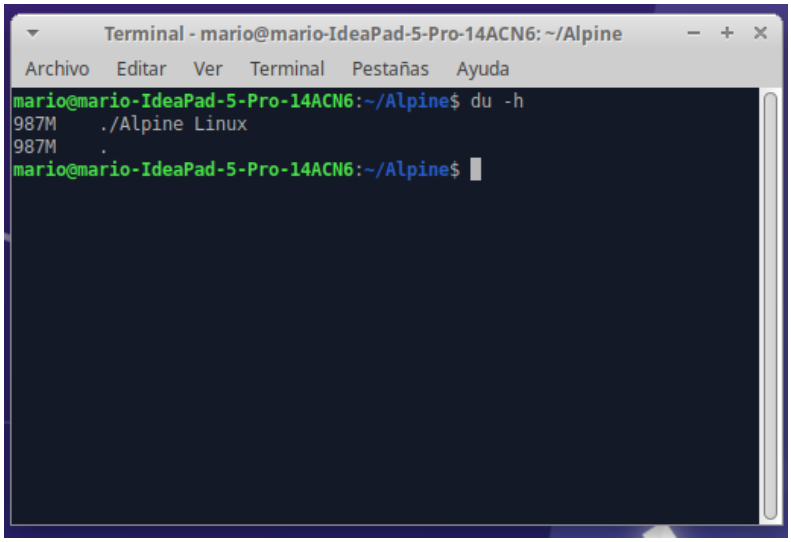

Figura 4.2: Espacio de disco ocupado por Alpine medido desde fuera de la máquina virtual

4.2. Desarrollo de la herramienta de despliegue de contenedores

4.2.1. Versión preliminar de la herramienta

La primera versión de la herramienta consistía en un *script* de *shell* que, a su vez, hacía uso de la interfaz de línea de comandos de Docker, ya que esta es la forma más sencilla de interactuar con el demonio de Docker. En Alpine, la *shell* que viene instalada por defecto es Almquist Shell (Ash), que cumple el estándar POSIX pero con un peso bastante inferior a otras alternativas más conocidas como Bash.

Aparte del propio Docker, este *script* utilizaba otros programas externos para su correcto funcionamiento: `yq`, para extraer la información de los ficheros YAML; `ipcalc` —incluido en BusyBox, que viene instalado por defecto en Alpine Linux—, para obtener la dirección IP de la red a partir de una dirección en formato CIDR; `xauth`, para la gestión de los permisos del sistema de ventanas X y `xfce4-terminal`, para la creación de las terminales que permitan la interacción con los contenedores.

El *script* completo se puede consultar en el apéndice A.

4.2.2. Versión final de la herramienta

Si bien es cierto que la versión de la herramienta escrita en Ash proporcionaba un rendimiento aceptable y sirvió para analizar de una forma sencilla qué opciones debían poseer los contenedores para su correcto funcionamiento, existían tres desventajas destacables.

En primer lugar, al utilizar la interfaz de línea de comandos de Docker se estaba empleando un intermediario entre nuestra aplicación y el demonio de Docker. Sin embargo, se puede interactuar directamente con el demonio de Docker a través de su API RESTful, que permite ejecutar cualquier orden disponible en Docker a través de peticiones HTTP dirigidas al *socket* Unix del demonio. A su vez, estas llamadas a la API se encuentran contenidas en los SDK de Docker Engine, disponibles tanto para Python como para Go [116]. Utilizando estos SDK, es posible crear clientes personalizados para Docker de una forma rápida y sencilla —de hecho, tanto la propia CLI de Docker como Docker Compose no son más que aplicaciones escritas en Go que hacen uso del SDK para la ejecución de sus tareas—.

En segundo lugar, en el *script* de Ash la gestión de cada contenedor y de cada red —ya fuera en el momento de su creación, destrucción o cualquier otra orden disponible— se llevaba a cabo de forma secuencial. Aunque se podría utilizar el operador `&` para mandar ciertas operaciones a segundo plano y seguir con la ejecución de otras partes del código, los sistemas de sincronización entre procesos en Ash son muy rudimentarios, lo que dificulta enormemente el control del flujo en caso de que haya que esperar a la finalización de alguna tarea o si se produce algún error. El uso de un lenguaje que permita la programación concurrente, junto al uso del SDK del Docker Engine, aumentaría notablemente el rendimiento de la aplicación.

Finalmente, si se tiene en cuenta que la ventaja de Ash es su peso reducido, esto hace que no se dispongan de algunos elementos comunes en otros lenguajes de programación;

por ejemplo, es imposible crear *arrays* de elementos. Aunque este problema puede ser solventable a través de otras construcciones disponibles, la sintaxis puede llegar a ser bastante ilegible, lo que dificultaría el mantenimiento de la herramienta a largo plazo. Por poner un ejemplo, este es el fragmento de código de la herramienta encargado de crear las terminales para la interacción con los contenedores —creado a partir de los ejemplos disponibles en [117]—:

```
crear_terminal() {
    primer_nombre=$1
    shift
    first=1
    for nombre; do
        if [ "$first" = 1 ]; then set --; first=0; fi
        set -- "$@" --tab -e "ash -c 'docker container attach $nombre;
        exec ash'"
    done
    xfce4-terminal -e "ash -c 'docker container attach $primer_nombre;
    exec ash'" "$@"
}
```

Ash, al igual que el resto de lenguajes de comandos disponibles en Linux, puede ser una alternativa recomendable para construir aplicaciones que sirvan para automatizar tareas, como en este caso la creación y gestión de contenedores. Pero si la complejidad de la aplicación aumenta, la mantenibilidad de estos *scripts* puede llegar a ser muy baja debido a múltiples razones: la falta de objetos o estructuras, la imposibilidad de establecer argumentos en la definición de las funciones o la dependencia hacia las variables globales, entre otros aspectos. De esta forma, cualquier cambio que sea necesario realizar en un futuro será más complicado de llevar a cabo y aumentará la probabilidad de que aparezcan nuevos errores.

Debido a estos motivos, se decidió reescribir la herramienta de despliegue y gestión de los contenedores en otro lenguaje de programación que cumpliese estos requisitos: uso de la API de Docker Engine, posibilidad de programación concurrente y alta mantenibilidad del código desarrollado.

Aunque existen bibliotecas creadas por la comunidad para otros lenguajes de programación, el SDK de Docker Engine se encuentra disponible oficialmente solo para Python y para Go. Cualquiera de estos dos lenguajes sería una buena elección para crear nuestro nuevo cliente; sin embargo, se optó por Go por dos motivos:

- Python es un lenguaje interpretado, mientras que Go es un lenguaje compilado. Esto significa que, en caso de utilizar Python, hubiese sido necesario instalar el intérprete de Python en la máquina virtual de Alpine, aumentando su peso. Además, en líneas generales, el rendimiento de los programas compilados suele ser superior al de los programas interpretados, ya que no es necesario traducir las instrucciones a código máquina en tiempo de ejecución.
- La concurrencia en Go es muy sencilla de implementar: solamente hace falta escribir el operador `go` delante de las llamadas a cada función que se quiera ejecutar concu-

rrentemente. Además, ofrece otras construcciones como los canales o los `errgroups` [118] que facilitan la comunicación entre los hilos creados.

El uso de Go permite además prescindir tanto de `yq` como de `ipcalc`, ya que se puede recurrir a otros paquetes que realicen la misma función: `yaml.v3` para la lectura del archivo YAML, desarrollado por Canonical [119], y `net`, que forma parte de la biblioteca estándar de Go [120]. Por defecto estos paquetes se enlazan estáticamente al binario generado, por lo que no existen problemas de dependencias en ejecución —aunque para evitar problemas de enlazado derivados del uso del paquete `net`, la compilación se debe realizar estableciendo la variable de entorno `CGO_ENABLED=0`—. Sin embargo, sigue siendo necesario que tanto `xauth` como `xfce4-terminal` se encuentren instalados en el sistema, debido a que no existen paquetes que puedan suplir su funcionalidad. Por ello, tendrán que ser ejecutados a través del paquete `os/exec` [121].

La única desventaja que puede tener Go frente al *script* de *shell* previamente desarrollado es su tamaño, tanto en líneas de código como en espacio ocupado en disco. El código desarrollado en Go contiene 859 líneas de código, frente a las 287 de la versión en Ash. Asimismo, el binario compilado de Go pesa aproximadamente 12 MB, mientras que el *script* solo usa unos 13 KB debido a que solo incluye texto plano. Sin embargo, consideramos que esto no supone un problema ya que en ningún caso se trata de un tamaño excesivamente grande, y las ventajas anteriormente mencionadas superan ampliamente a esta desventaja.

Para atajar el problema de la mantenibilidad, el desarrollo de la herramienta se ha separado en dos partes. Por un lado, se ha creado un paquete denominado LabsOnContainers, concebido como una API que encapsula las llamadas necesarias al SDK de Docker para crear y gestionar los entornos de prácticas. Por otra parte, se ha desarrollado en otro paquete —`main`— el ejecutable en sí, que en este caso se trata de una interfaz de línea de comandos que llama a las funciones de LabsOnContainers.

De esta manera, si en un futuro se quisiese crear una interfaz gráfica para gestionar los contenedores, solo sería necesario crear la parte de la vista, ya que el modelo está disponible para ser usado en el paquete de LabsOnContainers. Además, si se deseara aumentar las opciones disponibles a la hora de gestionar entornos —por ejemplo, obtener los procesos en ejecución en los contenedores del entorno—, solo habría que añadir una función a la API y llamarla desde el cliente.

A continuación se explican los comandos disponibles, la forma de ejecutarlos y los pasos que se siguen en cada uno de ellos, incluyendo algunos fragmentos relevantes del código. Asimismo, la documentación del paquete LabsOnContainers se puede consultar en [122]. Por último, en el apéndice B se incluye un manual de uso de la herramienta.

Creación del entorno

Para crear un entorno en el que se pueda llevar a cabo una práctica, es necesario ejecutar el comando `./LabsOnContainers -c <nombreFichero.yaml>`. Si el archivo existe y sigue el formato especificado en el apartado 3.4, se crea una estructura de tipo `LabEnviroment` con la información del archivo, y se llama a la función `CreateEnviroment` de LabsOnContainers.

```
func createLabEnviroment(filePath string) {
    fmt.Println("Creando el entorno de laboratorio...")

    file, err := os.ReadFile(filePath)
    if err != nil {
        fmt.Printf("error while opening file: %v\n", err)
        os.Exit(1)
    }

    var labEnv labsoncontainers.LabEnviroment

    err = yaml.Unmarshal(file, &labEnv)
    if err != nil {
        fmt.Printf("error while parsing yaml file: %v\n", err)
        os.Exit(1)
    }

    _, err = labsoncontainers.CreateEnviroment(&labEnv)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }

    (...)
```

```
// LabEnviroment represents the structure of a lab enviroment. It is
composed by the lab name and a list of LabContainer.
type LabEnviroment struct {
    LabName    string         `yaml:"nombre_practica"`
    Containers []LabContainer `yaml:"contenedores"`
}

// LabContainer represents the structure of a lab container. It is
composed by its name, the image name the container will use, a list of
LabNetwork and the background field: if it is set to true, a terminal
window will not be created for interacting with the container.
type LabContainer struct {
    Name       string       `yaml:"nombre"`
    Image      string       `yaml:"imagen"`
    Networks   []LabNetwork `yaml:"redes"`
    Background bool         `yaml:"background,omitempty"`
    ID         string
}

// LabNetwork represents the structure of a lab network. It is composed
by its name and, optionally, the IP that will be used by the container
in the network.
```

```
type LabNetwork struct {
    Name string `yaml:"nombre"`
    IP   string `yaml:"ip"`
}
```

La función `CreateEnviroment` realiza tres pasos: primero, destruye los entornos y sus contenedores y redes asociados en caso de que exista alguno con un nombre coincidente. A continuación, se crea una lista sin duplicados con los nombres de las redes y sus direcciones IP en el caso de que se hayan especificado. Finalmente, se crean las redes y los contenedores de forma concurrente, usando el paquete `errgroups` para cancelar todas las rutinas en el caso de que se produzca un error en alguna de ellas. Si efectivamente se produce un error, se destruye el entorno actual para eliminar aquellos contenedores y redes que sí se hayan podido crear.

```
// CreateEnviroment creates a lab enviroment using Docker Engine SDK and
LabEnviroment type.

// First, it destroy any other lab enviroment with the same name using
DestroyEnviroment, then it creates all the desired networks and finally,
it creates all the containers. On success, it returns a map of the
created containers names and their IDs.

// Note that, during container creation, it will not be pulled any
image, so the desired images will have to be previously built or pulled.
func CreateEnviroment(labEnv *LabEnviroment) (map[string]string, error)
{
    if labEnv.LabName == "" {
        return nil, fmt.Errorf("error while creating enviroment: lab
        name cannot be empty")
    }

    err := DestroyEnviroment(labEnv.LabName)
    if err != nil {
        return nil, fmt.Errorf("error while creating enviroment: %w",
        err)
    }

    networks, err := parseNetworks(labEnv)
    if err != nil {
        return nil, fmt.Errorf("error while creating enviroment: %w",
        err)
    }

    err = createNetworks(networks)
    if err != nil {
        createErr := fmt.Errorf("error while creating enviroment: %w",
        err)
```

```
        destroyErr := DestroyEnviroment(labEnv.LabName)
        if destroyErr != nil {
            createErr = fmt.Errorf("%w, %v", createErr, destroyErr)
        }
        return nil, createErr
    }

    containerIds, err := createContainers(labEnv.Containers,
    labEnv.LabName)
    if err != nil {
        createErr := fmt.Errorf("error while creating enviroment: %w",
        err)
        destroyErr := DestroyEnviroment(labEnv.LabName)
        if destroyErr != nil {
            createErr = fmt.Errorf("%w, %v", createErr, destroyErr)
        }
        return nil, createErr
    }

    return containerIds, nil
```

En este apartado solo se muestran las funciones relativas a la creación de contenedores y no las de creación de redes, debido a que son muy similares: en primer lugar se crea un `errgroup` para sincronizar las llamadas a las funciones de creación de cada contenedor o red. En estas funciones se llama a la API de Docker Engine utilizando el paquete `client` y especificando las opciones necesarias, según lo expuesto en la sección 3.2.

Las únicas diferencias notables son que, en la función `createContainer`, antes de crear el contenedor se debe crear la *cookie* de X11 para poder ejecutar aplicaciones gráficas. Esta tarea se realiza en la función `createX11Cookie`, llamando a `xauth` a través del paquete `os/exec` con los comandos especificados en el apartado 3.2.4. Además, después de la creación del contenedor se llama a la función `connectToNetworks`, que concurrentemente conecta los contenedores a las redes, previamente creadas, que le correspondan.

```
// createContainers concurrently creates (using errgroups) all the
specified containers.
func createContainers(containers []LabContainer, labName string)
(map[string]string, error) {
    containerIds := make(map[string]string)

    g, ctx := errgroup.WithContext(context.Background())

    for _, container := range containers {
        container := container
        g.Go(func() error {
            containerId, err := createContainer(ctx, &container,
            labName)
```

```
            if err == nil {
                containerIds[container.Name] = containerId
                return nil
            } else {
                return fmt.Errorf("error while creating container %v:
                %w", container.Name, err)
            }
        })
    }

    if err := g.Wait(); err != nil {
        return nil, fmt.Errorf("error while creating containers: %w",
        err)
    }

    return containerIds, nil
}
```

```
// createContainer creates the container. This function is equivalent to
run:

// docker container run --name (labName_labContainer.Name) --hostname
(labName_labContainer.Name) -d -it --cap-add=NET_ADMIN --init --env
DISPLAY --env XAUTHORITY=cookiePath --mount
type=bind,source="$(pwd)",target=/mnt/shared --mount
type=bind,source=/tmp/.X11-unix,target=/tmp/.X11-unix --mount
type=bind,source=cookiePath,target=cookiePath --label
background=labContainer.Background labContainer.Image

// If the container is successfully created, it is disconnected from the
bridge network and connected to the specified networks.
func createContainer(errGroupCtx context.Context, labContainer
*LabContainer, labName string) (string, error) {
    if labContainer.Name == "" {
        return "", fmt.Errorf("container name cannot be empty")
    }
    containerFullName := labName + "_" + labContainer.Name

    if labContainer.Image == "" {
        return "", fmt.Errorf("container image cannot be empty")
    }
    imageName := labContainer.Image

    cookiePath, err := createX11Cookie(containerFullName, labName)
    if err != nil {
        return "", fmt.Errorf("%v", err)
```

```
    }

    ctx := context.Background()
    cli, err := client.NewClientWithOpts(client.FromEnv,
    client.WithAPIVersionNegotiation())
    if err != nil {
        return "", fmt.Errorf("%v", err)
    }

    displayEnvVar := "DISPLAY=" + os.Getenv("DISPLAY")
    xauthorityEnvVar := "XAUTHORITY=" + cookiePath
    homeDir, err := os.UserHomeDir()
    if err != nil {
        return "", fmt.Errorf("%v", err)
    }

    resp, err := cli.ContainerCreate(ctx, &container.Config{
        Hostname:  containerFullName,
        Tty:       true,
        OpenStdin: true,
        Env: []string{
            displayEnvVar,
            xauthorityEnvVar,
        },
        Labels: map[string]string{
            "background": strconv.FormatBool(labContainer.Background),
        },
        Image: imageName,
    }, &container.HostConfig{
        Init: boolPointer(true),
        CapAdd: []string{
            "NET_ADMIN",
        },
        Mounts: []mount.Mount{
            {
                Type:   mount.TypeBind,
                Source: homeDir,
                Target: "/mnt/shared",
            },
            {
                Type:   mount.TypeBind,
                Source: "/tmp/.X11-unix",
                Target: "/tmp/.X11-unix",
            },
            {
                Type:   mount.TypeBind,
                Source: cookiePath,
```

```
                Target: cookiePath,
            },
        },
    }, nil, nil, containerFullName)
    if err != nil {
        return "", fmt.Errorf("%v", err)
    }
    containerID := resp.ID

    if err := cli.ContainerStart(ctx, containerID,
    types.ContainerStartOptions{}); err != nil {
        return "", fmt.Errorf("%v", err)
    }

    if err := cli.NetworkDisconnect(ctx, "bridge", containerID, false);
    err != nil {
        return "", fmt.Errorf("%v", err)
    }

    if err := connectToNetworks(labContainer.Networks, containerID,
    labName, labContainer.Name); err != nil {
        return "", fmt.Errorf("%w", err)
    }

    return containerID, nil
}
```

