Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Herramienta para la construcción automática de distribuciones Linux basadas en el proyecto Linux From Scratch

Autor: Gonzalo Gómez Gracia

Tutor: Antonio José Estepa Alonso

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Herramienta para la construcción automática de distribuciones Linux basadas en el proyecto Linux From Scratch

Autor:
Gonzalo Gómez Gracia

Tutor:
Antonio José Estepa Alonso
Doctor Ingeniero de Telecomunicación y Profesor Titular de la Universidad de Sevilla

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Grado: Herramienta para la construcción automática de distribuciones Linux basadas en el proyecto Linux From Scratch

Autor: Gonzalo Gómez Gracia
Tutor: Antonio José Estepa Alonso

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

A Rebeca

Agradecimientos

Estudiar una ingeniería es uno de los desafíos más grandes a los que me he tenido que enfrentar hasta ahora. No sólo por la dificultad en sí misma que conlleva estudiar algo de este tamaño, sino también por la necesidad de esfuerzo, constancia y dedicación que requiere.

Ahora que estoy a un paso de conseguir mi objetivo, miro hacia atrás en el tiempo y recuerdo que ser ingeniero no fue mi primer sueño cuando era pequeño. Mi primer sueño fue emular a mi padre, un luchador, una persona íntegra, leal y consciente de que sin esfuerzo nada es posible. El paso de los años me demostró que ese objetivo era difícil de conseguir, así que cambié de plan. Descubrí que mis otras aficiones, la ciencia, el estudio, las matemáticas, la física, la electrónica, las redes, los ordenadores, la robótica, la automatización y la programación entre muchas otras, podían unirse en un todo que despertaba mi curiosidad hasta más allá de donde mi capacidad de comprensión lograba entender, y me decidí a estudiar el grado en Ingeniería de las Tecnologías de Telecomunicación que estoy a punto de terminar.

Ha sido un camino muy largo y muy duro no sólo en el aspecto académico, también en el personal. Pero todo eso queda atrás como el resultado de un proceso de aprendizaje, como el camino que me ha hecho convertirme en lo que hoy en día soy. Es mi deseo que este camino haya servido para parecerme un poco más a él, mi referente. A mi padre.

Este camino, a pesar de las dificultades, ha estado plagado también de buenos momentos y de personas maravillosas. Algunas de ellas ya no están entre nosotros, pero su recuerdo quedará para siempre en mi memoria. Otras me acompañan día a día en esta lucha constante por realizarme como persona, y a todas ellas les doy las gracias.

A Rebeca, mi futura esposa y la luchadora que ha demostrado estar lo suficientemente loca por mí como para no salir corriendo. A mis padres, mi hermano Gorka, mis abuelos y el resto de mi familia, por nunca dejarme sólo. A mi madre, por ser mi ángel particular. A Chej, al que quiero como si fuera un hermano más. A Gregorio, un amigo indescriptible. A David y Fabi, compañeros de esta aventura. A Paco Gamero, que se merece lo mejor en la vida. A los doctores, enfermeros y auxiliares de la Unidad de Hematología del Hospital Virgen de Valme de Sevilla, y a todas y cada una de las personas que han formado parte de una manera u otra de todo este proceso, muchísimas gracias.

Gonzalo Gómez Gracia
Futuro graduado en Ingeniería de las Tecnologías de Telecomunicación

Sevilla, 2018

Resumen

Actualmente, el desarrollo y uso extendido de las tecnologías de virtualización, contenedores, orquestación, sistemas embebidos de ámbito general y la computación en la nube, así como la expansión del Internet de las Cosas, Internet of Things (IoT) por sus siglas en inglés, han hecho que el uso de GNU/Linux crezca en los últimos años. Sin embargo, algunas de estas aplicaciones necesitan de versiones de Linux no estándar y especializadas.

El proyecto *Linux from Scratch* reúne todas las instrucciones necesarias para construir un sistema operativo Linux desde cero. Dado que este proceso, más allá del aprendizaje sobre el funcionamiento interno de Linux que puede obtenerse tras una ejecución manual, resulta repetitivo y largo, el propio proyecto *Linux from Scratch* proporciona una herramienta de automatización que facilite esta tarea, llamada *jhalfs*.

El presente texto analiza las características del proyecto *Linux from Scratch*, la capacidad de personalización ofrecida al usuario final y la herramienta *jhalfs* para justificar y motivar el diseño de una nueva herramienta alternativa llamada *Linux from Scratch Builder*.

Esta nueva herramienta posee una estructura simple, requiere de la mínima intervención por parte del usuario y es capaz de construir las versiones más recientes de los libros *Linux from Scratch* y *Beyond Linux from Scratch*. Además, *LFSBuilder* proporciona opciones de personalización adicionales para adecuar el sistema operativo que resulta de su ejecución a las necesidades del usuario final, y ha obtenido mejores prestaciones en la comparativa de rendimiento realizada respecto a la herramienta oficial.

Abstract

Nowadays the constant usage and development of virtualization, containerized, *IoT*, and embedded technologies have bootstrapped GNU/Linux daily usage from the past few years. However, many of those use case need a specialized version of this operating system.

The *Linux from Scratch* project provides a step-by-step instructions for building your own customized Linux-based distribution from source code. Apart from the learning experience and because this is a long repetitive process, the *Linux from Scratch* project developers provide users with an automation tool called *jhalfs*.

This text pretends to analize both the *Linux from Scratch* project and the *jhalfs* tool, and its customization options to justify the development of a new alternative tool called *Linux from Scratch Builder*.

This new tool has a simpler structure, requires a little user intervention to do its job and can build latest *Linux from Scratch* and *Beyond Linux from Scratch* versions. Furthermore, it adds a customization system so the operating system built as a result can fit the system requirements. It also showed better values on performance tests in comparison with the official tool.

Índice Abreviado

Resumen V
Abstract VII
Índice Abreviado IX

1 Introducción y objetivos **1**
1.1 Personalizar un sistema operativo ya existente 2
1.2 Crear un sistema operativo personalizado desde cero 3

2 Linux from Scratch **9**
2.1 Linux from Scratch 12
2.2 Beyond Linux from Scratch 26
2.3 Automated Linux from Scratch 27

3 LFSBuilder **33**
3.1 Diseño de la herramienta 34
3.2 Dependencias 63
3.3 jhalfs vs. LFSBuilder 64

4 Desarrollo y pruebas **71**

5 Conclusiones y líneas de avance **75**

Apéndice A Mapa de Karnaugh **79**

Índice de Figuras 85
Índice de Tablas 87
Índice de Códigos 89
Bibliografía 91
Índice alfabético 93
Glosario 93

Índice

Resumen V
Abstract VII
Índice Abreviado IX

1 Introducción y objetivos **1**
1.1 Personalizar un sistema operativo ya existente 2
1.2 Crear un sistema operativo personalizado desde cero 3
1.2.1 Aprendizaje 3
1.2.2 Necesidad para desarrolladores y administradores de sistemas 4

2 Linux from Scratch **9**
2.1 Linux from Scratch 12
2.1.1 Preparación del entorno y requisitos previos 12
2.1.2 Toolchain 17
2.1.3 Sistema 18
2.1.4 Configuración y gestión de arranque 19
2.1.5 El interés de automatizar el proceso 25
2.2 Beyond Linux from Scratch 26
2.2.1 El interés de automatizar el proceso 26
2.3 Automated Linux from Scratch 27

3 LFSBuilder **33**
3.1 Diseño de la herramienta 34
3.1.1 Estructura y directorio principal 35
actions.py 35
builders.py 36
cli.py 37
components.py 38
config.py 39
downloader.py 42
lfsbuilder.py 43
printer.py 43
tools.py 43
xmlparser.py 44
3.1.2 Recetas 46
Definir un nuevo constructor 50
Definir un nuevo componente 51
3.1.3 Plantillas 51
script.tpl 51
setenv.tpl 52
3.1.4 Tests unitarios 53

3.1.5 Interfaz de línea de comandos 54
3.1.6 De los datos al *script* 57
3.2 Dependencias 63
3.3 jhalfs vs. LFSBuilder 64
3.3.1 Tecnologías 64
3.3.2 Dependencias 64
3.3.3 Rendimiento 65
3.3.4 Otras características 69

4 Desarrollo y pruebas 71

5 Conclusiones y líneas de avance 75

Apéndice A Mapa de Karnaugh 79

Índice de Figuras 85
Índice de Tablas 87
Índice de Códigos 89
Bibliografía 91
Índice alfabético 93
Glosario 93

1 Introducción y objetivos

The thing with Linux is that the developers themselves are actually customers too: that has always been an important part of Linux.

LINUS TORVALDS

GNU/Linux, más comúnmente conocido como *Linux*, es un sistema operativo gratuito, de código abierto, multiplataforma, multiusuario y multitarea. Estas características, unidas a su buen rendimiento, compatibilidad con el *hardware* antiguo, su configuración versátil y la cantidad de aplicaciones gratuitas disponibles y mantenidas por la propia comunidad de usuarios, hacen de GNU/Linux un sistema operativo muy completo y personalizable.

Históricamente, GNU/Linux deriva del sistema operativo conocido como UNIX*, nace de la unión del sistema operativo GNU y el kernel Linux, y forma la familia de sistemas operativos Linux más utilizada y extensa[11]. Existen multitud de distribuciones basadas en GNU/Linux[2], y otras tantas que no cumplen el estándar GNU o que implementan una parte del mismo[3], que conforman una amplia gama de opciones para el público general, por ejemplo en equipos de escritorio o móviles, pero no tan amplia para el uso específico. Algunas de estas distribuciones son Debian, CentOS, Arch Linux, PureOS o Fedora, entre otras. Otros sistemas operativos derivados de UNIX* son los de tipo BSD, de *Berkeley Software Distribution*. Ejemplos de esta familia de sistemas operativos son Mac OS X, FreeBSD y OpenBSD.

Hasta hace relativamente poco tiempo, el uso de Linux se limitaba mayormente al desarrollo de aplicaciones, la administración de sistemas y la industria del *software*. Actualmente, el desarrollo y uso extendido de las tecnologías de virtualización, contenedores, orquestación, sistemas embebidos de ámbito general y la computación en la nube, así como la expansión del Internet de las Cosas, *Internet of Things* (IoT) por sus siglas en inglés, han hecho que el uso de GNU/Linux crezca en los últimos años[4][9]. También se ha producido un aumento en el uso diario de Linux como entorno de escritorio. Sin embargo, algunas de estas aplicaciones necesitan de versiones de Linux no estándar y especializadas. Un caso a destacar en el uso de Linux es el de la lista *Top500*[6], lista que clasifica a los quinientos mejores súper computadores del mundo atendiendo a su rendimiento[1], y que está formada en su totalidad por computadoras que emplean versiones personalizadas del sistema operativo Linux.

Esta adaptación de un sistema operativo Linux, que puede enfocarse tanto a adecuar su apariencia o contenido a nuestros gustos personales como a mejorar el rendimiento del sistema en el dispositivo que lo ejecuta, puede hacerse fundamentalmente de dos maneras diferentes:

- Personalizar un sistema operativo ya existente.
- Crear un sistema operativo personalizado desde cero.

[1] Rendimiento evaluado usando el *Linpack Benchmark*: *https://www.top500.org/project/linpack/*

También pueden usarse soluciones mixtas de los dos grandes subtipos anteriores, por ejemplo usando para la creación de un sistema operativo propio algunos elementos de sistemas operativos ya existentes, como el *kernel*.

1.1 Personalizar un sistema operativo ya existente

Esta es quizá la solución más sencilla y también la más habitual. Consiste, básicamente, en modificar un sistema operativo conocido añadiendo y quitando los componentes deseados a la vez que se ajusta la configuración de los mismos a la pretendida. Es un proceso tedioso, que puede necesitar de un amplio espacio de tiempo para llevarse a cabo y que requiere de unos conocimientos especializados y avanzados para el usuario medio. Una forma de aligerar o facilitar este proceso es hacer uso de la automatización.

Para ello, podemos emplear programas o *scripts* que reproduzcan las modificaciones deseadas en el sistema operativo objetivo. Estos *scripts* requieren de un tiempo de desarrollo que puede ser un obstáculo para el uso de esta variante de personalización, además de que pueden llegar a tener un alto nivel de complejidad y requerir de unos conocimientos poco comunes en la mayor parte de la población.

Internet, también llamada coloquialmente como *la red de redes*, consiste en la unión lógica a nivel mundial de múltiples redes diferentes entre sí, esto es, heterogéneas, mediante la pila de protocolos *TCP/IP* desarrollada por el *IETF* y resumida en la norma *RFC 1180: a TCP/IP Tutorial*[2]. La invención y el posterior desarrollo de *Internet* ha conseguido conectar entre sí prácticamente todos los dispositivos electrónicos del globo, permitiendo a los usuarios de *Internet* publicar y generar contenido, por ejemplo en una página web, un blog, una plataforma de *microblogging*, contenido en formato vídeo, un *podcast* de audio, etc., y que este contenido esté disponible para el resto de usuarios de manera casi instantánea y prácticamente permanente en el tiempo. La gigantesca cantidad de información y datos que esto produce ha derivado a su vez en la creación y el desarrollo de nuevos perfiles profesionales, como el de *analista de datos*. A menor escala, la que engloba a la mayoría de usuarios, toda esta información supone una mayor probabilidad de encontrar la información que se busca o necesita en un momento determinado para multitud de temas dispares de una manera rápida y sencilla, pero también conlleva ciertos riesgos y aspectos a tener en cuenta, como la fiabilidad y credibilidad.

En el caso específico que nos atañe, *Internet* ha posibilitado que cientos, incluso varios miles, de esos *scripts* de personalización que nombrábamos anteriormente, estén a nuestro alcance, como suele decirse de manera informal, *a un click de distancia*. Por ejemplo, en plataformas de desarrollo colaborativo como *GitHub*[3], *BitBucket*[4], o *Savannah*[5], entre otras.

Muchos de estos programas y *scripts* han sido publicados, mantenidos y mejorados a lo largo del tiempo, o lo fueron originariamente y ahora otro usuario continua manteniendo el trabajo original, por grandes empresas tecnológicas y/o gente conocida, *celebrities* de la tecnología, la ingeniería del Software y el movimiento *Open Source* que liberan y comparten sus conocimientos con el resto de usuarios de manera altruista y desinteresada. Unos *scripts* de personalización que son serios candidatos a cumplir con su objetivo legítimo original: personalizar un sistema operativo ya existente. Sin embargo, dentro de todo este ecosistema de disponibilidad, gratuidad y facilidad, hay usuarios cuyas intenciones no son buenas, y que intentan, consiguiendo en muchas ocasiones, aprovecharse de los usuarios ingenuos que forman la zona ancha de esta particular *cadena alimenticia* basada en datos personales e información. El simple hecho de ejecutar un *script* de personalización que provenga de una fuente no fiable y/o nos limitemos a ejecutar sin saber qué hace ni cómo en nuestro sistema operativo, servidor de *hosting* o dispositivo genérico, podría hacer que el *depredador informático* creador de dicho *script*, tomara el control de los mismos, teniendo acceso también al resto de nuestros datos personales e información confidencial: datos médicos, datos bancarios, contactos, fotos, o datos laborales[12][1].

El peligro que ello conlleva debería hacernos reflexionar sobre qué queremos conseguir, cómo, y a qué precio. Así pues, parece lógico pensar que conocer qué instalamos en el sistema, de dónde proviene y cómo

[2] RFC 1180: *https://tools.ietf.org/html/rfc1180*

[3] GitHub Inc.: *https://github.com/*

[4] The BitBucket: *https://bitbucket.org/*

[5] The GNU Savannah: *http://savannah.gnu.org/*

funciona es una buena medida de protección. Pero, ¿por qué conformarse con conocer en profundidad una parte del sistema operativo cuando podemos conocerlo al completo?

1.2 Crear un sistema operativo personalizado desde cero

La opción de crear un sistema operativo personalizado partiendo de cero puede parecer extraña y es lógico suponer que no es sencilla. Para muchos, esta opción responde más a un deseo de aprender o a una necesidad técnica.

1.2.1 Aprendizaje

En el comportamiento humano pueden distinguirse dos tipos de conductas: una innata y otra aprendida. El aprendizaje, por su parte, responde a una curiosidad innata en el ser humano que provoca la modificación de la conducta como resultado de la experiencia.

Una de las figuras más importantes para la Psicología de la Educación es Jerome Seymour Bruner, psicólogo estadounidense fundador del *Centro de Estudios Cognitivos* de la Universidad de Harvard y cuyos trabajos han tenido una amplia influencia en el ámbito de la educación [8]. Para Bruner, el objetivo último de la enseñanza es conseguir que el alumno adquiera la comprensión *general* de la estructura de un área de conocimiento. La teoría de Bruner se basa en cuatro principios fundamentales; la motivación, la estructura, la secuencia y el reforzamiento.

La motivación es la condición que predispone al ser humano hacia el aprendizaje y su interés sólo se mantiene cuando existe una motivación intrínseca. Los motivos que incitan al aprendizaje son el instinto innato de curiosidad, la necesidad de desarrollar unas competencias y la reciprocidad. El instinto innato de curiosidad, por ejemplo, se manifiesta desde el nacimiento. Cuando una persona se encuentra en una situación para la que existen múltiples respuestas, esta curiosidad innata lo incita a buscar la información necesaria para resolver el problema.

El objetivo último de la enseñanza de una materia es la comprensión de la estructura fundamental de la misma, y adquirir la estructura de una asignatura es comprenderla de tal manera que podamos relacionar los conocimientos de la misma con los de otra. Para Bruner, el contenido de una asignatura debe estructurarse de tal forma que pueda transmitirse de manera sencilla y comprensible.

Por otro lado, Bruner propone el *aprendizaje por descubrimiento*, lo que implica un aprendizaje *inductivo*, es decir, debe partir de datos, hechos y situaciones particulares y también de experimentos prácticos, en lugar de basarse exclusivamente en las explicaciones de un profesor o el uso de la memoria, aunque también defiende su idoneidad en ciertas situaciones. Así, Bruner considera que se debe estimular a los alumnos a que descubran la estructura de la asignatura de manera autónoma por medio del descubrimiento guiado. Por último, Bruner acepta también el uso del llamado aprendizaje por recepción, ya que entiende que sería una pérdida de tiempo que cada estudiante tuviera que redescubrir todo el conocimiento que debe aprender.

En el caso que nos ocupa, construir un sistema operativo desde cero es una tarea amplia, tediosa y que requiere de conocimientos especializados. El proyecto *Linux from Scratch* proporciona un método paso a paso, con explicaciones del problema a resolver, su solución y las opciones de personalización de cada elemento en caso de que las hubiera. Relacionando además los conocimientos adquiridos en cada paso con los del paso anterior y posterior y con su papel en el objetivo final: el sistema operativo construido. Representa, así, un método de construcción modular en el que cada pieza sostiene la colocación de la pieza siguiente y es sostenida por la anterior. Podríamos decir, por tanto, que el proyecto *Linux from Scratch* cumple los requisitos expuestos por Bruner, y que ello lo convierte en una guía idónea para entender este campo de conocimiento y asegurar que el resultado tendrá un estructura comprensible y útil.

1.2.2 Necesidad para desarrolladores y administradores de sistemas

La tecnología es un campo en constante evolución. Una evolución que nos ha llevado a cambios tan sorprendentes como elevar el número de *bits* por segundo que nuestros equipos son capaces de manejar desde unos pocos *kilobytes* por segundo, o *kBps*, a las conexiones de diez o más *gigabits* por segundo, *Gb/s*, de las que disponemos hoy en día para las conexiones tipo *Ethernet*[6]. Mejoras y modificación de la frecuencia y modulación en los sistemas de Comunicaciones Digitales, cambios fundamentales que han permitido pasar de la comunicación oral entre dos localizaciones fijas, a poder comunicarse con un intercolutor con independencia de su ubicación[10]. Cambios en la arquitectura de ordenadores, como el paso de arquitecturas de 32 *bits* a las de 64 *bits*. Todos estos cambios, y otro muchos no nombrados aquí por abreviar, han significado también avances en la administración de sistemas y en la ingenieria del software. Como se comentaba en apartados anteriores, la aparición y auge de las tecnologías de microservicios, contenedores, virtualización y orquestación, como Docker o Kubernetes, también conocido este último como *k8s*, están dando visibilidad al sistema operativo Linux tanto por su habilidad de adaptación al entorno en el que se ejecuta como por su buen rendimiento.

La elección entre el uso de un modelo de virtualización u otro no es el objetivo de este texto, pero sí se pretende presentar los diferentes contextos en que usar un sistema operativo creado desde cero puede suponer una ventaja frente a los sistemas operativos convencionales.

Una arquitectura basada en *microservicios* hace referencia a un patrón de diseño para programas informáticos, generalmente aplicaciones web, compuesto por *microservicios* como su unidad mínima individual e indivisible que, mediante la comunicación de varios de estos componentes, haciendo uso del protocolo de nivel de aplicación *HTTP*, acrónimo de *HiperText Transfer Protocol* y definido en la RFC 2616 [7], o una RESTful API, *Representational State Transfer Application Programming Interface* [8], da forma a una aplicación mayor. Además, desde el punto de vista de la seguridad, poder separar la ejecución de los diferentes componentes y la información que manejan del resto de elementos es una buena práctica ya que la cantidad de información expuesta por cada nodo es sólo una parte del total de la información que se maneja.

Cada microservicio se presupone independiente del resto y debe poder ser desplegado sin afectar al correcto funcionamiento de los demás. Esta independencia entre microservicios puede conseguirse haciendo que cada componente esté aislado física o lógicamente del resto de microservicios que forman la aplicación. Podemos conseguir esta separación alojando cada uno de los microservicios que forman el sistema final en un contenedor o una máquina virtual de sistema diferentes.

Una máquina virtual de sistema, o máquina virtual de tipo 2, permite emular un ordenador y ejecutar un sistema operativo dentro de ella independiente del sistema operativo que la aloja. Esta independencia llega a todos los niveles, incluso a nivel fisico, ya que el sistema operativo hospedado dispone de los recursos de memoria *RAM* y Unidad de Control de Procesos, *CPU*, que le son expuestos desde el sistema anfitrión. Estas máquinas virtuales están desacopladas del sistema operativo huésped por una capa de *software* llamada *hipervisor*. La arquitectura típica de un sistema basado en máquinas virtuales con virtualización de tipo 2, está reflejada en la imagen 1.1a de la página 6.

Hipervisores de máquinas virtuales conocidos son *VMWare*, *VirtualBox* o *QEMU*.

[6] *Ethernet: IEEE 802 Working Group* *http://www.ieee802.org*

[7] RFC 2616: *https://tools.ietf.org/html/rfc2616*

[8] RESTful API: *https://www.w3.org/TR/2004/NOTE-ws-arch-20040211*

Las principales ventajas del uso de máquinas virtuales de este tipo son[4]:

- **Flexibilidad**: los recursos de la máquina anfitriona pueden usarse a conveniencia.
- **Eficiencia**: una máquina virtual funciona como una máquina independiente, que puede hacer un uso eficiente de los recursos de los que dispone. Separar los datos mejora la seguridad de la aplicación.
- **Copias de seguridad y recuperación**: las máquinas virtuales pueden almacenarse en el equipo anfitrión como un único archivo de disco, en formato *VMDK*, acrónimo de *Virtual Machine DisK*, *IMG* o similar, que pueden ser fácilmente copiados a otro dispositivo como copia de seguridad y ser restaurados en caso necesario.
- **Movilidad**: es sencillo mover una máquina virtual de una máquina anfitriona a otra en caso de fallos o necesidad. De hecho, algunos hipervisores de máquinas virtuales soportan hacer uso de esta característica de forma automática.
- **Libertad de elegir el sistema operativo**: diferentes sistemas operativos en distintas máquinas virtuales pueden coexistir en un mismo equipo anfitrión siempre que dispongan de una arquitectura común.

Como contrapartida, el uso de máquinas virtuales tiene los siguientes inconvenientes:

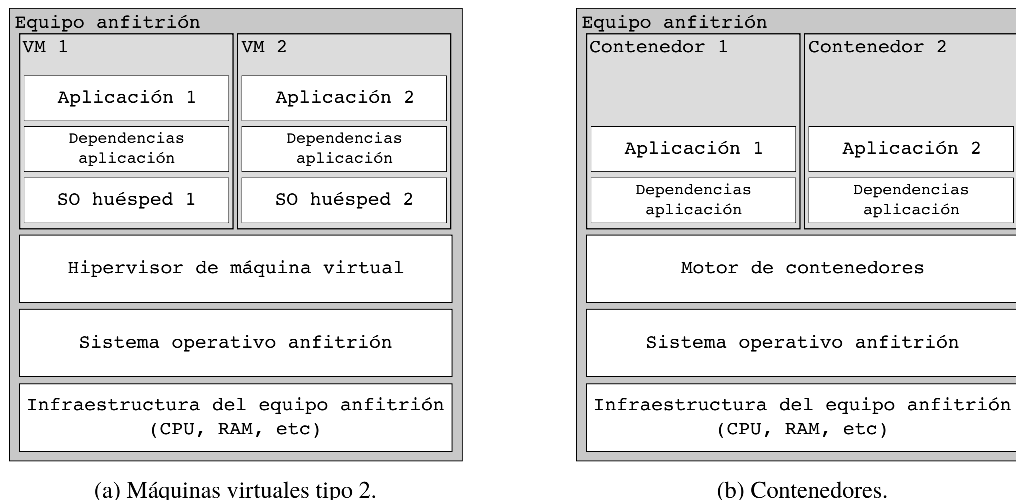
- **Compartir máquinas virtuales**: compartir máquinas virtuales a través de una red local puede consumir una mayor cantidad de tiempo debido al tamaño del archivo de disco de la máquina virtual.
- **Portabilidad**: a medida que se realizan cambios en una imagen de máquina virtual, estos cambios no se reflejan en todas las instancias de esa imagen.
- **Sobrecarga y desaprovechamiento de recursos**: el hipervisor de máquinas virtuales debe hacer de intermediario e interactuar con ambos sistemas operativos, lo que afecta al rendimiento de la máquina virtual. Además, el sistema operativo huésped puede tener unos requisitos de espacio, memoria *RAM* mínima y *CPU* que generan una ocupación de recursos que podría ser ineficaz.

Un contenedor Linux es una unidad estándard de software formada por el conjunto de todas las dependencias necesarias para ejecutar un programa específico sin la necesidad de incluir un *kernel* propio, y sustituyendo el hipervisor de máquinas virtuales por una aplicacion para la gestión de contenedores llamada motor de contenedores (*container engine*). Esta es la principal diferencia con respecto a la arquitectura basada en máquinas virtuales de la figura 1.1a de la página 6. Por otra parte, la arquitectura basada en contenedores también proporciona ese aislamiento de la aplicación con respecto al resto del sistema o de contenedores, a pesar de compartir el *kernel* del sistema anfitrión. Esto se consigue usando funciones nativas del *kernel* de Linux como son los *control groups*[9] y los *namespaces*[10], lo que implica que el uso de máquinas virtuales produce un mayor aislamiento entre procesos que el que produce el uso de contenedores, pero estos son más livianos que las máquinas virtuales y son más fáciles de mover entre una máquina y otra. Además, y debido también a que son más livianos, es posible ejecutar más contenedores que máquinas virtuales en una misma máquina, haciendo también un mejor uso de los recursos del sistema. Un esquema de la arquitectura basada en contenedores puede encontrarse en la figura 1.1b de la página 6.

Los contenedores han ganado notoriedad gracias a la aparición del motor de contenedores *Docker* [11]. Los contenedores Docker son diferentes a los contenedores Linux, pero no es el objetivo de este texto indicar las diferencias entre ambos. Sí hay que puntualizar aquí que el uso de contenedores *Docker* reducen, y pueden llegar a eliminar, los efectos producidos por las desventajas del uso de máquinas virtuales.

Una comparativa de ambos modelos de virtualización puede encontrarse en la figura 1.1 de la página 6.