Si el entorno se crea satisfactoriamente, `CreateEnviroment` devuelve un mapa con el nombre de cada contenedor y su ID. Para terminar, en la función `createLabEnviroment` del paquete `main` se llama a la función `GetEnviromentContainers` de LabsOnContainers, que devuelve la información de los contenedores creados, incluyendo la IP que se les ha asignado. Con esta información se llama a `printContainersInfo` y a `createTerminal-Windows`, que imprimen por la salida estándar la información de los contenedores creados y crean las terminales para la interacción con estos, respectivamente.

```
func createLabEnviroment(filePath string) {
    (...)

    containers, err :=
    labsoncontainers.GetEnviromentContainers(labEnv.LabName)
    if err != nil {
        fmt.Println(err)
        destroyErr := labsoncontainers.DestroyEnviroment(labEnv.LabName)
        if destroyErr != nil {
            fmt.Printf("error on labsoncontainers: %v\n", err)
        }
        os.Exit(1)
```

```
    }

    printContainersInfo(containers)

    err = createTerminalWindows(containers)
    if err != nil {
        fmt.Printf("error while creating terminal windows: %v\n", err)
        destroyErr := labsoncontainers.DestroyEnviroment(labEnv.LabName)
        if destroyErr != nil {
            fmt.Printf("error on labsoncontainers: %v\n", err)
        }
        os.Exit(1)
    }

    fmt.Println("Entorno creado exitosamente")
}
```

Existe una limitación durante el proceso de creación de las terminales, porque `xfce4-terminal` no permite su invocación desde programas con `setuid`, como en este caso. Para solventar esta limitación, se puede ejecutar el comando de creación de las terminales usando el UID real del usuario en lugar del efectivo. Pero, para que funcione correctamente, se debe añadir la siguiente línea al archivo `/etc/sudoers`, que permite que `usuario` pueda ejecutar `docker attach` pero no el resto de comandos de Docker. De esta forma, seguirá siendo imposible crear contenedores ni descargar imágenes de forma manual:

```
usuario ALL = (root) NOPASSWD: /usr/bin/docker *attach*
```

Destrucción del entorno

Para destruir el entorno creado con la opción anterior, se debe ejecutar el siguiente comando: `./LabsOnContainers -r <nombreEntorno>`. Esta opción elimina los contenedores, las redes y la carpeta donde se almacenan las *cookies* de X11 creadas, todo ello de forma concurrente al usar `errgroups`.

```
// destroyLabEnviroment destroys the specified lab enviroment using
LabsOnContainers API.
func destroyLabEnviroment(labName string) {
    fmt.Printf("Eliminando los contenedores y redes de %v...\n",
    labName)
    containersIds, err :=
    labsoncontainers.GetEnviromentContainers(labName)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
```

```
    if len(containersIds) > 0 {
        err := labsoncontainers.DestroyEnviroment(labName)
           if err != nil {
            fmt.Println(err)
            os.Exit(1)
           }
        fmt.Println("Contenedores y redes eliminados exitosamente")
    } else {
        fmt.Println("No existen contenedores asociados a", labName)
    }
}
```

```
// DestroyEnviroment removes all containers (including running
containers), networks and the X11 cookie directory of the provided lab
enviroment.
func DestroyEnviroment(labName string) error {
    err := destroyCookieDir(labName)
    if err != nil {
        return fmt.Errorf("error while destroying enviroment: %w", err)
    }

    containers, err := GetEnviromentContainers(labName)
    if err != nil {
        return fmt.Errorf("error while destroying enviroment: %w", err)
    }

    err = destroyContainers(containers)
    if err != nil {
        return fmt.Errorf("error while destroying enviroment: %w", err)
    }

    networksIds, err := GetEnviromentNetworks(labName)
    if err != nil {
        return fmt.Errorf("error while destroying enviroment: %w", err)
    }

    err = destroyNetworks(networksIds)
    if err != nil {
        return fmt.Errorf("error while destroying enviroment: %w", err)
    }

    return nil
}
```

Detención del entorno

Mediante `./LabsOnContainers -p <nombreEntorno>` se pueden parar todos los contenedores del entorno de prácticas de forma concurrente. Esto resulta de interés en el caso de que sea necesario detener los contenedores pero no se haya completado la práctica, para evitar así su destrucción y no tener que volver a repetir pasos que se hayan ejecutado anteriormente cuando se quiera terminar dicha práctica.

```
// stopLabEnviroment stops the containers of the specified lab
enviroment using LabsOnContainers API.
func stopLabEnviroment(labName string) {
    fmt.Printf("Deteniendo los contenedores y redes de %v...\n",
    labName)
    containersIds, err :=
    labsoncontainers.GetEnviromentContainers(labName)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }

    if len(containersIds) > 0 {
        err := labsoncontainers.StopEnviroment(labName)
           if err != nil {
            fmt.Println(err)
            os.Exit(1)
           }
        fmt.Println("Contenedores detenidos exitosamente")
    } else {
        fmt.Println("No existen contenedores asociados a", labName)
    }
}
```

```
// StopEnviroment stops all containers of the provided lab enviroment.
func StopEnviroment(labName string) error {
    containers, err := GetEnviromentContainers(labName)
    if err != nil {
        return fmt.Errorf("error while stopping enviroment: %w", err)
    }
    err = stopContainers(containers)
    if err != nil {
        return fmt.Errorf("error while stopping enviroment: %w", err)
    }
    return nil
}
```

Reactivación del entorno

Esta opción permite volver a arrancar los contenedores que se hayan detenido previamente para así poder continuar el trabajo. Con la ejecución de `./LabsOnContainers -l <nombreEntorno>` se lanzan los contenedores concurrentemente, se imprime por la salida estándar su información y se crean las terminales de los contenedores en primer plano.

```
// startLabEnviroment starts the containers of the specified lab
enviroment using LabsOnContainers API.
func startLabEnviroment(labName string) {
    fmt.Printf("Lanzando de nuevo los contenedores y redes de %v...\n",
    labName)
    containers, err := labsoncontainers.GetEnviromentContainers(labName)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }

    if len(containers) > 0 {
        err := labsoncontainers.StartEnviroment(labName)
           if err != nil {
            fmt.Println(err)
            os.Exit(1)
           }

        // It is necessary to call again to GetEnviromentContainers
        because we can only know the containers IPs after the restart
        containers, err =
        labsoncontainers.GetEnviromentContainers(labName)
        if err != nil {
            fmt.Println(err)
            os.Exit(1)
        }

        printContainersInfo(containers)

        err = createTerminalWindows(containers)
        if err != nil {
            fmt.Printf("error while creating terminal windows: %v\n",
            err)
            stopErr := labsoncontainers.StopEnviroment(labName)
            if stopErr != nil {
                fmt.Printf("error on labsoncontainers: %v\n", err)
            }
            os.Exit(1)
        }
```

```
        fmt.Println("Contenedores lanzados exitosamente")
    } else {
        fmt.Println("No existen contenedores asociados a", labName)
    }
}
```

```
// StartEnviroment starts all the containers of the provided lab
enviroment.
func StartEnviroment(labName string) error {
    containers, err := GetEnviromentContainers(labName)
    if err != nil {
        return fmt.Errorf("error while starting enviroment: %w", err)
    }

    err = startContainers(containers)
    if err != nil {
        return fmt.Errorf("error while starting enviroment: %w", err)
    }

    return nil
}
```

Inspeccionar entorno

Si se ejecuta `./LabsOnContainers -i <nombreEntorno>` es posible obtener información de bajo nivel relativa a los contenedores del entorno, como por ejemplo los volúmenes montados, las redes a las que están conectados o sus variables de entorno.

```
// inspectLabEnviroment prints to the standard output the result of
running inspect on the lab enviroment containers, using LabsOnContainers
API.
func inspectLabEnviroment(labName string) {
    fmt.Printf("Información de los contenedores de %v:\n", labName)
    containersIds, err :=
    labsoncontainers.GetEnviromentContainers(labName)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }

    if len(containersIds) > 0 {
        inspectMap, err := labsoncontainers.InspectEnviroment(labName)
           if err != nil {
            fmt.Println(err)
            os.Exit(1)
           }
```

```
        for container, inspect := range inspectMap {
            fmt.Println()
            fmt.Printf("%v:\n", container)
            fmt.Println(string(inspect))
        }
    } else {
        fmt.Println("No existen contenedores asociados a", labName)
    }
}
```

```
// InspectEnviroment returns low-level information of all the containers
of the provided lab enviroment. On success, it returns a map of the
containers' names as keys and their information as values.
func InspectEnviroment(labName string) (map[string][]byte, error) {
    containers, err := GetEnviromentContainers(labName)
    if err != nil {
        return nil, fmt.Errorf("error while inspecting enviroment: %w",
        err)
    }

    inspectMap, err := inspectContainers(containers)
    if err != nil {
        return nil, fmt.Errorf("error while inspecting enviroment: %w",
        err)
    }

    inspectJSONMap := make(map[string][]byte, len(inspectMap))

    for container, inspectInfo := range inspectMap {
        inspectJSON, err := json.MarshalIndent(inspectInfo, "", "    ")
        if err != nil {
            return nil, fmt.Errorf("error while inspecting enviroment:
            %w", err)
        }
        inspectJSONMap[container] = inspectJSON
    }

    return inspectJSONMap, nil
}
```

Capítulo 5

Resultados

5.1. Entorno de pruebas

El equipo que se ha utilizado para realizar las pruebas de despliegue de los contenedores ha sido un portátil Lenovo IdeaPad 5 Pro, que cuenta con un procesador AMD Ryzen 7 5800U de ocho núcleos, 16 GB de memoria RAM y un disco SSD de 512 GB. Sobre esta máquina se ha realizado una instalación mínima del sistema operativo Ubuntu 22.04, con XFCE como entorno de escritorio.

Hay que tener en cuenta que este sistema difiere de los disponibles en la Facultad de Informática de la Universidad Complutense de Madrid, especialmente al usar un disco de estado sólido en lugar de un disco mecánico, aspecto que, tal y como se había señalado en el apartado 3.1, puede afectar notablemente al rendimiento de las máquinas virtuales. No obstante, como ambos despliegues —prácticas con contenedores y con máquinas virtuales— se van a evaluar sobre el mismo entorno, se pueden estudiar las diferencias entre ambas implementaciones pudiendo suponer que se darán de idéntica manera en cualquier otro sistema.

Por otra parte, hubiera sido más conveniente usar Alpine Linux como sistema operativo base para poder realizar una comparación en un entorno más parecido a la máquina virtual descrita en el capítulo anterior. Sin embargo, VirtualBox no es compatible con Alpine, por lo que, en el caso de haber utilizado este sistema operativo, no se podrían haber realizado las pruebas comparativas entre el uso de contenedores y máquinas virtuales. Por ello, se ha optado por Ubuntu con XFCE para intentar obtener una semejanza mayor que si se hubiera usado GNOME, el entorno de escritorio por defecto de Ubuntu.

5.2. Prueba de concepto

La prueba de concepto realizada consiste en llevar a cabo la práctica 0 de la asignatura Redes y Seguridad II. Esta práctica es la primera que se realiza todos los años en esta asignatura, y la que más máquinas virtuales requiere. En ella se pide a los estudiantes efectuar configuraciones básicas de red para conectar todas las instancias entre sí, monitorizar el estado de la red con Wireshark y desplegar un servidor web con el módulo

SimpleHTTPServer de Python [123], al que posteriormente se accederá mediante Firefox desde otra máquina virtual. Todas las instancias usan la máquina virtual de Debian. En la figura 5.1 se muestra la topología de red que se debe construir en esta práctica:

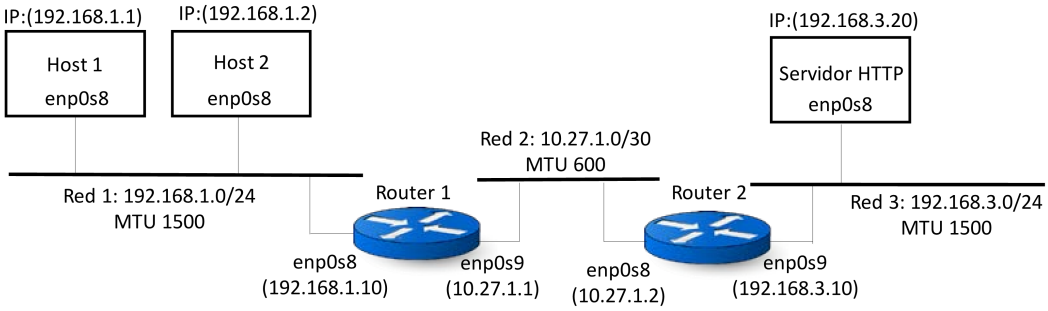

Figura 5.1: Topología de red de la práctica 0 de Redes y Seguridad II

Para realizar esta práctica con contenedores, se utilizará la imagen `rys`, descrita en la sección 3.2.2. Los contenedores se crearán mediante el siguiente archivo YAML, que sigue el formato detallado en el apartado 3.4:

```
nombre_practica: practica0
contenedores:
  - nombre: host1
    imagen: rys
    redes:
    - nombre: 1
  - nombre: host2
    imagen: rys
    redes:
    - nombre: 1
  - nombre: router1
    imagen: rys
    redes:
    - nombre: 1
    - nombre: 2
  - nombre: router2
    imagen: rys
    redes:
    - nombre: 2
    - nombre: 3
  - nombre: servidor
    imagen: rys
    redes:
    - nombre: 3
```

En la figura 5.2 podemos ver el resultado de ejecutar la orden `./LabsOnContainers -c practica0.yaml`, encargada de leer el fichero YAML y crear los contenedores y redes

en consonancia con lo descrito en dicho archivo. En la ventana donde se ha ejecutado el comando se muestra información de cada contenedor, como su identificador, el nombre de cada contenedor, su imagen, las redes a las que está conectado y la dirección IP que se le ha asignado en cada una de ellas. Para interactuar con ellos, se muestra una ventana de terminal con cinco pestañas, una para cada contenedor:

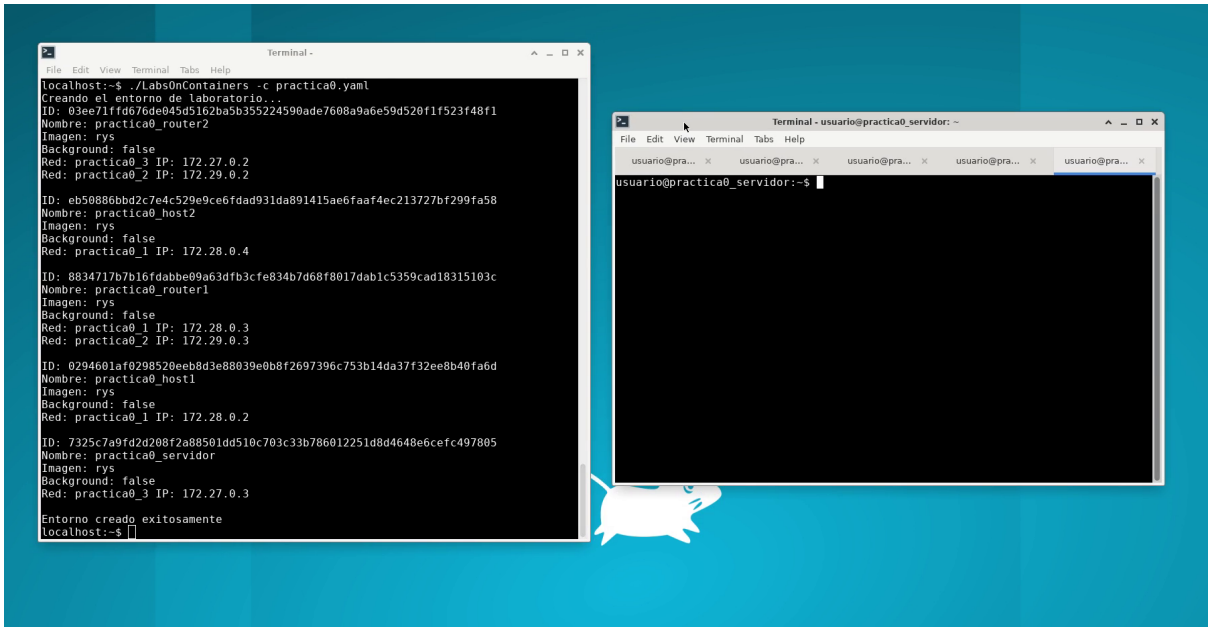

Figura 5.2: Estado tras haber ejecutado `./LabsOnContainers -c practica0.yaml`

Antes de configurar la red en cada contenedor se deben instalar todos los programas que vayan a hacer falta durante la práctica y que no estén incluidos en la imagen, en este caso Python, en el contenedor del servidor. Esto se debe a que, si se modifica la ruta por defecto que deben seguir los paquetes, se perderá la conexión a Internet. Este comportamiento resulta similar a lo que sucede actualmente con las máquinas virtuales, ya que normalmente se suele deshabilitar el adaptador NAT después de descargar los archivos necesarios para evitar interferencias durante el desarrollo de la práctica. No obstante, si se desea mantener la conectividad a Internet, se pueden configurar las tablas de enrutamiento para conectar los contenedores entre sí pero evitando modificar la ruta por defecto. En la figura 5.3 podemos ver que la instalación de Python se realiza de forma correcta dentro del contenedor:

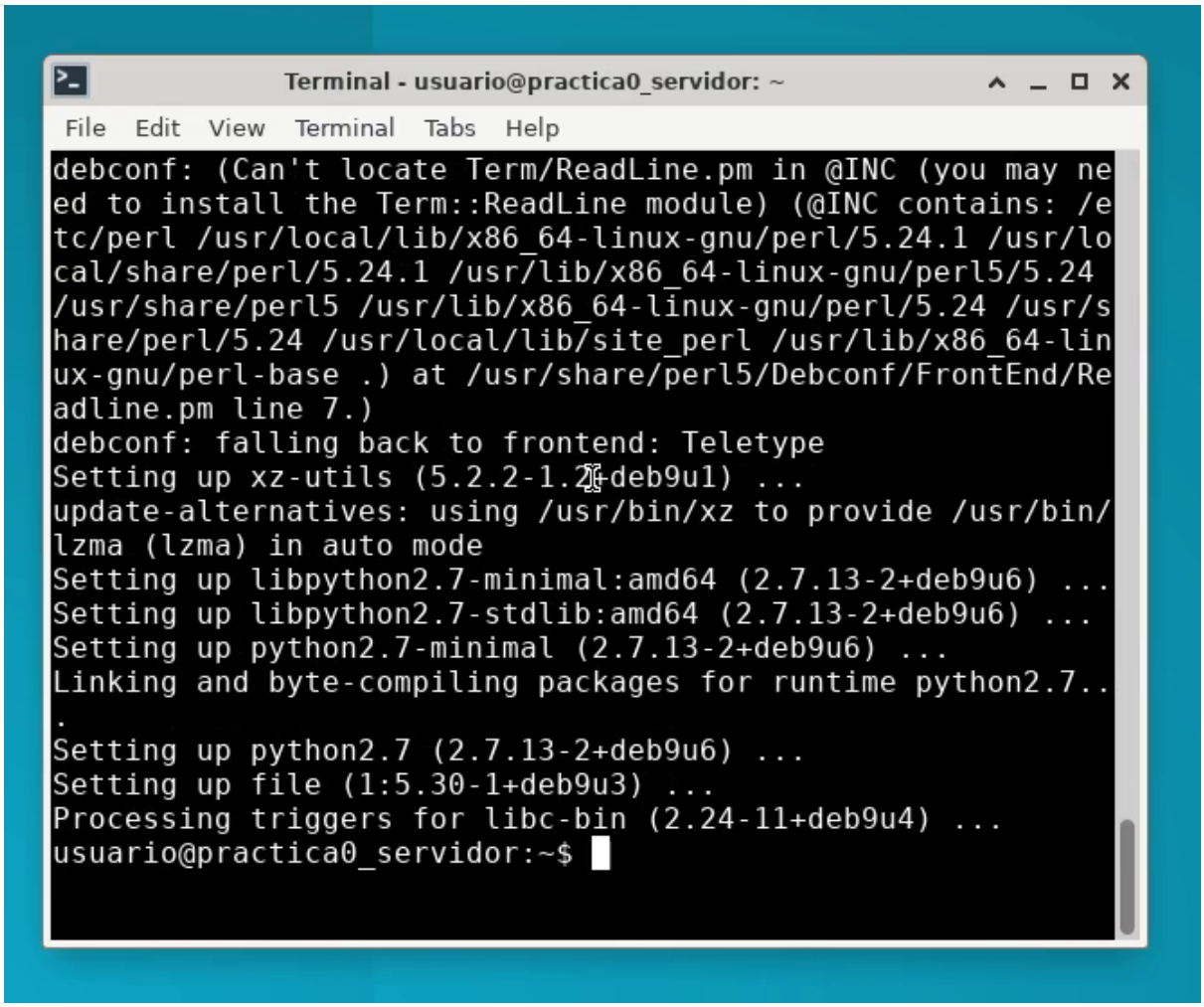

Figura 5.3: Instalación de Python en el contenedor del servidor

Tras instalar los programas necesarios, se puede configurar la red en cada contenedor utilizando los comandos de la *suite* `ip`. Podemos ver en la figura 5.4 que cada contenedor dispone de sus correspondientes direcciones IP, especificadas en la figura 5.1, y que la tabla de rutas está configurada de tal forma que puedan comunicarse unos con otros:

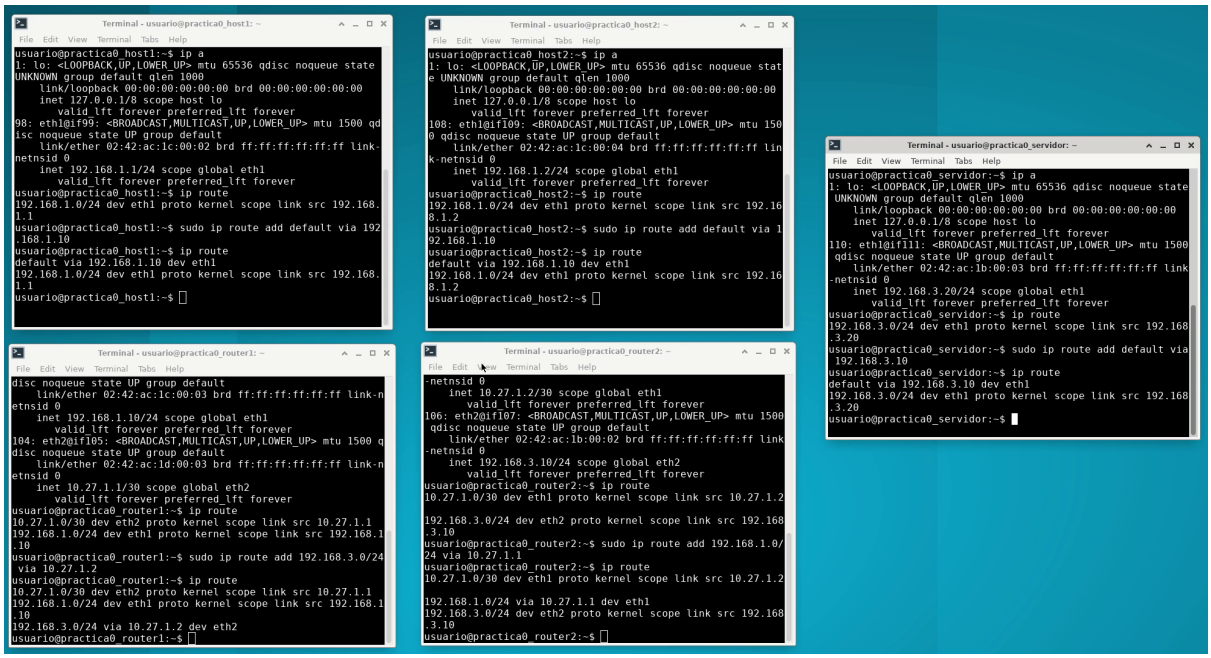

Figura 5.4: Configuración de red de los contenedores

A continuación, en la práctica se pide analizar mediante Wireshark qué ocurre cuando se manda un paquete ICMP con `ping` desde Host 2, primero a un contenedor de la misma red —Host 1— y luego a un contenedor de otra red —Servidor—, comprobando con `ip neigh` el estado de las tablas ARP antes y después de ejecutar los comandos. En las figuras 5.5 y 5.6 se muestra el resultado de ejecutar estas órdenes y el correcto funcionamiento en ambos casos:

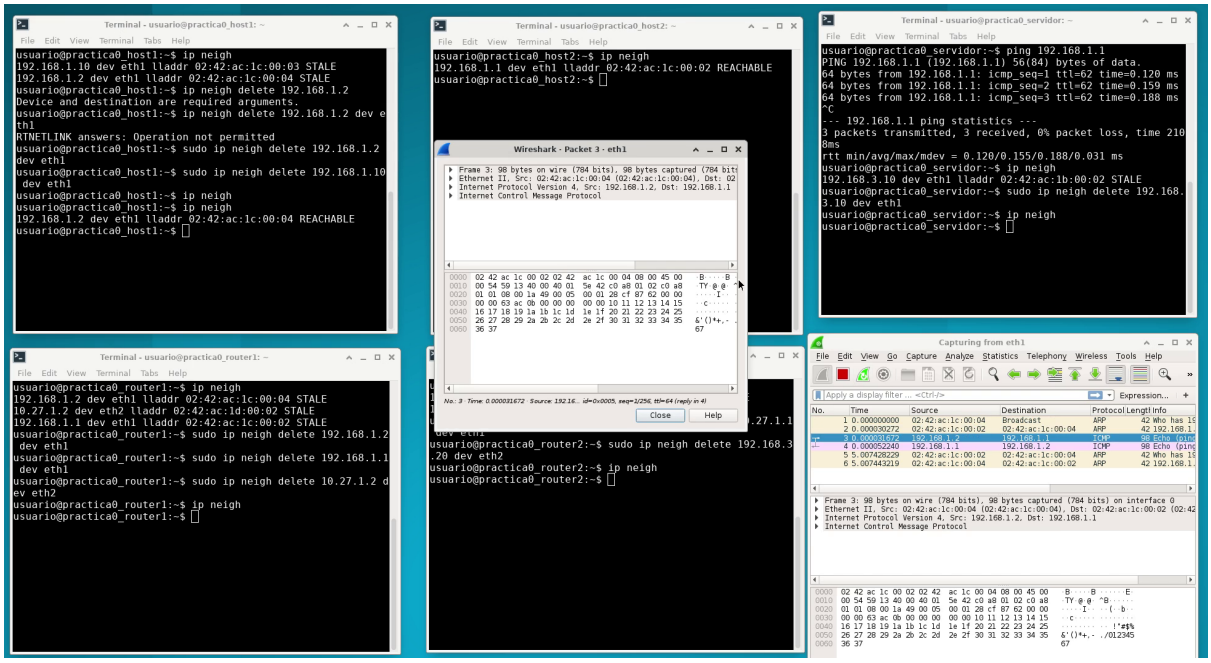

Figura 5.5: Resultado tras mandar un paquete ICMP en la misma red

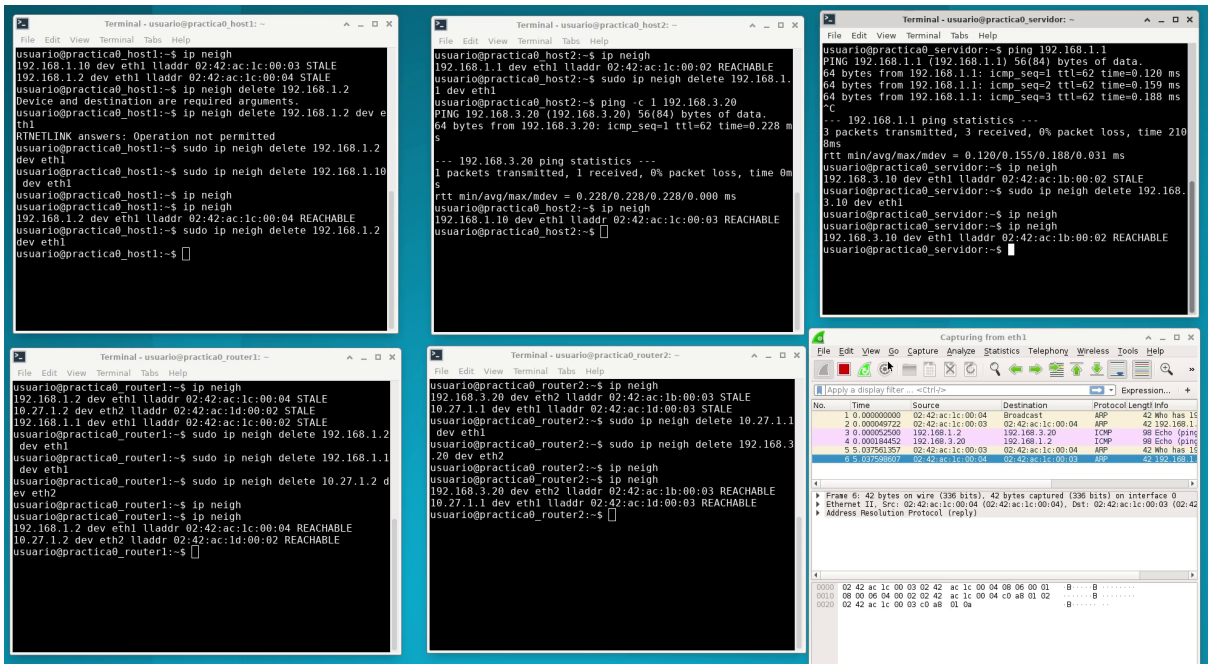

Figura 5.6: Resultado tras mandar un paquete ICMP a otra red

Finalmente, el último ejercicio de la práctica consiste en crear un servidor web con Python en la máquina Servidor y acceder a este desde Host 2 mediante Firefox. Además, se debe analizar el tráfico de la red con Wireshark en Host 2, Router 2 y Servidor. Podemos ver en la figura 5.7 que este ejercicio se puede llevar a cabo sin ningún tipo de inconveniente:

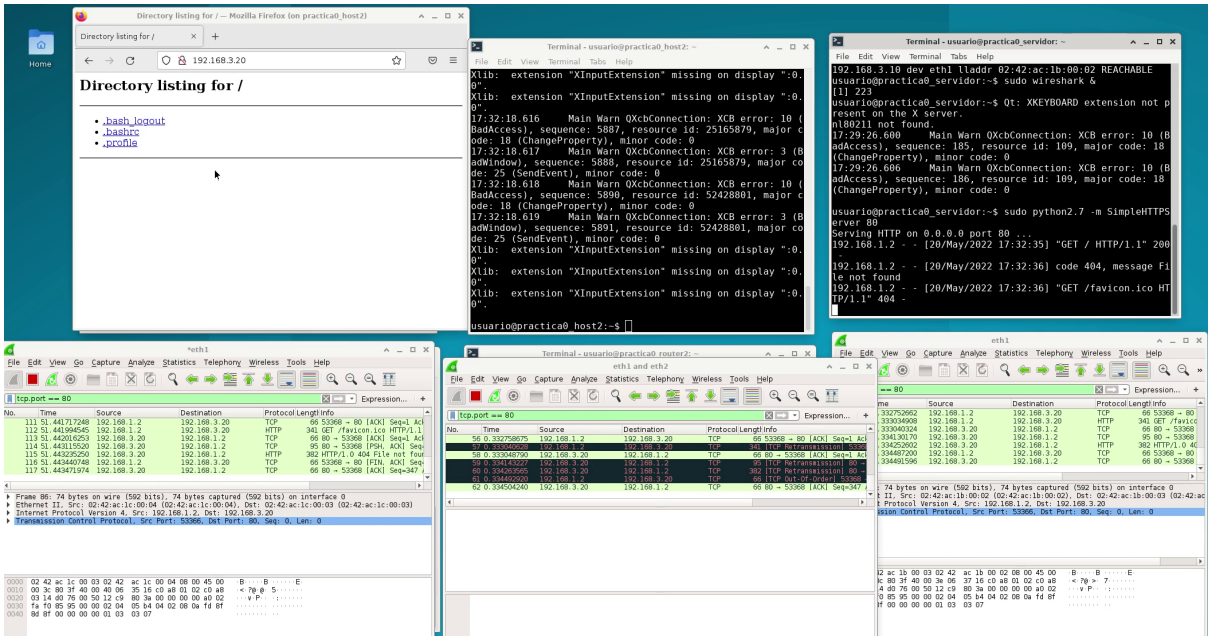

Figura 5.7: Resultado tras acceder al servidor web con Firefox desde Host 2

Una vez se ha realizado la práctica, se pueden destruir todos los contenedores y redes asociadas ejecutando el comando `./LabsOnContainers -r practica0`. Las ventanas de las terminales se quedan abiertas por si se desea reutilizarlas para interactuar con el sistema anfitrión. Se puede ver el resultado de ejecutar este comando en la figura 5.8:

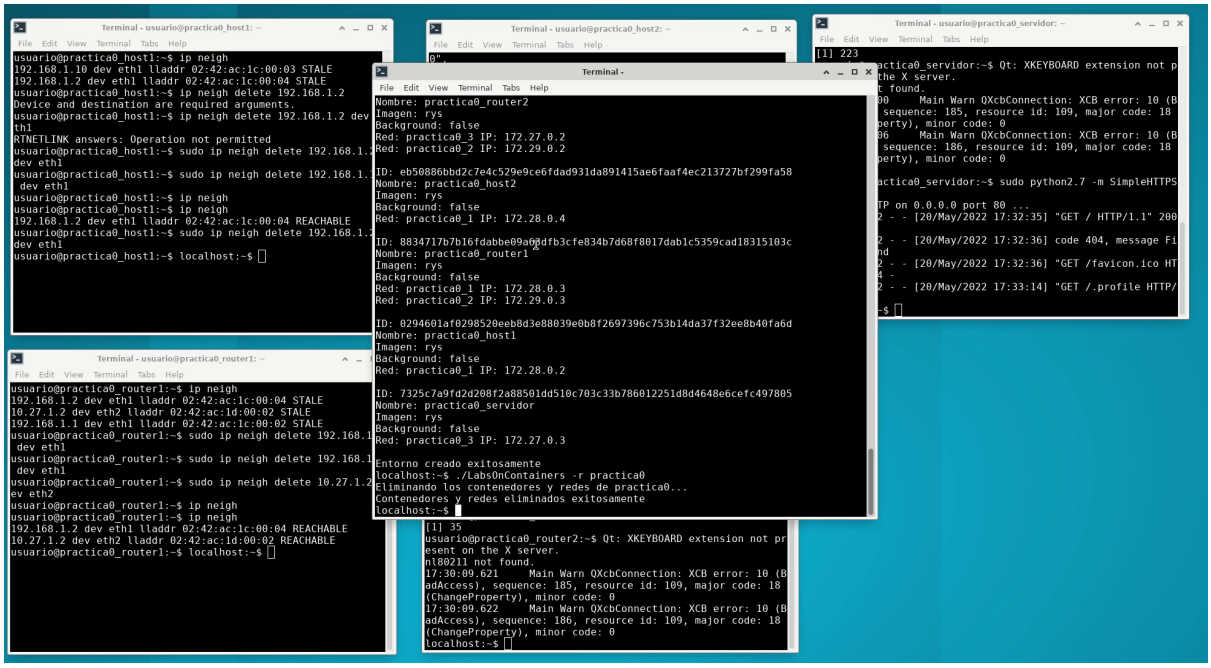

Figura 5.8: Estado tras haber ejecutado `./LabsOnContainers -r practica0`

Viendo estos resultados, podemos concluir que es totalmente viable realizar una práctica de una asignatura como Redes y Seguridad II utilizando contenedores con la configuración descrita en este trabajo, obteniendo una experiencia de uso satisfactoria y sin que se produzca ningún tipo de problema.

5.3. Evaluación del rendimiento

Como se ha visto en el anterior apartado, el objetivo de este trabajo, realizar una práctica de una asignatura de sistemas y redes con contenedores en lugar de con máquinas virtuales, puede ser cumplido. Pero, más allá de la experiencia de uso, resulta necesario evaluar su rendimiento para ver si, efectivamente, el uso de contenedores supone una ventaja o si, por el contrario, no existen diferencias notables. Por ello, en esta sección se comparan los contenedores y las máquinas virtuales en tres ámbitos: el espacio en disco que ocupan, el tiempo que tardan en crearse y el uso de CPU, memoria y entrada/salida durante una práctica.