[9] cgroups: kernel control groups *https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt*
[10] namespaces: página del manual para *linux namespaces* *http://man7.org/linux/man-pages/man7/namespaces.7.html*
[11] Docker: sistema para ejecutar aplicaciones en contenedores *https://www.docker.com*



(a) Máquinas virtuales tipo 2. (b) Contenedores.

Figura 1.1 Comparativa de arquitecturas de virtualización.

Para gestionar estos contenedores necesitamos un método de manejo y despliegue, lo que se conoce como *orquestación* de contenedores. Ejemplos de aplicaciones que nos permiten orquestar contenedores son *Kubernetes* [12] o *Docker Swarm* [13].

Otro punto interesante en las arquitecturas de virtualización es el uso de un sistema operativo que permita sacar ventaja, o reducir una desventaja al mínimo posible, del tipo de virtualización elegido. Una tendencia actual es la de usar un sistema operativo lo más pequeño posible, que ocupe poco espacio en memoria, y cuyo rendimiento sea óptimo. Además, si el sistema operativo está compuesto exclusivamente por las dependencias necesarias para ejecutar la aplicación que se quiere aislar, se consigue que la superficie de exposición del microservicio sea la mínima posible, elevando así la seguridad del mismo. Ejemplos de sistemas operativos livianos con uso extendido en contenedores y máquinas virtuales son CoreOS[14] , Alpine Linux[15] o minideb[16]. Por su parte, el proyecto *Linux from Scratch* proporciona en su página web las instrucciones necesarias para construir un sistema operativo Linux completo y asegura poder generar un sistema operativo lo suficientemente pequeño como para ser capaz de ejecutar un servidor web Apache con un tamaño de sólo 8 MB, ocho *megabytes*.

Aquí reside el objetivo principal de este texto, analizar las diferentes opciones de creación de sistemas operativos basados en el proyecto *Linux from Scratch* y su utilidad de automatización del proceso con los que justificar el desarrollo de una nueva herramienta que permita un mayor nivel de personalización y capacidad de adaptación a los requisitos de los usuarios del sistema operativo generado. Unos requisitos y una flexibilidad que las distribuciones Linux actuales pueden no ser capaces de ofrecer en determinadas situaciones.

El capítulo 2 presenta el proyecto *Linux from Scratch* y sus distintos subproyectos explicando qué partes de la creación de un sistema operativo aborda cada uno de ellos, examinando el método de consecución del objetivo principal de dicho subproyecto y los beneficios que aportaría un proceso de automatización. A continuación se analiza la herramienta *jhalfs* creada por los propios desarrolladores del proyecto para automatizar el proceso examinado anteriormente y se presentan los motivos que provocan la creación de la herramienta *LFSBuilder*.

[12]Kubernetes: *Automated container deployment, scaling, and management* https://kubernetes.io

[13]Docker Swarm: *Docker Swarm is native clustering for Docker.* https://docs.docker.com/engine/swarm/

[14]CoreOS: *página web* https://coreos.com

[15]Alpine Linux: *distribución Linux de propósito general, independiente y no comercial, diseñada para usuarios que aprecian la seguridad, simplicidad y la eficiencia.* https://alpinelinux.org

[16]minideb: *imagen minimalista basada en Debian construída específicamente para ser usada como imagen base en contenedores* https://github.com/bitnami/minideb

El capítulo 3 se centra en el diseño y la implementación de la herramienta *LFSBuilder*, su estructura interna, su capacidad de adaptación a los requisitos de construcción del usuario de la herramienta y una comparativa con la herramienta *jhalfs* mencionada anteriormente.

El capítulo 4 enumera las pruebas realizadas al código fuente y para la ejecución de la herramienta *LFSBuilder* dejando el capítulo 5 para mencionar las conclusiones del proyecto realizado, así como alguna de las muchas líneas de mejora de las que dispone una herramienta en constante evolución como es *LFSbuilder*.

2 Linux from Scratch

I never teach my pupils, I only attempt to provide the conditions in which they can learn

Albert Einstein

Linux from Scratch, abreviado comúnmente por las siglas LFS, es, tal y como puede leerse textualmente en su página web, un proyecto que provee al lector con instrucciones paso a paso para construir su propio sistema Linux personalizado a partir del código fuente. La primera versión del libro data del año 2001. El idioma original del proyecto es el inglés, aunque existe una traducción no oficial en español[1] y desactualizada desde la versión *6.3* de *Linux from Scratch* con fecha 30 de agosto de 2007. En cualquier caso, en este texto se analiza la versión más reciente del proyecto en su idioma original, examinando para cada uno de los dos subproyectos más relevantes, *Linux from Scratch* y *Beyond Linux from Scratch*, la estructura y el método de consecución de su objetivo principal, así como las conveniencia de automatización del proceso en cada caso. Por último se presenta el subproyecto de automatización *Automated Linux from Scratch*, analizando además la herramienta *jhalfs* desarrollada y mantenida por el propio proyecto *Linux from Scratch*.

Entre sus principales características y ventajas, los miembros del proyecto *Linux from Scratch* destacan los siguientes[5]:

- **Aprendizaje**: construir un sistema LFS enseña cómo funciona Linux internamente, cómo sus elementos funcionan en conjunto y dependen unos de otros y además cómo es posible personalizarlo para adecuarlo a nuestras preferencias y necesidades. También es un buen recurso práctico para aprender sobre la compilación de código fuente y resolución de dependencias.
- **Un sistema compacto**: al instalar una distribución Linux habitual, generalmente se instalan gran cantidad de programas y archivos que sólo ocupan espacio en nuestro disco o duro y que jamás usamos, junto con las aplicaciones y librerías que sí necesitamos. Los desarrolladores del proyecto aseguran que no es complicado construir un sistema Linux con un tamaño inferior a los 100 MB, cien *megabytes*. También afirman poder generar un sistema operativo lo suficientemente pequeño como para ser capaz de ejecutar un servidor web Apache con un tamaño de sólo 8 MB, ocho *megabytes*[2].
- **Flexibilidad**: *Linux from Scratch* construye un sistema operativo Linux mínimo. Si, como los desarrolladores, comparásemos ese sistema con la construcción de una casa, podríamos decir que el sistema generado es el esqueleto básico de la casa, a la que le faltan aún los sistemas de fontanería, la instalación eléctrica, la cocina, etc. Esto posibilita que cada usuario pueda ajustar la construcción de su sistema final a sus necesidades, o contentarse con el resultado básico. Los propios creadores del proyecto proporcionan otro subproyecto, llamado *Beyond Linux from Scratch*, BLFS, con las instrucciones necesarias para ampliar el sistema base con multitud de funciones y programas.

[1] para consultar la traducción no oficial en español visite la URL *http://www.escomposlinux.org/lfs-es/*

[2] Más información sobre cómo construir un sistema *Linux from Scratch* reducido puede encontrarse en la siguiente dirección web *http://www.linuxfromscratch.org/hints/downloads/files/small-lfs.txt*

- **Seguridad**: compilar el sistema al completo permite al usuario que lo desee auditar el código fuente que va a formar parte del sistema operativo final y aplicar en el mismo todos los parches de seguridad o solución de problemas que se deseen. Algunos de estos parches están incluidos por defecto en el libro, bien porque resuelven algún *bug* en el código fuente o fallo de seguridad, o bien porque son necesarios para poder ejecutar correctamente algún otro comando del proceso de construcción, como los tests.

Como se ha comentado en el punto *flexibilidad* de la lista anterior, *Linux from Scratch* está formado por varios subproyectos que permiten añadir distintas funcionalidades o realizar diferentes tareas relacionadas con el proyecto. Estos subproyectos y una breve descripción de los mismos se definen en la tabla 2.1.

Siglas	Nombre	Descripción
LFS	Linux from Scratch	Libro básico con instrucciones paso a paso para construir el sistema base.
BLFS	Beyond Linux from Scratch	Libro con instrucciones para extender y personalizar el sistema base. Incluye instrucciones para construir programas de uso común como *OpenSSH* o el servidor de páginas web *Apache*.
ALFS	Automated Linux from Scratch	Proyecto para automatizar la construcción del libro LFS.
CLFS	Cross Linux from Scratch	Instrucciones para realizar compilación cruzada a diferentes arquitecturas.
	Hints	Guías creadas por usuarios para personalizar algunas partes del sistema que no se incluyen en los libros LFS o BLFS.
	Patches	Colección de parches aplicables a los componentes del sistema.

Tabla 2.1 Subproyectos *Linux from Scratch*.

El proyecto *Linux from Scratch* intenta seguir tres estándares Linux lo más fielmente posible para asegurar la compatibilidad de los sistemas operativos generados a partir de *Linux from Scratch* con otras distribuciones o sistemas operativos Linux. Estos estándares son:

- **POSIX.1-2008**: la revisión más reciente de este estándar, concretamente la versión *POSIX.1-2017*[3], define la interfaz y el entorno de ejecución que deben ser comunes a los sistemas operativos para garantizar la portabilidad de aplicaciones a nivel de código fuente. Esto es, garantizar que un mismo código fuente podrá funcionar del mismo modo en dos sistemas operativos diferentes que implementen dicho estándar.
- **Filesystem Hierarchy Standard**: abreviado como FHS[4] y redactado por la *Linux Foundation*. Consiste en un conjunto de consejos y requisitos que deben cumplir la disposición de archivos y directorios en sistemas de tipo *UNIX*[*] con el fin de asegurar a usuarios y aplicaciones que se ejecuten en el sistema operativo, que la mayoría de archivos de configuración, binarios y directorios del sistema se encuentran en el lugar en el que se espera que estén. Por ejemplo, que en el directorio `bin`, en la ruta `/bin`, se encuentran los binarios y *scripts* comunes a todos los usuarios.

[3] POSIX.1-2017: *http://pubs.opengroup.org/onlinepubs/9699919799/*
[4] FHS: *http://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.html*

- **Linux Standard Base**: abreviado como LSB[5] y redactado también por la *Linux Foundation*, este estándard busca la compatibilidad entre las distintas distribuciones Linux.

Construir un sistema operativo requiere ciertos conocimientos previos de administración de sistemas Linux y construcción de *software* a partir del código fuente. Estos conocimientos son necesarios para poder ejecutar correctamente los comandos necesarios para construir los componentes del sistema y/o resolver problemas que puedan ocurrir durante la construcción de los mismos.

Habitualmente, la construcción de aplicaciones desde el código fuente conlleva realizar uno o varios de los siguientes pasos: descargar el archivo que contiene el código fuente, descomprimir el archivo, aplicar los parches necesarios, construir e instalar la aplicación haciendo uso de alguna utilidad de construcción de software como `cmake`, `make` o `ninja` y configurar la aplicación.

Podemos obtener el código fuente de la aplicación de diversas maneras, bien descargando una copia del mismo usando algún software para el control de versiones como `subversion` [6] o `git` [7] o bien descargando el archivo comprimido mediante una dirección web. Para este último método suelen usarse utilidades de línea de comandos como `wget` [8] o `curl` [9], que además permiten descargar varios ficheros automáticamente a partir de una lista de enlaces. Una vez descargado el código fuente, generalmente comprimido en formato `.tar.gz` con compresión `gzip` [10], podemos descomprimirlo usando la aplicación `tar` [11] en un terminal Linux.

Después de descomprimir el código fuente, y antes de construir la aplicación, es opcional (y a veces obligatorio) aplicar una serie de cambios al código fuente que resuelven fallos de funcionamiento o añaden nuevas funcionalidades al programa base. Estos cambios, llamados *parches*, vienen definidos en unos archivos con extensión `.patch` o `.diff` y se aplican en el código fuente haciendo uso del programa `patch`. También puede ser opcional la ejecución de otros comandos que preparen el código fuente antes de la compilación. El siguiente paso es construir el programa haciendo uso de alguna utilidad de construcción de *software*. La más extendida es `make`, y el proceso de construcción de *software* suele requerir ejecutar los siguientes comandos para limpiar un intento de construcción previo, configurar la construcción del programa a la máquina en que se va a construir, construirlo e instalarlo. Estos comandos aceptan parámetros por línea de comandos que permiten personalizar el resultado, y son: `make clean`, `configure`, `make`, y `make install`. Por último, y una vez se ha instalado el programa, las aplicaciones permiten ser configuradas para adecuar su funcionamiento al deseado por el usuario modificando algún archivo de configuración o ejecutando el programa con algún parámetro concreto por línea de comandos[12].

[5] LSB: *http://refspecs.linuxfoundation.org/lsb.shtml*
[6] subversion: *https://subversion.apache.org*
[7] git: *https://git-scm.com*
[8] wget: *https://www.gnu.org/software/wget/*
[9] curl: *https://curl.haxx.se*
[10] gzip: *https://www.gnu.org/software/gzip/*
[11] tar: *https://www.gnu.org/software/tar/*
[12] puede obtenerse más información sobre el proceso de construcción de programas a partir de su código fuente en la URL *http://moi.vonos.net/linux/beginners-installing-from-source/*

2.1 Linux from Scratch

Como se muestra en la tabla 2.1 de la página 10, *Linux from Scratch* es el subproyecto principal y está compuesto por el libro homónimo y los archivos en formato XML, *eXtensive Markup Language*, que constituyen el código fuente del mismo. Este libro genera el sistema operativo básico como resultado de las instrucciones incluidas en él. Está dividido en cuatro grandes partes lógicas, cada una de ella compuesta por varios capítulos, en los que se presentan los componentes en el orden requerido para resolver las distintas dependencias entre ellos. Construir el sistema operativo resultado del libro requiere leer y ejecutar los comandos proporcionados en dichas partes. La figura 2.1 resume este proceso en un diagrama de flujo básico.

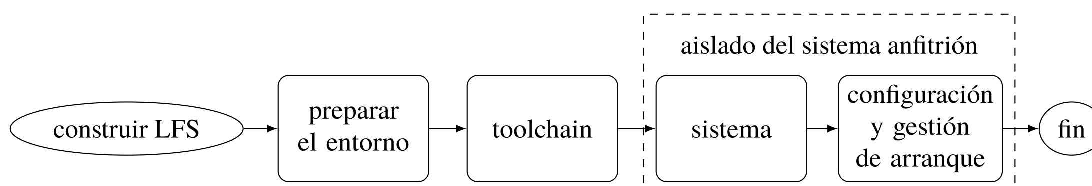


Figura 2.1 Diagrama de flujo para el libro *Linux from Scratch*.

La tabla 2.2 resume la correspondencia entre los distintos bloques del diagrama 2.1 y los capítulos del libro *Linux from Scratch*.

Bloque	Capítulos
Preparación del entorno	1-4
Toolchain	5
Sistema	6
Configuración y gestor de arranque	7-9

Tabla 2.2 Correspondencia entre los bloques del diagrama 2.1 y los capítulos del libro *Linux from Scratch*.

2.1.1 Preparación del entorno y requisitos previos

Linux from Scratch utiliza un sistema operativo Linux como base para construir el sistema operativo. Este sistema anfitrión provee los programas y librerías necesarios para construir un conjunto de programas, llamado *toolchain*[13], con los que construir el sistema final. Las dependencias del libro *Linux from Scratch* se listan en la tabla 3.5 de la página 63.

[13]más información en el apartado 2.1.2 Toolchain de la página 17

Herramienta	Comentario
Bash	Intérprete de línea de comandos. `/bin/sh` debe ser un enlace simbólico a `/bin/bash`.
Binutils	Utilidades básicas para el proceso de compilación de código fuente .
Bison	Generador de analizadores sintácticos. `/usr/bin/yacc` debe ser un enlace a `/usr/bin/bison`.
Bzip2	Programa para comprimir y descomprimir archivos.
Coreutils	Utilidades para manejar propiedades básicas del sistema.
Diffutils	Programas para mostrar diferencias entre archivos o directorios.
Findutils	Contiene programas para buscar archivos en un arbol de directorios.
Gawk	Programa para la manipulación de ficheros de texto. `/usr/bin/awk` debe ser un enlace simbólico a `/usr/bin/gawk`.
GCC	Incluyendo el compilador `g++`.
Glibc	Librería con rutinas principales del lenguaje de programación C.
Grep	Este programa permite realizar la busqueda de cadenas de texto dentro de archivos.
Gzip	Programa para comprimir y descomprimir archivos.
M4	Procesador de macros de ámbito general para la sustitución de cadenas de texto.
Make	Herramienta de construcción de código fuente.
Patch	Programa para realizar modificaciones en ficheros a partir de *parches*.
Perl	Lenguaje de programación.
Sed	Editor de flujo de datos. Generalmente usado para la sustitución y modificación de cadenas.
Tar	Programa para comprimir y descomprimir archivos.
Texinfo	Programa para leer y escribir páginas de información.
Xz	Programa para comprimir y descomprimir archivos.

Tabla 2.3 Dependencias del sistema anfitrión para construir el sistema operativo *Linux from Scratch*.

Antes de empezar a construir estas herramientas, debemos crear la partición o particiones de disco duro en que construir el sistema operativo *Linux from Scratch*, formatear dicha partición, montarla en algún lugar conocido del sistema operativo anfitrión y crear un usuario sin privilegios con el que construir estas herramientas. Los desarrolladores de *Linux from Scratch* recomiendan un tamaño mínimo de seis gigabytes, 6 *GB*, para esta partición, aunque un tamaño de veinte gigabytes, 20 *GB*, suele ser apropiado para un sistema operativo de uso común o primario. Debido a que es probable que no se disponga de la suficiente memoria *RAM* para el proceso de compilación, es también buena idea crear una parición de tipo *swap* para ser usada como memoria de intercambio durante el proceso de construcción, aunque podríamos usar directamente la partición *swap* del sistema anfitrión, en cuyo caso no sería necesario crear una nueva. Es cierto que el sistema *Linux from Scratch* final no ocupará tanto espacio necesariamente, pero este espacio de más permitirá un espacio de almacenamiento temporal suficiente a lo largo del proceso, por ejemplo durante la compilación de algunos componentes, y también añadir nuevas funcionalidades al sistema final, usando el libro *Beyond Linux from Scratch* o añadiendo nuestras propias funcionalidades o personalizaciones, quizás desde algún *script* de personalización provisto por terceros como se comentó en el apartado *1.1 Personalizar un sistema operativo ya existente* de la página 2.

Dado que es necesario utilizar un usuario sin privilegios para construir el *toolchain* por motivos de seguridad, es posible tener que crear dicho usuario. El motivo de compilar un programa con un usuario sin privilegios, excepto quizás la fase de instalación ejecutando el comando `make install`, es que los comandos `configure` y `make` pueden ejecutar comandos arbitrarios en el sistema, pudiendo modificar partes esenciales del sistema haciéndolo inservible. En Linux podemos usar los comandos `groupadd` y `useradd` para crear el grupo y usuario `lfs`. Un ejemplo de uso de estos comandos pueden encontrarse en el código 2.1.

```
# Creamos el grupo 'lfs'
groupadd lfs

# Creamos el usuario 'lfs' asociado al grupo 'lfs'.
useradd -s /bin/bash -g lfs -m -k /dev/null lfs
```

Código 2.1 Añadir el usuario y grupo `lfs`.

Los parámetros utilizados en el comando `useradd` anterior son los siguientes:

- `-s /bin/bash`: asigna la consola `/bin/bash` como predeterminada para el usuario creado.
- `-g lfs`: el grupo `lfs` es el grupo primario del usuario creado.
- `-m`: crea el directorio personal del usuario en `/home`.
- `-k /dev/null`: usa `/dev/null` como esqueleto base del directorio personal del usuario creado.

Pueden crearse nuevas particiones de disco usando las utilidades `cfdisk` y `fdisk`[14]. Hay que destacar aquí el peligro potencial de estas utilidades de gestión de particiones de disco duro, ya que podría modificarse erróneamente el particionado del disco duro para el sistema operativo anfitrión y hacerlo inservible. De ahora en adelante, supondremos que la partición de nombre `/dev/sdb1` es la partición en la que se va a construir el sistema operativo, y la partición de nombre `/dev/sdb2` es la partición que se utilizará como memoria de intercambio o *swap*. Una vez creadas las particiones hay que darles formato con el programa `mkfs`[15]. Los comandos del código 2.2 inferior muestran los comandos que han de ejecutarse como usuario administrador del sistema *root* para formatear las particiones `/dev/sdb1` y `/dev/sdb2` como partición en formato *ext4* y *swap* respectivamente.

[14]fdisk: consultar la entrada del manual ejecutando la instrucción de línea de comandos `man 8 fdisk` para más información.
[15]mkfs: consultar la entrada del manual ejecutando la instrucción de línea de comandos `man 8 mkfs` para más información.

```
# Formatear la partición '/dev/sdb1' como ext4
mkfs -v -t ext4 /dev/sdb1

# Formatear la partición swap
mkswap /dev/sdb2
```

Código 2.2 Formatear las particiones `/dev/sdb1` y `/dev/sdb2`.

Para montar la nueva partición `/dev/sdb1`, Linux dispone del programa `mount`. Además, *Linux from Scratch* aconseja añadir la variable de entorno `LFS` apuntando al directorio en el cual se va a montar la partición `/dev/sdb1` para referirse a este punto de montaje de manera sencilla, por ejemplo en el directorio `/mnt/lfs`. Los comandos a ejecutar para crear esa variable de entorno, crear el punto de montaje, montar la partición y habilitar la partición `/dev/sdb2` como *swap* están recogidos en el código 2.3.

```
# Definir la variable de entorno 'LFS'
LFS=/mnt/lfs
export $LFS

# Crear el punto de montaje
mkdir -pv $LFS

# Montar la partición '/dev/sdb1'
mount -v -t ext4 /dev/sdb1 $LFS

# Habilitar partición '/dev/sdb2' como swap
/sbin/swapon -v /dev/sdb2
```

Código 2.3 Definir la variable de entorno *LFS* y montar las particiones.

Otras variables de entorno y opciones del intérprete de línea de comandos necesarias para construir el sistema *Linux from Scratch* son las siguientes:

- `set +h`: este comando desactiva el cacheo de localización de programas del intérprete de línea de comandos *bash* obligando al intérprete a buscar en los directorios contenidos en la variable de entorno `PATH` cada vez que se necesite ejecutar un archivo binario. Esto, aunque obliga a buscar múltiples veces un mismo archivo binario y puede parecer poco útil o productivo, supone la clave para aislar el sistema operativo en construcción del sistema anfitrión [16].
- `umask 022`: el programa `umask` establece la máscara de creación de ficheros y directorios por defecto para el usuario al que se configura. El parámetro `022` proporcionado configura los permisos por defecto para los nuevos ficheros y directorios del usuario restando el valor proporcionado a los valores por defecto. Estos son, `666` y `777` para ficheros y directorios respectivamente. Así pues, los ficheros creados bajo la influencia del comando `umask 022` tendrán permisos `644` por defecto y los directorios tendrán permisos `755` por defecto, asegurando que los mismos sólo puedan ser creados y borrados por su propietario, pero legibles y ejecutables por cualquier otro usuario.
- `LC_ALL=POSIX`: la variable de entorno `LC_ALL`, sobreescribe el valor del resto de variables de entorno relacionadas con la configuración del idioma. El valor `POSIX` o `C` obliga a los programas a utilizar el idioma por defecto para mostrar mensajes por pantalla y asegura el correcto funcionamiento del sistema dentro del *chroot* al construir el sistema operativo final.
- `LFS_TGT=$(uname -m)-lfs-linux-gnu`: la variable de entorno `LFS_TGT` proporciona información sobre la arquitectura y el tipo de máquina a usar para construir el *toolchain* temporal del apartado *2.1.2 Toolchain*.

[16] más información en el apartado *2.1.2 Toolchain* de la página 17

- `PATH=/tools/bin:/bin:/usr/bin`: la variable de entorno `PATH` define los directorios en que se buscarán los binarios requeridos para la ejecución de un comando. Al añadir el directorio `/tools/bin` a la variable de entorno `PATH`, todos los programas generados e instalados en el *toolchain* estarán disponibles por el intérprete de línea de comandos.

Los comandos necesarios para configurar este entorno están recogidos en el código 2.4 y pueden ejecutarse directamente en el terminal de línea de comandos en que se va a construir el sistema, o añadirlos al archivo de configuración `${HOME}/.bashrc` del usuario sin privilegios que vayamos a utilizar para construir el *toolchain* consiguiendo que las variables de entorno estén configuradas de una sesión a otra en caso de que no vayamos a construirlo de una sola vez. La variable de entorno `LFS` definida anteriormente vuelve a definirse aquí por comodidad.

```
# Desactivar la caché de localización de binarios
set +h

# Definir la máscara de creación de ficheros
umask 022

# Definir la variable de entorno 'LC_ALL'
LC_ALL=POSIX
export LC_ALL

# Definir la variable de entorno 'LFS'
LFS=/mnt/lfs
export LFS

# Definir la variable de entorno 'LFS_TGT'
LFS_TGT=$(uname -m)-lfs-linux-gnu
export LFS_TGT

# Definir la variable de entorno 'PATH'
PATH=/tools/bin:/bin:/usr/bin
export PATH
```

Código 2.4 Otras variables de entorno y opciones del intérprete de línea de comandos.

A continuación, el libro crea el directorio `$LFS/sources` para descargar dentro los archivos con el código fuente de los distintos componentes del sistema operativo y los parches necesarios. Además, el libro aconseja activar el *sticky bit*[17] en dicho directorio para evitar que los archivos con el código fuente sean eliminados por otros usuarios. Así mismo, el libro *Linux from Scratch* provee una lista con todos los enlaces necesarios para descargar los distintos paquetes de código fuente y parches. Este archivo, de nombre `wget-list`[18] puede descargarse de internet con el comando `wget`. Los comandos necesarios para crear y descargar los archivos del código fuente en el directorio `$LFS/sources` se encuentran en el código 2.5.

[17] sticky bit: bit de permisos Linux que, activado en un determinado directorio, permite únicamente al usuario dueño del directorio o al usuario *root*, borrar o renombrar los archivos del directorio.

[18] wget-list: la lista de URLs para la última versión estable del libro *Linux from Scratch* se encuentra en la URL *http://www.linuxfromscratch.org/lfs/view/stable/wget-list*

```
# Crear el directorio '$LFS/sources'
mkdir -v ${LFS}/sources

# Activar el sticky bit en el directorio '$LFS/sources'
chmod -v a+wt ${LFS}/sources

# Descargar todos los archivos cuyas URLs
# se encuentran en el fichero 'wget-list'
wget --input-file=wget-list --continue --directory-prefix=${LFS}/sources
```

Código 2.5 Añadir el directorio `sources` y descargar los archivos necesarios.

Por último, es necesario crear el directorio `tools` dentro del punto de montaje referenciado por la variable de entorno `LFS` para construir el *toolchain* dentro de este directorio. También es necesario crear un enlace simbólico a este directorio en el directorio raíz del sistema operativo anfitrión para que los distintos binarios generados en el *toolchain* estén disponibles durante la creación del propio *toolchain*. Los comandos requeridos para cumplir estos requisitos pueden encontrarse en el código 2.6.

```
# Crear el directorio '${LFS}/tools'
mkdir -v ${LFS}/tools

# Crear un enlace simbólico al directorio raíz
ln -sv ${LFS}/tools /
```

Código 2.6 Añadir el directorio `tools` y crear el enlace simbólico necesario.

2.1.2 Toolchain

Un *toolchain* informático es un conjunto de herramientas y programas que se utilizan como base para construir o crear otro producto informático diferente. En este apartado del libro se construyen todos los programas necesarios para compilar el resto del sistema dentro del entorno generado en la sección anterior.

Estos programas se usan como herramientas temporales y no formarán parte del sistema operativo final. Esto es así para independizar el sistema operativo creado del anfitrión, en caso de que las herramientas propias del sistema operativo anfitrión hayan sido modificadas o personalizadas de alguna manera, bien por el usuario de dicho sistema operativo o bien por la comunidad de desarrolladores del mismo.