Hay que recordar que la idea de este proyecto es ejecutar los contenedores dentro de una máquina virtual con el fin de mejorar su aislamiento. Este hecho puede suponer una reducción en su rendimiento y que las diferencias apreciadas no sean tan significativas debido a la sobrecarga producida por la máquina virtual de Alpine. Por este motivo, además de ejecutar los contenedores sobre la máquina virtual, también se van a realizar las pruebas de rendimiento ejecutando los contenedores de forma nativa sobre Ubuntu, para así disponer de una comparativa más completa.

Cabe destacar que, para llevar a cabo estas pruebas, se podrían haber realizado ajustes de rendimiento para obtener unos resultados más consistentes, como por ejemplo desconectar la red, modificar el algoritmo de escalado de frecuencia o cambiar la prioridad de los procesos [124]. Sin embargo, se ha decidido no emplear ninguna de estas técnicas para que los resultados sean más parecidos a los que se obtendrían en un entorno real. Por otra parte, como el sistema donde se han efectuado las pruebas es un ordenador portátil, todas las pruebas se han realizado con el ordenador conectado a la corriente eléctrica para evitar variaciones producidas por el estado de la batería.

5.3.1. Espacio de disco utilizado

En la tabla 5.1 podemos ver una comparativa del espacio en disco utilizado por cada una de las alternativas al realizar la práctica 0 de Redes y Seguridad II. Este espacio ha sido medido tras crear todas las instancias en lugar de al final de su realización para evitar fluctuaciones derivadas de las acciones realizadas dentro de cada contenedor o de cada máquina virtual. El espacio ocupado por los contenedores ha sido medido con el comando `docker system df`, y el de las máquinas virtuales, comprobando el tamaño de las carpetas de cada máquina virtual mediante `du -h`.

Contenedores (nativo)	Contenedores (sobre Alpine)	Máquinas virtuales
818 MB	2,6 GB	7 GB

Tabla 5.1: Comparativa del espacio utilizado (los colores indican cómo de favorable resulta cada alternativa, siendo de menos a más favorable rojo, amarillo y verde en este orden)

La alternativa que menos espacio ocupa es ejecutar los contenedores de forma nativa. Gracias a no tener que cargar un sistema operativo completo y a que solo se disponen de los programas imprescindibles para realizar las prácticas, el peso de la imagen `rys` resulta muy reducido. Además, gracias al sistema de archivos por capas, los contenedores no emplean espacio adicional a menos que realicen escrituras sobre el propio contenedor, por lo que nada más crear el entorno de prácticas el espacio utilizado por cada uno de ellos es prácticamente nulo.

Utilizar contenedores sobre la máquina virtual de Alpine supone ocupar un total de 2,6 GB de espacio. Este espacio se puede separar entre el peso de la imagen de los contenedores —que es igual al del caso anterior, 818 MB— y el peso de la propia máquina de Alpine, que supondría unos 1,8 GB aproximadamente.

Por último, podemos observar que cualquiera de estas dos alternativas es mucho más beneficiosa que seguir empleando máquinas virtuales, al menos en lo que a espacio de disco se refiere. Las cinco máquinas virtuales ocupan un total de 7 GB, de los cuales 6,7 GB se corresponden a la máquina virtual base de Debian y los 300 MB restantes a las máquinas virtuales clonadas, ya que, a pesar de usar la opción de clonación enlazada de VirtualBox que permite reutilizar el disco duro de la máquinas virtuales base, cada una de las instacias clonadas ocupa unos 60 MB adicionales.

Así, podemos concluir que el uso de contenedores puede suponer una ganancia de entre 4,4 y 6,2 GB con respecto al uso de máquinas virtuales.

5.3.2. Tiempo de creación

Al igual que en la sección anterior, se ha tomado el caso de la práctica 0 de Redes y Seguridad II para ver cuánto tiempo ocupa la creación de los contenedores y de las máquinas virtuales. En el caso de los contenedores, se ha medido mediante la herramienta `time` el tiempo que tarda en ejecutarse `./LabsOnContainers -c practica0.yaml`. Para las máquinas virtuales, aunque en la actualidad el proceso de clonado y encendido se realiza de forma interactiva, se ha desarrollado el siguiente *script* para que la comparativa se realice en igualdad de condiciones:

```
#!/bin/sh

set -e

VBoxManage snapshot rys2122 take Snapshot

VBoxManage clonevm rys2122 --snapshot Snapshot --options link --name
"Host 1" --register
VBoxManage clonevm rys2122 --snapshot Snapshot --options link --name
"Host 2" --register
VBoxManage clonevm rys2122 --snapshot Snapshot --options link --name
"Router 1" --register
VBoxManage clonevm rys2122 --snapshot Snapshot --options link --name
"Router 2" --register
VBoxManage clonevm rys2122 --snapshot Snapshot --options link --name
"Servidor HTTP" --register

VBoxManage modifyvm "Host 1" --nic1 nat --cableconnected1 on --nic2
intnet --intnet2 "Red 1" --cableconnected2 on
VBoxManage modifyvm "Host 2" --nic1 nat --cableconnected1 on --nic2
intnet --intnet2 "Red 1" --cableconnected2 on

VBoxManage modifyvm "Router 1" --nic1 nat --cableconnected1 on --nic2
intnet --intnet2 "Red 1" --cableconnected2 on --nic3 intnet --intnet3
"Red 2" --cableconnected3 on
VBoxManage modifyvm "Router 2" --nic1 nat --cableconnected1 on --nic2
intnet --intnet2 "Red 2" --cableconnected2 on --nic3 intnet --intnet3
"Red 3" --cableconnected3 on
VBoxManage modifyvm "Servidor HTTP" --nic1 nat --cableconnected1 on
--nic2 intnet --intnet2 "Red 3" --cableconnected2 on

VBoxManage startvm "Host 1"
VBoxManage startvm "Host 2"
VBoxManage startvm "Router 1"
VBoxManage startvm "Router 2"
VBoxManage startvm "Servidor HTTP"
```

No obstante, en el caso de las máquinas virtuales no solo se debe medir el tiempo que tardan en ejecutarse estos comandos, sino que también se debe tener en cuenta el tiempo que tarda en arrancar el sistema operativo en sí. Para ello, se ha utilizado la herramienta `systemd-analyze`, que mide el tiempo desde que el *kernel* empieza a cargarse hasta que todos los servicios de arranque han terminado de iniciarse. Por tanto, el tiempo de encendido de las máquinas virtuales se expresará como la suma entre el tiempo que tarda en ejecutarse el script de VirtualBox, medido con `time`, y la mediana de los tiempos obtenidos con `systemd-analyze`, para evitar usar valores extremos.

Por otra parte, también se ha comparado el tiempo que tardan en ejecutarse sobre la máquina virtual de Alpine las versiones de Go y de Ash de la herramienta de despliegue de los contenedores, con el fin de saber si el hecho de utilizar concurrencia y un lenguaje compilado ha supuesto una mejora en los tiempos de creación.

En todos los casos se han realizado diez medidas, descartando la primera ejecución para evitar diferencias por motivos como la preparación de la caché del disco. En la tabla 5.2 se muestran los resultados, expresados como la media de los tiempos obtenidos y la desviación estándar calculada.

Contenedores (sobre Alpine – Go)	**Contenedores (nativo – Go)**	**Contenedores (sobre Alpine – Ash)**	**Máquinas virtuales**
1,403 ± 0,08 s	2,367 ± 0,082 s	2,459 ± 0,032 s	37,608 ± 0,086 s

Tabla 5.2: Comparativa de tiempos de creación del entorno (los colores indican cómo de favorable resulta cada alternativa, siendo de menos a más favorable rojo, amarillo y verde en este orden)

Contrariamente a lo que se podría intuir, los mejores resultados se obtienen al crear los contenedores sobre la máquina virtual de Alpine, con hasta casi un segundo de diferencia a si se creasen sobre Ubuntu. Esta diferencia podría deberse a que, a pesar de haber realizado una instalación mínima, en Ubuntu existen más servicios en ejecución que en Alpine, por lo que es posible que se haya destinado más tiempo de ejecución a otros procesos existentes en el sistema.

Por otra parte, podemos confirmar que la versión en Go es más rápida que la versión de Ash, obteniendo una mejora de un 75,27 % en los tiempos de creación con las mismas condiciones de entorno. Además, los resultados obtenidos con `time` nos permiten confirmar a qué se debe esta diferencia, ya que, si calculamos el tiempo de procesador —suma del tiempo de procesador en modo usuario, `user`, y en modo sistema, `sys`— en ambos casos, la versión de Ash ocupa de media 0,94 segundos, mientras que en el caso de Go el tiempo de procesador es de tan solo 0,01 segundos. Por tanto, podemos asegurar que ha habido una optimización notable de los tiempos de creación gracias al uso de Go y de la concurrencia, encontrándose ahora la mayor limitación en los tiempos de espera por entrada/salida, debido a la comunicación con Docker.

Finalmente, podemos apreciar que las máquinas virtuales ofrecen unos tiempos hasta 37 veces peores a los de los contenedores. Estos tiempos se pueden desgranar entre lo que se tarda en ejecutar el *script* de clonado y creación de las máquinas virtuales —3,9 segundos de media, superior incluso al tiempo de la versión no concurrente en Ash— y el tiempo

de encendido —33,71 segundos de media—. Se puede achacar esta enorme diferencia al hecho de que, con los contenedores, no se debe cargar desde cero el núcleo del sistema operativo ni tantos servicios, en comparación con las máquinas virtuales.

5.3.3. Uso de CPU, memoria y entrada/salida

Para comprobar las diferencias en cuanto al uso de procesador, memoria RAM y entrada/salida, se ha llevado a cabo la siguiente prueba, inspirada en lo que podría ser un caso real: recreando la topología de la práctica 0 de Redes y Seguridad II, se ha desplegado el servidor web de Python sobre la máquina Servidor, y desde Host 1 se han estado realizando peticiones web a este servidor cada segundo utilizando `watch -n 1 wget 192.168.3.20`. Además, en todas las instancias se ha estado monitorizando el estado de la red mediante Wireshark.

El rendimiento se ha monitorizado durante un minuto usando la herramienta `sar` [125], ejecutando el siguiente comando: `sar -urbh 1 60`, que muestra estadísticas del uso de CPU, memoria y E/S 60 veces con un intervalo de un segundo. Una vez se completa la ejecución, se muestra una media de todos los valores durante el tiempo especificado.

En todos los casos se han realizado tres mediciones desde el sistema anfitrión, escogiendo la mediana para descartar valores extremos. En las tabla 5.3 se muestra el porcentaje de CPU usado, de memoria y las transferencias de entrada/salida por segundo de cada alternativa:

	Contenedores (nativo)	**Contenedores (sobre Alpine)**	**Máquinas virtuales**
Uso de CPU	0,7 %	1 %	1,3 %
Uso de memoria (valor base: 6,8 %)	10,6 %	16,3 %	45 %
Transferencias por segundo	4,5	2,55	7,63

Tabla 5.3: Comparativa del uso de CPU, memoria y entrada/salida (los colores indican cómo de favorable resulta cada alternativa, siendo de menos a más favorable rojo, amarillo y verde en este orden)

Como se puede observar en la tabla, los contenedores ejecutados nativamente obtienen los mejores resultados en cuanto a uso de CPU, aunque no existen enormes diferencias entre las tres opciones. Algo similar ocurre con la entrada/salida, ya que, si bien se podría haber esperado que tanto la máquina virtual de Alpine como las de Debian hubieran obtenido peores resultados, tampoco ha habido diferencias notables, e incluso la máquina virtual de Alpine ha sido la alternativa menos demandante en cuanto a transferencias de entrada/salida. Como la prueba realizada no era intensiva ni en uso de CPU ni en uso de E/S, y que es posible que se hayan podido producir interferencias al realizar las mediciones desde el sistema anfitrión, no podemos concluir que ninguna de las opciones ofrezca el mejor rendimiento en ninguno de estos dos ámbitos.

Donde sí podemos advertir una mayor desigualdad es en el uso de memoria RAM. En todos los casos se partía de que el porcentaje de memoria RAM usada por Ubuntu antes

de crear los entornos virtuales era del 6,8 %. Los contenedores ejecutados nativamente suponen la mejor alternativa debido a que, al contrario de lo que sucede con las máquinas virtuales, no requieren que se les asigne memoria estáticamente desde su creación, sino que pueden demandarla de manera dinámica. Los contenedores ejecutados sobre la máquina virtual de Alpine también obtienen un buen rendimiento, con un 5,7 % de diferencia entre esta opción y la ejecución nativa, diferencia que podemos atribuir al uso de la máquina virtual. Por último, las máquinas virtuales vuelven a ser la peor opción de las tres, disparándose su porcentaje de uso de memoria hasta el 45 %.

Se debe tener en cuenta que en nuestro caso contábamos con una máquina con 16 GB de RAM, por lo que puede parecer que este porcentaje de uso no es elevado. Sin embargo, si la cantidad de RAM disponible fuera de 8 GB —configuración habitual en muchos ordenadores utilizados por estudiantes—, el uso de cinco máquinas virtuales podría suponer que el porcentaje de memoria RAM utilizada fuese de hasta el 90 %, condición que podría generar un importante cuello de botella y una experiencia de uso muy pobre. Por el contrario, con los contenedores —tanto de forma nativa como sobre una máquina virtual— este porcentaje se mantendría entre el 21,2 % y el 32,6 %, dependiendo de la opción escogida. En conclusión, se puede afirmar que el uso de contenedores supone una importante mejora con respecto a las máquinas virtuales en cuanto a uso de memoria RAM.

5.3.4. Discusión

Tal y como se ha visto en los anteriores tres apartados, se puede asegurar que el uso de contenedores es una mejor opción en comparación con el despliegue actual con máquinas virtuales, al menos en cuanto a uso de espacio en disco, memoria RAM y tiempos de creación.

Por otra parte, también podemos observar que ejecutar los contenedores sobre una máquina virtual tampoco ha supuesto una sobrecarga elevada. Uno de los motivos se debe a la elección de Alpine Linux como sistema operativo base de la máquina virtual, ya que, gracias a su sencillez, se ha conseguido un espacio de disco bastante reducido y unos tiempos de creación de los contenedores inferiores incluso a la ejecución nativa. El único aspecto mejorable podría ser el consumo de memoria, que es superior al de los contenedores sin máquina virtual debido a la asignación estática de RAM, pero consideramos que se mantiene dentro de unos márgenes razonables y que no supone un grave impacto en el rendimiento global del sistema.

5.4. Análisis de seguridad

Aparte de evaluar las diferencias de rendimiento entre los contenedores y las máquinas virtuales, también es necesario realizar un análisis de seguridad de la implementación para asegurar que optar por este despliegue no supone un riesgo para la integridad del sistema anfitrión.

En primer lugar, la mayor garantía de seguridad que ofrece el despliegue diseñado en este proyecto es que los contenedores se ejecutan sobre una máquina virtual. Este hecho supone que, incluso si un usuario malicioso o un proceso lograse explotar alguna vulnerabilidad para superar el aislamiento proporcionado por los contenedores, todavía se encontraría

en un entorno separado de la máquina anfitriona, por lo que los riesgos de que el sistema se vea dañado se reducen notablemente.

No obstante, también podemos enumerar otras medidas que se han tomado para aumentar aún más la seguridad. Por ejemplo, el hecho de que el usuario de la máquina virtual no disponga de permisos de superusuario ni pertenezca al grupo `docker` imposibilita que se puedan crear contenedores con opciones peligrosas, cuestión que ya se había mencionado en la sección 2.3.5. Además, hay que recordar que la interacción de las aplicaciones gráficas de los contenedores con el servidor X del anfitrión se realiza mediante *untrusted cookies*, lo que evita que un programa malicioso pueda llegar a interferir con las ventanas del sistema anfitrión.

La seguridad del entorno y de los contenedores ha sido analizada con Docker Bench for Security [126], una herramienta que «comprueba si se cumplen una serie de buenas prácticas a la hora de desplegar contenedores en producción». El informe completo puede consultarse en el apéndice C.

Algunas de estas prácticas no pueden aplicarse a este proyecto: por ejemplo, esta herramienta considera como falta añadir *capabilities* adicionales, directriz que no se puede satisfacer debido a que resulta imprescindible que los contenedores dispongan de `NET_ADMIN` para realizar las prácticas. Por otra parte, como esta herramienta está enfocada a entornos de producción, existen directivas cuya implementación no tiene sentido en este ámbito, como por ejemplo establecer el reinicio automático de los contenedores en caso de fallo.

En cuanto a las recomendaciones que sí se pueden aplicar a este proyecto, podemos ver que muchas de ellas se han llevado a cabo, como evitar compartir los *namespaces* del anfitrión con los contenedores, no montar directorios sensibles del anfitrión o no exponer puertos privilegiados. Sin embargo, existen otras buenas prácticas que no se han implementado, como configurar la auditoría de los ficheros de Docker, emplear los *user namespaces* o aplicar perfiles personalizados de AppArmor y SELinux. El haberlo hecho habría supuesto un gran esfuerzo y una gran cantidad de tiempo, sobre todo teniendo en cuenta que en este proyecto no se pretendía implementar un sistema en producción sino realizar una primera aproximación para conocer si era viable el uso de contenedores para realizar prácticas académicas. Además, hay que tener en cuenta que los contenedores se están ejecutando sobre una máquina virtual, por lo que no resulta tan prioritario establecer unas medidas de seguridad tan estrictas. No obstante, somos conscientes de la importancia de estas recomendaciones, y podrían ser de utilidad si en el futuro se decidiese implementar este proyecto dentro de los sistemas de producción de los laboratorios.