Todos los programas compilados en esta fase se instalaran en el directorio `${LFS}/tools` creado en el código 2.6 para mantenerlos separados tanto de los programas que conforman el sistema operativo anfitrión, como de los programas que serán compilados en el siguiente bloque del diagrama 2.1 de la página 12, el *sistema*, y que formarán parte del sistema operativo *Linux from Scratch*. Esto, unido al uso del comando `set +h` y haber incluido el directorio `tools` como primera opción en la variable de entorno `PATH`, hacen que los nuevos binarios sean utilizados a medida que están disponibles.

Las herramientas que conforman el *toolchain* de *Linux from Scratch* se construyen en dos fases. En una primera fase se construyen los componentes *binutils*, *gcc*, la *API* del *kernel* y *glibc*, que además resuelven una serie de dependencias circulares en el resto del proceso. En la segunda fase se construyen los verdaderos componentes del *toolchain*, incluidos una nueva versión de los componentes antes mencionados.

Binutils instala su propio ensamblador y enlazador en dos localizaciones diferentes del directorio `tools` usando enlaces no simbólicos. Al instalar *binutils* primero conseguimos que los componentes *gcc* y *glibc* realicen su configuración en función de las características de este *binutils* y no del que está disponible en el sistema operativo anfitrión. Esto es más importante de lo que parece porque es un paso fundamental en la independencia de *Linux from Scratch* frente a la distribución anfitriona. Construir el *toolchain* usando unos

mal configurados *gcc* y *glibc* puede derivar en un *toolchain* inservible para construir el resto del sistema.

Después de instalar los componentes *binutils* y *gcc*, instalamos en el directorio `${LFS}/tools` la *API* y las librerías del *kernel* de Linux para que el componente *glibc* que se instala posteriormente se configure para usar las funcionalidades proporcionadas por dichas librerías. Este componente se configura con una serie de parámetros que hacen referencia a los componentes construidos anteriormente. Una vez se ha construido *glibc*, volvemos a construir los componentes *binutils* y *gcc*. Llegados a este punto, el *toolchain* está lo suficientemente aislado del sistema operativo anfitrión como para poder construir el resto de componentes que lo forman sin depender del sistema anfitrión.

La mayor parte del proceso restante de contrucción del *toolchain* no tiene nada de especial, es sólo cuestión de seguir las instrucciones proporcionadas por el libro. En los últimos pasos del capítulo, se remueven algunos elementos innecesarios de los binarios producidos para reducir el espacio que ocupa este *toolchain* en la partición del disco duro y se cambia el dueño del directorio `tools` a `root` para evitar que un futuro usuario del sistema operativo *Linux from Scratch*, recordemos que el usuario sin privilegios `lfs` creado en el código 2.1 de la página 13 no existe en el sistema final en este punto, con el mismo *ID* de usuario se convertiría en el dueño de dichos binarios, siendo posible realizar un uso malicioso de los mismos.

Hay que destacar una característica de este proceso que también proporciona aislamiento frente al sistema anfitrión. Este proceso es propio de la compilación cruzada, en la que los programas instalados bajo el mismo directorio cooperan entre ellas para construirse conjuntamente. Así, la variable `LFS_TGT` que define la arquitectura objetivo para la que se compilan los programas, asegura que la primera compilación de los componentes *binutils* y *gcc* producen herramientas compatibles con otras arquitecturas. En nuestro caso concreto, y según el valor que se le otorgó a dicha variable de entorno en el código 2.4 de la página 16, se generan unos binarios para la misma arquitectura en la que se construye. Aprovechando también el hecho de que, por definición, unas herramientas producidas con compilación cruzada no pueden depender en modo alguno del sistema anfitrión, conseguimos aislar el *toolchain*, que era uno de los objetivos principales. Así, podríamos decir que este es un caso particular de compilación cruzada.

2.1.3 Sistema

Este bloque del diagrama de flujo 2.1 de la página 12 es el que se encarga de construir el sistema operativo final. Es, como puede verse en el mencionado diagrama, el primero de los bloques que se construyen aislado del sistema. Recordemos que gran parte de este aislamiento se ha conseguido al construir el *toolchain*, y la parte que falta se consigue cambiando la raíz del sistema con el comando `chroot` y configurando ese entorno. Hay que puntualizar aquí varias cosas:

1. Este bloque modifica el *toolchain* hasta el punto de hacerlo inservible para futuros procesos de construcción. Si se desea usar el mismo *toolchain* para construir distintos sistemas *Linux from Scratch* con el fin de evitar tener que construir el *toolchain* múltiples veces, es necesario guardar una copia de seguridad del mismo antes de empezar a construir los componentes que forman este bloque.
2. El usuario `root` también debe configurar las variables de entorno `LFS` y `LFS_TGT`.
3. La variable de entorno `PATH` toma un valor diferente para, al igual que se hacía en los apartados anteriores, ir utilizando los nuevos binarios a medida que están disponibles. Así, el directorio `/tools/bin` pasa de estar en primera posición a la última, consiguiendo así que los binarios construidos en el *toolchain* sólo se utilicen como último recurso. El nuevo valor de la variable `PATH` es `PATH=/bin:/usr/bin:/sbin:/usr/sbin:/tools/bin`.
4. Al *chroot* se entra haciendo uso del commando `/tools/bin/env -i` para entrar al *chroot* con un entorno limpio, sin verse influenciado por el entorno del usuario del sistema anfitrión que llamó al comando `chroot`.
5. Dentro del *chroot*, el *prompt* del intérprete de comandos mostrará el mensaje `I have no name!` hasta que el fichero `/etc/passwd` sea creado.

Una vez se haya completado este bloque, el *toolchain* ya no es necesario, por lo que el directorio `${LFS}/tools` puede borrarse. Esto elimina algunos binarios que son necesarios para construir programas proporcionados por el libro *Beyond Linux from Scratch*. Sin embargo, en caso de ser necesarios, pueden volver a construirse dichas dependencias con las instrucciones contenidas en dicho libro.

2.1.4 Configuración y gestión de arranque

El último bloque del diagrama 2.1 de la página 12 configura el sistema creado en el bloque anterior y también configura el proceso de arranque del sistema *Linux from Scratch*. Como en este bloque el *toolchain* ya no es necesario y es posible que se haya borrado el directorio `tools`, en este bloque utilzaremos un nuevo valor para la variable de entorno `PATH` sin el directorio que contenía el *toolchain*. Así, el valor de la variable de entorno `PATH` es `PATH=/bin:/usr/bin:/sbin:/usr/sbin`.

Arrancar un sistema Linux es un proceso complejo. Es necesario montar múltiples sistemas de archivos virtuales y físicos, iniciar los dispositivos de almacenamiento y de entrada y salida, montar las particiones *swap*, comprobar la integridad de los sistemas de archivos, fijar la hora del reloj del sistema, configurar y activar los dispositivos de red, y ejecutar cualquier otra tarea que el administrador del sistema considere necesaria. Todos estos procesos deben organizarse de alguna manera para garantizar que se ejecutan en el orden adecuado. El diagrama de flujo que representa los pasos necesarios para arrancar un sistema operativo está reflejado en la figura 2.2.

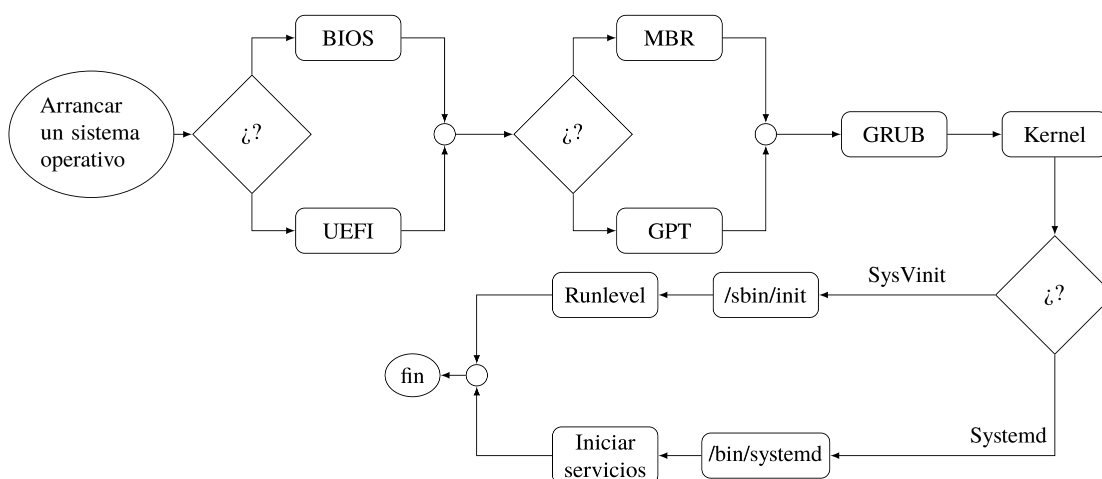


Figura 2.2 Diagrama de flujo para el arranque de un sistema operativo.

BIOS, *Basic Input/Output System*, es el *firmware*[19] que se instala en un ordenador y que se ejecuta en primer lugar cuando éste se enciende. Realiza algunas comprobaciones de integridad en el sistema, lo que se conoce como proceso *POST*, siglas de *Power On Self Test*. Tras estas comprobaciones, el *BIOS* busca un cargador de arranque *MBR* o *GPT* en alguno de los dispositivos de almacenamiento del sistema, como los discos duros, el lector de CD o los puertos *USB*[20]. Este orden puede modificarse desde el menú de configuración accesible durante el encendido de la máquina

UEFI, o *Unified Extensible Firmware Interface*, es una versión más moderna de *BIOS* desarrollada por el *Unified EFI Forum* para convertirlo en el *firmware* por defecto. *UEFI* es capaz de realizar diagnósticos y reparaciones a través de la red, y además *UEFI* tiene mayor número de funcionalidades que *BIOS*, como funciones criptográficas o un intérprete de comandos con el que ejecutar otras aplicaciones o actualizar el propio *firmware*. Tanto la arquitectura como los *drivers* son independientes del tipo de *CPU* del que disponga la máquina, incluida la arquitectura ARM, *Advanced RISC Machines*, ordenadores con un conjunto

[19] firmware: *programa informático que controla los circuitos electrónicos de un cierto dispositivo a muy bajo nivel.*

[20] USB: *estándard de comunicación entre ordenadores, periféricos y otros dispositivos electrónicos que permite compartir información y alimentación eléctrica. Más información en la URL http://www.usb.org/home*

de instrucciones reducido. Una vez que *UEFI* ha realizado las comprobaciones necesarias busca y ejecuta un cargador de arranque al igual que *BIOS*.

En resumen podríamos decir que el *firmware* del equipo se dedica a comprobar los dispositivos físicos y a ejecutar el cargador de arranque del dispositivo elegido.

El *Master Boot Record*, o *MBR*, es un sector especial del disco duro, normalmente posicionado en la primera partición del mismo y con un tamaño inferior de 512 *bytes*. Contiene el código necesario para realizar comprobaciones de integridad, información sobre las diferentes particiones del disco duro y la información para inicializar el gestor de arranque, siendo *GRUB* el más utilizado. Otro gestor de arranque conocido, y anterior a *GRUB*, es *LILO*. Actualmente, el *MBR* está siendo reemplazado por *GPT* como cargador de arranque por defecto.

GPT, o *GUID Partition Table*, es un estándard de particionado de discos duros usando identificadores globales únicos. Forma parte del *firmware UEFI*, pero puede ser usado también con un *firmware* de tipo *BIOS* debido a las limitaciones de tamaño de las tablas de partición de *MBR*. Comparado con *MBR*, *GPT* puede manejar discos duros de tamaño superior a los 2 *TB*, dos *terabytes*, y hasta 128 particiones primarias en lugar de las cuatro que es capaz de manejar *MBR*.

GRUB son las siglas de *GRand Unified Bootloader* y es el gestor de arranque más utilizado por las diferentes distribuciones Linux. Este programa permite elegir qué versión o imagen del *kernel* Linux instalada quiere ejecutarse mediante el menú de selección que aparece al principio de su ejecución. El componente *GRUB* se instala en el directorio `/boot` y su archivo de configuración es el fichero `/boot/grub/grub.cfg`. Antes de configurar *GRUB*, es necesario instalarlo en el primer sector del disco duro, donde residen el *MBR* o *GPT* para que sea reconocido y ejecutado por el cargador de arranque seleccionado en el bloque anterior del diagrama 2.2 de la página 19.

El proceso de instalación de *GRUB* es peligroso, ya que podría inutilizar el proceso de arranque. Para contrarrestar este inconveniente, el libro *Linux from Scratch* recomienda grabar una copia de seguridad del *GRUB*. Los pasos a seguir para realizar esta copia de seguridad se recogen en el código 2.7 y hace uso del comando `grub-mkrescue` que fue instalado al construir el componente *GRUB* durante la construcción del sistema en la sección 2.1.3.

```
# Cambiar al directorio /tmp
cd /tmp

# Realizar la copia de seguridad usando 'grub-mkrescue'
grub-mkrescue --output=grub-img.iso
```

Código 2.7 Hacer una copia de seguridad del *GRUB* con el programa `grub-mkrescue`.

Una vez ha terminado el proceso de grabación de la copia de seguridad anterior podemos seguir con el proceso de instalación de *GRUB*. *GRUB* usa su propio formato de nombres para los discos duros y particiones. Esta convención sigue el patrón `(hdX,Y)`, donde *X* es número del disco duro e *Y* es el número de la partición, empezando ambos valores por el número cero. Así, nuestra partición principal `/dev/sdb1` se corresponde con el valor `(hd1,1)`, ya que al disco `/dev/sda` le corresponde el identificador `(hd0)`. Para instalar el componente *GRUB* en el disco `/dev/sdb` es necesario ejecutar el comando `grub-install` del código 2.8.

```
# Instalar GRUB en el disco /dev/sdb
grub-install /dev/sdb
```

Código 2.8 Instalar *GRUB* en el disco `/dev/sdb` usando el programa `grub-install`.

Por último sólo sería necesario configurar el gestor de arranque *GRUB* modificando, o escribiendo uno nuevo en caso de que no existiera previamente, su archivo de configuración `/boot/grub/grub.cfg`. Este archivo de configuración se compone de una serie de opciones que controlan el comportamiento de *GRUB* y de su menú más una serie de entradas, denominadas *menuentry*, que definen los comandos necesarios para inicializar la versión del *kernel* de un determinado sistema operativo, en el que se especifica también la partición en que se aloja, o para definir otros comportamientos personalizados, como por ejemplo apagar el equipo.

El código 2.9 muestra el contenido del archivo de configuración `/boot/grub/grub.cfg` con dos entradas. La primera de estas entradas permiten ejecutar el *kernel* 4.15.3 construido para la versión 8.2 del sistema operativo *Linux from Scratch* alojado en la ruta `/boot/vmlinuz-4.15.3-lfs-8.2` de la partición `/dev/sdb1`. La otra entrada permite apagar el equipo ejecutando el comando `halt`.

```
# Begin /boot/grub/grub.cfg
set default=0
set timeout=5

insmod ext2
set root=(hd1,1)

menuentry "GNU/Linux, Linux 4.15.3-lfs-8.2" {
        linux   /boot/vmlinuz-4.15.3-lfs-8.2 root=/dev/sdb1 ro
}

menuentry "Apagar el equipo" {
    echo "Apagando el equipo..."
    halt
}
```

Código 2.9 Ejemplo de archivo de configuración `/boot/grub/grub.cfg`.

Siguiendo con el proceso indicado en el diagrama 2.2 de la página 19, una vez que el gestor de arranque *GRUB* ha inicializado el *kernel*, le cede a éste el control de la máquina. El *kernel* es el núcleo del sistema operativo Linux, y se trata de un software gratuito y de código abierto escrito en lenguaje C que fue desarrollado por Linus Torvalds en 1991. Actualmente es desarrollado y mantenido por la comunidad de desarrolladores que forman la *Linux Kernel Organization*, que a su vez forma parte de la *Linux Foundation*. Entre las funcionalidades del *kernel* se encuentran las de compartición de librerías, gestión de memoria física y virtual, la multitarea, y todo lo relacionado con los dispositivos y protocolos de red.

La primera tarea que realiza el *kernel* cuando toma el control de la máquina es inicializar el planificador del sistema con el *PID*, identificador de proceso, número cero. A continuación monta el sistema de archivos indicado por el parámetro `root=/dev/sdb1` en el directorio raíz o `/` e inicia el proceso correspondiente al sistema de inicio, *SysVinit* o *Systemd*, que inicia los demonios o servicios del sistema, como la red o el servidor de impresión *CUPS*[21] y monta el resto de sistemas de ficheros recogidos en el fichero de configuración `/etc/fstab`.

El libro *Linux from Scratch* gestiona el proceso de cargar el *kernel* de Linux usando el gestor de arranque *GRUB*, concretamente la versión denominada *GRUB Legacy*, y una vez que el *kernel* se ha cargado, el proceso

[21]CUPS: *sistema de impresión de código abierto que utiliza el protocolo de impresión Internet Printing Protocol, IPP. Más información en la URL https://www.cups.org/*

continúa usando uno de los dos principales sistemas de inicio: *SysVinit* o *Systemd*. Además, el proyecto *Linux from Scratch* dispone de una versión de sus libros *Linux from Scratch* y *Beyond Linux from Scratch* para cada sistema de arranque.

SysVinit era el sistema de inicio predeterminado en la mayoría de distribuciones Linux hasta el año 2015 en que fue mayoritariamente sustitutido por *Systemd*. Es el gestor de inicio por defecto en *Linux from Scratch*, aunque también está disponible otra versión del libro que emplea *Systemd* en su lugar.

SysVinit está formado por el programa `/sbin/init`. Es un proceso demonio iniciado por el *kernel* en el momento del arranque con el ID de proceso número 1. El fichero de configuración principal es el archivo `/etc/inittab` y el *script* de inicio principal es `/etc/rc.d/rc.sysinit`, que configura el entorno básico del sistema, inicia la memoria de intercambio *swap* y ejecuta el resto de pasos requeridos para la inicialización del sistema, cambiando también al nivel de ejecución por defecto. *SysVinit* define siete niveles de ejecución diferentes. Los nombres y la característica principal de cada nivel de ejecución se resume en la tabla 2.4.

Runlevel	Nombre	Características
0	Apagado	Apaga el sistema. No marcar como nivel por defecto.
1	Usuario único	El sistema arranca como usuario administrador para realizar tareas de mantenimiento. En este nivel de ejecución, la red no está disponible.
2	Multiusuario	Modo multiusuario sin capacidades de red.
3	Multiusuario completo	Igual que el modo número dos pero con funciones de red.
4	Sin uso	Disponible para ser definido por el usuario
5	X11	Modo multiusuario con servidor gráfico.
6	Reinicio	Reinicia el sistema. No marcar como nivel por defecto.

Tabla 2.4 Descripción de los niveles de ejecución del gestor de inicio *SysVinit*.

Los distintos niveles de ejecución definen una serie de estados del sistema compuestos por los servicios listados en los diferentes directorios `/etc/rc.d/rc{0..6}.d`. Estos directorios contienen enlaces simbólicos a los *scripts* contenidos en el directorio `/etc/init.d`. El nombre de estos enlaces sigue el patrón `[K,S][0..99][nombre]` dónde la letra `K` significa *kill*, e instruye al proceso `init` a que pare el servicio especificado por *nombre*. La letra `S` por el contrario, significa *start*, e instruye al proceso `init` a que inicie el servicio *nombre*. El número comprendido entre 0 y 99 indica el orden en que debe llamarse al *script* de servicio *nombre*.

Si el sistema de inicio en uso es *SysVinit*, la ejecución del comando `ps -p 1` en el intérprete de línea de comandos muestra el contenido del código 2.10.

```
# Proceso número 1 de un sistema configurado con SysVinit
$ ps -p 1
  PID TTY          TIME CMD
    1 ?        00:00:00 init
```

Código 2.10 Salida del comando `ps -p 1` para *SysVinit*.

Además de ejecutar el nivel de ejecución requerido, el programa `/sbin/init` monta en el sistema el resto de sistemas de ficheros listados en el archivo `/etc/fstab`. Este archivo contiene la información necesaria para montar un sistema de archivos concreto en una ruta concreta y con unas opciones específicas. También puede definir los sistemas de archivo que se montan a través de la red utilizando el protocolo de sistemas de archivos distribuidos *NFS*[22]. Las líneas del fichero `/etc/fstab` siguen la estructura `[sistema de archivos] [punto de montaje] [formato] [opciones] [dump] [pass]`. Cada una de estas opciones tienen un significado y formato específicos:

- **sistema de archivos**: este atributo identifica el sistema de archivos que se desea montar. La identificación se realiza utilizando la ruta de la partición, por ejemplo `/dev/sdb1` o un *UUID*, acrónimo de *Universal Unique Indentifier*, un número de identificación de 128 bits. En este último caso, el parámetro se escribe en el archivo de configuración con el prefijo `UUID=`.
- **punto de montaje**: es la ruta absoluta al directorio destino en que se montará un sistema de archivos.
- **formato**: indica el formato del sistema de archivos en que se va a montar. Formatos comunes son `ext2`, `ext3`, `ext4`, `vfat`, `swap`, `proc` o `nfs` entre otros.
- **opciones**: algunas opciones comunes definen el comportamiento del sistema de archivos montado como los permisos de lectura y escritura, si debe ser automáticamente montado durante el inicio del sistema, qué hacer si se produce algún error, si la escritura en el sistema de archivos es síncrona o asíncrona, o si se permite la ejecución de binarios, entre otras. Puede obtenerse más información sobre las opciones disponibles en la entrada del manual de Linux del comando `mount`, concretamente la accesible en su sección número ocho mediante el comando `man 8 mount`.
- **dump**: opción binaria en la que `1` significa *verdadero* y `0` significa *falso*, para activar un método de copia de seguridad en caso de error. Este parámetro debe, por lo general, tomar el valor *falso*.
- **pass**: opción binaria que indica si deben realizarse comprobaciones de integridad sobre el sistema de archivos o no. El sistema de archivos montado en el directorio raíz siempre debe tener esta opción activada.

Un ejemplo de archivo `/etc/fstab` para el sistema de inicio *SysVinit* se muestra en el código 2.11.

```
# Begin /etc/fstab

# file system mount-point type   options             dump  fsck
#                                                          order

/dev/sdb1    /            ext4    defaults             1     1
/dev/sdb2    swap         swap    pri=1                0     0
proc         /proc        proc    nosuid,noexec,nodev  0     0
sysfs        /sys         sysfs   nosuid,noexec,nodev  0     0
devpts       /dev/pts     devpts  gid=5,mode=620       0     0
tmpfs        /run         tmpfs   defaults             0     0
devtmpfs     /dev         devtmpfs mode=0755,nosuid    0     0

# End /etc/fstab
```

Código 2.11 Ejemplo de archivo `/etc/fstab` para *SysVinit*.

Systemd[23] nació como sustituto de *SysVinit* para convertirse en el sistema de inicio común entre las diferentes distribuciones Linux. Su binario principal es `systemd` y está alojado en el directorio `/bin`. Su fichero principal de configuración se encuentra en el directorio `/etc/systemd` y se llama `system.conf`.

[22]NFS: *Networking File System.*

[23]systemd: visite su página principal para más información *https://www.freedesktop.org/wiki/Software/systemd/*

Entre sus principales características destacan la paralelización de ejecución de servicios, el uso de *Linux sockets*[24] y *Desktop Bus* (D-Bus)[25] para inicializar los servicios que componen el arranque del sistema operativo, la posibilidad de iniciar, parar y reiniciar servicios del sistema bajo demanda, mantener el seguimiento de los procesos que usan *cgroups*, y es compatible con dos tipos de *scripts* de inicio: los propios de *SysVinit* y los *Linux Standard Base*, *LSB*. A parte de todo lo anterior, *Systemd* también implementa un sistema de gestión de dependencias entre servicios.

Systemd implementa varios comandos a ejecutar con la utilidad de control `systemctl`. Algunos de estos comandos están recogidos en la tabla 2.5[26]:

Nombre	Descripción
list-units	Lista las unidades cargadas en el sistema.
list-sockets	Muestra los *sockets* en uso.
start	Arranca la unidad, o servicio, indicado.
stop	Para el servicio indicado.
reload	Recarga la configuración de un determinado servicio.
restart	Inicia, o reinicia si ya estaba en ejecución, el servicio indicado.
kill	Envía la señal indicada como parámetro al servicio deseado.
status	Muestra el estado actual del servicio indicado.

Tabla 2.5 Resumen de comandos del programa `systemctl`.

Si el sistema de inicio en uso es *Systemd*, la ejecución del comando `ps -p 1` en el intérprete de línea de comandos muestra el contenido del código 2.12.

```
# Proceso número 1 de un sistema configurado con Systemd
$ ps -p 1
  PID TTY          TIME CMD
    1 ?        00:00:00 systemd
```

Código 2.12 Salida del comando `ps -p 1` para *systemd*.

Los servicios, o *unidades*, que componen *Systemd* se definen como archivos de configuración, habitualmente con extensión `.service`, que incluyen los parámetros necesarios para iniciar, parar y configurar el servicio. A modo de ejemplo, el fichero que controla el servicio de conexión remota *SSH*, `ssh.service`, se muestra en el código 2.13.

[24]linux sockets: *un socket Linux es un fichero utilizado para la intercomunicación entre procesos de una misma máquina (IPC).*

[25]D-Bus: software para la intercomunicación entre procesos (IPC) y llamadas de procedimiento remoto (RPC).

[26]más información ejecutando el comando `systemctl --help`

```
[Unit]
Description=OpenBSD Secure Shell server
After=network.target auditd.service
ConditionPathExists=!/etc/ssh/sshd_not_to_be_run

[Service]
EnvironmentFile=-/etc/default/ssh
ExecStart=/usr/sbin/sshd -D $SSHD_OPTS
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartPreventExitStatus=255
Type=notify

[Install]
WantedBy=multi-user.target
Alias=sshd.service
```

Código 2.13 Contenido del fichero `sshd.service` para la gestión del servicio *SSH* en *Systemd*.

Por último, un ejemplo del archivo de configuración `/etc/fstab` para el sistema de inicio *Systemd* se muestra en el código 2.14

```
# Begin /etc/fstab

# file system mount-point type   options          dump  fsck
#                                                        order

/dev/sdb1    /           ext4    defaults          1     1
/dev/sdb2    swap        swap    pri=1             0     0

# End /etc/fstab
```

Código 2.14 Ejemplo de archivo `/etc/fstab` para *Systemd*.

2.1.5 El interés de automatizar el proceso

Una vez leído, comprendido, y construidos todos los componentes que forman cada uno de los bloques *toolchain*, *sistema* y *configuración y gestión de arranque* del diagrama 2.1 de la página 12 y que componen el sistema operativo *Linux from Scratch*, se hace evidente que el proceso es largo, tedioso y repetitivo.

El tiempo estimado de construcción del sistema operativo resultado del libro *Linux from Scratch* es de 190 minutos en una arquitectura de 64 *bits*. Esto es, 3 horas y 10 minutos, una cantidad de tiempo demasiado grande para ser empleado en un proceso repetitivo cuyo objetivo no sea el aprendizaje sino la necesidad, como se comentó en el apartado *1.2.2 Necesidad para desarrolladores y administradores de sistemas* de la página 4.

En ese caso, emplear más de 3 horas en un proceso de este tipo por necesidad puede suponer una pérdida de tiempo y la aparición de sentimientos negativos tales como la pereza o el hastío. Recordemos que el proceso conlleva, para cada componente que forman los distintos bloques de la figura 2.1 de la página 12 la realización de las siguientes tareas:

1. Descomprimir el código fuente correspondiente al componente que se desea construir.
2. Aplicar los parches necesarios en el código fuente.
3. Ejecutar otros comandos para preparar el código fuente o el entorno de construcción.
4. Construir el componente, generalmente ejecutando los comandos `configure`, `make` y `make install`.
5. Configurar el componente instalado.
6. Borrar el código fuente descomprimido.

Los pasos anteriores pueden parecer pocos y sencillos, pero el libro está compuesto por un total de 121 componentes, lo que convierte el proceso en la realización de, aproximadamente, 600 operaciones debido a que no todos los componentes requieren compilación ni realizar todos los pasos listados anteriormente. Esto, unido al tiempo requerido y a la necesidad de tener los componentes del sistema actualizados cada cierto tiempo, convierte la construcción del sistema operativo *Linux from Scratch* en un candidato idóneo para la automatización y sus ventajas principales, como la realización de tareas repetitivas minimizando esfuerzos y el desarrollo de procesos sin la intervención directa del ser humano, donde sólo la supervisión o el mantenimiento periódico son requeridos.

Para este nuevo contexto, el proyecto *Linux from Scratch* dispone del subproyecto *Automated Linux from Scratch*, que implementa la herramienta *jhalfs* para la automatización de construcción del libro *Linux from Scratch*. Véase el apartado *2.3 Automated Linux from Scratch* de la página 27 para más información al respecto.

2.2 Beyond Linux from Scratch

Después de construir el sistema *Linux from Scratch*, el sistema operativo resultado es un sistema Linux que podemos considerar completo, pero básico. La configuración de red, por ejemplo, se realiza de manera estática para una interfaz de tipo *Ethernet 802.3*, pero no se dice nada de dispositivos *Ethernet 802.11*, el protocolo conocido comúnmente como *Wi-Fi*, ni la configuración dinámica usando el protocolo *DHCP*, o *Dynamic Host Configuration Protocol*, un protocolo de configuración dinámica de direcciones de red definido en la RFC 2131 [27]. Tampoco se instalan las utilidades para la compartición de sistemas de archivos en red, *NFS* o *SAMBA*[28], ni la herramienta de conexión remota *OpenSSH* o el servidor de impresión *CUPS*. Algunas de estas utilidades son prácticamente usadas a diario por cientos de desarrolladores y administradores de sistemas de todo el mundo. Para cubrir estas necesidades, y otras muchas como por ejemplo la disponibilidad de un entorno de escritorio, y contribuyendo a la personalización del sistema operativo básico creado en la sección anterior, el proyecto *Linux from Scratch* desarrolla el subproyecto *Beyond Linux from Scratch*.

Al igual que el anterior, *Beyond Linux from Scratch* pone a disposición de los usuarios un libro con las instrucciones de construcción y configuración de los diferentes componentes, además de los ficheros *XML* con los que compilar el libro. El proceso para la construcción de las diferentes utilidades sigue los mismos pasos que para los componentes del libro *Linux from Scratch*, con el paso añadido de que es necesario descargar los ficheros comprimidos del código fuente como un paso extra para cada componente, ya que no es necesario instalar todos los componentes que forman este libro, sino sólo aquellos que sean de interés.

En cuanto a las dependencias de los componentes, el libro proporciona tres tipos de dependencias para cada uno ordenadas de mayor a menor obligatoriedad, y deja a expensas del usuario cumplir las dependencias recomendadas y opcionales.

2.2.1 El interés de automatizar el proceso

Una vez más, nos encontramos ante un proceso repetitivo, tanto o más extenso que el del libro *Linux from Scratch*, pero que también es candidato a ser automatizado incluso en el caso de que sólo se desee construir parcialmente. Es aún más susceptible de ser automatizado si se pretende construir ambos subproyectos de manera consecutiva.

[27] RFC 2131: *https://tools.ietf.org/html/rfc2131*

[28] SAMBA: *implementación open source del protocolo SMB para la compartición de sistemas de archivos entre Windows y Linux*

Por contra, automatizar el subproyecto *Beyond Linux from Scratch* es una tarea muy complicada y extensa debido a que los componentes no se presentan de manera lineal como en el libro *Linux from Scratch*, hay componentes que no se instalan en los directorios predeterminados, existen dependencias circulares cuya solución puede no ser trivial, etc.

Así, el subproyecto *Automated Linux from Scratch* implementa en la herramienta *jhalfs* la posibilidad de crear los *scripts* para construir dichos componentes y el archivo *Makefile* que gobierna la ejecución de los diferentes *scripts* con el añadido de que la herramienta *make* intenta ayudar con la gestión de dependencias entre componentes. Véase el apartado *2.3 Automated Linux from Scratch* para más información al respecto.

2.3 Automated Linux from Scratch

El subproyecto *Automated Linux from Scratch* ha creado una herramienta, llamada *jhalfs*, para automatizar la construcción de los libros *Linux from Scratch* y *Beyond Linux from Scratch* que además es utilizada por los desarrolladores del proyecto como herramienta oficial para realizar pruebas de funcionamiento con los comandos de las diferentes versiones del libro *Linux from Scratch*.

La herramienta *jhalfs*[29] está compuesta por una serie de *scripts* escritos en el lenguaje de programación *Bash*, utilizado por el intérprete de comandos Linux del mismo nombre. Estos *scripts* contienen un conjunto de comandos que se ejecutan de manera secuencial. Además, `bash` es el intérprete de comandos por defecto para la mayoría de distribuciones Linux, por lo que la herramienta funciona por defecto en la mayoría de distribuciones. Como otros requisitos, *jhalfs* depende de varias herramientas para descargar el código fuente del libro, los archivos comprimidos con el código fuente y extraer los comandos a ejecutar. A groso modo, estas dependencias son los programas `svn`[30], `xsltproc` y `wget`. Para mostrar los diálogos del menú de configuración de la herramienta, el programa *jhalfs* hace uso del programa `dialog` y la librería `ncurses`.

El menú de configuración de la herramienta *jhalfs* presenta varias opciones: *ajustes del libro*, *ajustes generales*, *opciones de construcción*, *configuración del sistema* y *opciones avanzadas*.

El submenú *ajustes del libro* permite elegir qué libro y versión se desea utilizar, tanto el subproyecto como la versión del sistema de inicio a emplear. En el caso de construir el libro *Linux from Scratch* es posible añadir soporte para el libro *Beyond Linux from Scratch*. También está disponible la opción de dar soporte para construir componentes propios después de la construcción de cualquiera de los libros.

El submenú *ajustes generales* se encarga de configurar el directorio base de la instalación, y de elegir si la construcción empezará justo después de configurar la herramienta y generar los archivos necesarios o no. El submenú *opciones de construcción*, permite utilizar un archivo `fstab` propio en lugar del generado por el libro, y decidir si el *kernel* de Linux será construido o no durante el proceso de construcción del sistema. También añade la posibilidad de utilizar un gestor de paquetes, por ejemplo `dpkg`, creando los ficheros de descripción típicos de dicho gestor de paquetes a la hora de instalar los componentes del sistema final. Este submenú también ofrece la posibilidad de crear ficheros de *log* con los mensajes generados durante el proceso de construcción de cada componente.

En el submenú de *configuración del sistema* se definen la zona horaria o el idioma en los que se configurará el sistema operativo generado, el archivo de configuración con el que compilar el *kernel* y la configuración de red. El submenú de *opciones avanzadas* está diseñado para tareas de mantenimiento y optimización de la compilación como el número de trabajaos que el comando `make` puede ejecutar en paralelo.

Una vez configurada la herramienta *jhalfs*, ésta crea en el directorio de instalación, digamos `/mnt/lfs`, una serie de directorios y ficheros adicionales: *jhalfs*, `sources` y `tools`. El directorio `tools` contiene los parches y ficheros comprimidos con el código fuente de los programas a instalar, y el directorio `tools`

[29]jhalfs: *puede descargarse el código fuente de la herramienta jhalfs ejecutando el comando* `svn co svn://svn.linuxfromscratch.org/ALFS/jhalfs/trunk jhalfs`

[30]svn: binario del sistema de control de versiones *Subversion*.

es el directorio en el que se instalará el *toolchain*. Por su parte, el directorio *jhalfs* es el directorio principal de la herramienta *jhalfs*. Este directorio contiene los *scripts* de *bash* con las funciones necesarias para construir los componentes y un archivo *Makefile* que gobierna la construcción del sistema *Linux from Scratch* en el orden correcto. También pueden encontrarse en este directorio los comandos que se van a ejecutar, separados por capítulo y componente. Estos *scripts* pueden editarse para personalizar los diferentes aspectos de la compilación o la configuración de un programa.

El nombre de los *scripts* tiene el formato `[000..999]-[nombre]`, donde el número de tres cifras indica el orden de ejecución de dicho *script*. De acuerdo con los desarrolladores de la aplicación *jhalfs*, hay dos maneras de personalizar[31] la construcción del sistema operativo *Linux from Scratch*. Una se centra en modificar cómo se construye el sistema básico, y la otra se centra en cómo se añaden componentes y configuraciones propias.

La forma recomendada de modificar la manera en que se construye el sistema base es modificando los *scripts* que identifican a cada componente en el directorio `/mnt/lfs/jhalfs/lfs-commands/chapter{5-8}`. Así, para eliminar la construcción de algún componente basta con eliminar el *script* correspondiente del directorio adecuado. Del mismo modo, para cambiar el orden de ejecución de los diferentes *scripts* sólo es necesario modificar el número de tres cifras que forma parte del nombre de los *scripts*. Para cambiar la versión de algún componente es necesario modificar el nombre de su archivo comprimido en el fichero `/mnt/lfs/jhalfs/pkg_tarball_list` reflejando la versión que queremos compilar y colocar el archivo comprimido con el código fuente de la versión deseada en el directorio `/mnt/lfs/sources`.

Si lo que se desea es añadir un nuevo componente al proceso de construcción de algún bloque de los que forman el libro *Linux from Scratch*, primero hay que decidir en qué momento se quiere construir dicho componente, después escribir el *script* con los comandos necesarios para construirlo y por último nombrar este nuevo *script* con los tres mismos dígitos que el componente al que se quiere preceder y añadir un nuevo dígito antes del nombre del componente. Por ejemplo, si durante la fase de configuración del programa *jhalfs* decidimos utilizar el gestor de paquetes *dpkg* para añadir la funcionalidad de dicho gestor de paquetes en el sistema operativo *Linux from Scratch*, el programa *jhalfs* genera un nuevo *script*, llamado `065-1-dpkg`, que se ejecutará antes del *script* `065-stripping`. En este caso concreto, la herramienta *jhalfs* está programada para añadir el *script* `065-1-dpkg` sin la intervención del usuario, aunque este no es el comportamiento general, donde es el usuario quien debe añadir manualmente el mencionado *script*.

La construcción de los componentes que forman parte del libro *Beyond Linux from Scratch* puede realizarse en dos momentos diferentes. Si se hace en un sistema en ejecución, el proceso es el mismo que para construir el libro *Linux from Scratch* pero seleccionando el subproyecto *Beyond Linux from Scratch* en el submenú *ajustes del libro* a la hora de configurar la herramienta *jhalfs*. Después de ejecutar la herramienta, son necesarios ejecutar una serie de pasos adicionales:

- Configurar los privilegios de superusuario para el usuario del sistema.
- Instalar los archivos de inicialización del intérprete de comandos `bash`. Este paso no es obligatorio, pero sí recomendado porque algunas instrucciones del libro *Beyond Linux from Scratch* dependen de estos archivos.
- La herramienta no tiene forma de saber qué versión de *Linux from Scratch* se instaló, por lo que se mostrarán todos los paquetes en el menú de configuración como si no se hubiera instalado ninguno. Si se conoce la versión del repositorio *subversion* que se instaló, puede ejecutarse el *script* `update-lfs.sh`. En caso de conocerse dicha versión o haber modificado la versión de algún componente base del libro, es necesario crear ficheros vacíos manualmente con el nombre `[componente]-[version]` en el subdirectorio `tracking` del directorio principal de la herramienta y ejecutar la herramienta. También es necesario realizar este paso si se ha instalado algún componente del libro *Beyond Linux from Scratch* manualmente.

Si se desea instalar componentes de este libro después de la construcción del sistema operativo *Linux from Scratch* sin llegar a arrancar el sistema operativo creado, lo que hace la herramienta *jhalfs* es instalar las

[31] más información en el fichero `README.CUSTOM` de la herramienta *jhalfs*.

herramientas en el directorio `/mnt/lfs/blfs-root` y sus dependencias se construyen antes de agregar las herramientas o componentes propios. Una vez se inicia el sistema *Linux from Scratch* creado igualmente hay que realizar una serie de pasos adicionales:

- Es necesario crear un usuario y entrar al sistema con dicho usuario para usar la herramienta `blfs-tool`. Configurar los permisos de superusuario para este nuevo usuario. Este paso no es obligatorio, pero sí recomendado.
- Mover el directorio `/blfs-root` al directorio principal del usuario creado y asignarle la propiedad del directorio y sus archivos.

Alcanzado este punto en ambas variantes, es posible editar los *scripts* alojados en el directorio `blfs-root/work/scripts`. A continuación es necesario generar el archivo *Makefile* que gobernará la construcción de los componentes utilizando el script `gen-makefile.sh` del directorio `blfs-root` desde el directorio `work` y ejecutar dicho *Makefile* para instalar los componentes y sus dependencias en el orden adecuado.

Por último, si lo que se desea es añadir a la herramienta un conjunto de componentes propios que serán construidos después del proceso de construcción de *Linux from Scratch* o *Beyond Linux from Scratch*, es necesario activar la opción *añadir soporte para herramientas propias* del submenú *ajustes del libro* e incluir el script correspondiente al componente que deseamos construir en el directorio `custom/config` del directorio que contiene el código fuente de la aplicación *jhalfs*. Los desarrolladores de la aplicación incluyen una serie de scripts de ejemplo. En el código 2.15 se muestra el contenido del *script* de instalación del programa `dhcpcd` que habilita la opción de configurar la red usando el protocolo *DHCP*.

```
#
# $Id: 964-dhcpcd 4026 2018-01-13 09:08:56Z pierre $
#
# dhcpcd is an implementation of the DHCP client specified in RFC2131.
# This is useful for connecting your computer to a network which uses
# DHCP to assign network addresses.
#

PKG="dhcpcd"
PKG_VERSION="6.9.3"
PKG_FILE="dhcpcd-${PKG_VERSION}.tar.xz"
URL="http://roy.marples.name/downloads/dhcpcd/${PKG_FILE}"
MD5="8357d023c4687d27bc6ea7964236b2a6"
for i in PATCH{1..10}; do
   unset $i
done

( cat << "xEOFx"

./configure --libexecdir=/lib/dhcpcd \
            --dbdir=/var/lib/dhcpcd &&
make

make install

# Add the following to boot scripts.
#make install-service-dhcpcd

# more configuration?
xEOFx
) > tmp
```

Código 2.15 Ejemplo de archivo `/etc/fstab` para *Systemd*.

A modo de resumen, las diferentes opciones de personalización de las que dispone la herramienta *jhalfs* se muestran en la tabla 2.6:

Modificación	Acción
Modificar opciones de compilación	Modificar el *script* del directorio `/mnt/lfs/jhalfs/lfs-commands`.
Borrar componente	Eliminar su *script* del directorio `/mnt/lfs/jhalfs/lfs-commands`.
Cambiar el orden de ejecución	Modificar el número de tres cifras del nombre del *script*.
Cambiar versión	Modificar su entrada en la lista `/mnt/lfs/jhalfs/pkg_tarball_list` e incluir el nuevo fichero en el directorio `/mnt/lfs/sources`.
Añadir componentes BLFS	Si el sistema está en ejecución y no se configuró previamente, ejecutar la herramienta *jhalfs* seleccionando el libro *Beyond Linux from Scratch* en el submenú *ajustes del libro*. Este método requiere ejecutar una serie de pasos adicionales.
	Si la herramienta se configuró para añadir componentes del libro *Beyond Linux from Scratch* en el momento de construir el sistema base, entonces hay que realizar algunos pasos adicionales antes de crear el fichero *Makefile* que permitirá instalar dichos componentes una vez el sistema básico esté en ejecución.
Añadir componentes propios	Activar la opción del menú de configuración y añadir los scripts en el subdirectorio `custom/config` del directorio *jhalfs* creado en el punto de montaje para la construcción del sistema *Linux from Scratch*.

Tabla 2.6 Resumen de modificaciones en el programa *jhalfs* para la construcción del sistema base.

Después de lo expuesto hasta ahora, resulta evidente que la herramienta *jhalfs* es compleja, tanto en su estructura como en su utilización. Es cierto que gran parte del proceso está automatizado y es cuestión de, dicho de manera informal, *darle a un botón* para que se construya el sistema, pero sólo en caso de construir el sistema *Linux from Scratch*, ya que ampliar el sistema base usando las instrucciones del libro *Beyond Linux from Scratch* requiere de cierto trabajo adicional por parte del usuario como se acaba de comentar.

La principales ventajas de *jhalfs* radican en el uso de *bash* como lenguaje de programación, que lo hace portable a la mayoría de distribuciones Linux, y el uso del programa `make` para gestionar las dependencias. Otro punto fuerte de esta aplicación es la lógica para calcular las dependencias de los componentes del libro *Beyond Linux from Scratch*, nada trivial teniendo en cuenta que es fácil encontrar dependencias circulares.

Sin embargo, el uso del lenguaje *bash* es a la vez una ventaja y una desventaja, ya que no es el lenguaje más adecuado para aplicaciones complejas como esta. Tampoco es trivial el análisis y la extracción de los comandos necesarios para construir los programas del sistema, que depende de una herramienta externa como `xsltproc`, que a su vez forma parte de un conjunto de librerías. Esta dependencia no suele incorporarse por defecto en la mayoría de distribuciones Linux, por lo que obliga al usuario de la herramienta *jhalfs* a ocupar espacio adicional en su disco duro instalando varias utilidades para cumplir esta dependencia, y posiblemente nunca usar el resto de utilidades instaladas.

A su vez, el proceso de modificación de los *scripts* es totalmente manual, y debe reproducirse cada vez que se utilice la herramienta *jhalfs* para generar un sistema *Linux from Scratch* ya que los *scripts* de construcción de los componentes se autogeneran en cada ejecución del programa de automatización. La inclusión de componentes propios a lo largo del proceso de construcción de los libros *Linux from Scratch* o *Beyond Linux*

from Scratch es sencilla, pero está limitada por el hecho de que sólo puede añadirse un único dígito extra al nombre del *script*, lo que permite incluir como máximo diez *scripts* antes de la ejecución de cualquiera de los generados por *jhalfs*. Añadir componentes a continuación de la construcción de cualquiera de los libros sí es un proceso sencillo, aunque tampoco es reproducible de manera automática entre ejecuciones de *jhalfs*.

Unido a lo anterior, y como motivación extra al desarrollo de una nueva herramienta estructurada de forma simple, que requiera la mínima intervención del usuario durante el proceso y que permita un mayor nivel de personalización del resultado, está el hecho de que los propios desarrolladores de *jhalfs* reconocen que su herramienta no soporta las últimas versiones[32] de los libros *Linux from Scratch* y *Beyond Linux from Scratch*. Concretamente, ellos reconocen soporte hasta la versión *8.0* que fue publicada el 25 de febrero de 2017 tanto para la versión estable del programa como para la versión en desarrollo.

De todo ello nace la idea de desarrollar el programa *LFSBuilder*.

[32]más información de las versiones soportadas por la herramienta *jhalfs* en la URL *http://wiki.linuxfromscratch.org/alfs/wiki/SupportedBooks*

3 LFSBuilder

Inteligence is the ability to avoid doing work, yet getting the work done.

LINUS TORVALDS

LFSBuilder, acrónimo de *Linux from Scratch Builder*, es una herramienta pensada para automatizar la ejecución de los comandos presentes en los libros *Linux from Scratch* y *Beyond Linux from Scratch* dando libertad al usuario de personalizar el resultado.

El programa *LFSBuilder* está escrito en Python, un lenguaje de prototipado rápido, usando el paradigma de la programación orientada a objetos para modelar los diferentes elementos que componen los libros del proyecto *Linux from Scratch*. Además, el programa *LFSBuilder* está construido en dos niveles diferentes para separar la lógica de construcción del sistema de las opciones de personalización. Como puede verse en la figura 3.1, la lógica del programa funciona usando la información proporcionada por los elementos de la capa superior.

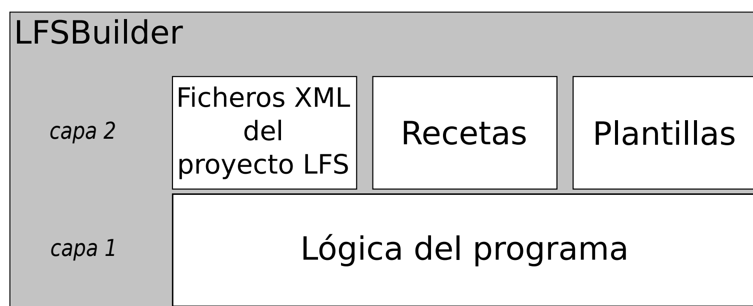


Figura 3.1 Diagrama de capas de la aplicación *LFSBuilder*.

De acuerdo con la figura 3.1, la personalización del sistema proviene de tres fuentes diferentes: los ficheros XML para la versión de los libros *Linux from Scratch* o *Beyond Linux from Scratch* deseada, los valores definidos por el usuario para cada componente que hay que construir y las plantillas de ficheros básicos que utiliza el programa. Esta capa de abstracción por encima de la lógica del programa facilita la personalización del sistema operativo generado sin obligar al usuario de la herramienta a modificar el código fuente principal, sino el archivo de configuración en formato *YAML* y/o la lógica que gobiernan las diferentes propiedades de un componente. De este modo, un usuario podría cometer un error e inutilizar la construcción de un componente concreto, pero el resto de los componentes no se verían afectados por dicha modificación.

3.1 Diseño de la herramienta

La herramienta *LFSBuilder* implementa los mismos bloques que el diagrama de flujo para el libro *Linux from Scratch* de la figura 2.1 de la página 12, con la novedad de que añade un nuevo elemento al final del proceso, el de *recolección y limpieza del entorno*. La figura 3.2 muestra el nuevo diagrama de flujo mencionado.

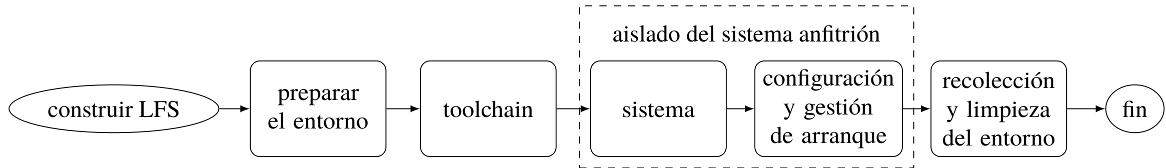


Figura 3.2 Diagrama de flujo para construir el sistema operativo *Linux from Scratch* usando la herramienta *Linux from Scratch Builder*.

Este nuevo bloque se encarga de desmontar el directorio en el que se construye el sistema operativo y los directorios del sistema anfitrión necesarios para realizar la compilación en los bloques *sistema* y *configuración y gestión de arranque*. Cada bloque del diagrama anterior se corresponde con un objeto capaz de construir una lista de *componentes*. Así, a los elementos lógicos básicos que forman la herramienta y que se acaban de nombrar, los componentes `builders` y `components`, hay que incorporar los elementos `LFSBuilder`, `cli`, `actions`, `xmlparser` `downloader`, `printer` y `tools`. Cada uno de estos componentes tiene una función específica dentro de la herramienta *LFSBuilder*.

El componente principal del programa es `LFSBuilder`. El usuario interactúa con este componente mediante la interfaz de línea de comandos definida por el componente `cli`. `LFSBuilder` hace uso de las acciones definidas en el componente `actions` para hacer modificaciones en la configuración de la herramienta o comprobar si es posible construir la secuencia de constructores proporcionada.

Como se ha comentado en el párrafo anterior, cada constructor o `builder`, dispone de una lista de componentes que debe construir. Estos constructores reciben el nombre en inglés de los bloques del diagrama 3.2 de la página 34. Es decir `provider`, `toolchain`, `system`, `configuration` y `collector`. Los componentes del libro *Beyond Linux from Scratch* pueden construirse haciendo uso del constructor `blfs`. Los datos que definen tanto a los elementos constructores como a los componentes que se construyen provienen de tres fuentes de información: los valores de configuración definidos en el fichero `config.py`, los comandos extraídos de los ficheros XML de los libros, y la configuración personalizada que se aloja en el directorio `recipes`. La interfaz de línea de comandos modifica los valores del fichero `config.py` y la información definida en el directorio `recipes` prevalece sobre los presentes en el libro.

Así, el componente `xmlparser` se encarga de parsear los archivos XML que conforman el libro para extraer los comandos a ejecutar y escribirlos en un archivo XML dentro del subdirectorio `tmp` del directorio principal del programa para hacerlos disponibles al constructor adecuado. El componente `downloader` se encarga de descargar, a petición del usuario, los ficheros XML del libro solicitado y los archivos comprimidos con el código fuente de los diferentes componentes que integran el sistema operativo. Por su parte, el componente *printer* muestra los mensajes generados al usuario por la salida estándar. Por último, el componente `tools` alberga un conjunto de funciones necesarias para el correcto funcionamiento de los diferentes componentes de la herramienta *LFSBuilder*.

3.1.1 Estructura y directorio principal

Estructuralmente, la herramienta está dividida en varios ficheros y directorios compuestos por el código fuente que contiene la lógica del programa y archivos de configuración. La estructura de directorios está recogida en la figura 3.3.

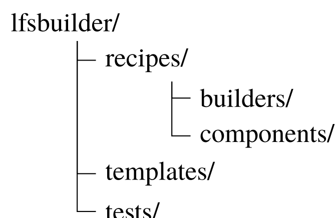


Figura 3.3 Estructura de directorios de la herramienta *LFSBuilder*.

El directorio `lfsbuilder` es el directorio principal del programa. Contiene los archivos de código fuente del programa y el archivo de configuración principal. La lista inferior enumera los diferentes ficheros que componen este directorio y una pequeña descripción de cada uno de ellos.

- **actions.py**: acciones personalizadas para los argumentos de la interfaz de línea de comandos.
- **builders.py**: implementa la lógica necesaria para construir la lista de componentes correspondiente.
- **cli.py**: interfaz de línea de comandos. Hace uso del módulo nativo de Python llamado `argparse`.
- **components.py**: este fichero contiene la lógica necesaria para construir un componente, requiera compilación o no.
- **config.py**: fichero de configuración principal. Varios de sus elementos pueden modificarse a través de la línea de comandos.
- **downloader.py**: fichero encargado de descargar los ficheros XML de los libros y el código fuente de los programas que se van a construir.
- **lfsbuilder.py**: fichero principal del programa.
- **printer.py**: muestra los mensajes producidos durante la ejecución del programa.
- **tools.py**: fichero con funciones de uso común como escritura/lectura de ficheros, sustitutir elementos en un archivo, leer la receta de un determinado componente, ejecutar programas, etc.
- **xmlparser.py**: lógica para analizar los ficheros XML que contienen los comandos proporcionados por el libro para construir los diferentes componentes del sistema operativo. También escribe los comandos en ficheros individuales para cada constructor.

actions.py

El fichero `actions.py` implementa dos clases que heredan de la clase `Action` definida en el módulo base de Python `argparse`, utilizado para la definición de argumentos de línea de comandos. La primera clase, `ModifyBuildersList`, se encarga de validar que la lista de constructores pasada como argumentos es válida. Así, la herramienta no permite intentar construir dos veces el mismo constructor, ni el constructor `blfs` antes que alguno de los constructores propios del libro *Linux from Scratch*. También comprueba que el orden de los constructores del libro *Linux from Scratch* sea la correcta. La otra clase definida en este archivo es `SetConfigOption`, que modifica los valores de configuración del archivo `config.py` en función de ciertos argumentos de línea de comandos. Véase el apartado *3.1.5 Interfaz de línea de comandos* de la página 54 para más información.

La figura 3.4 muestra las clases definidas en este fichero.

ModifyBuildersList
builders_list builder_index_dict
__call__() check_builders_dupes() check_builders_order()

SetConfigOption
__call__()

Figura 3.4 Clases implementadas en el fichero `actions.py`.

builders.py

El constructor es uno de los elementos básicos del programa *LFSBuilder*, junto con los componentes. Existen dos tipos de constructores `InfrastructureComponentsBuilder` y `ComponentsBuilders`. Ambos constructores heredan de las clases `BaseBuilder` y `BaseComponentsBuilder`, modificando los atributos necesarios para su correcto funcionamiento.