El último punto que se debe analizar es la seguridad de las imágenes empleadas por los contenedores. Tal y como se ha expuesto en el apartado 3.5, se ha decidido que los estudiantes no puedan descargarse ni construir nuevas imágenes, evitando así que puedan utilizar imágenes que incluyan opciones peligrosas. En cuanto a la imagen `rys`, se ha realizado un análisis de vulnerabilidades con `docker scan`, que indica que existen vulnerabilidades tanto en los programas instalados como en la propia imagen de Debian. Sin embargo, solo siete de estas vulnerabilidades son de alto impacto. La recomendación en este caso sería actualizar Debian a la última versión, suponiendo que no se necesiten las características específicas de Debian 9 para la realización de las prácticas. Por otra parte, las vulnerabilidades del resto de aplicaciones resultan más difíciles de solucionar

porque se depende de las versiones disponibles en los repositorios de paquetes de Debian. No se ha incluido el análisis completo en esta memoria debido a que el número de hojas que ocupaba era excesivo, pero en la figura 5.9 se puede ver una captura de los resultados obtenidos:

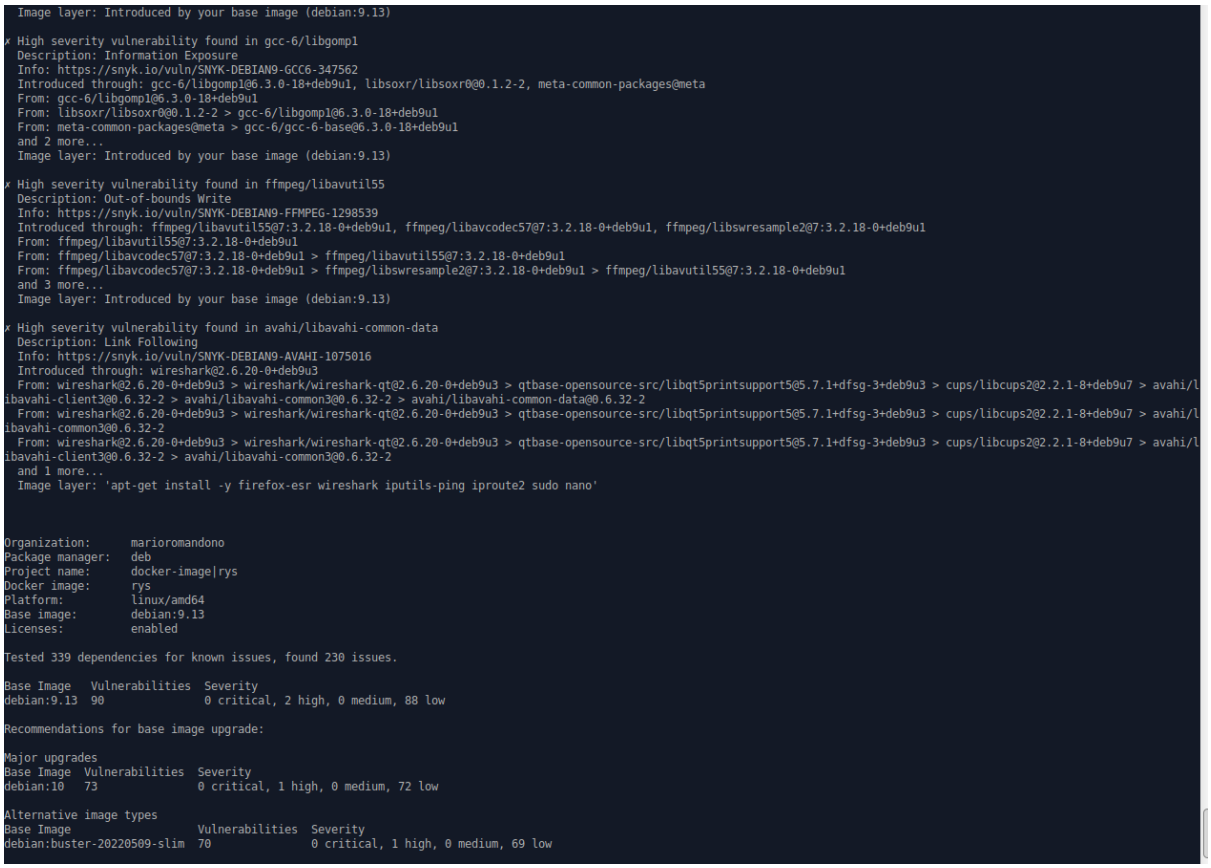

Figura 5.9: Análisis de vulnerabilidades de la imagen `rys`

En resumen, podemos finalizar este análisis de seguridad afirmando que, si bien se podrían haber aplicado otras medidas adicionales, este proyecto cuenta con un nivel de seguridad aceptable para poder ser candidato a la sustitución de la implementación actual de los laboratorios, gracias al uso de una máquina virtual como barrera de aislamiento extra y al control de las opciones disponibles a la hora de crear contenedores e imágenes, que evitan que se puedan generar situaciones de riesgo por parte de usuarios maliciosos.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este Trabajo Fin de Grado se pretendía evaluar la puesta en funcionamiento de un laboratorio docente empleando contenedores en lugar de máquinas virtuales, con el fin de proporcionar una solución a la problemática actual en las prácticas de las asignaturas de sistemas y redes de las titulaciones de la Facultad de Informática de la Universidad Complutense de Madrid.

Tras efectuar un estudio de cómo funciona la virtualización a nivel de sistema operativo y analizar qué características debían tener los contenedores para poder sustituir a las máquinas virtuales actuales, se diseñó una solución que utilizase Docker como *software* de gestión de contenedores, Alpine Linux y VirtualBox para crear una máquina virtual que sirviese como base para la ejecución de los contenedores, y Go para desarrollar una herramienta que automatizase el proceso de despliegue de los contenedores.

El funcionamiento de esta propuesta fue evaluado en una prueba de concepto consistente en realizar la práctica 0 de la asignatura Redes y Seguridad II, tarea que se pudo llevar a cabo de manera exitosa. Además, se comparó el rendimiento entre la solución planteada la actual, que utiliza máquinas virtuales, obteniendo que los contenedores ofrecen mejores resultados en cuanto a espacio de disco ocupado, tiempos de creación y uso de memoria RAM. Finalmente, se examinó la seguridad de este proyecto, observando que, gracias al uso de la máquina virtual de Alpine Linux y a las medidas implementadas durante la creación de los contenedores, se puede alcanzar un nivel de seguridad comparable con el del despliegue actual.

Por estos motivos, podemos concluir que la implementación de un laboratorio mediante contenedores es totalmente viable, obteniendo los mismos resultados de aprendizaje por parte de los estudiantes. Además, esta propuesta supone una mejor alternativa con respecto al uso de las máquinas virtuales actuales debido a que proporciona un rendimiento muy superior, sin que la seguridad del sistema se vea comprometida en ningún caso. Por último, el uso de contenedores puede ser aprovechado para, a través de nuevas prácticas o modificando las ya existentes, añadir objetivos relacionados con la gestión y administración de contenedores, ya que se trata de un conocimiento que puede resultar muy necesario de cara al futuro profesional de los estudiantes.

6.2. Trabajo futuro

Este proyecto pretende ser una primera aproximación para comprobar si el uso de contenedores para realizar prácticas académicas era más beneficioso que seguir empleando máquinas virtuales. Por este motivo, existen bastantes posibilidades de realizar mejoras sobre la implementación en el futuro.

Sobre la herramienta de despliegue de contenedores, una posible mejora sería añadir la capacidad de que pueda escribir en un fichero de *log* las operaciones que se van ejecutando, con el fin de poder auditar su comportamiento en caso necesario. También se podría añadir una opción a la herramienta que guardase el estado de los contenedores en un archivo, lo que permitiría que, por ejemplo, un estudiante pudiera comenzar una práctica en los laboratorios de la facultad y completarla más tarde en su ordenador personal. Por último, a pesar de que se ha alcanzado un alto nivel de optimización gracias al uso de concurrencia, sería interesante realizar pruebas de perfilado para conocer si es posible aplicar más mejoras de rendimiento a nivel de código.

Otro cambio que se podría realizar es eliminar la restricción estricta de que se deban usar imágenes locales para la creación de los contenedores. Para garantizar que la seguridad del sistema no se pone en riesgo, se podría comprobar que la imagen seleccionada pertenece a un registro confiable, así como verificar que no se ha visto modificada durante el proceso de descarga mediante Docker Content Trust.

Si bien es cierto que el uso de la máquina virtual de Alpine proporciona un necesario nivel de aislamiento extra, se podrían evaluar otras alternativas que lograsen eliminar la necesidad de esta máquina virtual, pero otorgando el mismo grado de seguridad. Una posibilidad es volver a comprobar si es posible la ejecución de contenedores *rootless*, revisando especialmente que no existan problemas con los permisos de los ficheros compartidos a través de los *bind mounts*, tal y como se había especificado en la sección 3.6. Otra opción recomendable es estudiar el uso de *runtimes* diferentes a runc, como gVisor [127], Sysbox [128] o Kata Containers [129]. Estos *runtimes* proporcionan diversas alternativas para incrementar el nivel de seguridad de los contenedores, como el uso de máquinas virtuales ligeras o *kernels* a nivel de usuario que evitan la interacción directa entre el contenedor y el núcleo del sistema operativo.

Finalmente, se podrían añadir nuevas funcionalidades que mejorasen la experiencia durante la realización de las prácticas, como la creación de una interfaz gráfica que permitiese la definición visual de los contenedores deseados o, basándonos en trabajos como Labtainers, agregar la posibilidad de obtener automáticamente información de las tareas ejecutadas dentro de los contenedores para poder evaluar el desempeño de los estudiantes durante la práctica.

Chapter 6

Conclusions and future work

6.1. Conclusions

In this Final Degree Project, the aim was to evaluate the implementation of a teaching laboratory using containers instead of virtual machines, in order to provide a solution to the current problem in the practices of the systems and networks subjects of the degrees of the Faculty of Computer Science of the Complutense University of Madrid.

After conducting a study of how OS-level virtualization works and analyzing what features containers should have to be able to replace the current virtual machines, a solution was designed using Docker as container management software, Alpine Linux and VirtualBox to create a virtual machine to serve as a base for running the containers, and Go to develop a tool to automate the container deployment process.

The functioning of this proposal was evaluated in a proof of concept consisting of performing practice 0 of the Computer Networks Security II subject, a task that was successfully carried out. In addition, the performance between the proposed solution and the current one, which uses virtual machines, was compared, obtaining that containers offer better results in terms of occupied disk space, creation times and RAM memory usage. Finally, the security of this project was examined, observing that, thanks to the use of the Alpine Linux virtual machine and the measures implemented during the creation of the containers, a level of security comparable to that of the current deployment can be achieved.

For these reasons, we can conclude that the implementation of a laboratory using containers is totally viable, obtaining the same learning results for the students. In addition, this proposal is a better alternative to the use of the current virtual machines because it provides a much higher performance, without compromising the security of the system in any case. Finally, the use of containers can be used to add objectives related to the management and administration of containers through new practices or by modifying existing ones, since this knowledge may be very necessary for the students' professional future.

6.2. Future work

This project is intended to be a first approach to test whether the use of containers for academic practices was more beneficial than continuing to use virtual machines. For this reason, there are many possibilities to improve the implementation in the future.

Regarding the container deployment tool, a possible improvement would be to add the ability to write to a log file the operations that are being executed, in order to be able to audit its behavior if necessary. Also, an option could be added to the tool that would save the status of the containers in a file, which would allow, for example, a student to start a practice in the faculty labs and complete it later on his or her personal computer. Finally, although a high level of optimization has been achieved through the use of concurrency, it would be interesting to perform profiling tests to see if further performance improvements can be applied at the code level.

Another change that could be made is to remove the strict restriction that local images must be used for container creation. To ensure that system security is not put at risk, it could be checked that the selected image belongs to a trusted registry, as well as verifying that it has not been modified during the download process using Docker Content Trust.

While it is true that the use of the Alpine virtual machine provides a necessary extra level of isolation, other alternatives could be evaluated that would eliminate the need for this virtual machine, but providing the same degree of security. One possibility is to recheck if the execution of rootless containers is possible, checking especially that there are no problems with the permissions of the shared files through bind mounts, as specified in the section 3.6. Another recommended option is to study the use of different runtimes to runc, such as gVisor [127], Sysbox [128] or Kata Containers [129]. These runtimes provide several alternatives to increase the security level of containers, such as the use of lightweight virtual machines or user-level kernels that avoid direct interaction between the container and the operating system kernel.

Finally, new functionalities could be added to improve the experience during the practices, such as the creation of a graphical interface that would allow the visual definition of the desired containers or, based on works such as Labtainers, adding the possibility of automatically obtaining information of the tasks executed inside the containers to be able to evaluate the performance of the students during the practice.

Bibliografía

[1] "Amazon Web Services", *Amazon*, 2022. [En línea]. Disponible en: https://aws.amazon.com/es/ [Accedido: 13-05-2022].

[2] "Microsoft Azure", *Microsoft*, 2022. [En línea]. Disponible en: https://azure.microsoft.com/es-es/ [Accedido: 13-05-2022].

[3] H. E. Schaffer, S. F. Averitt, M. I. Hoit, A. Peeler, E. D. Sills y M. A. Vouk, "NCSU's Virtual Computing Lab: A Cloud Computing Solution", *Computer*, vol. 42, n.º 7, págs. 94-97, 2009. DOI: 10.1109/MC.2009.230.

[4] M. C. Murphy y M. McClelland, "Computer Lab to Go: A "Cloud" Computing Implementation", en *The Proceedings of the Information Systems Education Conference*, 2008. [En línea]. Disponible en: https://proc.edsig.org/2008/2343/ISECON.2008.Murphy.pdf.

[5] "What is a Container?", *Docker*, 2022. [En línea]. Disponible en: https://www.docker.com/resources/what-container/ [Accedido: 13-05-2022].

[6] J. H. Sianipar, C. Willems y C. Meinel, "A container-based virtual laboratory for internet security e-learning", *International Journal of Learning and Teaching*, vol. 2, n.º 2, págs. 121-128, dic. de 2016. DOI: 10.18178/ijlt.2.2.121-128.

[7] "Labtainers", *Center for Cybersecurity, Cyber Operations - Naval Postgraduate School*, 2022. [En línea]. Disponible en: https://nps.edu/web/c3o/labtainers [Accedido: 24-04-2022].

[8] M. F. Thompson y C. E. Irvine, "Individualizing Cybersecurity Lab Exercises with Labtainers", *IEEE Security & Privacy*, vol. 16, n.º 2, págs. 91-95, 2018. DOI: 10.1109/MSP.2018.1870862.

[9] G. Chen, "CvLabs: A Container Based Interactive Virtual Lab for IT Education", Georgia Institute of Technology, inf. téc., 2020. [En línea]. Disponible en: http://hdl.handle.net/1853/64898.

[10] Victoria Fedoseenko. "A brief history of virtualization, or why do we divide something at all", *ISPsystem*, 25 de nov. de 2019. [En línea]. Disponible en: https://www.ispsystem.com/news/brief-history-of-virtualization [Accedido: 01-03-2022].

[11] O. Agesen, A. Garthwaite, J. Sheldon y P. Subrahmanyam, "The evolution of an x86 virtual machine monitor", *ACM SIGOPS Operating Systems Review*, vol. 44, n.º 4, págs. 3-18, 2010. DOI: 10.1145/1899928.1899930.

[12] *Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture*, Intel Corporation, abr. de 2005.

[13] *AMD64 Virtualization Codenamed "Pacifica" Technology Secure Virtual Machine Architecture Reference Manual*, Advanced Micro Devices, mayo de 2005.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt y A. Warfield, "Xen and the Art of Virtualization", *ACM SIGOPS Operating Systems Review*, vol. 37, n.º 5, págs. 164-177, oct. de 2003, ISSN: 0163-5980. DOI: 10.1145/1165389.945462.

[15] Avi Kivity. "[PATCH 0/7] KVM: Kernel-based Virtual Machine", *Linux-Kernel Archive*, 19 de oct. de 2006. [En línea]. Disponible en: https://lkml.iu.edu/hypermail/linux/kernel/0610.2/1369.html [Accedido: 07-03-2022].

[16] "Oracle VM VirtualBox", *Oracle*, 2022. [En línea]. Disponible en: https://www.virtualbox.org/ [Accedido: 07-03-2022].

[17] "Firecracker", *Amazon Web Services*, 2022. [En línea]. Disponible en: https://firecracker-microvm.github.io/ [Accedido: 07-03-2022].

[18] "Understanding Full Virtualization, Paravirtualization, and Hardware Assist", VMWare, Inc, 11 de mar. de 2008. [En línea]. Disponible en: https://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html [Accedido: 02-03-2021].

[19] D. Barrett y G. Kipper, "How Virtualization Happens", en *Virtualization and Forensics*, D. Barrett y G. Kipper, eds., Boston: Syngress, 2010, págs. 3-24. DOI: https://doi.org/10.1016/B978-1-59749-557-8.00001-1.

[20] R. J. Creasy, "The Origin of the VM/370 Time-Sharing System", *IBM Journal of Research and Development*, vol. 25, n.º 5, págs. 483-490, 1981. DOI: 10.1147/rd.255.0483.

[21] "Paravirt_ops", *The Linux Kernel Documentation*, 2019. [En línea]. Disponible en: https://www.kernel.org/doc/html/latest/virt/paravirt_ops.html [Accedido: 05-03-2022].

[22] "DomU Support for Xen", *Xen Project Wiki*, 2017. [En línea]. Disponible en: https://wiki.xenproject.org/wiki/DomU_Support_for_Xen [Accedido: 05-03-2022].

[23] "Kernel-based Virtual Machine", *KVM*, 2022. [En línea]. Disponible en: https://www.linux-kvm.org/page/Main_Page [Accedido: 15-05-2022].

[24] "Xen Project", *The Linux Fundation*, 2022. [En línea]. Disponible en: https://xenproject.org/ [Accedido: 15-05-2022].

[25] "Hyper-V Technology Overview", *Microsoft Docs*, 29 de jul. de 2021. [En línea]. Disponible en: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview [Accedido: 15-05-2022].

[26] "VMware ESXi", *VMWare*, 2022. [En línea]. Disponible en: https://www.vmware.com/products/esxi-and-esx.html [Accedido: 15-05-2022].

[27] "VMware Workstation Pro", *VMWare*, 2022. [En línea]. Disponible en: https://www.vmware.com/products/workstation-pro.html [Accedido: 15-05-2022].

[28] "Parallels Desktop", *Parallels*, 2022. [En línea]. Disponible en: https://www.parallels.com/products/desktop/ [Accedido: 15-05-2022].