El constructor de tipo `InfrastructureComponentsBuilder` se encarga de construir los componentes que modifican el entorno de construcción del sistema, como generar y dar formato a un archivo de imagen de disco duro para ser usado como máquina virtual, montar el directorio `sources` dentro del punto de montaje para no ocupar espacio innecesario dentro de la imagen de disco, y limpiar el entorno al final de la compilación. Un `ComponentsBuilder` es, como su propio nombre indica, un constructor de componentes del sistema final.

Estos constructores se generan de manera dinámica a lo largo de la ejecución de la herramienta *LFSBuilder* en función de las características del constructor definidas en la lógica del programa y de los valores configurados en la receta correspondiente a dicho constructor. Esta es la finalidad de la clase `BuilderGenerator`. Más información en la sección *3.1.2 Recetas* de la página 46.

La imagen 3.5 muestra el esquema de clases definido en este archivo y la relación entre las mismas.

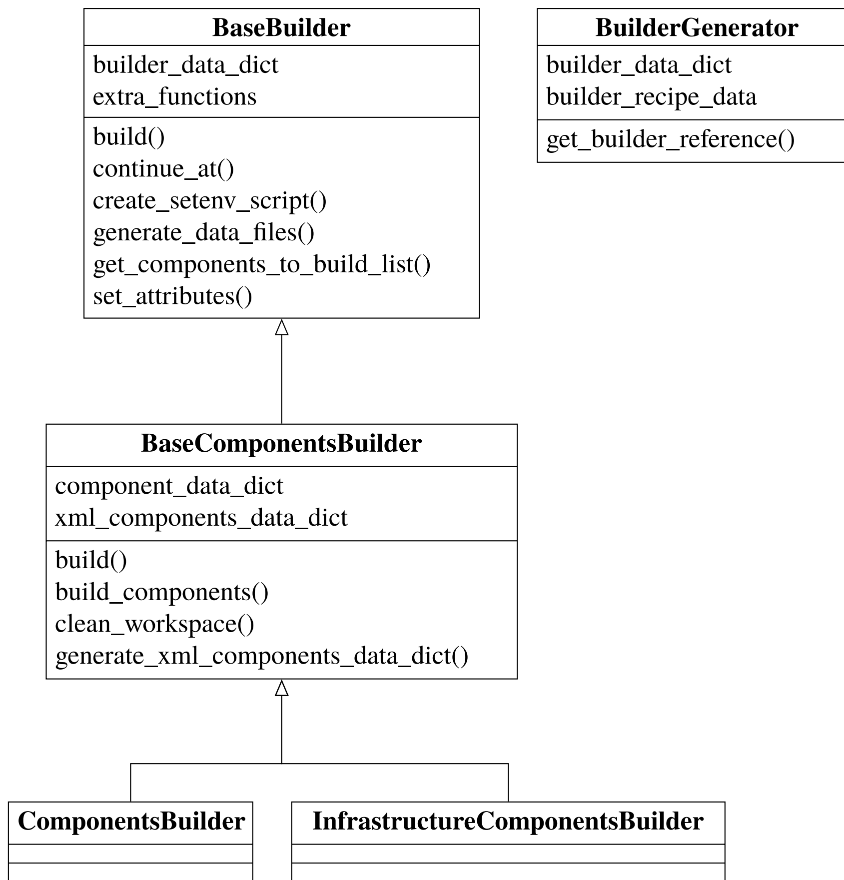


Figura 3.5 Clases implementadas en el fichero `builders.py`.

cli.py

Este fichero implementa el componente `cli` nombrado anteriormente. La clase `Cli` define los argumentos de línea de comandos para la herramienta *LFSBuilder* y sus diferentes comandos haciendo uso del módulo `argparse` de Python.

El diagrama de la clase `Cli` puede consultarse en la figura 3.6.

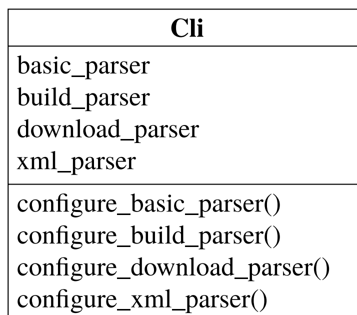


Figura 3.6 Clases implementadas en el fichero `cli.py`.

components.py

Los componentes son el otro elemento importante de la herramienta *LFSBuilder*. Son los que se encargan de compilar el código fuente de un programa e instalar los binarios y archivos de configuración generados en el sistema final. También posee la lógica necesaria para construir los componentes desde dentro o fuera del `chroot` del punto de montaje en función del constructor que esté intentando construir dicho componente o la información definida en su correspondiente receta. Más información en la sección *3.1.2 Recetas* de la página 46.

Cada componente a construir se corresponde con un objeto de tipo `CompilableComponent` o `SystemConfigurationComponent` dependiendo de si el componente a construir requiere ejecutar el proceso de compilación, por ejemplo el componente *gcc*, o simplemente modifica o añade elementos al sistema final, como el componente *stripping*, que se encarga de reducir el tamaño de los binarios creados en el bloque *toolchain* pero no construye nada por sí mismo. Al igual que ocurría con los constructores, la clase `ComponentGenerator` crea un objeto de alguno de los dos tipos anteriores de forma dinámica durante el proceso de construcción.

La imagen 3.7 muestra las distintas clases definidas en el fichero `components.py`.

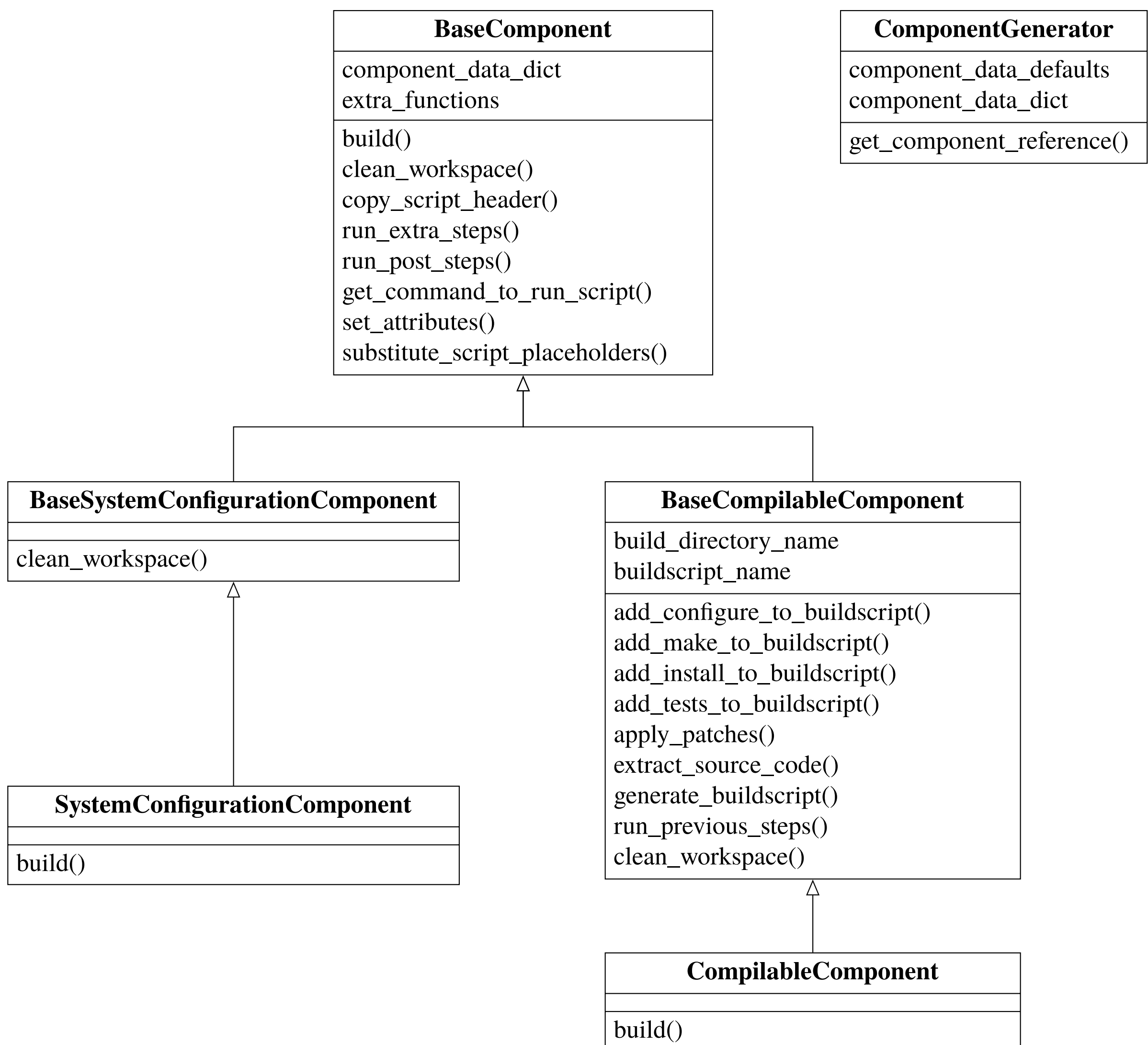


Figura 3.7 Clases implementadas en el fichero `components.py`.

config.py

El fichero `config.py` es el fichero principal de configuración de la herramienta *LFSBuilder*. Permite configurar, entre otras cosas, el nombre del usuario sin privilegios que se utilizará para construir los componentes del *toolchain*, el directorio base de construcción en el que también se monta la partición de disco duro o el fichero `img` para crear una máquina virtual con el contenido del sistema operativo *Linux from Scratch*, el sistema de inicio a utilizar, la versión de los libros *Linux from Scratch* y *Beyond Linux from Scratch* que se va a utilizar, o las opciones de compilación quedan valor a la variable de entorno `MAKEFLAGS`.

La tabla 3.1 mostrada a continuación lista las opciones disponibles en el archivo `config.py`. Todas las variables son de tipo *cadena de texto* salvo que se indique lo contrario.

Opción	Descripción
BASE_DIRECTORY	Directorio base en el que instalar el sistema *Linux from Scratch*.
NON_PRIVILEGED_USERNAME	Nombre del usuario sin privilegios que se utilizará para construir los componentes como no `root`.
LFS_VERSION	Versión del libro que se desea construir.
MOUNT_SOURCES_DIRECTORY	Permite elegir si se desea montar el directorio `sources` o no. Opción binaria.
SOURCES_ORIG_DIRECTORY	Ruta al directorio `sources` que se desea montar. Puede usarse una ruta relativa al directorio que contiene el código fuente de la herramienta *LFSBuilder* usando la cadena `@@LFSBUILDER_SRC_DIRECTORY@@`.
MAKEFLAGS	Define las opciones de compilación para la herramienta `make`.
ROOT_PASSWD	Contraseña para el usuario `root` del sistema creado.
NON_PRIVILEGED_USERNAME_PASSWD	Contraseña del usuario `NON_PRIVILEGED_USERNAME` en caso de contruir el componente *createuser*.
GENERATE_DATA_FILES	Opción binaria para decidir si los archivos de datos que genera el analizador de XML deben generarse o no. El proceso de construcción fallará en caso de que estos ficheros no existan en el archivo `tmp`.
RESTORE_XML_BACKUPS	Opción binaria para dejar los archivos XML de los libros sin modificar.

Opción	Descripción
GENERATE_IMG_FILE	Esta opción binaria permite decidir si generar un archivo `IMG` para construir el sistema operativo en él o no.
MOUNT_IMG_FILE	Opción para montar o no el archivo de imagen de disco. Opción binaria.
IMG_FILENAME	Ruta al fichero `IMG` que se quiere montar como `BASE_DIRECTORY`. Puede usarse una ruta relativa al directorio que contiene el código fuente de la herramienta *LFSBuilder* usando la cadena `@@LFSBUILDER_SRC_DIRECTORY@@`.
IMG_SIZE	Tamaño del fichero `IMG` a montar. Las unidades son `M` para *megabyte* y `G` para *gigabyte*. Se recomienda usar un tamaño alrededor de 10 *gigabytes*.
CUSTOM_COMPONENTS_TO_BUILD	Atributo binario para decidir si los componentes del libro *Linux from Scratch* deben construirse en el orden indicado por el libro o en un orden personalizado. Si se desea un orden personalizado, éste viene definido por la lista `components_to_build` de la receta correspondiente a cada constructor.
SYSV	Opción binaria para elegir el sistema de inicio *SysVinit*.
SYSTEMD	Opción binaria para elegir el sistema de inicio *Systemd*.
INCLUDE_MESON_BUILDER	Desde la versión *8.2* de *Linux from Scratch*, es necesario construir los componentes *python*, *ninja* y *meson* para el sistema de inicio *Systemd*. Esta opción es sólo obligatoria para este sistema de inicio y la herramienta fallará en caso de no cumplir este requisito.
SAVE_TOOLCHAIN	Atributo binario que activa, o desactiva, la opción de guardar el `toolchain` antes de empezar con el bloque del `sistema`.
SAVE_TOOLCHAIN_FILENAME	En caso de activar la opción anterior, el `toolchain` se comprimirá en un archivo de extensión `.tar.gz` Si el valor de este parámetro es `lfsbuilder-toolchain-@@LFS_VERSION@@`, entonces el nombre del fichero resultante es `[SAVE_TOOLCHAIN_FILENAME]-[fecha].tar.gz`.
DELETE_TOOLS	Selecciona si el directorio `tools` será borrado después de construir el sistema operativo, cuando ya no es necesario. Opción binaria.
TIMEZONE	Zona horaria a utilizar para configurar el sistema. Puede usarse el comando `tzselect` para obtener la zona horaria deseada.

Opción	Descripción
PAPER_SIZE	Valor de la variable de entorno `PAGE` para configuración del componente *groff*. El valor `letter` es común en los Estados Unidos, y el valor `A4` en el resto del mundo.
KEYMAP	Valor de configuración para el idioma del teclado.
CONSOLE_FONT	Fuente de la consola.
LANG	Variable de entorno para la configuración del idioma. Se recomienda hacer uso de la codificación `UTF-8`.
LOCALE	Valor del resto de variables de entorno encargadas de configurar distintos aspectos del idioma.
CHARMAP	Codificación por defecto del sistema de caracteres. Se recomienda hacer uso de la codificación `UTF-8`.
ROOT_PARTITION_NAME	Nombre del disco que contiene la partición de instalación del sistema. Se usa en el archivo de configuración de `GRUB` para seleccionar la partición raíz al iniciar el *kernel* del sistema operativo *Linux from Scratch*.
ROOT_PARTITION_NUMBER	Número de la partición del disco que contiene la instalación del sistema. Opción utilizada para la configuración de `GRUB`.
GRUB_ROOT_PARTITION_NAME	Nombre del dispositivo en el que se debe instalar el componente `GRUB`.
GRUB_ROOT_PARTITION_NUMBER	Nombre del disco duro y partición a incluir en el archivo de configuración del `GRUB`. Este valor le indica al `GRUB` en qué disco duro debe buscar el *kernel* a iniciar.
FILESYSTEM_PARTITION_TYPE	Formato para el fichero de disco duro o la partición en la que se instalará *Linux from Scratch*.
SWAP_PARTITION_NAME	Nombre y número de la partición *swap* para el archivo `/etc/fstab`.
ETH0_IP_ADDRESS	Dirección IP con la que configurar la interfaz de red `eth0`.
ETH0_GATEWAY_ADDRESS	Dirección IP de la puerta de enlace para la interfaz de red `eth0`.
ETH0_BROADCAST_ADDRESS	Dirección IP de difusión para la interfaz de red `eth0`.
ETH0_MASK	Máscara de red para la interfaz `eth0`.
DOMAIN_NAME	Nombre de dominio a utilizar para la configuración de red.
DNS_ADDRESS_1 DNS_ADDRESS_2	Direcciones IP usadas por el protocolo de descubrimiento dinámico de dominios de red *DNS*.

Opción	Descripción
HOSTNAME	Nombre del equipo del sistema operativo construido.
DISTRIBUTION_NAME	Nombre de la distribución construida.
DISTRIBUTION_VERSION	Versión de la distribución construida.
DISTRIBUTION_DESCRIPTION	Descripción de la distribución construida.
REMOVE_REBOOT_COMPONENT	Opción binaria. Este parámetro desactiva la ejecución del componente *reboot* del libro *Linux from Scratch*. Su valor debe ser siempre `True` porque el constructor `collector` realiza su función de forma más adecuada.
MOUNT_SYSTEM_BUILDER_DIRECTORIES	Opción binaria que decide si los directorios del sistema deben montarse en el directorio base de construcción antes de iniciar el proceso o no.
VERBOSE	Opción binaria que muestra mayor cantidad de información al usuario.
DEBUG_SCRIPTS	Muestra los comandos que componen un *script* a medida que se ejecutan. Opción binaria.
CONTINUE_AT	Esta opción permite iniciar la construcción de un determinado constructor por un determinado componente. Sólo es válido para los constructores `toolchain`, `system`, `configuration` y `blfs`.

Tabla 3.1 Opciones del fichero `config.py`.

downloader.py

En el archivo `downloader.py` se define la clase encargada de descargar los diferentes archivos de código fuente, parches y de clonar los repositorios de los libros. La figura 3.8 muestra el diseño de esta clase.

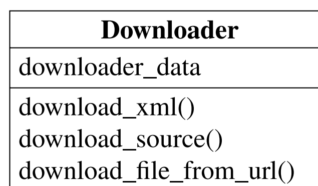


Figura 3.8 Diagrama de la clase definida en el fichero `downloader.py`.

El método `download_xml()` se encarga de clonar la versión adecuada de los repositorios *Linux from Scratch* y *Beyond Linux from Scratch* correspondiente al valor `LFS_VERSION` del archivo `config.py`. Por otra parte, el método `download_sources()` descarga los ficheros comprimidos de código fuente. Estas descargas se hacen dentro de los subdirectorio `tmp` y `tmp/sources` respectivamente.

lfsbuilder.py

Fichero principal de la herramienta *LFSBuilder*. Se encarga de interactuar con el usuario mediante la interfaz de línea de comandos, proporcionando los comandos `build`, `download` y `parse`, que construyen una lista de constructores proporcionados, descargan los archivos XML o el código fuente de los componentes para el libro elegido, y analizan los ficheros XML generando, según el constructor, la lista de componentes incluidos en el libro, y los comandos a ejecutar para cada uno de ellos en las correspondientes etapas de construcción del componente.

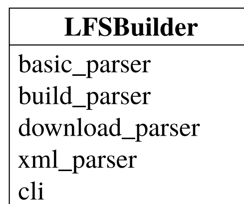


Figura 3.9 Diagrama de la clase definida en el fichero `lfsbuilder.py`.

Más información sobre la interfaz de línea de comandos en el apartado *3.1.5 Interfaz de línea de comandos* de la página 54.

printer.py

En el fichero `printer.py` sólo se definen una serie de funciones básicas para mostrar mensajes en la salida estándar con colores y un formato específico, `[hora] [mensaje]`. Además, los mensajes de error terminan la ejecución del programa devolviendo el código de estado 1.

Cada tipo de mensaje se muestra con un color diferente. Esta información se encuentra recogida en la tabla 3.3.

Tipo de mensaje	Color
Información	Azul
Aviso	Amarillo
Error	Rojo

Tabla 3.2 Tipos de mensaje mostrados por la herramienta *LFSBuilder*.

tools.py

El archivo `tools.py` contiene multitud de funciones comunes para ser utilizadas por el resto de componentes y clases. Funciones de lectura/escritura de ficheros, de listas, de diccionarios, sustituciones, modificación de comandos y ficheros, lectura de recetas, etc.

xmlparser.py

La herramienta *LFSBuilder*, al contrario que la herramienta oficial *jhalfs*, no utiliza utilidades externas para analizar los ficheros XML, sino que hace uso de la clase `xml.etree.ElementTree.XMLParser` definida en el módulo `xml` nativo de Python. El diseño de la clase `LFSXmlParser` queda reflejado en la figura 3.10.

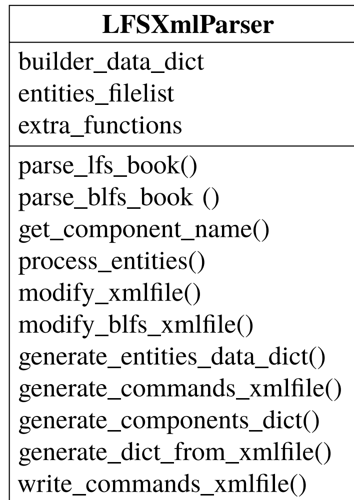


Figura 3.10 Diagrama de la clase definida en el fichero `xmlparser.py`.

En el caso del libro *Linux from Scratch*, hay dos tipos de ficheros que requieren ser analizados, los ficheros de entidades, y los ficheros de comandos. Sin embargo, los ficheros que forman el libro *Beyond Linux from Scratch* mezclan ambos tipos de datos. La herramienta *LFSBuilder* posee la lógica necesaria para analizar ambos libros indistintamente.

Los ficheros de entidades, con extensión `.ent`, contienen información sobre los distintos componentes y el libro tales cómo versiones, URL para la descarga del código fuente, sumas de comprobación *md5* o tiempos estimados de construcción. Estos valores son utilizados tanto en los ficheros de comandos como en los propios ficheros de entidades a modo de variables. El formato de estos marcadores es, como ejemplo para el atributo `bash-version`, el siguiente `&bash-version;`. Un pequeño ejemplo del formato de los ficheros de entidades puede verse en el código 3.1.

```
<!ENTITY bash-version "4.4.18">
<!ENTITY bash-size "9,242 KB">
<!ENTITY bash-url "&gnu;bash/bash-&bash-version;.tar.gz">
<!ENTITY bash-md5 "518e2c187cc11a17040f0915dddce54e">
<!ENTITY bash-home "&gnu-software;bash/">
<!ENTITY bash-ch5-du "61 MB">
<!ENTITY bash-ch5-sbu "0.4 SBU">
<!ENTITY bash-ch6-du "56 MB">
<!ENTITY bash-ch6-sbu "2.0 SBU">
```

Código 3.1 Ejemplo de entidades XML.

Como puede apreciarse, las líneas contenidas en este fichero siguen una estructura simple: primero se indica que se está definiendo una entidad XML, a continuación el nombre del atributo y por último el valor. Así pues, la clase `LFSXmlParser` definida en este fichero se encarga de leer y extraer los atributos y valores de las diferentes líneas para almacenarlos en un diccionario, estructura básica de Python que consiste en un conjunto de elementos de tipo clave-valor, para ser sustituidos en el momento en que aparezca un marcador del tipo `&atributo;` en cualquier fichero de comandos o entidades.

Por su parte, los ficheros XML están divididos en subdirectorios con el nombre de los capítulos que forman el libro. Cada directorio contiene un fichero XML con el nombre del capítulo y que hace de índice enumerando los componentes que forman dicho capítulo y el orden en que deben ser construidos. La clase `LFSXmlParser` analiza este archivo para conocer el nombre de los componentes que forman el constructor y los escribe en un fichero llamado `[constructor]_components_to_build.txt` en el directorio `tmp`. Este archivo será usado posteriormente por el constructor mencionado para conocer los componentes que deben ser construidos en caso de que el parámetro `CUSTOM_COMPONENTS_TO_BUILD` del fichero principal de configuración tenga el valor `False`. En otro caso, el contenido del archivo es despreciado por la herramienta, pero se deja en el directorio `tmp` para consulta del usuario.

Los ficheros XML que contienen los comandos necesarios para construir un determinado componente contienen a su vez una determinada estructura. Así, los comandos que hay que ejecutar se envuelven por las etiquetas `<screen><userinput>` y `</userinput></screen>` más un atributo que indica el paso al que pertenece. Por ejemplo, los comandos que deben ejecutarse antes de compilar el código fuente utilizan la etiqueta `<userinput remap="pre">` y los comandos que forman parte de la configuración del código fuente se etiquetan como `<userinput remap="configure">`. El resto de pasos se etiquetan de manera similar, y los no etiquetados son añadidos a los pasos posteriores a la compilación e instalación del código fuente. El código 3.2 muestra una parte del archivo XML con las instrucciones para construir el componente *gmp*.

```
<screen><userinput remap="configure">./configure --prefix=/tools \
        --enable-mpbsd</userinput></screen>

    <para>Compile the package:</para>

<screen><userinput remap="make">make</userinput></screen>

    <para>To test the results, issue:</para>

<screen><userinput remap="test">make check</userinput></screen>

    <para>Install the package:</para>

<screen><userinput remap="install">make install</userinput></screen>
```

Código 3.2 Contenido parcial del fichero `gmp.xml`.

La clase `LFSXmlParser` hace una copia de seguridad del archivo en primer lugar, modifica el archivo y lee y clasifica cada comando, añadiéndolo al mismo diccionario en que añadió las entidades leídas previamente usando la clave `[componente]-[paso]`, por ejemplo `gcc-configure`. Por último, una vez ha leídos y clasificados todos los comandos de los componentes de un determinado constructor, los escribe en el fichero de nombre `[constructor]_data.xml` para ser leído por el constructor nombrado y restaura el contenido del archivo XML original.

Las modificaciones a los archivos provienen de dos fuentes. La lógica del programa implementa un conjunto de modificaciones comunes y las recetas de cada componente pueden especificar modificaciones concretas mediante los parámetros `components_substitution_list`, `disable_commands_list` y `comment_out_list`. Puede obtenerse más información sobre estos parámetros en la sección *3.1.2 Recetas*.

3.1.2 Recetas

Una *receta* define uno de los elementos básicos de la herramienta *LFSBuilder* nombrados anteriormente: los constructores y los componentes. Está formada por un directorio y dos archivos: un fichero obligatorio en formato YAML, nombrado, al igual que el directorio, como el elemento que define, que describe atributos del elemento, y un fichero opcional de funciones, llamado `functions.py` que permite redefinir el comportamiento de varias funciones de la herramienta u otras propias para personalizar la construcción del elemento de la manera deseada.

Dentro del directorio `recipes` de la herramienta hay otros dos directorios, `builders` y `components`, que almacenan los directorios que contienen las definiciones de cada elemento. A modo de ejemplo, dentro del directorio `recipes/builders/toolchain` se encuentran los ficheros `toolchain.yaml` y `functions.py`.

El fichero `[nombre].yaml` es obligatorio, y su contenido mínimo ha de ser el atributo `name`. Este atributo define el valor de varias variables necesarias para el correcto funcionamiento de las clases `BuilderGenerator` y `ComponentGenerator`. Estas clases agrupan los datos del elemento que se han obtenido de los ficheros creados por el objeto `LFSXmlParser` y el contenido de la receta, dando prioridad a los datos incluidos en esta última para que la personalización deseada por el usuario siempre tenga mayor prioridad que la información del libro. Con estos datos, el programa genera una instancia de la clase adecuada en tiempo de ejecución. Por defecto, se instancia un objeto de la clase `ComponentsBuilder` o `CompilableComponent` para los constructores y los componentes respectivamente. Los parámetros `base_builder` y `base_component` permiten seleccionar la clase que se instanciará en el generador correspondiente.

Las opciones disponibles son las siguientes:

Parámetro	Descripción
package_name	Nombre del fichero del código comprimido.
previous	Comandos a ejecutar en la fase de pasos previos a la compilación. Los parches se aplican en un paso anterior a este, justo después de descomprimir el código fuente.
configure	Comandos a ejecutar para configurar el código fuente antes de la compilación. Este atributo sustituye el comando proporcionado por el libro al completo.
configure_options	Opciones extra para el comando `configure`. Se añaden a continuación de las proporcionadas por el libro.
make	Comando a ejecutar en la fase de compilación del código fuente.
make_options	Opciones extra para el comando `make`. Se añaden a continuación de las proporcionadas por el libro.
install	Comando a ejecutar en la fase de instalación de los binarios y ficheros construidos.
install_options	Opciones extra para el comando `make install`. Se añaden a continuación de las proporcionadas por el libro.

Parámetro	Descripción
test	Comando a ejecutar en lugar de los comandos `make test` o `make check` proporcionados por el libro.
test_options	Opciones extra para el comando `make test`. Se añaden a continuación de las proporcionadas por el libro.
include_tests	Parámetro binario que decide si los tests deben o no ejecutarse.
post	Comandos a ejecutar después del proceso de configuración. Sobreescribe a los propuestos por el libro.
env_PATH_value	valor de la variable de entorno `PATH` para construir los componentes de un determinado constructor.
book	Libro al que pertenece el componente.
run_as_username	Nombre del usuario que se utilizará para construir el componente.
run_into_chroot	Indica si el elemento debe construirse desde dentro del `chroot` o desde fuera.
version	Versión del componente que se desea construir.
comment_out_list	Lista de comandos a comentar en los ficheros XML.
components_to_build	Lista de componentes, y orden, que un determinado constructor se encarga de construir si el parámetro `CUSTOM_COMPONENTS_TO_BUILD` está activo.
component_substitution_list	Lista de sustituciones a realizar en el archivo XML del componente.
chapters_list	Lista de capítulos que forma un constructor del libro *Linux from Scratch*.
disable_commands_list	Lista de comandos que se deshabilitarán en el fichero XML del componente y serán ignorados por la clase `LFSXmlParser`.
runscript_cmd	Comando con el que ejecutar los *scripts* de compilación.

Tabla 3.3 Parámetros para las recetas de la herramienta *LFSBuilder*.

Mención aparte merece la instalación de servicios para los componentes del libro *Beyond Linux from Scratch*. En este libro, los servicios se encuentran incluidos en un archivo comprimido diferente, y es necesario extraer estos servicios e instalarlos en el sistema. Para ello, se ha desarrollado una función en el fichero `tools.py` llamada `modify_blfs_component_bootscript_install` que recoge el valor del parámetro `bootscript_install_cmd` y lo incluye en los pasos necesarios para instalar el servicio. La lógica de esta función es capaz de instalar servicios tanto para *SysVinit* como *Systemd*.

El código 3.3 muestra los pasos necesarios para instalar el servicio `opensshd` en su versión para el sistema de inicio *SysVinit*. El proceso es el mismo para otros servicios, simplemente cambiando el comando `make install-sshd` del paso número 2 por el indicado en el parámetro `bootscript_install_cmd` de la receta del componente en cuestión.

```
# 1. Extraer los ficheros de instalación de servicios
cd /mnt/lfs/sources
tar xf blfs-bootscripts-20180105.tar.*
cd blfs-bootscripts-20180105

# 2. Instalar el servicio o script de inicio necesario
make install-sshd

# 3. Volver al directorio 'sources' y borrar los archivos descomprimidos
cd /mnt/lfs/sources
rm -rf blfs-bootscripts-20180105
```

Código 3.3 Comandos para la instalación del servicio `opensshd` con la herramienta *LFSBuilder*.

El código 3.4 incluye, a modo de ejemplo, el contenido de la receta `openssh.yaml` para el programa *OpenSSH*.

```
---
name: 'openssh'
book: 'blfs'

# Nombre del usuario para el que se autorizará la clave
# pública 'vms.pub' presente en el subdirectorio 'files'.
# Puede usarse el valor 'config.NON_PRIVILEGED_USERNAME' para
# configurar el acceso para dicho usuario.
# ADVERTENCIA: usar el usuario 'root' puede ser peligroso.
openssh_username: 'config.NON_PRIVILEGED_USERNAME'
openssh_public_key_filename: 'vms.pub'

# Comando para la instalación del servicio.
bootscript_install_cmd: 'make install-sshd'

# El fichero XML del componente 'ssh' ejecuta comandos usando
# condiciones lógicas. Esto es: patch && configure && make
# Como los parches se aplican en un paso previo, el comando 'configure' no
# se ejecutaría nunca, dado que el comando 'patch' devuelve FALSO, por lo que
# la condición lógica termina en este punto:
# patch && configure && make --> 0 && configure && make
# Con la siguiente sustitución se fuerza al comando 'patch'
# a devolver VERDADERO.
component_substitution_list: ['.patch &amp;&amp;',
                              '.patch || true']

disable_commands_list: ['ssh-keygen',
                        "sed 's@d/login@d/sshd@g' /etc/pam.d/'"]

extra_download_urls:
    - '&patch-root;/openssh-&openssh-version;-openssl-1.1.0-1.patch'
```

Código 3.4 Contenido de la receta `openssh.yaml` para el programa *OpenSSH*.

Por su parte, el fichero opcional `functions.py` permite a un elemento sobreescribir algunas de las funciones de la lógica del programa para acomodar su funcionamiento a sus necesidades específicas. Así, el componente *kernel* sobreescribe la función `run_previous_steps` para, además de ejecutar la función contenida en la lógica del programa, copiar el archivo de configuración para compilar el núcleo del sistema operativo con él. La necesidad de sobreescribir esta función reside en que los pasos previos que va a ejecutar el libro es hacer una limpieza del directorio de compilación para asegurarse de que no hay ningún fichero de configuración de una versión antigua o erróneo, por lo que el directorio se queda sin configuración. Después de ejecutar dicha limpieza, el libro solicita al usuario generar un nuevo fichero de configuración haciendo uso de un menú al estilo del menú de configuración de la herramienta *jhalfs*. Sin embargo, esto requiriría la intervención del usuario y pararía temporalmente la automatización del proceso. Posibilitar la sobreescritura de esta función soluciona el problema haciendo que el usuario indique en la receta del componente qué fichero de configuración de los presentes en el subdirectorio `files` de la receta quiere utilizar y copiándolo al directorio de compilación. El código 3.5 muestra la función `run_previous_step` comentada anteriormente.

```
def run_previous_steps(component_data_dict, parent_function):

    # Call parent function
    parent_function()

    print("Copying custom '.config' file")
    filename = os.path.join(component_data_dict["lfsbuilder_src_directory"],
                            "recipes",
                            "components",
                            "kernel",
                            "files",
                            component_data_dict["kernel_config_filename"])

    tools.copy_file(filename,
                    os.path.join(component_data_dict["extracted_directory"],
                    ".config")
    )
```

Código 3.5 Uso del fichero `functions.py` para sobreescribir la función `run_previous_steps` en el componente *kernel*.

Como puede apreciarse en el código de ejemplo, estas funciones reciben como primer parámetro el diccionario con los valores actuales del componente tales como nombre, versión, comandos a ejecutar en cada uno de los pasos de construcción, etc. y una referencia a la función que la lógica del programa hubiera ejecutado de no haber sido sobreescrita, lo que se llama función padre o `parent_function`. Estos parámetros son pasados como argumento siempre, aunque no sean necesarios.

Los valores del objeto componente presentes en el diccionario pueden ser modificados desde el fichero `functions.py` en cualquier momento, quedando actualizado el diccionario presente en la lógica del programa desde ese momento. La función `set_attributes` se ejecuta justo después de instanciar el objeto correspondiente, por lo que permite personalizar este diccionario antes de realizar cualquier otra etapa de la construcción del elemento. Esto permite, por ejemplo, modificar valores del diccionario en función de los valores de ciertos parámetros del fichero de configuración principal o de otros parámetros definidos en su correspondiente receta, que también están disponibles en el diccionario pasado como primer argumento.

Las funciones disponibles para ser sobreescritas, así como su objetivo principal, se enumeran en la siguiente tabla:

Función	Descripción
apply_patches	Función que busca y aplica un parche en el código fuente del componente.
build	Define los pasos a ejecutar para construir los componentes de un determinado constructor.
extract_source_code	Define la forma en que se extrae el código fuente de un componente.
get_components_to_build_list	Función que define la lista de componentes a construir en un cierto constructor.
modify_xmlfile	Modifica el fichero XML del libro para un componente antes de realizar el análisis de los comandos.
run_post_steps	Pasos a ejecutar una vez que el componente se ha construido e instalado.
run_previous_steps	Pasos a ejecutar después de aplicar el parche al código fuente y antes de iniciar el proceso de construcción.
set_attributes	Permite modificar los atributos del objeto antes de iniciar la construcción.

Tabla 3.4 Funciones disponibles para sobreescritura en el fichero `functions.py`.

Los atributos de la receta YAML y la posibilidad de definir las funciones personalizadas presentadas anteriormente, suponen una capa de personalización extra y, si bien es verdad que el fichero `functions.py` requiere conocimientos de programación en el lenguaje *Python*, facilitan la personalización de los diferentes componentes y constructores de la herramienta. Además, permite crear nuevos elementos de manera sencilla.

Definir un nuevo constructor

La creación de un nuevo constructor requiere crear un nuevo directorio con el nombre del nuevo constructor dentro de la ruta `recipes/builders` y un fichero YAML con el mismo nombre que contenga la información básica del mismo. Es obligatorio definir los atributos `name`, `env_PATH_value` y `components_to_build` mientras el resto de atributos son opcionales. El código 3.6 muestra el contenido de la receta que define el constructor `provider`. En ella se indica que el constructor es de tipo `InfrastructureComponnetsBuilder`, que construye sus componentes desde fuera del `chroot`, no pertenece a ninguno de los libros del proyecto *Linux from Scratch*, que la variable de entorno `PATH` no hace uso del *toolchain*, y además que este constructor comprende a cuatro componentes.

```
---
name: 'provider'
base_builder: 'InfrastructureComponentsBuilder'
build_into_chroot: false
book: 'custom'
env_PATH_value: '/bin:/usr/bin:/sbin:/usr/sbin'

components_to_build: ['imggenerator', 'imgmount', 'mount', 'sources']
```

Código 3.6 Contenido de la receta `provider.yaml` para el constructor `provider`.

Definir un nuevo componente

Generar un nuevo componente requiere, al igual que crear un constructor nuevo, crear un directorio nuevo con el nombre del componente en la ruta `recipes/components` y al menos el fichero YAML que lo define. Si quisieramos, por ejemplo, añadir un nuevo componente llamado *redis* que instale la base de datos en memoria *Redis*[1] en nuestro sistema *Linux from Scratch*, tendríamos que crear un nuevo directorio `redis` en la ruta `recipes/components/redis` y el fichero `redis.yaml` con el contenido del código 3.7 mostrado a continuación.

```
---
name: 'redis'
book: 'custom'

extra_download_urls: ['http://download.redis.io/redis-stable.tar.gz']

make: 'make V=1'
install: 'make install V=1'
```

Código 3.7 Receta para el componente Redis.

Como puede verse en la receta anterior, este componente no pertenece a ningún libro de los publicados en el proyecto `Linux from Scratch` ni necesita ejecutar el comando `configure` ni otros pasos previos o posteriores a la instalación.

3.1.3 Plantillas

El otro elemento de personalización del programa *LFSBuilder* son las plantillas. Una plantilla es un fichero que se usa como esqueleto base para realizar ciertas acciones del programa. Inicialmente, la herramienta *Linux from Scratch Builder* define dos plantillas: `script.tpl` y `setenv.tpl`.

script.tpl

Esta plantilla es el esqueleto de los scripts generados para ejecutar los diferentes comandos de cada uno de los pasos de construcción de un componente. Este fichero base es copiado por el programa al directorio de construcción del componente y renombrado según el paso al que pertenece. Una vez el fichero destino existe, la herramienta *LFSBuilder* añade los comandos que debe ejecutar y por último ejecuta dicho *script*.

El fichero `script.tpl` es muy simple. Únicamente incorpora una serie de comentarios con los que dar información y contexto al usuario en caso de necesitar depurar un error de ejecución e importa el entorno de compilación definido para el constructor al que pertenece el componente en construcción. El contenido del fichero `script.tpl` se muestra en el código 3.8.

```
#!/bin/bash
# Component key name: '@@LFS_COMPONENT_KEYNAME@@'
# Builder name: '@@LFS_BUILDER_NAME@@'

# Load custom environment
. @@LFS_SETENV_FILE@@
```

Código 3.8 Contenido del fichero `script.tpl`.

[1] Redis: más información en la URL *https://redis.io*

Así, el fichero `script.tpl` informa del nombre del componente que se está construyendo y el constructor al que pertenece. Por último, incluye el entorno definido por la plantilla `setenv.tpl` y personalizado en la receta de cada constructor. Las cadenas `@@LFS_[atributo]@@` definen una serie de marcadores que la herramienta intenta sustituir de manera autónoma con los valores del archivo de configuración `config.py` y la lista de sustituciones `component_substitution_list` definida en la receta del componente. Además, la herramienta *LFSBuilder* mostrará un mensaje de error en caso de que exista una cadena `@@LFS_[atributo]@@` pendiente de sustituir en el *script* que se va a ejecutar.

setenv.tpl

La herramienta *LFSBuilder* adecua el entorno de compilación a las características de cada constructor. Así, cada constructor genera un *script* llamado `setenv.sh` que es incluido en los diferentes *scripts* utilizados para construir los componentes. Esto incrementa el aislamiento de la construcción del componente con el resto del sistema.

Este aislamiento extra se consigue con el comando utilizado para la ejecución del *script*, por defecto `env -i /bin/bash`, y que puede personalizarse con el atributo `runscript_cmd` de la receta del constructor. Así, el comando `env -i` asegura que el *script* se ejecuta con un entorno de compilación limpio, sin interferencias de las variables de entorno del usuario que está ejecutando la herramienta *LFSBuilder*. Dado que el entorno está vacío cuando se inicia la ejecución del *script*, incluir el contenido del *script* `setenv.sh` generado por el constructor asegura que el componente se compilará en el entorno deseado.

El entorno de compilación utilizado en la herramienta *Linux from Scratch Builder* es una modificación del entorno descrito en la sección *2.1.1 Preparación del entorno y requisitos previos* de la página 12 al que se añaden nuevas opciones y variables. Estas nuevas opciones se enumeran en la lista siguiente:

- `set -u`: esta opción del intérprete *bash* produce que la ejecución de un comando falle en caso de que en dicho comando se intente utilizar alguna variable desconocida. Es, por tanto, una medida de protección ya que el comando `sudo rm -rf /${UNSET_VARIABLE}*` fallará en lugar de ejecutar el comando `sudo rm -rf /*`.
- `PATH`: esta variable de entorno indica al intérprete de la línea de comandos en qué directorios debe buscar los binarios a utilizar.
- `set -e`: esta opción del intérprete *bash* produce que la ejecución de un *script* se pare inmediatamente en caso de que algún comando falle.
- `MAKEFLAGS`: esta variable de entorno define opciones para el comando `make`.
- `TERM`: la variable de entorno `TERM` define el tipo de terminal en uso. Según el tipo de terminal especificado pueden variar los colores y la tipografía empleados.

3.1.4 Tests unitarios

Como parte del proceso de diseño e implementación de la herramienta `Linux from Scratch Builder` se han incluido una serie de tests unitarios en el directorio `tests` del código fuente del programa.

Un test unitario representa la prueba mínima que puede realizarse sobre un fragmento de código para comprobar su correcto funcionamiento. Estas pruebas ayudan al proceso de desarrollo ya que permiten comprobar si los requisitos que comprueba un test en concreto se cumplen o no, o si una modificación del código fuente provoca que algún requisito deje de cumplirse correctamente. Todo lo anterior permite que el código fuente de una aplicación tenga, por lo general, menos fallos que una aplicación sin tests unitarios, una mayor calidad, y que los fallos se descubran lo antes posible.

Los test unitarios deben cumplir las siguientes características:

1. **Nombre claro y conciso**: el nombre de un test unitario debe ser lo suficientemente descriptivo, conciso y claro. Esto permite conocer fácilmente qué hace el test y qué parte del código analiza.
2. **Independencia**: los tests unitarios deben ser independientes entre sí y poder ejecutarse individual y aleatoriamente sin que ello afecte al resultado.
3. **Automatizable**: si los tests unitarios pueden ejecutarse de manera automática significa que pueden ser ejecutados periódicamente sin necesidad de intervención manual. La mayoría de programas dedicados al control de versiones de *software* permiten ejecutar algún comando automáticamente después de realizar cambios. La ejecución de los tests unitarios en dicho punto ayudaría a, como se ha mencionado anteriormente, descubrir fallos introducidos en la última modificación de código.

Existe, además, un método de desarrollo de *software* basado en tests llamado *Test Driven Development*, o TDD. En esta metodología, los tests unitarios se desarrollan antes que el propio programa, como implementación de los requisitos, y posteriormente se programa el código fuente del programa que cumplen dichos tests. Esta no es la metodología en que se basa la herramienta *LFSBuilder*, pero se menciona en este punto para puntualizar la importancia de los tests en el proceso de desarrollo de un *software*.

Los tests del programa *LFSBuilder* están implementados en Python usando la librería `unittest` nativa y realizan pruebas en varios de los componentes del programa, como el archivo de configuración o la interfaz de línea de comandos.

Es posible ejecutar todos los tests a la vez o únicamente los tests relativos a un componente específico. El código 3.9 muestra los comandos necesarios.

```
# Ejecutar todos los tests unitarios presentes en el directorio 'tests'
python -B -m unittest discover --verbose tests/

# Ejecutar únicamente los tests unitarios relativos al fichero de configuración
python -B -m unittest --verbose tests.test_config
```

Código 3.9 Comandos para ejecutar los tests unitarios de la herramienta *LFSBuilder*.

Actualmente, la herramienta *LFSBuilder* dispone de setenta y seis, 76, tests unitarios que comprueban el correcto funcionamiento o tipo de dato de los diferentes elementos del programa. Siguiendo el ejemplo mostrado en el código 3.9 anterior, los tests para el fichero de configuración `config.py` comprueban que el tipo de dato de cada atributo son los esperados por el programa.

3.1.5 Interfaz de línea de comandos

La interfaz de línea de comandos de la herramienta *LFSBuilder* permite al usuario interactuar con la misma. Está desarrollada con el módulo nativo de Python llamado `argparse`.

Este módulo permite definir de manera sencilla las opciones de la interfaz que se desea implementar y las opciones a ejecutar para cada una de ellas. Existen una serie de acciones predeterminadas, pero también permite definir acciones personalizadas, como las definidas en el fichero `actions.py`.

En el caso de la herramienta *LFSBuilder*, la interfaz de línea de comandos responde al siguiente diagrama de casos de uso.

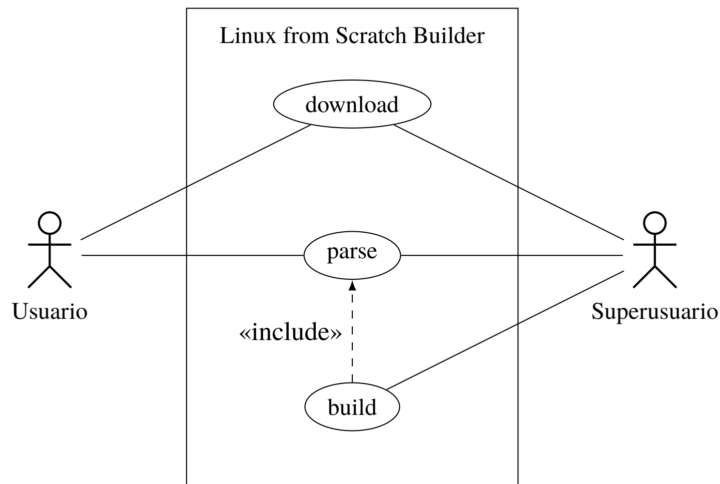


Figura 3.11 Diagrama de casos de uso de la aplicación *LFSBuilder*.

Como puede verse en la figura anterior, un usuario sin privilegios es capaz de ejecutar tanto los comandos de descarga de información como los comandos de análisis y generación de ficheros de comandos para los diferentes constructores de los libros *Linux from Scratch* y *Beyond Linux from Scratch*. En cuanto al último comando, `build`, la herramienta requiere ser ejecutada por un usuario con privilegios por la necesidad de ejecutar ciertas operaciones en el sistema generalmente atribuidas exclusivamente a usuarios administrativos, por ejemplo el uso de los comandos `chroot` y `losetup`.

Así, la estructura de la interfaz de línea de comandos que responde al diagrama presentado en la figura 3.11 se muestra en el código 3.10.

```
[sudo] python -B lfsbuilder.py [opciones lfsbuilder] comando [opciones comando]
```

Código 3.10 Estructura de la interfaz de línea de comandos para la herramienta *LFSBuilder*.

Como se ha comentado anteriormente, el comando `build` requiere privilegios de superusuario en la máquina para poder hacer uso de funciones especiales del sistema, de ahí que el comando `sudo` se optativo. La opción `-B` de Python instruye al intérprete a no generar los ficheros `.pyc` con el *byte code* del correspondiente módulo con extensión `.py`. Esta opción no es obligatoria, aunque sí recomendable ya que el uso de los ficheros precompilados con extensión `.pyc` puede dar lugar a errores en ejecuciones sucesivas en caso de que sean ejecutados utilizando una versión del intérprete diferente a la que los generó.

Las `opciones de lfsbuilder` permiten dar valor a ciertos parámetros del fichero `config.py` para modificar el comportamiento de la herramienta durante su ejecución. Los valores impuestos por línea de comandos tienen preferencia frente a aquellos presentes en el fichero de configuración. El parámetro

`--help` ofrece más información al respecto de las opciones disponibles. El argumento `comando` indica a la herramienta *Linux from Scratch Builder* la acción que se desea ejecutar. Las acciones disponibles son `download`, `parse` y `build`. Estas opciones están resumidas en el apartado *3.1.1 lfsbuilder.py* de la página 43.

La independencia entre las distintas acciones de la herramienta facilita la ejecución de las mismas según sea necesario. Esto implica, como contrapartida, el tener que ejecutar más pasos para obtener el resultado completo, pero permite ahorrar tiempo con acciones innecesarias. Por ejemplo, la ejecución de los comandos de descarga podría no ser necesaria en caso de que la máquina empleada para la construcción del sistema operativo *Linux from Scratch* sólo tuviera conexión a la red local y recibiera los ficheros de código fuente a través de otra máquina de la misma red mediante el protocolo *NFS*. Así, se ha decidido dar la posibilidad de ejecutar las diferentes acciones por separado, aunque por diseño se ha decidido incluir la ejecución del comando `parse` como parte del comando `build` para realizar ambas acciones con cada constructor que se pretende construir de manera automática.

Las `opciones del comando` dan libertad al usuario para personalizar la ejecución del comando en cuestión. Estas opciones permiten elegir el tipo de elemento a descargar, el sistema de inicio, si se debe o no montar el directorio `sources` en el directorio base, o si se desea iniciar el proceso de ejecución por un componente concreto, entre otros. Como úlitmo argumento, la línea de comandos debe recibir una lista de constructores con los que interactuar.

En el caso de los libros *Linux from Scratch* y *Beyond Linux from Scratch*, la lista de constructores debe cumplir una serie de restricciones en cuanto al orden y la presencia o no de ciertos elementos. Estas limitaciones son las siguientes:

- Un constructor no puede aparecer duplicado en la lista de constructores.
- Los constructores del libro *Linux from Scratch* siguen el orden siguiente: `toolchain`, `system` y `configuration`.
- Estos constructores pueden construirse individualmente en cualquier orden, pero necesitan ser construidos en el orden correcto cuando hay varios presentes.
- El constructor `blfs` correspondiente al libro *Beyond Linux from Scratch* puede construirse por sí mismo.
- En presencia de alguno de los constructores del libro *Linux from Scratch* y el constructor `blfs`, el constructor perteneciente al libro *Beyond Linux from Scratch* debe ser construido en último lugar.

Las restricciones referidas a la no duplicidad de constructores y al libro *Beyond Linux from Scratch* se han implementado en el código fuente de la herramienta *LFSBuilder* mediante el uso de estructuras condicionales dada su simplicidad.

La lógica necesaria para implementar el resto de restricciones de la lista anterior consiste en una función lógica de cinco variables obtenida como solución del correspondiente *Mapa de Karnaugh*. Se ha optado por esta solución debido a que el problema consiste en verificar la validez o no de una cierta combinación de variables binarias. El código 3.11 muestra la función `check_lfs_builders_order` encargada de comprobar dicha validez.

```
def check_lfs_builders_order(t, s, c, m, la):

    # This function checks if provided 'lfs' book's builders combination
    # is valid (order does matter) by implementing solution for the
        corresponding
    # 5 variable's Karnaugh map where inputs are:
    #
    # t: whether the 'toolchain' builder is pretended to be built or not
    #
    # s: whether the 'system' builder is pretended to be built or not
    #
    # c: whether the 'configuration' builder is pretended to be built or not
    #
    # m: whether the 'toolchain' builder is pretended to be built first or not.
    # That is, if its index value is the minimum between them.
    #
    # la: whether the 'system' builder is pretended to be built before
    # the 'configuration' builder or not. That's true if its index value
    # is the minimum between them.


    # Ensure we are getting boolean values as input
    t = bool(t)
    s = bool(s)
    c = bool(c)
    m = bool(m)
    la = bool(la)

    return (not(t) and la) or \
           (not(s) and not(c)) or \
           (not(t) and not(c)) or \
           (not(t) and not(s) and m) or \
           (not(t) and not(s) and not(m)) or \
           (s and m and la) or \
           (not(c) and m)
```

Código 3.11 Función `check_lfs_builders_order` como solución de un Mapa de Karnaugh.

El proceso de obtención de dicha función y la solución del mencionado *Mapa de Karnaugh* puede consultarse con mayor detalle en el *apéndice A: Mapa de Karnaugh* de la página 79.

3.1.6 De los datos al *script*

El proceso de construcción de un componente en la herramienta *Linux from Scratch Builder* conlleva varios pasos. A grandes rasgos, este proceso requiere leer los datos del libro y la receta del componente, generar los *scripts* de construcción, y ejecutarlos. Describiremos el proceso en esta sección construyendo el componente *kernel* del constructor `configuration` del libro *Linux from Scratch* como ejemplo.

Los procesos de creación de constructores y componentes se explican en las secciones *3.1.2 Definir un nuevo constructor* y *3.1.2 Definir un nuevo componente* de las páginas 50 y 51 respectivamente, por lo que se obvian en esta sección. El contenido de la receta para la construcción del componente *kernel* puede consultarse en el código 3.12.

```
---
name: 'kernel'
package_name: 'linux'
kernel_config_filename: 'kernel_config_version_4_15_3_arch_x86_64'

disable_commands_list: ['make menuconfig',
                        'mount --bind /boot /mnt/lfs/boot']

# We use the 'configure' step, which is empty on the book, to ensure that
# provided config file is valid for the kernel version we are trying to build
# before actually building it with 'make', by running the
# 'make olddefconfig' command.
#
# By doing this, we ensure that the 'make oldconfig' command,
# which run an interactive session, won't be run and that all the
# required options for the new kernel version are set with its default values.
#
# Uncomment below line to activate this functionality

# configure: 'make olddefconfig'
```

Código 3.12 Receta en formato YAML para el componente *kernel* del libro *Linux from Scratch*.

Una vez definidas las recetas necesarias, el siguiente paso es descargar los ficheros XML del libro al que pertenece el componente, si fuera necesario, y el código fuente del mismo. Para ello, la interfaz de línea de comandos de la herramienta *LFSBuilder* dispone de la acción `download`. El código 3.13 muestra los comandos necesarios para descargar tanto los ficheros XML del libro *Beyond Linux from Scratch*, como el código fuente de los componentes que forman dicho constructor.

```
# Descarga de los ficheros XML para el libro Beyond Linux from Scratch
python -B lfsbuilder.py download --xml lfs

# Descarga de los ficheros de código fuente de los componentes del libro BLFS
python -B lfsbuilder.py download --sources lfs
```

Código 3.13 Comandos para la descarga de los ficheros XML y el código fuente de los componentes del libro *Beyond Linux from Scratch*.