[29] M. Kaschke. "Virtual Machine (VM) vs. Container", *Medium*, 24 de nov. de 2020. [En línea]. Disponible en: https://mkaschke.medium.com/virtual-machine-vm-vs-container-13ab51f4c177 [Accedido: 10-03-2022].

[30] T. Hildred. "The History of Containers", *Red Hat Blog*, 28 de ago. de 2015. [En línea]. Disponible en: https://www.redhat.com/en/blog/history-containers [Accedido: 11-03-2022].

[31] C. Siebenmann. "The somewhat surprising history of chroot()", *Wandering Thoughts*, 25 de ago. de 2015. [En línea]. Disponible en: https://utcc.utoronto.ca/~cks/space/blog/unix/ChrootHistory [Accedido: 08-03-2022].

[32] "Jails", en *The FreeBSD Handbook*, The FreeBSD Project, ed. 2021. [En línea]. Disponible en: https://docs.freebsd.org/en/books/handbook/jails/.

[33] "Overview", *Linux-VServer*, 22 de abr. de 2013. [En línea]. Disponible en: http://linux-vserver.org/Overview [Accedido: 11-03-2022].

[34] P. B. Menage, "Adding Generic Process Containers to the Linux Kernel", en *Proceedings of the Ottawa Linux Symposium*, 2007, págs. 45-58. [En línea]. Disponible en: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-45-58.pdf.

[35] M. Kerrisk. "Namespaces in operation, part 1: namespaces overview", *LWN.net*, 4 de ene. de 2013. [En línea]. Disponible en: https://lwn.net/Articles/531114/ [Accedido: 11-03-2022].

[36] "What's LXC?", *linuxcontainers.org*, 28 de sep. de 2021. [En línea]. Disponible en: https://linuxcontainers.org/lxc/introduction/ [Accedido: 12-03-2022].

[37] "Why Docker?", *Docker*, 2022. [En línea]. Disponible en: https://www.docker.com/why-docker [Accedido: 12-03-2022].

[38] M. Eder, "Hypervisor- vs. Container-based Virtualization", en *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 2015, págs. 1-7. DOI: 10.2313/NET-2016-07-1_01.

[39] S. Hykes. "Docker 0.9: introducing execution drivers and libcontainer", *Docker Blog*, 10 de mar. de 2014. [En línea]. Disponible en: https://www.docker.com/blog/docker-0-9-introducing-execution-drivers-and-libcontainer/ [Accedido: 12-03-2022].

[40] J. Barr. "AWS Elastic Beanstalk for Docker", *AWS News Blog*, 23 de abr. de 2014. [En línea]. Disponible en: https://aws.amazon.com/es/blogs/aws/aws-elastic-beanstalk-for-docker/ [Accedido: 12-03-2022].

[41] "What is Kubernetes?", *Kubernetes*, 23 de jul. de 2021. [En línea]. Disponible en: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ [Accedido: 12-03-2022].

[42] "About the Open Container Initiative", *Open Container Initiative*, 2020. [En línea]. Disponible en: https://opencontainers.org/about/overview/ [Accedido: 12-03-2022].

[43] Open Container Iniciative. "OCI Image Format Specification", *repositorio de GitHub*, 2022. [En línea]. Disponible en: https://github.com/opencontainers/image-spec [Accedido: 12-03-2022].

[44] Open Container Iniciative. "Open Container Initiative Runtime Specification", *repositorio de GitHub*, 2022. [En línea]. Disponible en: https://github.com/opencontainers/runtime-spec [Accedido: 12-03-2022].

[45] T. Brown, "Bringing Docker To Windows Developers with Windows Server Containers", *MSDN Magazine Issues*, vol. 32, n.º 4, abr. de 2017. [En línea]. Disponible en: https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/april/containers-bringing-docker-to-windows-developers-with-windows-server-containers.

[46] "Docker overview", *Docker*, 6 de feb. de 2022. [En línea]. Disponible en: https://docs.docker.com/get-started/overview/ [Accedido: 13-03-2022].

[47] "containerd", *containerd*, 2022. [En línea]. Disponible en: https://containerd.io/ [Accedido: 13-03-2022].

[48] Open Containers Iniciative. "runc", *repositorio de GitHub*, 2022. [En línea]. Disponible en: https://github.com/opencontainers/runc [Accedido: 13-03-2022].

[49] S. Hykes. "Introducing Moby Project: a new open-source project to advance the software containerization movement", *Docker Blog*, 18 de abr. de 2017. [En línea]. Disponible en: https://www.docker.com/blog/introducing-the-moby-project/ [Accedido: 12-03-2022].

[50] "What is Podman?", *Podman*, 30 de jun. de 2021. [En línea]. Disponible en: https://podman.io/whatis.html [Accedido: 13-03-2022].

[51] containers. "containers", *repositorio de GitHub*, 2022. [En línea]. Disponible en: https://github.com/containers [Accedido: 13-03-2022].

[52] P. Sethi. "Rootless containers with Podman: The basics", *Red Hat Developer Blog*, 25 de sep. de 2020. [En línea]. Disponible en: https://developers.redhat.com/blog/2020/09/25/rootless-containers-with-podman-the-basics [Accedido: 13-03-2022].

[53] "Run the Docker daemon as a non-root user (Rootless mode)", *Docker*, 17 de dic. de 2021. [En línea]. Disponible en: https://docs.docker.com/engine/security/rootless/ [Accedido: 13-03-2022].

[54] containers. "crun", *repositorio de GitHub*, 2022. [En línea]. Disponible en: https://github.com/containers/crun [Accedido: 13-03-2022].

[55] containers. "Container Tools Guide", *repositorio de GitHub*, 24 de ago. de 2021. [En línea]. Disponible en: https://github.com/containers/buildah/blob/main/docs/containertools/README.md [Accedido: 13-03-2022].

[56] *Building, running, and managing Linux containers on Red Hat Enterprise Linux 8*, Red Hat, 8 de mar. de 2022, pág. 10. [En línea]. Disponible en: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/pdf/building_running_and_managing_containers/red_hat_enterprise_linux-8-building_running_and_managing_containers-en-us.pdf.

[57] "What is rkt?", *Red Hat*, 23 de jun. de 2021. [En línea]. Disponible en: https://www.redhat.com/es/topics/containers/what-is-rkt [Accedido: 13-03-2022].

[58] rkt. "Ending and archiving the rkt project", *repositorio de GitHub*, 5 de feb. de 2020. [En línea]. Disponible en: https://github.com/rkt/rkt/issues/4024 [Accedido: 13-03-2022].

[59] App Container. "App Container", *repositorio de GitHub*, 5 de feb. de 2020. [En línea]. Disponible en: https://github.com/appc/spec [Accedido: 13-03-2022].

[60] "What is LXD?", *linuxcontainers.org*, 15 de nov. de 2021. [En línea]. Disponible en: https://linuxcontainers.org/lxd/introduction/ [Accedido: 13-03-2022].

[61] "python", *Docker Hub*, 2022. [En línea]. Disponible en: https://hub.docker.com/_/python [Accedido: 14-03-2022].

[62] W. Felter, A. Ferreira, R. Rajamony y J. Rubio, "An updated performance comparison of virtual machines and Linux containers", IBM Research Division, inf. téc., 21 de jul. de 2014. [En línea]. Disponible en: https://dominoweb.draco.res.ibm.com/reports/rc25482.pdf [Accedido: 02-05-2022].

[63] R. Morabito, J. Kjällman y M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison", en *2015 IEEE International Conference on Cloud Engineering*, 2015, págs. 386-393. DOI: 10.1109/IC2E.2015.74.

[64] Z. Li, M. Kihl, Q. Lu y J. A. Andersson, "Performance Overhead Comparison between Hypervisor and Container Based Virtualization", en *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, 2017, págs. 955-962. DOI: 10.1109/AINA.2017.79.

[65] R.-S. Schmoll, T. Fischer, H. Salah y F. H. P. Fitzek, "Comparing and Evaluating Application-specific Boot Times of Virtualized Instances", en *2019 IEEE 2nd 5G World Forum (5GWF)*, 2019, págs. 602-606. DOI: 10.1109/5GWF.2019.8911615.

[66] S. Natarajan, R. Krishnan, A. Ghanwani, D. Krishnaswamy, P. Willis, A. Chaudhary y F. Huici, "An Analysis of Lightweight Virtualization Technologies for NFV", Internet Engineering Task Force, inf. téc., jul. de 2016. [En línea]. Disponible en: https://datatracker.ietf.org/doc/html/draft-natarajan-nfvrg-containers-for-nfv-03 [Accedido: 02-05-2022].

[67] C. G. Kominos, N. Seyvet y K. Vandikas, "Bare-metal, virtual machines and containers in OpenStack", en *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, 2017, págs. 36-43. DOI: 10.1109/ICIN.2017.7899247.

[68] S. Shirinbab, L. Lundberg y E. Casalicchio, "Performance evaluation of containers and virtual machines when running Cassandra workload concurrently", *Concurrency and Computation: Practice and Experience*, vol. 32, n.º 17, 2020. DOI: https://doi.org/10.1002/cpe.5693.

[69] T. Adufu, J. Choi e Y. Kim, "Is container-based technology a winner for high performance scientific applications?", en *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2015, págs. 507-510. DOI: 10.1109/APNOMS.2015.7275379.

[70] M. T. Chung, N. Quang-Hung, M.-T. Nguyen y N. Thoai, "Using Docker in high performance computing applications", en *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, 2016, págs. 52-57. DOI: 10.1109/CCE.2016.7562612.

[71] G. E. de Velp, E. Rivière y R. Sadre, "Understanding the Performance of Container Execution Environments", en *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*, Association for Computing Machinery, 2020, págs. 37-42. DOI: 10.1145/3429885.3429967.

[72] S. Sultan, I. Ahmad y T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead", *IEEE Access*, vol. 7, págs. 52 976-52 996, 2019. DOI: 10.1109/ACCESS.2019.2911732.

[73] T. Bui, *Analysis of Docker Security*, arXiv, 2015. DOI: 10.48550/ARXIV.1501.02967.

[74] "Docker security", *Docker*, 10 de abr. de 2021. [En línea]. Disponible en: https://docs.docker.com/engine/security/ [Accedido: 07-05-2022].

[75] M. Calizo. "Secure your containers with SELinux", *Opensource.com*, 18 de nov. de 2020. [En línea]. Disponible en: https://opensource.com/article/20/11/selinux-containers [Accedido: 07-05-2022].

[76] "AppArmor security profiles for Docker", *Docker*, 6 de ago. de 2021. [En línea]. Disponible en: https://docs.docker.com/engine/security/apparmor/ [Accedido: 07-05-2022].

[77] "Improving Linux container security with seccomp", *Red Hat Blog*, 15 de jun. de 2020. [En línea]. Disponible en: https://www.redhat.com/sysadmin/container-security-seccomp [Accedido: 07-05-2022].

[78] "Vulnerability scanning for Docker local images", *Docker*, 15 de dic. de 2021. [En línea]. Disponible en: https://docs.docker.com/engine/scan/ [Accedido: 07-05-2022].

[79] "Content trust in Docker", *Docker*, 16 de ago. de 2021. [En línea]. Disponible en: https://docs.docker.com/engine/security/trust/ [Accedido: 07-05-2022].

[80] "Debian – The Universal Operating System", *Debian*, 2 de mar. de 2022. [En línea]. Disponible en: https://www.debian.org/index.en.html [Accedido: 16-05-2022].

[81] "Wireshark", *Wireshark*, 2022. [En línea]. Disponible en: https://www.wireshark.org/ [Accedido: 16-05-2022].

[82] "The netfilter.org iptables project", *Netfilter*, 2022. [En línea]. Disponible en: https://www.netfilter.org/projects/iptables/index.html [Accedido: 16-05-2022].

[83] "OpenVPN", *OpenVPN*, 2022. [En línea]. Disponible en: https://openvpn.net/ [Accedido: 16-05-2022].

[84] "The Apache HTTP Server Project", *The Apache Software Foundation*, 2022. [En línea]. Disponible en: https://httpd.apache.org/ [Accedido: 16-05-2022].

[85] "Kali Linux", *OffSec Services Limited*, 2022. [En línea]. Disponible en: https://www.kali.org/ [Accedido: 16-05-2022].

[86] "Nmap: the Network Mapper", *Nmap*, 2022. [En línea]. Disponible en: https://nmap.org/ [Accedido: 16-05-2022].

[87] "OWASP Zed Attack Proxy (ZAP)", *The OWASP Foundation*, 2022. [En línea]. Disponible en: https://www.zaproxy.org/ [Accedido: 16-05-2022].

[88] "Metasploitable 2 Exploitability Guide", *Rapid7*, 2022. [En línea]. Disponible en: https://docs.rapid7.com/metasploit/metasploitable-2-exploitability-guide/ [Accedido: 16-05-2022].

[89] "I/O Virtualization Bottlenecks in Cloud Computing Today", en *Proceedings of the 2nd Conference on I/O Virtualization*, USENIX Association, 2010. [En línea]. Disponible en: https://www.jeffshafer.com/publications/papers/shafer-wiov2010.pdf.

[90] LXC. "Ubuntu template", *repositorio de GitHub*, 3 de mayo de 2022. [En línea]. Disponible en: https://github.com/lxc/lxc-ci/blob/master/images/ubuntu.yaml [Accedido: 18-05-2022].

[91] "debian", *Docker Hub*, 2022. [En línea]. Disponible en: https://hub.docker.com/_/debian [Accedido: 18-05-2022].

[92] "X.org", *X.org Foundation*, 23 de oct. de 2020. [En línea]. Disponible en: https://www.x.org/wiki/ [Accedido: 22-05-2022].

[93] M. Viereck. "Short setups to provide X display to container", *repositorio de GitHub*, 29 de abr. de 2021. [En línea]. Disponible en: https://github.com/mviereck/x11docker/wiki/Short-setups-to-provide-X-display-to-container [Accedido: 22-05-2022].

[94] "Use bind mounts", *Docker*, 25 de abr. de 2022. [En línea]. Disponible en: https://docs.docker.com/storage/bind-mounts/ [Accedido: 22-05-2022].

[95] M. Viereck. "Untrusted cookie for container applications", *repositorio de GitHub*, 17 de feb. de 2022. [En línea]. Disponible en: https://github.com/mviereck/x11docker/wiki/X-authentication-with-cookies-and-xhost-(%22No-protocol-specified%22-error)#untrusted-cookie-for-container-applications [Accedido: 22-05-2022].

[96] "docker run", *Docker*, 11 de ago. de 2021. [En línea]. Disponible en: https://docs.docker.com/engine/reference/commandline/run/ [Accedido: 03-12-2021].

[97] "ubuntu", *Docker Hub*, 2021. [En línea]. Disponible en: https://hub.docker.com/_/ubuntu [Accedido: 03-12-2021].

[98] "Overview of Docker Compose", *Docker*, 15 de oct. de 2021. [En línea]. Disponible en: https://docs.docker.com/compose/ [Accedido: 03-12-2021].

[99] "mysql", *Docker Hub*, 2022. [En línea]. Disponible en: https://hub.docker.com/_/mysql [Accedido: 22-05-2022].

[100] "httpd", *Docker Hub*, 2022. [En línea]. Disponible en: https://hub.docker.com/_/httpd [Accedido: 22-05-2022].

[101] "Runtime options with Memory, CPUs, and GPUs", *Docker*, 19 de nov. de 2021. [En línea]. Disponible en: https://docs.docker.com/config/containers/resource_constraints/ [Accedido: 04-12-2021].

[102] B. Baude. "Rootless containers with Podman: The basics", *Enable Sysadmin*, 28 de oct. de 2019. [En línea]. Disponible en: https://www.redhat.com/sysadmin/container-networking-podman [Accedido: 23-05-2022].

[103] "Install Docker Engine", *Docker*, 15 de oct. de 2021. [En línea]. Disponible en: https://docs.docker.com/engine/install/ [Accedido: 05-11-2021].

[104] "Docker Desktop Overview", *Docker*, 15 de oct. de 2021. [En línea]. Disponible en: https://docs.docker.com/desktop/ [Accedido: 05-11-2021].

[105] *Oracle® VM VirtualBox® User Manual*, ver. 6.1.28, Oracle Corporation, 2021, pág. 51. [En línea]. Disponible en: https://download.virtualbox.org/virtualbox/6.1.28/UserManual.pdf [Accedido: 05-11-2021].

[106] "Running Docker Desktop in nested virtualization scenarios", *Docker*, 29 de sep. de 2021. [En línea]. Disponible en: https://docs.docker.com/desktop/windows/troubleshoot/#running-docker-desktop-in-nested-virtualization-scenarios [Accedido: 05-11-2021].

[107] M. Coleman. "Containers and VMs Together", *Docker Blog*, 8 de abr. de 2016. [En línea]. Disponible en: https://www.docker.com/blog/containers-and-vms-together [Accedido: 05-11-2021].

[108] "Fedora CoreOS Documentation", *Fedora Project*, 27 de mayo de 2021. [En línea]. Disponible en: https://docs.fedoraproject.org/en-US/fedora-coreos/ [Accedido: 23-11-2021].

[109] "Fedora Silverblue User Guide", *Fedora Project*, 28 de sep. de 2021. [En línea]. Disponible en: https://docs.fedoraproject.org/en-US/fedora-silverblue/ [Accedido: 23-11-2021].

[110] "Flatcar Container Linux", *Flatcar*, 2021. [En línea]. Disponible en: https://www.flatcar.org/ [Accedido: 23-11-2021].

[111] "Overview of RancherOS", *Rancher*, 20 de oct. de 2021. [En línea]. Disponible en: https://rancher.com/docs/os/v1.x/en/ [Accedido: 23-11-2021].

[112] "About", *Alpine Linux*, 2021. [En línea]. Disponible en: https://alpinelinux.org/about/ [Accedido: 23-11-2021].

[113] *Debian GNU/Linux Installation Guide*, Debian, 27 de mar. de 2022. [En línea]. Disponible en: https://www.debian.org/releases/stable/amd64/install.en.pdf [Accedido: 15-05-2022].

[114] *Fedora Silverblue User Guide*, Fedora, 22 de feb. de 2022. [En línea]. Disponible en: https://docs.fedoraproject.org/en-US/fedora-silverblue/ [Accedido: 15-05-2022].

[115] "Alpine Linux Wiki", *Alpine Linux*, 2022. [En línea]. Disponible en: https://wiki.alpinelinux.org/wiki/Main_Page [Accedido: 15-05-2022].