Una vez descargados los ficheros de código fuente y los ficheros XML del libro, es posible ejecutar la acción `build` para que se inicie el proceso de construcción. El código 3.14 muestra el comando necesario que ejecuta dicha opción. En este código forzamos a crear y montar un fichero en el que construir el sistema operativo con los argumentos `--generate-img-file` y `--mount-img-file`, y a montar el directorio `sources` en el directorio de construcción con el argumento `--mount-sources`.

```
# Construir los componentes del libro Beyond Linux from Scratch
python -B lfsbuilder.py build --generate-img-file \
                              --mount-img-file \
                              --mount-sources lfs
```

Código 3.14 Comando para la construcción de los componentes del libro *Beyond Linux from Scratch*.

Una vez se inicia la ejecución del comando `build`, el proceso de construcción se describe en la lista de la página siguiente. El resto de esta página se deja en blanco a propósito para facilitar al lector el seguimiento del proceso.

1. Primero se realizan las comprobaciones y modificaciones necesarias de la lista de constructores. Así, se añaden los constructores `provider` y `collector` al principio y al final de la misma, respectivamente.
2. A continuación se realizan una serie de comprobaciones y modificación de la configuración proporcionada por el usuario para personalizar la ejecución de la herramienta *LFSBuilder*.
3. El objeto `LFSBuilder` itera con la lista de constructores para construir cada uno de los elementos de la lista. Este proceso está compuesto, para cada constructor, por los siguientes pasos:
   - **3.**1 Generar un objeto `BuilderGenerator` que lea la información de la receta del constructor en cuestión e instancie, uniendo la información leída y una serie de valores por defecto y/o de configuración, un objeto del tipo de constructor requerido en la receta de forma dinámica. Obtener la lista de componentes a construir ejecutando, si existe, la función homónima presente en el fichero `functions.py`.
   - **3.**2 Desechar este objeto `BuilderGenerator`.
   - **3.**3 Llamar a la función `set_attributes()` del objeto constructor generado dinámicamente. Esta función intenta llamar a la función homónima del fichero opcional `functions.py` presente en el directorio de definición del constructor en caso de existir.La utilidad principal de esta función es la de generar y/o modificar atributos del propio constructor.
   - **3.**4 Llamar a la función `build()` del objeto constructor que, en caso de estar sobreescrita en el fichero `functions.py` es ejecutada.
   - **3.**5 En caso de no existir dicha función o realizar una llamada a la función padre dentro de dicha función sobreescrita, lo primero es generar el fichero con las diferentes variables de entorno a partir de la plantilla `setenv.tpl` presentada en la página 52 y parsear los ficheros XML del libro con los comandos necesarios, generando el fichero de comandos en el directorio `tmp`.
   - **3.**6 A continuación, para cada componente:
     - **3.**6.1 Instanciar un objeto de tipo `ComponentGenerator` para, al igual que pasaba con los constructores, instanciar de manera dinámica el objeto componente a construir. Este objeto se genera mediante la información proporcionada por los archivos XML del libro, en caso de que el constructor pertenezca a alguno de los libros del proyecto *Linux from Scratch*, la información presente en la receta del componente, la configuración personalizada por el usuario desde el archivo `config.py` o la línea de comandos, y algunos parámetros del constructor, como el nombre.
     - **3.**6.2 Desechar el objeto `ComponentGenerator`.
     - **3.**6.3 Llamar a la función `set_attributes()` si está presente en el fichero `functions.py` del directorio del componente o ejecutar la implementada en la lógica de la herramienta *LFSBuilder*. La utilidad principal de esta función es la de modificar parcialmente alguno de los comandos presentes en el libro o los propios atributos del objeto en función de ciertos valores de configuración o de otros atributos del mismo objeto.
     - **3.**6.4 Llamar a la función `build()` del objeto componente que, en caso de estar sobreescrita en el fichero `functions.py` es ejecutada.
     - **3.**6.5 Limpiar el espacio de trabajo utilizado durante el proceso de construcción.
     - **3.**6.6 Liberar la memoria utilizada por el objeto componente.
   - **3.**7 Limpiar el espacio de trabajo utilizado durante el proceso de construcción.
   - **3.**8 Liberar la memoria utilizada por el objeto constructor.

Si observamos con un poco más de profundidad el proceso de construcción del componente, esto es, la llamada a la función `build`, para construir un componente son necesarias las siguientes acciones:

1. **Descomprimir el código fuente.** Esta lógica es capaz de extraer el código fuente comprimido de un componente. Puede personalizarse en el fichero `functions.py` del directorio del componente.
2. Aplicar los parches al código fuente. Este método, `apply_patches`, también puede sobreescribirse.
3. Ejecutar los pasos previos almacenados en el diccionario `component_data_dict` del objeto componente con la clave `previous`. Para ello, se genera un *script* llamado `previous.sh` a partir de la plantilla `script.tpl` mencionada en la página 51 dentro del directorio del código fuente descomprimido. A continuación se ejecuta este *script* desde fuera o dentro del `chroot` del directorio principal según lo indicado por el constructor y/o el propio componente, usando como comando de ejecución el valor del atributo `runscript_cmd`.
4. Generar y ejecutar el *script* de compilación del código fuente es el siguiente paso. Para ello se genera el *script* `compile.sh` dentro del directorio del código fuente del componente y se ejecuta de la misma manera que el *script* anterior. Los valores que componen este *script* vienen definidos por los atributos `configure`, `make`, `make test` y `make install`, además de otros atributos opcionales listados en la tabla 3.3 de la página 47.
5. Por último, una vez el componente ha sido construido se ejecutan los comandos incluidos en el atributo `post` para configurar o realizar las modificaciones necesarias. Para ello, al igual que en pasos anteriores, se genera y ejecuta el *script* `post.sh` con los valores almacenados en el diccionario del objeto componente en construcción.

El proceso explicado es el que siguen los objetos instanciados de tipo `CompilableComponent`. Para la construcción de componentes del tipo `SystemConfigurationComponent`, el proceso anterior se reduce al último punto exclusivamente debido a que estos componentes no dependen de ningún código fuente ni requieren de compilación. Únicamente configuran y modifican ciertas partes del sistema operativo en construcción.

Continuando con el caso concreto del componente *kernel*, una vez que se instancia la clase `CompilableComponent`, se genera un diccionario con los datos de la receta mostrada en el código 3.12 de la página 57 y los datos obtenidos del libro. El código 3.15 mostrado a continuación contiene los datos extraídos del libro *Linux from Scratch* para el componente *kernel*.

```
<component name="kernel">
    <version/>
    <md5/>
    <url/>
    <require_build_dir>0</require_build_dir>
    <previous>make mrproper</previous>
    <configure/>
    <make>make</make>
    <test/>
    <install>make modules_install
cp -iv arch/x86/boot/bzImage /boot/vmlinuz-4.18-lfs-8.3-rc1
cp -iv System.map /boot/System.map-4.18
cp -iv .config /boot/config-4.18
install -d /usr/share/doc/linux-4.18
cp -r Documentation/* /usr/share/doc/linux-4.18</install>
           <post>install -v -m755 -d /etc/modprobe.d
cat &gt; /etc/modprobe.d/usb.conf &lt;&lt; &quot;EOF&quot;
# Begin /etc/modprobe.d/usb.conf

install ohci_hcd /sbin/modprobe ehci_hcd ; /sbin/modprobe -i ohci_hcd ; true
install uhci_hcd /sbin/modprobe ehci_hcd ; /sbin/modprobe -i uhci_hcd ; true

# End /etc/modprobe.d/usb.conf
EOF</post>
</component>
```

Código 3.15 Comandos extraídos del libro *Linux from Scratch* para el componente *kernel*.

En el siguiente paso, el programa busca y descomprime dentro del directorio `sources` el fichero que contiene el código fuente del kernel de Linux. Una vez descomprimido pasa a ejecutar los comandos indicados por el atributo `previous` a menos que la función `run_previous_steps` esté sobreescrita como es el caso del componente *kernel*. Así, lo que en realidad se ejecuta es la función mostrada en el código 3.16.

```
def run_previous_steps(component_data_dict, parent_function):

    # Call parent function
    parent_function()

    print("Copying custom \'.config\' file")
    filename = os.path.join(component_data_dict["lfsbuilder_src_directory"],
                            "recipes",
                            "components",
                            "kernel",
                            "files",
                            component_data_dict["kernel_config_filename"])

    tools.copy_file(filename,
                    os.path.join(component_data_dict["extracted_directory"], ".
                        config"))
```

Código 3.16 Función `run_previous_steps` sobreescrita por el componente *kernel*.

Como puede extraerse del código 3.16, la función llama en primer lugar a la función homónima presente en la lógica del programa, por lo que ejecuta el comando `make mrproper` para eliminar los posibles archivos de configuración del kernel presentes en el directorio, y luego copia el fichero indicado por el

atributo `kernel_config_filename` presente en el subdirectorio `files` de la receta con el objetivo de compilar el kernel usando esa configuración específica.

El kernel de Linux tiene una peculiaridad, y es que requiere usar un archivo de configuración adaptado a los parámetros de la versión que se pretende construir. Debido a esto, es necesario o bien generar un fichero de configuración para cada nueva versión del kernel o bien dejar que el propio archivo `Makefile` actualice el fichero de configuración presente con los valores por defecto de los nuevos parámetros. En caso de querer hacer uso de esta funcionalidad es necesario, tal y como se indica en la receta del kernel de la página 57, descomentar una línea para agregar el comando `make olddefconfig` al proceso de compilación que se desarrolla en el *script* `compile.sh`.

El contenido del *script* `compile.sh` puede consultarse en el código 3.17 que incluye los valores de los atributos `make` e `install`.

```
#!/bin/bash

# Component key name: 'kernel'
# Builder name: 'configuration'

# Load custom environment
. /setenv.sh

make
make modules_install
cp -iv arch/x86/boot/bzImage /boot/vmlinuz-4.18-lfs-8.3-rc1
cp -iv System.map /boot/System.map-4.18
cp -iv .config /boot/config-4.18
install -d /usr/share/doc/linux-4.18
cp -r Documentation/* /usr/share/doc/linux-4.18
```

Código 3.17 Contenido del *script* `compile.sh` para la construcción del componente *kernel*.

Una vez que el componente se ha compilado e instalado, la herramienta *LFSBuilder* ejecuta el paso de configuración con la función `run_post_steps` y el valor del atributo `post`. En el caso del kernel de Linux, el contenido del *script* que se ejecuta se muestra en el código 3.18.

```
#!/bin/bash

# Component key name: 'kernel'
# Builder name: 'configuration'

# Load custom environment
. /setenv.sh

install -v -m755 -d /etc/modprobe.d

cat &gt; /etc/modprobe.d/usb.conf &lt;&lt; &quot;EOF&quot;
# Begin /etc/modprobe.d/usb.conf

install ohci_hcd /sbin/modprobe ehci_hcd ; /sbin/modprobe -i ohci_hcd ; true
install uhci_hcd /sbin/modprobe ehci_hcd ; /sbin/modprobe -i uhci_hcd ; true

# End /etc/modprobe.d/usb.conf
EOF
```

Código 3.18 Contenido del *script* `post.sh` para la construcción del componente *kernel*.

3.2 Dependencias

La herramienta *LFSBuilder* ha sido diseñada y desarrollada buscando ser lo más autónoma posible y con el mínimo de requisitos necesarios.

Los requisitos propios del libro no pueden obviarse en este punto, a pesar de no ser dependencias estrictas del programa *LFSBuilder*, puesto que solventan dependencias necesarias para la construcción de ciertos componentes en la etapa de creación del `toolchain`. Estas dependencias se enumeran en la tabla 3.5 a tres columnas.

make	patch	libstdc++
gcc	g++	gcc-c++
gawk	glibc	coreutils
texinfo	m4	bison
ncurses-devel	tar	xz
subversion	gzip	wget

Tabla 3.5 Dependencias del libro *Linux from Scratch*.

En cuanto a la herramienta *LFSBuilder*, los tres únicos requisitos son: el sistema de control de versiones `subversion` para poder descargar los ficheros XML del libro, el intérprete Python en su versión `2.7.x` para poder ejecutar el programa, y el paquete `PyYAML` para poder interactuar con las diferentes recetas que componen la herramienta. Este paquete puede ser instalado usando el gestor de paquetes `pip`, ejecutando el comando presente en el código 3.19 de la página 64.

```
# Instalar el paquete PyYAML usando el gestor pip
pip install PyYAML
```

Código 3.19 Comando para la instalación del paquete `PyYAML` usando el gestor `pip`.

Como se comentaba al principio de esta sección, una de las ideas detrás de la herramienta *LFSBuilder* es la de tener el menor número de requisitos posibles para que pueda ser empleada en prácticamente cualquier sistema operativo Linux. En esta línea, el único requisito que no viene por defecto en una distribución habitual de Python es un módulo para el tratamiento de ficheros en formato YAML, de ahí la necesidad de instalar el paquete `PyYAML` usando `pip`. En caso de existir en un futuro un módulo nativo incluido para tal fin, este requisito podría eliminarse.

En cuanto a subversion es necesario destacar aquí que los propios desarrolladores del proyecto *Linux from Scratch* ponen a disposición de los usuarios los ficheros HTML para ser descargados. Python dispone de un modulo nativo para parsear ficheros en tal formato, pero es más sencillo trabajar con los ficheros XML originales que con los ficheros HTML ya procesados porque estos últimos no hacen distinción alguna entre comandos como sí ocurre en el caso de los ficheros XML. Además, para el uso de subversión desde Python se ha descartado incluir la instalación de una librería externa mediante `pip` porque el uso de subversión es tan escueto en el código fuente de la herramienta *LFSBuilder* que no compensa el uso espacio extra requerido en la máquina para instalar la librería y sus dependencias frente al hipotético uso que se haría de dicha librería.

Un ejemplo de uso de librerías nativas frente a librerías externas es el caso del parser de ficheros XML. Si bien existen alternativas más sencillas de usar como es el caso del paquete `Beautiful Soup`, se ha preferido utilizar el módulo nativo `xml` de Python para reducir la instalación de componentes adicionales en la máquina que ejecuta la herramienta.

3.3 jhalfs vs. LFSBuilder

La comparación de ambas herramientas es una tarea compleja, puesto que utilizan tecnologías y paradigmas de programación diferentes. Aún así, es posible nombrar ciertos aspectos que distinguen una herramienta de otra.

3.3.1 Tecnologías

El primer aspecto, y al mismo tiempo el que representa la diferencia más obvia, es el de la tecnología utilizada para su desarrollo. Bash en el caso de *jhalfs* y Python para *LFSBuilder*. Bash es un lenguaje muy potente, pero tiende a ser complicado a medida que es necesario añadir funcionalidades algo más avanzadas, como es el caso del análisis de ficheros XML o similares, o enlazar unos comandos con otros mediante el uso de *pipes* o tuberías. Python, sin embargo, facilita mucho estas tareas al tratarse de un lenguaje de más alto nivel y algo más próximo al lenguaje humano, lo que facilita su lectura y comprensión. Además, Python es multiplataforma y de propósito general, y puede adaptarse a multitud de tareas de forma más o menos simple. Ambos intérpretes suelen estar incluidos en la mayoría de distribuciones Linux por defecto.

3.3.2 Dependencias

En cuanto a requisitos, los requisitos de la herramienta *LFSBuilder* se enumeraron en el apartado *3.2 Dependencias* de la página 63, y los de *jhalfs* se comentaron a lo largo de la sección *2.3 Automated Linux from Scratch* de la página 27, pero se listan a modo de comparativa en la tabla 3.6 siguiente.

jhalfs	LFSBuilder
subversion	subversion
bash	Python
wget	PyYAML
libxslt	
libxslt-dev	
libxml2	
libxml2-dev	
libxml2-utils	
xsltproc	
libncurses5-dev	
libncursesw5-dev	
libtinfo	

Tabla 3.6 Comparativa de dependencias entre las herramientas *jhalfs* y *LFSBuilder*.

Como puede verse en la tabla anterior, la herramienta *jhalfs* tiene más requisitos que la herramienta *LFSBuilder*, la mayoría de ellos relativos al tratamiento de ficheros XML debido a que el lenguaje Bash no es capaz de parsear estos ficheros de forma nativa. *LFSBuilder*, en cambio, solventa este requisito mediante el uso de la librería `xml` nativa de Python, tal y como se comentó en apartados anteriores, ahorrando así el espacio de disco duro para la instalación de dichas dependencias.

Algo similar ocurre con las dependencias de *jhalfs* relacionadas con `ncurses`. La herramienta *jhalfs* necesita estas librerías para mostrar el menú de configuración al ejecutar el comando `make menuconfig` con el que configurar e iniciar la herramienta. El programa *LFSBuilder*, sin embargo, emplea una interfaz de línea de comandos para interactuar con el usuario.

3.3.3 Rendimiento

Otro aspecto en el que podríamos realizar una comparativa es en cuanto al rendimiento de ambas herramientas. Es cierto que las pruebas de rendimiento pueden ser un tanto capciosas puesto que dependen de factores externos y no sólo de aquellos inherentes a la aplicación o el lenguaje de programación en cuestión, sino también al sistema operativo, a los recursos de la máquina y a la carga de trabajo soportada durante el análisis. Aún así, en el caso que nos ocupa se ha ejecutado cinco veces el proceso de construcción del libro *Linux from Scratch* con cada herramienta dentro de una máquina virtual con el sistema operativo Debian 9.5 de 64 bits, kernel 4.9.0-7-amd64, una unidad de CPU y 1 gigabyte de memoria RAM. La versión del libro testeada es la *8.0*, versión más reciente soportada[2] por la herramienta *jhalfs*. Además, ambas herramientas han utilizado el mismo fichero de configuración para construir el *kernel* de Linux.

[2] más información de las versiones soportadas por la herramienta *jhalfs* en la URL *http://wiki.linuxfromscratch.org/alfs/wiki/SupportedBooks*

A continuación, la tabla 3.7 muestra los resultados de la ejecución de ambos programas para varios apartados estadísticos. Estos datos se han extraído de la información proporcionada por el comando `/usr/bin/time --verbose`.

	jhalfs					LFSBuilder				
Parámetro	Valor 1	Valor 2	Valor 3	Valor 4	Valor 5	Valor 1	Valor 2	Valor 3	Valor 4	Valor 5
Tiempo total	4:27:34	4:32:27	4:47:21	4:48:17	4:46:01	2:29:58	2:17:41	2:19:03	2:12:14	2:13:43
Valor medio	4:40:20					2:18:32				
% uso de CPU	69	73	71	71	73	79	73	72	73	74
Valor medio	71					74				
Uso de memoria máximo	907 MB	913 MB	885 MB	888 MB	879 MB	817 MB	835 MB	819 MB	832 MB	832 MB
Valor medio	894 MB					827 MB				
Cambios de contexto voluntarios	28.650.606	32.867.105	31.956.487	32.401.772	31.016.072	5.586.153	5.563.453	5.518.837	5.713.650	5.595.025
Valor medio	31.378.408					5.595.424				
Uso de swap	0	0	0	0	0	0	0	0	0	0
Valor medio	0					0				

Tabla 3.7 Comparativa de ejecución entre las herramientas *jhalfs* y *LFSBuilder*.

A continuación se analizan algunos de los resultados mostrados en la tabla 3.7 anterior. Uno de los datos más llamativos de la tabla 3.7 anterior es la del tiempo requerido para la construcción del sistema operativo. Como puede observarse, la herramienta *jhalfs* necesitó más tiempo que *LFSBuilder* en completar la misma tarea. La valoración de estos datos depende mucho del tipo de máquina y sus características de CPU y memoria RAM. En este caso, ambas máquinas eran iguales en ambos parámetros, y la utilidad que se le ha dado a las máquinas ha sido de total dedicación a la construcción del sistema operativo *Linux from Scratch*.

Una diferencia de tiempo cercana a las dos horas es una diferencia sorprendente. Analizando la construcción de los diferentes componentes del proceso se ha descubierto que la diferencia de tiempo antes mencionada recae, prácticamente en su totalidad, en el tiempo requerido para la construcción del componente `gcc` del sistema final. Así, el tiempo requerido por la herramienta *LFSBuilder* para construir este componente es de unos 20 minutos, mientras que la herramienta *jhalfs* necesitó alrededor de los 140 minutos, más de dos horas.

Para validar estos datos y el correcto funcionamiento del sistema operativo resultante tras la ejecución de la herramienta *LFSBuilder* se han llevado a cabo las siguientes comprobaciones:

1. Ambas herramientas utilizan las mismas órdenes y parámetros para la construcción del componente `gcc` dado que las obtienen de la misma versión del libro y no realizan ninguna modificación sobre las mismas.
2. Los desarrolladores del libro *Linux from Scratch* emplean la herramienta *jhalfs* para probar los comandos presentes en cada una de las versiones del libro y luego publican estos datos, así como los diferentes *logs* de compilación de los componentes, en Internet para la consulta de los usuarios. Gracias a esto, se ha comprobado que el tiempo requerido por *jhalfs* para la construcción del mencionado componente en una máquina con CPU Intel i7-4700MQ y 8 GB de memoria RAM fue de 207 minutos y 50 segundos[3]. Así pues, el tiempo de construcción requerido por la herramienta *jhalfs* en el entorno de pruebas utilizado puede considerarse válido a ojos de los datos oficiales.
3. Para descartar que este largo espacio de tiempo requerido se deba a la versión de `gcc` presente en la versión *8.0* del libro *Linux from Scratch*, se ha comprobado el reporte de la versión más reciente del libro, concretamente la versión *8.3-rc1*, en la que se necesitaron 144 minutos y 47 segundos para compilar la versión 8.2.0 de `gcc`.
4. Tras el proceso de construcción del componente `gcc` se realizan una serie de validaciones y pruebas[4] para comprobar el correcto funcionamiento de la utilidad construida. Estas pruebas son realizadas satisfactoriamente a lo largo del proceso de construcción del sistema operativo *Linux from Scratch*.
5. Debido al valor de la variable de entorno `PATH` y la opción `set +h` del intérprete de comandos `bash` que utiliza la herramienta `LFSBuilder` durante la construcción del componente `gcc` para el

[3] consultar el tiempo empleado para la construcción del componente `gcc-6.3.0` en la URL *http://www.linuxfromscratch.org/lfs/build-logs/8.0-rc1/i7-4700MQ/SVN-20170213-SBU_DU-2017-02-13.report*

[4] consulte estas comprobaciones en la URL *http://www.linuxfromscratch.org/lfs/view/stable/chapter06/gcc.html*

sistema operativo definitivo puede asegurarse que el binario generado es utilizado a partir de este punto tal y como se mencionó en el apartado *2.1.1 Preparación del entorno y requisitos previos* de la página 12, por lo que el resto de componentes se han construido empleando el mencionado binario, lo que demuestra su correcto funcionamiento.

6. Por último, y para evaluar el correcto funcionamiento del componente `gcc` en el sistema en funcionamiento se han llevado a cabo, también de manera satisfactoria, las mismas comprobaciones recomendadas por el libro *Linux from Scratch* durante la construcción del mencionado componente.

El dato del uso de memoria tiene una difícil interpretación. Por una parte, el hecho de que *jhalfs* haya usado más memoria RAM de la disponible que *LFSBuilder* podría significar que *jhalfs* hace un mejor uso de la misma, sin embargo este hecho no se refleja en una mejoría del resto de apartados bajo análisis. Al mismo tiempo, el menor consumo de memoria por parte de *LFSBuilder* significa que se trata de un proceso más liviano para la máquina que lo ejecuta.

La imagen 3.12 muestra el uso de memoria RAM de *LFSBuilder* durante uno de los procesos de construcción anteriores. Los datos fueron obtenidos con el programa para la monitorización de memoria `mprof` [5]. La monitorización de memoria empleada ha incluido la memoria utilizada por los procesos hijos derivados del proceso principal que ejecuta la herramienta *LFSBuilder*. Esta información es proporcionada por el parámetro `--include-children` de `mprof`. Como puede observarse, el valor máximo se alcanza en momentos muy concretos y excasos a lo largo de todo el proceso, y el valor medio del uso de memoria es relativamente bajo, alrededor de los 193 MB.

[5] puede obtenerse más información sobre `memory_profiler` en la URL *https://github.com/pythonprofilers/memory_profiler*

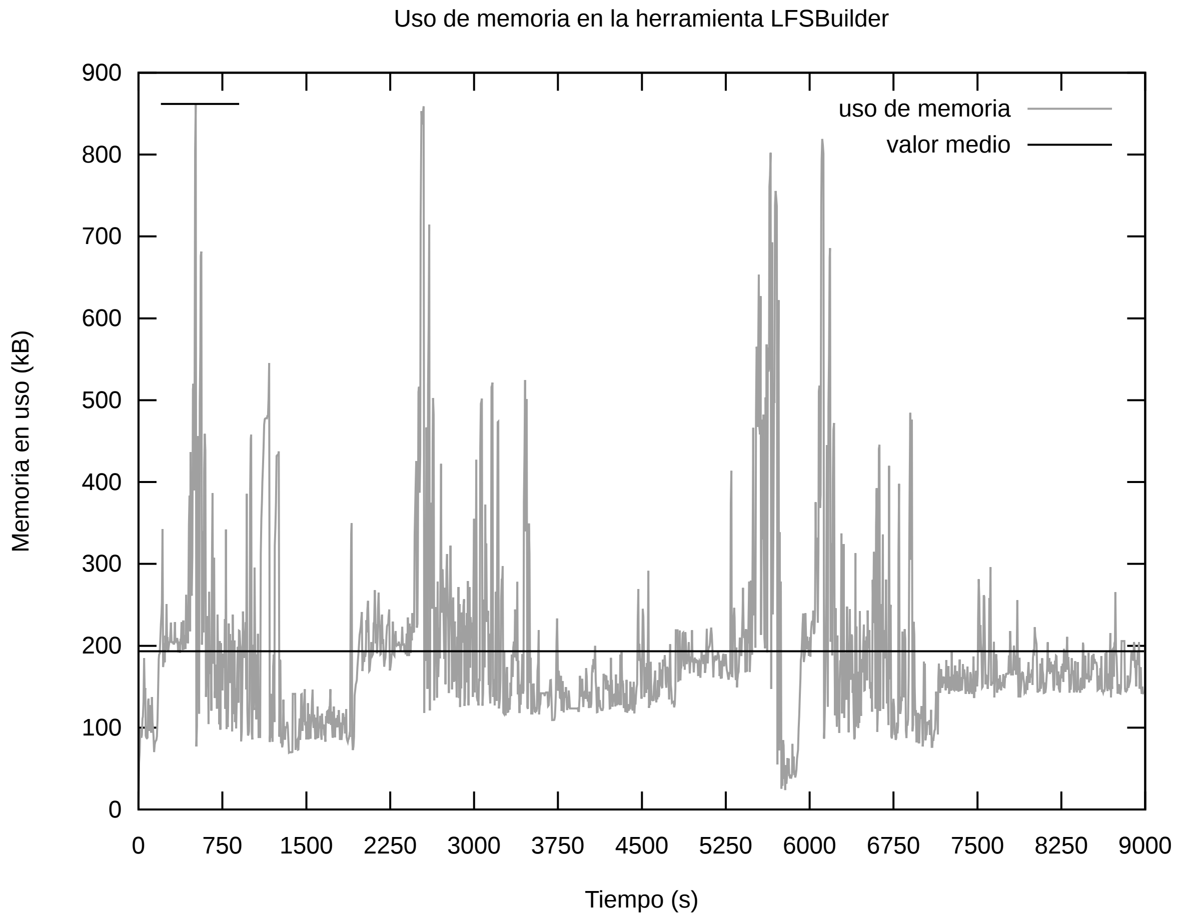


Figura 3.12 Uso de memoria durante la ejecución de la herramienta *LFSBuilder*.

El dato sobre el número de cambios voluntarios de contexto refleja el número de veces que el proceso liberó la CPU durante su periodo de uso porque se encontraba a la espera de un evento externo, por ejemplo una operación de entrada y salida o esperando el final de la ejecución de una *script* de compilación en el caso de *LFSBuilder*. En este apartado estadístico la herramienta *LFSBuilder* también obtuvo mejores resultados al necesitar, en media, un 17,8 % de los cambios de contexto voluntarios que necesitó *jhalfs*.