[116] "Develop with Docker Engine SDKs", *Docker*, 15 de ene. de 2022. [En línea]. Disponible en: https://docs.docker.com/engine/api/sdk/ [Accedido: 23-04-2022].

[117] "I'm trying to put a command in a variable, but the complex cases always fail!", *BashFAQ*, 12 de feb. de 2021. [En línea]. Disponible en: http://mywiki.wooledge.org/BashFAQ/050 [Accedido: 23-04-2022].

[118] "errgroups", *Go Packages*, 20 de feb. de 2021. [En línea]. Disponible en: https://pkg.go.dev/golang.org/x/sync/errgroup [Accedido: 24-04-2022].

[119] "YAML support for the Go language", *Go Packages*, 7 de ene. de 2021. [En línea]. Disponible en: https://pkg.go.dev/gopkg.in/yaml.v3 [Accedido: 23-04-2022].

[120] "net", *Go Packages*, 12 de abr. de 2022. [En línea]. Disponible en: https://pkg.go.dev/net [Accedido: 23-04-2022].

[121] "os/exec", *Go Packages*, 12 de abr. de 2022. [En línea]. Disponible en: https://pkg.go.dev/os/exec [Accedido: 23-04-2022].

[122] "labsoncontainers", *Go Packages*, 12 de abr. de 2022. [En línea]. Disponible en: https://pkg.go.dev/github.com/marioromandono/labsoncontainers [Accedido: 15-05-2022].

[123] *SimpleHTTPServer*, Python 2.7.18 documentation, 19 de jun. de 2020. [En línea]. Disponible en: https://docs.python.org/2/library/simplehttpserver.html [Accedido: 21-05-2022].

[124] D. Bakhvalov. "How to get consistent results when benchmarking on Linux?", *Easyperf.net*, 2 de ago. de 2019. [En línea]. Disponible en: https://easyperf.net/blog/2019/08/02/Perf-measurement-environment-on-Linux [Accedido: 21-05-2022].

[125] S. Godard. "sar manual page", *Sysstat Utilities Home Page*, ene. de 2022. [En línea]. Disponible en: http://sebastien.godard.pagesperso-orange.fr/man_sar.html [Accedido: 21-05-2022].

[126] Docker. "Docker Bench for Security", *repositorio de GitHub*, 2022. [En línea]. Disponible en: https://github.com/docker/docker-bench-security [Accedido: 22-05-2022].

[127] "gVisor", *gVisor*, 2022. [En línea]. Disponible en: https://gvisor.dev [Accedido: 24-05-2022].

[128] "Sysbox", *Nestybox*, 2022. [En línea]. Disponible en: https://www.nestybox.com/sysbox [Accedido: 24-05-2022].

[129] "Kata Containers", *Kata Containers*, 2022. [En línea]. Disponible en: https://katacontainers.io [Accedido: 24-05-2022].

Índice de figuras

2.1. Funcionamiento de la virtualización completa con soporte de hardware . 12
2.2. Funcionamiento de la paravirtualización 13
2.3. Comparación entre hipervisores y contenedores 15

4.1. Espacio de disco ocupado por Alpine medido desde dentro de la máquina virtual 44
4.2. Espacio de disco ocupado por Alpine medido desde fuera de la máquina virtual 44

5.1. Topología de red de la práctica 0 de Redes y Seguridad II 61
5.2. Estado tras haber ejecutado `./LabsOnContainers -c practica0.yaml` . 62
5.3. Instalación de Python en el contenedor del servidor 63
5.4. Configuración de red de los contenedores 64
5.5. Resultado tras mandar un paquete ICMP en la misma red 64
5.6. Resultado tras mandar un paquete ICMP a otra red 65
5.7. Resultado tras acceder al servidor web con Firefox desde Host 2 65
5.8. Estado tras haber ejecutado `./LabsOnContainers -r practica0` 66
5.9. Análisis de vulnerabilidades de la imagen `rys` 73

Índice de tablas

3.1. Comparativa entre hipervisores de tipo 2 36
3.2. Comparativa entre distribuciones Linux 39

5.1. Comparativa del espacio utilizado 67
5.2. Comparativa de tiempos de creación del entorno 69
5.3. Comparativa del uso de CPU, memoria y entrada/salida 70

Apéndice A

Código de la versión preliminar de la herramienta de despliegue

```
#!/bin/sh

run_container() {
    docker container run \
        --name "$nombre" \
        --hostname "$nombre" \
        -d -it --cap-add=NET_ADMIN \
        --init \
        --env DISPLAY \
        --env XAUTHORITY=/cookie \
        --mount type=bind,source="$(pwd)",target=/mnt/shared \
        --mount type=bind,source=/tmp/.X11-unix,target=/tmp/.X11-unix \
        --mount type=bind,source="${Cookiefile}",target=/cookie \
        --label background="$background" \
        "$imagen"
}

crear_cookie() {
    if [ ! -d ~/.cookies ]; then
        mkdir ~/.cookies
    fi
    Cookiefile=~/.cookies/"$nombre"_cookie
    :> "$Cookiefile"
    xauth -f "$Cookiefile" generate "$DISPLAY" . untrusted timeout 3600
    Cookie="$(xauth -f "$Cookiefile" nlist "$DISPLAY" | sed -e
    's/^..../ffff/')"
    echo "$Cookie" | xauth -f "$Cookiefile" nmerge -
}

crear_terminal() {
```

```
    primer_nombre=$1
    shift
    first=1
    for nombre; do
        if [ "$first" = 1 ]; then set --; first=0; fi
        set -- "$@" --tab -e "ash -c 'docker container attach $nombre;
        exec ash'"
    done
    xfce4-terminal -e "ash -c 'docker container attach $primer_nombre;
    exec ash'" "$@"
}

error() {
    echo "$*" 1>&2
    exit 1
}

uso() {
    echo "Uso: $0 <-c fichero | -l nombre_practica | -i nombre_practica
    | -p nombre_practica | -r nombre_practica | -h>"
    echo ""
    echo "-c: Crea el entorno de contenedores a partir del fichero YAML
    proporcionado (se destruyen los contenedores asociados al entorno
    del fichero)"
    echo "-l: Ejecuta todos los contenedores asociados al entorno
    proporcionado"
    echo "-i: Muestra la información de todos los contenedores asociados
    al entorno proporcionado"
    echo "-p: Detiene todos los contenedores asociados al entorno
    proporcionado"
    echo "-r: Destruye todos los contenedores asociados al entorno
    proporcionado"
    echo "-h: Muestra este mensaje de ayuda"
}

abortar_creacion() {
    destruir_entorno > /dev/null 2>&1
    error "$1"
}

mostrar_informacion_contenedor() {
    _nombre="$1"
    _imagen=$(docker inspect --format='{{.Config.Image}}' "$_nombre")
    _background=$(docker inspect
    --format='{{.Config.Labels.background}}' "$_nombre")

    echo "Nombre del contenedor: $_nombre"
```

```
    echo "Imagen: $_imagen"
    echo "Background: $_background"
    for _red in $(docker inspect --format='{{range $key, $value :=
    .NetworkSettings.Networks}}{{ println $key}}{{end}}' $_nombre | sed
    '/^$/d'); do
        _ip=$(docker container inspect -f '{{ (index
        .NetworkSettings.Networks "'"$_red"'").IPAddress }}' $_nombre)
        echo "Red: $_red IP: $_ip"
    done
    echo ""
}

destruir_entorno() {
    if [ "$(docker ps -aq -f name="$nombre_practica")" ]; then
        echo "Eliminando los contenedores y redes de
        $nombre_practica..."
        docker container rm -f $(docker ps -a --filter
        name="$nombre_practica" --format '{{.Names}}') > /dev/null ||
        error "No se ha podido borrar el entorno $nombre_practica"
        if [ "$(docker network ls -q --filter name="$nombre_practica")"
        ]; then
            docker network rm $(docker network ls -q --filter
            name="$nombre_practica") > /dev/null || error "No se ha
            podido borrar el entorno $nombre_practica"
        fi
        echo "Contenedores y redes eliminados exitosamente"
    else
        echo "No existen contenedores asociados a $nombre_practica"
    fi
}

lanzar_entorno() {
    if [ "$(docker ps -aq -f name="$nombre_practica")" ]; then
        echo "Lanzando de nuevo los contenedores de $nombre_practica..."
        listaContenedoresTerminal=""
        for nombre in $(docker ps -a --filter name="$nombre_practica"
        --format '{{.Names}}'); do
            docker container start "$nombre" > /dev/null || error "No se
            ha podido lanzar el contenedor $nombre"

            if [ "$(docker inspect -f '{{.Config.Labels.background}}'
            "$nombre" )" = "false" ]; then
                if [ -z "$listaContenedoresTerminal" ]; then
                    listaContenedoresTerminal="$nombre"
                else
                    listaContenedoresTermi-
                    nal="$listaContenedoresTerminal
                    $nombre"
```

```
                fi
            fi

            mostrar_informacion_contenedor "$nombre"
        done

        crear_terminal $listaContenedoresTerminal

        echo "Contenedores lanzados exitosamente"
    else
        echo "No existen contenedores asociados a $nombre_practica"
    fi
}

parar_entorno() {
    if [ "$(docker ps -aq -f name="$nombre_practica")" ]; then
        echo "Deteniendo los contenedores de $nombre_practica..."
        docker container stop $(docker ps -a --filter
        name="$nombre_practica" --format '{{.Names}}') > /dev/null ||
        error "No se ha podido parar el entorno $nombre_practica"
        echo "Contenedores detenidos exitosamente"
    else
        echo "No existen contenedores asociados a $nombre_practica"
    fi
}

inspeccionar_entorno() {
    if [ "$(docker ps -aq -f name="$nombre_practica")" ]; then
        echo "Información de los contenedores de $nombre_practica"
        docker container inspect $(docker ps -a --filter
        name="$nombre_practica" --format '{{.Names}}') || error "No se
        ha podido parar el entorno $nombre_practica"
    else
        echo "No existen contenedores asociados a $nombre_practica"
    fi
}

crear_entorno() {
    nombre_practica=$(yq e '.nombre_practica' "$fichero")
    if [ "$nombre_practica" = "null" ]; then
        error "Se debe especificar el campo nombre_practica en el
        fichero"
    fi

    # Si ya existen contenedores asociados a la práctica, se destruyen
    previamente
    if [ "$(docker ps -aq -f name="$nombre_practica")" ]; then
```

```
        destruir_entorno
        echo "" # Se imprime esta línea vacía para dejar más bonita la
        salida
    fi

    numero_contenedores=$(yq e '.contenedores | length' "$fichero")
    if [ "$numero_contenedores" = 0 ]; then
        error "Se deben especificar los contenedores deseados en el
        campo contenedores del fichero"
    fi

    # Primero se crean todos los contenedores
    i=0
    listaContenedoresTerminal=""
    while [ $i -lt "$numero_contenedores" ]
    do
        nombre_contenedor=$(yq e '.contenedores['"$i"'].nombre'
        "$fichero")
        if [ "$nombre_contenedor" = "null" ]; then
            abortar_creacion "Se debe especificar el campo nombre en
            contenedores[$i]"
        fi

        # Se concatena el nombre de la practica con el nombre del
        contenedor para poder hacer operaciones asociadas a entornos de
        prácticas
        nombre="${nombre_practica}_${nombre_contenedor}"

        imagen=$(yq e '.contenedores['"$i"'].imagen' "$fichero")
        if [ "$imagen" = "null" ]; then
            abortar_creacion "Se debe especificar el campo imagen en
            contenedores[$i]"
        fi

        background=$(yq e '.contenedores['"$i"'].background' "$fichero")
        if [ "$background" != true ]; then
            background=false
            if [ -z "$listaContenedoresTerminal" ]; then
                listaContenedoresTerminal="$nombre"
            else
                listaContenedoresTerminal="$listaContenedoresTerminal
                $nombre"
            fi
        fi

        # Se crea la cookie de X11 para usar aplicaciones con GUI
        crear_cookie || abortar_creacion "Error al crear la cookie para
        $nombre"
```

```
        # Se crea el contenedor y se le desconecta del adaptador bridge
        run_container > /dev/null || abortar_creacion "Error en la
        creación del contenedor $nombre"
        docker network disconnect bridge "$nombre" || abortar_creacion
        "Error en la creación del contenedor $nombre"

        numero_redes=$(yq e '.contenedores['"$i"'].redes | length'
        "$fichero")
        if [ "$numero_redes" = 0 ]; then
            abortar_creacion "Se debe especificar el campo redes en
            contenedores[$i]"
        fi

        j=0
        while [ $j -lt "$numero_redes" ]
        do
            red_fichero=$(yq e
            '.contenedores['"$i"'].redes['"$j"'].nombre' "$fichero")
            if [ "$red_fichero" = "null" ]; then
                abortar_creacion "Se debe especificar el campo nombre en
                contenedores[$i].redes[$j]"
            fi
            red="${nombre_practica}_red_${red_fichero}"

            ip=$(yq e '.contenedores['"$i"'].redes['"$j"'].ip'
            "$fichero")
            if [ "$ip" = "null" ]; then
                subnet_option=""
                ip_option=""
            else
                # Comprueba que la IP introducida esté en formato CIDR
                válido
                if [ "$(ipcalc -s -n "$ip" )" ]; then
                    direccion_red="$(ipcalc -n "$ip" | sed
                    's/NETWORK=//')"
                    prefijo="$(ipcalc -p "$ip" | sed 's/PREFIX=//')"
                    subnet_option="--subnet=$direccion_red/$prefijo"

                    ip=$(echo "$ip" | sed 's/\/[0-9]*//')
                    ip_option="--ip $ip"
                else
                    abortar_creacion "Debe proporcionar una IP válida en
                    .contenedores[$i].redes[$j]"
                fi
            fi
```

```
            # Solo se crea la red si no existe previamente
            docker network inspect "$red" > /dev/null 2>&1 || docker
            network create $subnet_option "$red" > /dev/null || \
              abortar_creacion "No se ha podido crear la red $red"

            docker network connect $ip_option "$red" "$nombre" ||
            abortar_creacion "No se ha podido conectar el contenedor
            $nombre"

            j=$((j+1))
        done

        mostrar_informacion_contenedor "$nombre"

        i=$((i+1))
    done

    # Si todos los contenedores se han creado exitosamente, se abren las
    terminales
    if [ -n "$listaContenedoresTerminal" ]; then
        crear_terminal $listaContenedoresTerminal
    fi

    echo "Contenedores creados exitosamente"
}

if [ ! "$(which yq)" ]; then
    error "yq debe estar instalado en el sistema para utilizar este
    script"
fi

if [ ! "$(which xauth)" ]; then
    error "xauth debe estar instalado en el sistema para utilizar este
    script"
fi

if [ ! "$(which docker)" ]; then
    error "docker debe estar instalado en el sistema para utilizar este
    script"
fi

# Controla que el comando se ejecute solo con un flag
opcion=false

while getopts ":c:l:r:p:i:h" o; do
    case "${o}" in
        c)
```

```
            $opcion && error "Solo se puede especificar una opción
            <-c|-l|-r|-p|-i|-h>"
            fichero=${OPTARG}

            if [ ! -f "$fichero" ]; then
                error "El fichero $fichero no existe"
            fi
            comando="crear"
            opcion=true
            ;;
        l)
            $opcion && error "Solo se puede especificar una opción
            <-c|-l|-r|-p|-i|-h>"
            nombre_practica=${OPTARG}

            if [ -z "$nombre_practica" ]; then
                error "Se debe proporcionar el valor nombre_practica"
            fi

            comando="lanzar"
            opcion=true
            ;;
        r)
            $opcion && error "Solo se puede especificar una opción
            <-c|-l|-r|-p|-i|-h>"
            nombre_practica=${OPTARG}

            if [ -z "$nombre_practica" ]; then
                error "Se debe proporcionar el valor nombre_practica"
            fi

            comando="destruir"
            opcion=true
            ;;
        p)
            $opcion && error "Solo se puede especificar una opción
            <-c|-l|-r|-p|-i|-h>"
            nombre_practica=${OPTARG}

            if [ -z "$nombre_practica" ]; then
                error "Se debe proporcionar el valor nombre_practica"
            fi

            comando="parar"
            opcion=true
            ;;
        i)
```

```
            $opcion && error "Solo se puede especificar una opción
            <-c|-l|-r|-p|-i|-h|>"
            nombre_practica=${OPTARG}

            if [ -z "$nombre_practica" ]; then
                error "Se debe proporcionar el valor nombre_practica"
            fi

            comando="inspeccionar"
            opcion=true
            ;;
        h)
            $opcion && error "Solo se puede especificar una opción
            <-c|-l|-r|-p|-i|-h>"

            uso
            exit 0
            ;;
        *)
            echo "Opción ${o} no válida"
            uso
            exit 1
            ;;
    esac
done
shift $((OPTIND-1))

# Quitamos la extensión en caso de que exista
if [ -n "$nombre_practica" ]; then
    nombre_practica=${nombre_practica%.*}
fi

case "$comando" in
    crear)
        crear_entorno && exit
        ;;
    lanzar)
        lanzar_entorno && exit
        ;;
    destruir)
        destruir_entorno && exit
        ;;
    parar)
        parar_entorno && exit
        ;;
    inspeccionar)
        inspeccionar_entorno && exit
```

```
        ;;
    *)
        uso && exit 1
        ;;
esac
```

Apéndice B

Manual de uso de la herramienta de despliegue de contenedores

Para llevar a cabo una práctica mediante la herramienta de despliegue de contenedores, es necesario crear en primer lugar un archivo YAML en el que se definan los contenedores requeridos. A continuación se muestra un ejemplo de la sintaxis que debe tener este fichero:

```
nombre_practica: practica0
contenedores:
  - nombre: host1
    imagen: rys
    redes:
    - nombre: 1
  - nombre: host2
    imagen: rys
    redes:
    - nombre: 1
  - nombre: router1
    imagen: rys
    redes:
    - nombre: 1
    - nombre: 2
  - nombre: router2
    imagen: rys
    redes:
    - nombre: 2
    - nombre: 3
  - nombre: servidor
    imagen: rys
    background: true
    redes:
    - nombre: 3
      ip: 192.168.1.10/24
```

Todos los campos son obligatorios, salvo `background` e `ip`. Si no se proporcionan, por defecto se creará una ventana de terminal para el contenedor y la asignación de la dirección IP se hará de forma automática. En el caso de definir la dirección IP, deberá estar escrita en formato CIDR.

Una vez se haya creado el archivo YAML, se debe ejecutar el comando `./LabsOnContainers -c <nombreArchivo.yaml>`, que creará los contenedores y redes del fichero, y lanzará las terminales para poder interactuar con los contenedores.

Si en algún momento se desean detener los contenedores pero guardando su estado, por ejemplo para continuar con el desarrollo de la práctica más tarde, se puede ejecutar `./LabsOnContainers -p <nombre_practica>`, especificando como nombre de la práctica el mismo que se haya establecido en el fichero YAML durante la creación del entorno. Cuando se desee continuar con la práctica, se puede ejecutar `./LabsOnContainers -l <nombre_practica>` para reactivar los contenedores.

Durante la realización se puede ejecutar en cualquier momento el comando `./LabsOnContainers -i <nombre_practica>`, que mostrará información de bajo nivel acerca de los contenedores de la práctica, como por ejemplo los volúmenes montados, las redes a las que están conectados o sus variables de entorno.

Finalmente, una vez se haya terminado de realizar la práctica, se debe ejecutar `./LabsOnContainers -r <nombre_practica>`, que eliminará los contenedores y redes asociados.

Apéndice C

Resultado de Docker Bench for Security

```
Section A - Check results

[INFO] 1 - Host Configuration
[INFO] 1.1 - Linux Hosts Specific Configuration
[WARN] 1.1.1 - Ensure a separate partition for containers has been
created (Automated)
[INFO] 1.1.2 - Ensure only trusted users are allowed to control Docker
daemon (Automated)
[INFO]       * Users:
[WARN] 1.1.3 - Ensure auditing is configured for the Docker daemon
(Automated)
[WARN] 1.1.4 - Ensure auditing is configured for Docker files and
directories -/run/containerd (Automated)
[WARN] 1.1.5 - Ensure auditing is configured for Docker files and
directories - /var/lib/docker (Automated)
[WARN] 1.1.6 - Ensure auditing is configured for Docker files and
directories - /etc/docker (Automated)
[INFO] 1.1.7 - Ensure auditing is configured for Docker files and
directories - docker.service (Automated)
[INFO]       * File not found
[INFO] 1.1.8 - Ensure auditing is configured for Docker files and
directories - containerd.sock (Automated)
[INFO]       * File not found
[INFO] 1.1.9 - Ensure auditing is configured for Docker files and
directories - docker.socket (Automated)
[INFO]       * File not found
[INFO] 1.1.10 - Ensure auditing is configured for Docker files and
directories - /etc/default/docker (Automated)
[INFO]       * File not found
[INFO] 1.1.11 - Ensure auditing is configured for Dockerfiles and
directories - /etc/docker/daemon.json (Automated)
```

```
[INFO]        * File not found
[WARN] 1.1.12 - 1.1.12 Ensure auditing is configured for Dockerfiles and
directories - /etc/containerd/config.toml (Automated)
[INFO] 1.1.13 - Ensure auditing is configured for Docker files and
directories - /etc/sysconfig/docker (Automated)
[INFO]        * File not found
[WARN] 1.1.14 - Ensure auditing is configured for Docker files and
directories - /usr/bin/containerd (Automated)
[WARN] 1.1.15 - Ensure auditing is configured for Docker files and
directories - /usr/bin/containerd-shim (Automated)
[WARN] 1.1.16 - Ensure auditing is configured for Docker files and
directories - /usr/bin/containerd-shim-runc-v1 (Automated)
[WARN] 1.1.17 - Ensure auditing is configured for Docker files and
directories - /usr/bin/containerd-shim-runc-v2 (Automated)
[WARN] 1.1.18 - Ensure auditing is configured for Docker files and
directories - /usr/bin/runc (Automated)
[INFO] 1.2 - General Configuration
[NOTE] 1.2.1 - Ensure the container host has been Hardened (Manual)
[PASS] 1.2.2 - Ensure that the version of Docker is up to date (Manual)
[INFO]        * Using 20.10.15, verify is it up to date as deemed
necessary

[INFO] 2 - Docker daemon configuration
[NOTE] 2.1 - Run the Docker daemon as a non-root user, if possible
(Manual)
[WARN] 2.2 - Ensure network traffic is restricted between containers on
the default bridge (Scored)
[PASS] 2.3 - Ensure the logging level is set to 'info' (Scored)
[PASS] 2.4 - Ensure Docker is allowed to make changes to iptables
(Scored)
[PASS] 2.5 - Ensure insecure registries are not used (Scored)
[PASS] 2.6 - Ensure aufs storage driver is not used (Scored)
[INFO] 2.7 - Ensure TLS authentication for Docker daemon is configured
(Scored)
[INFO]      * Docker daemon not listening on TCP
[INFO] 2.8 - Ensure the default ulimit is configured appropriately
(Manual)
[INFO]      * Default ulimit doesn't appear to be set
[WARN] 2.9 - Enable user namespace support (Scored)
[PASS] 2.10 - Ensure the default cgroup usage has been confirmed
(Scored)
[PASS] 2.11 - Ensure base device size is not changed until needed
(Scored)
[WARN] 2.12 - Ensure that authorization for Docker client commands is
enabled (Scored)
[WARN] 2.13 - Ensure centralized and remote logging is configured
(Scored)
```

```
[WARN] 2.14 - Ensure containers are restricted from acquiring new
privileges (Scored)
[WARN] 2.15 - Ensure live restore is enabled (Scored)
[WARN] 2.16 - Ensure Userland Proxy is Disabled (Scored)
[PASS] 2.17 - Ensure that a daemon-wide custom seccomp profile is
applied if appropriate (Manual)
[INFO] Ensure that experimental features are not implemented in
production (Scored) (Deprecated)

[INFO] 3 - Docker daemon configuration files
[INFO] 3.1 - Ensure that the docker.service file ownership is set to
root:root (Automated)
[INFO]      * File not found
[INFO] 3.2 - Ensure that docker.service file permissions are
appropriately set (Automated)
[INFO]      * File not found
[INFO] 3.3 - Ensure that docker.socket file ownership is set to
root:root (Automated)
[INFO]      * File not found
[INFO] 3.4 - Ensure that docker.socket file permissions are set to 644
or more restrictive (Automated)
[INFO]      * File not found
[PASS] 3.5 - Ensure that the /etc/docker directory ownership is set to
root:root (Automated)
[PASS] 3.6 - Ensure that /etc/docker directory permissions are set to
755 or more restrictively (Automated)
[INFO] 3.7 - Ensure that registry certificate file ownership is set to
root:root (Automated)
[INFO]      * Directory not found
[INFO] 3.8 - Ensure that registry certificate file permissions are set
to 444 or more restrictively (Automated)
[INFO]      * Directory not found
[INFO] 3.9 - Ensure that TLS CA certificate file ownership is set to
root:root (Automated)
[INFO]      * No TLS CA certificate found
[INFO] 3.10 - Ensure that TLS CA certificate file permissions are set to
444 or more restrictively (Automated)
[INFO]       * No TLS CA certificate found
[INFO] 3.11 - Ensure that Docker server certificate file ownership is
set to root:root (Automated)
[INFO]       * No TLS Server certificate found
[INFO] 3.12 - Ensure that the Docker server certificate file permissions
are set to 444 or more restrictively (Automated)
[INFO]       * No TLS Server certificate found
[INFO] 3.13 - Ensure that the Docker server certificate key file
ownership is set to root:root (Automated)
[INFO]       * No TLS Key found
```

```
[INFO] 3.14 - Ensure that the Docker server certificate key file
permissions are set to 400 (Automated)
[INFO]       * No TLS Key found
[PASS] 3.15 - Ensure that the Docker socket file ownership is set to
root:docker (Automated)
[PASS] 3.16 - Ensure that the Docker socket file permissions are set to
660 or more restrictively (Automated)
[INFO] 3.17 - Ensure that the daemon.json file ownership is set to
root:root (Automated)
[INFO]       * File not found
[INFO] 3.18 - Ensure that daemon.json file permissions are set to 644 or
more restrictive (Automated)
[INFO]       * File not found
[INFO] 3.19 - Ensure that the /etc/default/docker file ownership is set
to root:root (Automated)
[INFO]       * File not found
[INFO] 3.20 - Ensure that the /etc/sysconfig/docker file permissions are
set to 644 or more restrictively (Automated)
[INFO]       * File not found
[INFO] 3.21 - Ensure that the /etc/sysconfig/docker file ownership is
set to root:root (Automated)
[INFO]       * File not found
[INFO] 3.22 - Ensure that the /etc/default/docker file permissions are
set to 644 or more restrictively (Automated)
[INFO]       * File not found
[INFO] 3.23 - Ensure that the Containerd socket file ownership is set to
root:root (Automated)
[INFO]       * File not found
[INFO] 3.24 - Ensure that the Containerd socket file permissions are set
to 660 or more restrictively (Automated)
[INFO]       * File not found

[INFO] 4 - Container Images and Build File
[PASS] 4.1 - Ensure that a user for the container has been created
(Automated)
[NOTE] 4.2 - Ensure that containers use only trusted base images
(Manual)
[NOTE] 4.3 - Ensure that unnecessary packages are not installed in the
container (Manual)
[NOTE] 4.4 - Ensure images are scanned and rebuilt to include security
patches (Manual)
[WARN] 4.5 - Ensure Content trust for Docker is Enabled (Automated)
[WARN] 4.6 - Ensure that HEALTHCHECK instructions have been added to
container images (Automated)
[WARN]      * No Healthcheck found: [rys:latest]
[INFO] 4.7 - Ensure update instructions are not used alone in the
Dockerfile (Manual)
```

```
[INFO]      * Update instruction found: [rys:latest]
[NOTE] 4.8 - Ensure setuid and setgid permissions are removed (Manual)
[PASS] 4.9 - Ensure that COPY is used instead of ADD in Dockerfiles
(Manual)
[NOTE] 4.10 - Ensure secrets are not stored in Dockerfiles (Manual)
[NOTE] 4.11 - Ensure only verified packages are installed (Manual)
[NOTE] 4.12 - Ensure all signed artifacts are validated (Manual)

[INFO] 5 - Container Runtime
[WARN] 5.1 - Ensure that, if applicable, an AppArmor Profile is enabled
(Automated)
[WARN]      * No AppArmorProfile Found: practica0_router2
[WARN]      * No AppArmorProfile Found: practica0_host2
[WARN]      * No AppArmorProfile Found: practica0_router1
[WARN]      * No AppArmorProfile Found: practica0_host1
[WARN]      * No AppArmorProfile Found: practica0_servidor
[WARN] 5.2 - Ensure that, if applicable, SELinux security options are
set (Automated)
[WARN]      * No SecurityOptions Found: practica0_router2
[WARN]      * No SecurityOptions Found: practica0_host2
[WARN]      * No SecurityOptions Found: practica0_router1
[WARN]      * No SecurityOptions Found: practica0_host1
[WARN]      * No SecurityOptions Found: practica0_servidor
[WARN] 5.3 - Ensure that Linux kernel capabilities are restricted within
containers (Automated)
[WARN]      * Capabilities added: CapAdd=[NET_ADMIN] to
practica0_router2
[WARN]      * Capabilities added: CapAdd=[NET_ADMIN] to practica0_host2
[WARN]      * Capabilities added: CapAdd=[NET_ADMIN] to
practica0_router1
[WARN]      * Capabilities added: CapAdd=[NET_ADMIN] to practica0_host1
[WARN]      * Capabilities added: CapAdd=[NET_ADMIN] to
practica0_servidor
[PASS] 5.4 - Ensure that privileged containers are not used (Automated)
[PASS] 5.5 - Ensure sensitive host system directories are not mounted on
containers (Automated)
[PASS] 5.6 - Ensure sshd is not run within containers (Automated)
[PASS] 5.7 - Ensure privileged ports are not mapped within containers
(Automated)
[PASS] 5.8 - Ensure that only needed ports are open on the container
(Manual)
[PASS] 5.9 - Ensure that the host's network namespace is not shared
(Automated)
[WARN] 5.10 - Ensure that the memory usage for containers is limited
(Automated)
[WARN]       * Container running without memory restrictions:
practica0_router2
```

```
[WARN]       * Container running without memory restrictions:
practica0_host2
[WARN]       * Container running without memory restrictions:
practica0_router1
[WARN]       * Container running without memory restrictions:
practica0_host1
[WARN]       * Container running without memory restrictions:
practica0_servidor
[WARN] 5.11 - Ensure that CPU priority is set appropriately on
containers (Automated)
[WARN]       * Container running without CPU restrictions:
practica0_router2
[WARN]       * Container running without CPU restrictions:
practica0_host2
[WARN]       * Container running without CPU restrictions:
practica0_router1
[WARN]       * Container running without CPU restrictions:
practica0_host1
[WARN]       * Container running without CPU restrictions:
practica0_servidor
[WARN] 5.12 - Ensure that the container's root filesystem is mounted as
read only (Automated)
[WARN]       * Container running with root FS mounted R/W:
practica0_router2
[WARN]       * Container running with root FS mounted R/W:
practica0_host2
[WARN]       * Container running with root FS mounted R/W:
practica0_router1
[WARN]       * Container running with root FS mounted R/W:
practica0_host1
[WARN]       * Container running with root FS mounted R/W:
practica0_servidor
[PASS] 5.13 - Ensure that incoming container traffic is bound to a
specific host interface (Automated)
[WARN] 5.14 - Ensure that the 'on-failure' container restart policy is
set to '5' (Automated)
[WARN]       * MaximumRetryCount is not set to 5: practica0_router2
[WARN]       * MaximumRetryCount is not set to 5: practica0_host2
[WARN]       * MaximumRetryCount is not set to 5: practica0_router1
[WARN]       * MaximumRetryCount is not set to 5: practica0_host1
[WARN]       * MaximumRetryCount is not set to 5: practica0_servidor
[PASS] 5.15 - Ensure that the host's process namespace is not shared
(Automated)
[PASS] 5.16 - Ensure that the host's IPC namespace is not shared
(Automated)
[PASS] 5.17 - Ensure that host devices are not directly exposed to
containers (Manual)
```

```
[INFO] 5.18 - Ensure that the default ulimit is overwritten at runtime
if needed (Manual)
[INFO]       * Container no default ulimit override: practica0_router2
[INFO]       * Container no default ulimit override: practica0_host2
[INFO]       * Container no default ulimit override: practica0_router1
[INFO]       * Container no default ulimit override: practica0_host1
[INFO]       * Container no default ulimit override: practica0_servidor
[PASS] 5.19 - Ensure mount propagation mode is not set to shared
(Automated)
[PASS] 5.20 - Ensure that the host's UTS namespace is not shared
(Automated)
[PASS] 5.21 - Ensure the default seccomp profile is not Disabled
(Automated)
[NOTE] 5.22 - Ensure that docker exec commands are not used with the
privileged option (Automated)
[NOTE] 5.23 - Ensure that docker exec commands are not used with the
user=root option (Manual)
[PASS] 5.24 - Ensure that cgroup usage is confirmed (Automated)
[WARN] 5.25 - Ensure that the container is restricted from acquiring
additional privileges (Automated)
[WARN]       * Privileges not restricted: practica0_router2
[WARN]       * Privileges not restricted: practica0_host2
[WARN]       * Privileges not restricted: practica0_router1
[WARN]       * Privileges not restricted: practica0_host1
[WARN]       * Privileges not restricted: practica0_servidor
[WARN] 5.26 - Ensure that container health is checked at runtime
(Automated)
[WARN]       * Health check not set: practica0_router2
[WARN]       * Health check not set: practica0_host2
[WARN]       * Health check not set: practica0_router1
[WARN]       * Health check not set: practica0_host1
[WARN]       * Health check not set: practica0_servidor
[INFO] 5.27 - Ensure that Docker commands always make use of the latest
version of their image (Manual)
[WARN] 5.28 - Ensure that the PIDs cgroup limit is used (Automated)
[WARN]       * PIDs limit not set: practica0_router2
[WARN]       * PIDs limit not set: practica0_host2
[WARN]       * PIDs limit not set: practica0_router1
[WARN]       * PIDs limit not set: practica0_host1
[WARN]       * PIDs limit not set: practica0_servidor
[PASS] 5.29 - Ensure that Docker's default bridge 'docker0' is not used
(Manual)
[PASS] 5.30 - Ensure that the host's user namespaces are not shared
(Automated)
[PASS] 5.31 - Ensure that the Docker socket is not mounted inside any
containers (Automated)
```

```
[INFO] 6 - Docker Security Operations
[INFO] 6.1 - Ensure that image sprawl is avoided (Manual)
[INFO]      * There are currently: 1 images
[INFO] 6.2 - Ensure that container sprawl is avoided (Manual)
[INFO]      * There are currently a total of 5 containers, with 5 of
them currently running

[INFO] 7 - Docker Swarm Configuration
[PASS] 7.1 - Ensure swarm mode is not Enabled, if not needed (Automated)
[PASS] 7.2 - Ensure that the minimum number of manager nodes have been
created in a swarm (Automated) (Swarm mode not enabled)
[PASS] 7.3 - Ensure that swarm services are bound to a specific host
interface (Automated) (Swarm mode not enabled)
[PASS] 7.4 - Ensure that all Docker swarm overlay networks are encrypted
(Automated)
[PASS] 7.5 - Ensure that Docker's secret management commands are used
for managing secrets in a swarm cluster (Manual) (Swarm mode not
enabled)
[PASS] 7.6 - Ensure that swarm manager is run in auto-lock mode
(Automated) (Swarm mode not enabled)
[PASS] 7.7 - Ensure that the swarm manager auto-lock key is rotated
periodically (Manual) (Swarm mode not enabled)
[PASS] 7.8 - Ensure that node certificates are rotated as appropriate
(Manual) (Swarm mode not enabled)
[PASS] 7.9 - Ensure that CA certificates are rotated as appropriate
(Manual) (Swarm mode not enabled)
[PASS] 7.10 - Ensure that management plane traffic is separated from
data plane traffic (Manual) (Swarm mode not enabled)


Section C - Score

[INFO] Checks: 117
[INFO] Score: 1
```

Mario Román Dono
2021 - 2022
Ult. actualización 30 de mayo de 2022
LaTeX lic. LPPL & powered by TEFLON CC-ZERO