3.3.4 Otras características

La herramienta *jhalfs* incorpora la posibilidad de instalar un gestor de paquetes, como por ejemplo `dpkg`, el gestor de paquetes para sistemas operativos basados en Debian, y configurar los componentes construidos para ser tratados por el gestor de paquetes elegido una vez que el sistema operativo `Linux from Scratch` haya sido construido. También, la herramienta *jhalfs* almacena en un directorio los diferentes *logs* de los componentes que construye, lo que facilita el proceso de depuración y revisión de la compilación. Estas son características de las que *LFSBuilder* no dispone, pero que serán añadidas en un futuro.

Otra característica implementada en la herramienta *jhalfs* pero que *LFSBuilder* no incorpora es la gestión de dependencias para el libro *Beyond Linux from Scratch*. Esta es una tarea muy compleja que requiere, en muchos casos, la resolución de dependencias circulares. En cualquier caso, los desarrolladores de *jhalfs* no aseguran el buen funcionamiento de esta funcionalidad. Otro intento de resolver estos árboles de dependencias para el libro *Beyond Linux from Scratch* es el script `blfs-deps` disponible en el subproyecto *Hints* del proyecto *Linux from Scratch*, aunque no ha sido actualizado desde su primera versión publicada en el año 2004.

Por último, es necesario destacar en esta peculiar comparativa que la herramienta oficial *jhalfs* no soporta versiones del libro *Linux from Scratch* posteriores a la *8.0*, pero la herramienta *LFSBuilder* sí. Fue precisamente esta limitación de la herramienta *jhalfs* uno de los motivos del desarrolo de *Linux from Scratch Builder*.

4 Desarrollo y pruebas

If you're going to try, go all the way. Otherwise, don't even start.

CHARLES BUKOWSKI

El desarrollo de una herramienta como *LFSBuilder* conlleva un extenso periodo de diseño, análisis, aprendizaje y pruebas. En concreto, para desarrollar este programa ha sido necesario aprender sobre programación orientada a objetos, el lenguaje Python[7], el sistema operativo Linux[14], compilación, redes[13], y UML, entre otros.

El desarrollo del código fuente se ha realizado en un ordenador de escritorio con sistema operativo Ubuntu 16.04 LTS, procesador Intel® Core i7-6700 de 64 bits, con 8 núcleos a una frecuencia de 3.40 GHz, y 16 gigabytes de memroria RAM. En cuanto a las aplicaciones de edición empleadas se han utilizado los editores de texto `emacs` en su versión 24.5.1 y la versión 7.4 del editor `vim`. La versión de Python utilizada ha sido la 2.7.12. Para la orientación a objetos y los diagramas de flujo se ha hecho uso de la aplicación para sistemas operativos Linux `dia`, en su versión 0.97.3, una aplicación para el diseño de diagramas. La organización de tareas y control de progreso se ha llevado a cabo mediante el plugin `org-mode` [1] del editor de textos `emacs`.

En cuanto a las pruebas realizadas, el código ha sido comprobado con varios *linters* o analizadores de código para verificar que tanto el código Python como los ficheros en formato YAML se adecuan al estilo estándard. Los ficheros en formato YAML han sido analizados con el programa `yamllint`. Los ficheros de código Python se han analizado con los *linters* `pylint`, `pycodestyle` para adecuar el código a la gúia de estilo PEP-8[2].

El uso de `pylint` ha requerido deshabilitar la comprobación de ciertos códigos de error que se enumeran en la tabla 4.1.

Código	Descripción
C0103	El nombre de la variable, objeto, función, módulo o longitud de la línea supera el número de caracteres permitidos.
C0325	Paréntesis innecesarios en una expresión.
R0903	La clase tiene muy pocos métodos públicos.
W1401	Caracter de escape `\` erróneo en la expresión.

Tabla 4.1 Códigos de error deshabilitados para el *linter* `pylint`.

[1] org-mode: *https://orgmode.org*
[2] PEP-8: *https://www.python.org/dev/peps/pep-0008*

Los motivos que justifican la omisión de estos códigos de error se enumeran en la lista siguiente.

- Los códigos de error `C0103` y `C0325`, relativos al número de caracteres y los paréntesis, han sido desechados por cuestiones de estilo y legibilidad del código. Cumplir con la limitación de caracteres convertía ciertas partes del código en un auténtico código *espagueti* imposible de leer o nombres de variables no descriptivos. En cuanto a los paréntesis, algunas expresiones lógicas resultan más legibles agrupando términos con paréntesis, pero en ciertas ocasiones el analizador de código `pylint` consideraba que no eran necesarios. Así, y para favorecer la legibilidad del código, se han descartado ambos códigos de error. Además, la propia guía de estilo indica que el número máximo de caracteres que componen una línea puede modificarse desde los 79 que recomienda la guía de estilo, hasta 99 caracteres o hasta el número de caracteres que los usuarios acuerden usar. La herramienta *LFSBuilder* intenta no superar el límite alternativo recomendado de 99 caracteres a menos que sea estrictamente necesario por motivos de legibilidad.
- El código `R0903` evita que una clase sea demasiado opaca para el resto del programa porque dispone de muy pocos métodos públicos. Este es el caso, por ejemplo, de la clase `Downloader`, implementada en el módulo `downloader.py`. Esta clase, en concreto, tal y como se comentó en la subsección *3.1.1 downloader.py* de la página 42, sólo interactúa con la interfaz de línea de comandos para descargar los ficheros XML o los ficheros de código fuente de uno de los dos libros del subproyecto *Linux from Scratch*, por lo que sólo necesita tener un único método accesible desde el exterior. Debido a esto, el código de error `R0903` ha sido deshabilitado.
- En cuanto al código `W1401`, la inclusión en la lógica del programa de ciertas cadenas `XML` para deshabilitar a la hora de modificar los fichero XML de algún componente o el uso de caracteres especiales que requieren ser escapados con una barra invertida, o *backslash*, obligan a desestimar los fallos producidos por el mencionado código.

Para comprobar el correcto funcionamiento de la herramienta se han realizado múltiples ejecuciones en diferentes entornos y configuraciones. La mayoría de pruebas se han realizado en máquinas virtuales de 64 bits y de 32 bits, instalando ambos gestores de inicio y algunos componentes del libro *Beyond Linux from Scratch* además de otros componentes propios. Las pruebas se han llevado a cabo en los siguientes sistemas operativos y características de la máquina.

Sistema Operativo	Kernel	CPU	RAM
CentOS Linux 7.5.1804 64 bits	3.10.0-862.11.6.el7.x86_64	1	1GB
Debian 9.5 64 bits	4.9.0-7-amd64	1	1GB
Debian 9.5 64 bits	4.9.0-7-amd64	1	2GB
Ubuntu 16.04 64 bits	4.4.0-133-generic	8	16GB
Linux Mint 19 32bits	4.15.0-20-generic	1	512MB

Tabla 4.2 Entornos empleados para probar el funcionamiento de la herramienta *LFSBuilder*.

En todas las pruebas realizadas en los entornos descritos en la tabla 4.2 anterior, la herramienta `Linux from Scratch Builder` ha generado como resultado un fichero en formato `raw` compuesto por una única partición, formateada con el sistema de archivos `ext4`, y que posteriormente ha sido convertido a un formato de máquina virtual, concretamente VMDK, para ser probado en el hipervisor de máquinas virtuales VirtualBox. Es cierto que *LFSBuilder* es capaz de trabajar con múltiples sistemas de archivos y que un fichero que luego pueda convertirse a un formato de máquina virtual no es la salida obligatoria del programa, pero se ha hecho uso de esta opción por su comodidad para probar el resultado y la seguridad que implica estar generando el sistema operativo en un fichero independiente para no producir cambios en la máquina anfitriona en caso de error. Además, y puesto que este es un uso específico de la herramienta, la conversión del fichero *IMG* generado al formato de máquina virtual necesario se deja a cargo del usuario para no sobrecargar las máquinas de todos los posibles usuarios con requisitos opcionales.

Uno de los problemas más importantes a los que ha sido necesario hacer frente han sido, además de errores varios de compilación y falta de espacio en disco, los *kernel panic*. Un *Kernel panic* es un error referente a sistemas operativos de tipo Unix* del que no es posible recuperarse, lo que provoca que el sistema operativo nunca llegue a arrancar. Puede deberse a diversos motivos, desde una mala configuración en el momento de construir el *kernel*, a un problema en algún archivo de configuración. El único *kernel panic* detectado durante el desarrollo de *LFSBuilder* fue el reflejado en el código 4.1, referente a un sistema de archivos desconocido.

```
Kernel panic-not syncing: VFS: unable to mount root fs on unknown block(0,0)
```

Código 4.1 Mensaje de error para el *kernel panic* referido a un sistema de archivos desconocido.

Básicamente este mensaje de error indica que el sistema de archivos indicado en la directiva `root` del fichero de configuración de GRUB, `/boot/grub/grub.cfg`, está mal configurado y su directiva `root` apunta a un sistema de archivos inexistente, por lo que no puede iniciarse el sistema operativo. Solucionar este error de configuración hace que el temido *kernel panic* no se produzca y el sistema operativo arranque correctamente.

Además, y para probar de manera autónoma los cambios realizados en el código fuente, se ha hecho uso de la aplicación web para integración continua `Jenkins`. Esta aplicación web desarrollada en el lenguaje multiplataforma Java es un servidor de automatización para procesos de integración y entrega continuos, o CI/CD (*continuos integration and continuos development* por sus siglas en inglés). En el caso de `Linux from Scratch Builder`, el uso de `Jenkins` permite ejecutar automáticamente tanto los tests unitarios como un proceso de construcción bajo demanda o de manera periódica para comprobar el correcto funcionamiento de los cambios realizados en el código.

Durante el proceso de desarrollo de la herramienta *LFSBuilder* se configuraron dos trabajos en `Jenkinks` que ejecutaban tanto los tests unitarios como el comando para construir el sistema operativo *Linux from Scratch* mencionado en el código 3.14 de la página 58 una vez al día.

5 Conclusiones y líneas de avance

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing

LFSBuilder resuelve el problema de creación de un sistema operativo Linux desde cero de manera automática a la vez que permite una mayor capacidad de personalización y adaptación del sistema operativo resultante a las necesidades del usuario final. Mayor flexibilidad y mejor rendimiento son las características principales que diferencian a *LFSBuilder* de la herramienta oficial *jhalfs* tal y como se ha visto en apartados anteriores.

Durante el proceso de diseño, análisis e implementación de la herramienta `Linux from Scratch Builder` han quedado patentes los beneficios que aporta un campo como el de la automatización a la industria tecnológica actual. Reconocer los procesos susceptibles de ser automatizados representa, a veces, un desafío quizás mayor que el proceso de automatización en sí mismo, pero ello genera también unos beneficios y un aprendizaje muy valiosos para el futuro.

La automatización supone la eliminación de errores humanos por despiste o errores tipográficos. Si algo que está automatizado funciona correctamente, lo hará en ocasiones sucesivas siempre y cuando no se modifique el proceso entre una ejecución y otra. Además, la automatización implica un ahorro de tiempo y reducción de esfuerzos que puede representar, en la industria, incluso consecuencias económicas, ya que invertir el tiempo en automatizar procesos repetitivos como el que se ha descrito en este texto aumenta la capacidad de trabajo, puesto que el tiempo que antes se empleaba en generar un sistema operativo propio adecuado a las necesidades de un desarrollador o equipo de trabajo puede emplearse ahora en resolver otros errores del código o tareas más prioritarias.

La realización de este proyecto ha supuesto un reto personal importante, no solo por el desafío que supone la creación de un sistema operativo desde cero, sino también por lo que representa enfrentarse a la automatización de dicho proceso. El aprendizaje agregado del sistema operativo Linux, programación, compilación, protocolos y administración de redes, contenedores, virtualización, computación en la nube y otras tecnologías actuales como Docker y Kubernetes no hacen sino mejorar la sensación con la que finaliza esta etapa formativa.

Los objetivos cumplidos en este proyecto se muestran en la lista siguiente:

- Análisis del proyecto *Linux from Scratch* para evaluar los procesos a automatizar.
- Análisis de la herramienta oficial *jhalfs* para la automatización del proceso de construcción de un sistema operativo *Linux from Scratch*.
- Diseño y desarrollo de una herramienta alternativa que ofrece las siguientes características:
  - Mayor personalización del proceso de construcción y del resultado debido a la implementación de un sistema de plantillas y recetas.
  - Mejor rendimiento que la herramienta *jhalfs*.
  - Compatible con las versiones más recientes de los libros *Linux from Scratch* y *Beyond Linux from Scratch* así como de gran parte de las versiones anteriores. Este último punto ha sido probado a partir de la versión *7.0* publicado en 2011.

Respecto a las líneas de mejora, y a pesar de que la herramienta *LFSBuilder* pueda parecer bastante completa, es obligatorio reconocer que hay muchas mejoras que realizar, algunas de las cuales se enumeran a continuación.

- **Gestión de errores**: actualmente la herramienta *LFSBuilder* no realiza gestión de errores más allá de terminar la ejecución del programa en ese mismo momento e informar al usuario. Si bien es verdad que ofrece la posibilidad de volver a ejecutar el proceso de construcción continuando por el componente que falló usando la opción `--continue-at`, deshacer los pasos realizados para ese componente hasta el momento del fallo podría ser una buena opción, asegurando además que el componente se reconstruiría en un entorno limpio de sus intentos previos. En cualquier caso, esta opción no debería contemplar el borrado de los *scripts* generados hasta el momento del fallo, pues pueden servir para depurar errores más facilmente.
- **Errores en la descarga**: la herramienta *LFSBuilder* deja de ejecutarse si se produce algún error tratando de descargar un fichero de código fuente. Dado que estos problemas pueden ocurrir por motivos irremediables, por ejemplo porque el servidor que aloja el fichero solicitado haya sufrido una avería, el proyecto *Linux from Scratch* dispone de un servidor *FTP* alternativo para descargar los ficheros necesarios. Los propios mantenedores de proyecto solicitan que no se abuse de su servidor de respaldo, sería conveniente implementar la lógica en la herramienta *LFSBuilder* que utilizase dicho servidor sólo en caso de fallo.
- **Almacenaje de *logs***: el código fuente desarrollado hasta la fecha no almacena de ninguna forma los diferentes *logs* producidos a lo largo del proceso de construcción. Almacenar estos logs dentro del subdirectorio `tmp` de la herramienta ayudaría a depurar errores o analizar el proceso de construcción del componente.
- **Mediciones de tiempo**: con el objetivo de contabilizar el tiempo transcurrido para construir cada componente y detectar posibles cuellos de botella en el proceso de construcción del sistema operativo, emplear una clase que mida, almacene y guarde el tiempo empleado para los principales pasos del proceso de construcción puede ser un buen primer paso a definir y desarrollar. Con los datos obtenidos podrían generarse multitud de gráficas y análisis estadísticos que permitan analizar, a lo largo del tiempo, si un cambio en el código fuente supone una mejora o no en alguno de los aspectos analizados.
- **Entorno gráfico**: el libro *Beyond Linux from Scratch* permite instalar y configurar un entorno de escritorio para el sistema operativo *Linux from Scratch*, pero según el entorno de escritorio que se desee utilizar esta tarea puede ser compleja. Disponer de la lógica necesaria para hacer transparente al usuario la mayor parte de la instalación de estos componentes, seleccionando el entorno de escritorio que se quiere instalar como una nueva opción en el fichero de configuración, sería una gran funcionalidad para el programa *LFSBuilder*.
- **Gestión de dependencias**: otro problema del libro *Beyond Linux from Scratch* es la gestión de dependencias. Ya se comentó en el apartado *3.3 jhalfs vs. LFSBuilder* de la página 64 que la herramienta `jhalfs` dispone de cierta lógica con la que intentar resolver este problema. Una lógica de este tipo sería un buen añadido a *LFSBuilder* dado que en la actualidad es el usuario final quien debe añadir los componentes en el orden correcto a la lista `components_to_build` del constructor `blfs` para que sean construidos.

- **Refactorizar el módulo *tools***: actualmente el módulo `tools.py` es en realidad un fichero de funciones. Modificar la estructura de este fichero y convertirlo en un módulo o paquete de Python ayudaría a mejorar la organización del código fuente y facilitar la modificación del mismo.
- **Gestión de paquetes**: hoy día, un gestor de paquetes es una herramienta prácticamente fundamental en la mayoría de distribuciones Linux. El objetivo principal de *Linux from Scratch Builder* es generar un sistema operativo a medida y personalizado. Así pues, permitir el uso opcional de un determinado gestor de paquetes incrementaría las posibilidad de los usuarios para personalizar su sistema operativo o distribución Linux usando la herramienta *LFSBuilder*.

Estas y otras muchas mejoras tienen cabida en la herramienta expuesta en este texto puesto que no es un trabajo terminado, sino un proyecto en constante evolución. Tal y como dice la cita de Alan Turing que encabeza este capítulo, no podemos ver más alla del futuro inmediato, pero es seguro que siempre habrá cosas por hacer.

Apéndice A
Mapa de Karnaugh

Un mapa de Karnaugh es una utilidad matemática empleada en problemas lógicos combinacionales para simplificar funciones algebraicas Booleanas de manera simple.

Los mapas de Karnaugh se emplean en circuitos lógicos combinacionales de los que se conoce, o puede obtenerse fácilmente, su tabla de verdad. Así, mediante la representación bidimensional de la mencionada tabla de verdad se identifican los patrones que validan la tabla de verdad y se agrupan en potencias de 2. Puesto que la tabla de verdad de una función de n variables posee 2^n filas, el mapa de Karnaugh correspondiente debe poseer también 2^n cuadrados. Las variables de la expresión son ordenadas mediante codificación Gray, de manera que sólo el valor de una de las variables lógicas de entrada varía entre celdas adyacentes. La transferencia de los términos de la tabla de verdad al mapa de Karnaugh se realiza de forma directa, albergando un 0 ó un 1, dependiendo del valor que toma la función para cada celda. Al agrupar en potencias de dos, es decir $\{2^n\}_{n=0,1,2,...}$, se asegura que en todos los grupos seleccionados los valores de entrada tendrán la misma salida para el conjunto de entrada agrupado, por lo que la función lógica se va reduciendo con cada subgrupo. Para realizar correctamente una reducción mediante mapa de Karnaugh es obligatorio que todos las celdas con valor uno formen parte de una agrupación, aunque el grupo esté formado por una única celda individual.

La herramienta *LFSBuilder* resuelve la combinación lógica de varias variables booleanas para decidir si la combinación de constructores que se pretende construir es válida o no. Recordemos primero cuáles son estas variables y qué representan.

- **t**: el bit t indica si el constructor *toolchain* va a construirse o no.
- **s**: el bit s marca si el constructor *system* va a ser construido.
- **c**: el bit c indica si el constructor *configuration* va a construirse o no.
- **min**: el bit m indica si el índice del *toolchain* es el menor de la lista de constructores con la que se va a trabajar. Esto significa, por tanto, que el *toolchain* es el primer elemento de esa lista.
- **la**: parecido al anterior, este bit indica si en la comparativa posicional entre los constructores *system* y *configuration*, el primero tiene un índice menor que *configuration*.

La tabla de verdad que gobierna este circuito combinacional es la mostrada a dos columnas en la tabla A.1.

t	s	c	m	la	F	t	s	c	m	la	F
0	0	0	0	0	1	1	0	0	0	0	1
0	0	0	0	1	1	1	0	0	0	1	1
0	0	0	1	0	1	1	0	0	1	0	1
0	0	0	1	1	1	1	0	0	1	1	1
0	0	1	0	0	1	1	0	1	0	0	0
0	0	1	0	1	1	1	0	1	0	1	0
0	0	1	1	0	1	1	0	1	1	0	0
0	0	1	1	1	1	1	0	1	1	1	0
0	1	0	0	0	1	1	1	0	0	0	0
0	1	0	0	1	1	1	1	0	0	1	0
0	1	0	1	0	1	1	1	0	1	0	1
0	1	0	1	1	1	1	1	0	1	1	1
0	1	1	0	0	0	1	1	1	0	0	0
0	1	1	0	1	1	1	1	1	0	1	0
0	1	1	1	0	0	1	1	1	1	0	0
0	1	1	1	1	1	1	1	1	1	1	1

Tabla A.1 Tabla de verdad para el circuito combinacional de la herramienta *LFSBuilder*.

A partir de esta tabla de verdad se genera el mapa de Karnaugh mostrado a continuación.

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	1	1	0
10	1	1	1	1

$t = 0$

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	1

$t = 1$

Figura A.1 Mapa de Karnaugh para la tabla de verdad mostrada en la tabla A.1.

Cuya resolución paso a paso se detalla como sigue:

1. Agrupamos el primer término obteniendo la expresión $f_1 = \tilde{t} \cdot la$

$t = 0$

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	1	1	0
10	1	1	1	1

$t = 1$

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	1

Figura A.2 Obtención del primer término para el mapa de Karnaugh mostrada en la figura A.1.

2. Agrupamos otro término en el mapa de la izquierda obteniendo la expresión $f_2 = \tilde{t} \cdot \tilde{c}$

$t = 0$

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	1	1	0
10	1	1	1	1

$t = 1$

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	1

Figura A.3 Obtención del segundo término para el mapa de Karnaugh mostrada en la figura A.1.

3. Seleccionamos un tercer término en el mismo mapa del que obtenemos la expresión $f_3 = \tilde{t} \cdot \tilde{s} \cdot \tilde{m}$

$t = 0$ (filas: s, c; columnas: m, la)

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	1	1	0
10	1	1	1	1

$t = 1$ (filas: s, c; columnas: m, la)

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	1

Figura A.4 Obtención del tercer término para el mapa de Karnaugh mostrada en la figura A.1.

4. Agrupamos un nuevo término en el mismo mapa obteniendo la expresión $f_4 = \tilde{s} \cdot \tilde{c} \cdot m$

$t = 0$ (filas: s, c; columnas: m, la)

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	1	1	0
10	1	1	1	1

$t = 1$ (filas: s, c; columnas: m, la)

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	1

Figura A.5 Obtención del cuarto término para el mapa de Karnaugh mostrada en la figura A.1.

5. Seleccionamos un primer término en ambos mapas obteniendo la expresión $f_5 = \tilde{s} \cdot \tilde{c}$

t = 0 (filas: *s*, *c*; columnas: *m*, *la*)

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	1	1	0
10	1	1	1	1

t = 1 (filas: *s*, *c*; columnas: *m*, *la*)

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	1

Figura A.6 Obtención del quinto término para el mapa de Karnaugh mostrada en la figura A.1.

6. Agrupamos otro nuevo término entre ambos mapas obteniendo la expresión $f_6 = s \cdot m \cdot la$

t = 0 (filas: *s*, *c*; columnas: *m*, *la*)

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	1	1	0
10	1	1	1	1

t = 1 (filas: *s*, *c*; columnas: *m*, *la*)

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	1

Figura A.7 Obtención del sexto término para el mapa de Karnaugh mostrada en la figura A.1.

7. Por último agrupamos un término más entre ambos mapas obteniendo la expresión $f_7 = \tilde{c} \cdot m$

$t=0$

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	1	1	0
10	1	1	1	1

$t=1$

s, c \ m, la	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	1

Figura A.8 Obtención del séptimo término para el mapa de Karnaugh mostrada en la figura A.1.

Sumando los términos generados en los pasos anteriores se obtiene la función mínima que valida la tabla de verdad A.1 de la página 80. Esta función es,

$$F\left(t,s,c,m,la\right)=\left(\tilde{t}\cdot la\right)+\left(\tilde{t}\cdot\tilde{c}\right)+\left(\tilde{t}\cdot\tilde{s}\cdot\tilde{m}\right)+\left(\tilde{s}\cdot\tilde{c}\cdot m\right)+\left(\tilde{s}\cdot\tilde{c}\right)+\left(s\cdot m\cdot la\right)+\left(\tilde{c}\cdot m\right)$$

Índice de Figuras

1.1	Comparativa de arquitecturas de virtualización	6
2.1	Diagrama de flujo para el libro *Linux from Scratch*	12
2.2	Diagrama de flujo para el arranque de un sistema operativo	19
3.1	Diagrama de capas de la aplicación *LFSBuilder*	33
3.2	Diagrama de flujo para construir el sistema operativo *Linux from Scratch* usando la herramienta *Linux from Scratch Builder*	34
3.3	Estructura de directorios de la herramienta *LFSBuilder*	35
3.4	Clases implementadas en el fichero `actions.py`	36
3.5	Clases implementadas en el fichero `builders.py`	37
3.6	Clases implementadas en el fichero `cli.py`	37
3.7	Clases implementadas en el fichero `components.py`	38
3.8	Diagrama de la clase definida en el fichero `downloader.py`	42
3.9	Diagrama de la clase definida en el fichero `lfsbuilder.py`	43
3.10	Diagrama de la clase definida en el fichero `xmlparser.py`	44
3.11	Diagrama de casos de uso de la aplicación *LFSBuilder*	54
3.12	Uso de memoria durante la ejecución de la herramienta *LFSBuilder*	68
A.1	Mapa de Karnaugh para la tabla de verdad mostrada en la tabla A.1	80
A.2	Obtención del primer término para el mapa de Karnaugh mostrada en la figura A.1	81
A.3	Obtención del segundo término para el mapa de Karnaugh mostrada en la figura A.1	81
A.4	Obtención del tercer término para el mapa de Karnaugh mostrada en la figura A.1	82
A.5	Obtención del cuarto término para el mapa de Karnaugh mostrada en la figura A.1	82
A.6	Obtención del quinto término para el mapa de Karnaugh mostrada en la figura A.1	83
A.7	Obtención del sexto término para el mapa de Karnaugh mostrada en la figura A.1	83
A.8	Obtención del séptimo término para el mapa de Karnaugh mostrada en la figura A.1	84

Índice de Tablas

2.1 Subproyectos *Linux from Scratch* 10
2.2 Correspondencia entre los bloques del diagrama 2.1 y los capítulos del libro *Linux from Scratch* 12
2.3 Dependencias del sistema anfitrión para construir el sistema operativo *Linux from Scratch* 13
2.4 Descripción de los niveles de ejecución del gestor de inicio *SysVinit* 22
2.5 Resumen de comandos del programa `systemctl` 24
2.6 Resumen de modificaciones en el programa *jhalfs* para la construcción del sistema base 31

3.1 Opciones del fichero `config.py` 42
3.2 Tipos de mensaje mostrados por la herramienta *LFSBuilder* 43
3.3 Parámetros para las recetas de la herramienta *LFSBuilder* 47
3.4 Funciones disponibles para sobreescritura en el fichero `functions.py` 50
3.5 Dependencias del libro *Linux from Scratch* 63
3.6 Comparativa de dependencias entre las herramientas *jhalfs* y *LFSBuilder* 65
3.7 Comparativa de ejecución entre las herramientas *jhalfs* y *LFSBuilder* 66

4.1 Códigos de error deshabilitados para el *linter* `pylint` 71
4.2 Entornos empleados para probar el funcionamiento de la herramienta *LFSBuilder* 72

A.1 Tabla de verdad para el circuito combinacional de la herramienta *LFSBuilder* 80

Índice de Códigos

2.1 Añadir el usuario y grupo `lfs` 13
2.2 Formatear las particiones `/dev/sdb1` y `/dev/sdb2` 15
2.3 Definir la variable de entorno *LFS* y montar las particiones 15
2.4 Otras variables de entorno y opciones del intérprete de línea de comandos 16
2.5 Añadir el directorio `sources` y descargar los archivos necesarios 17
2.6 Añadir el directorio `tools` y crear el enlace simbólico necesario 17
2.7 Hacer una copia de seguridad del *GRUB* con el programa `grub-mkrescue` 20
2.8 Instalar *GRUB* en el disco `/dev/sdb` usando el programa `grub-install` 21
2.9 Ejemplo de archivo de configuración `/boot/grub/grub.cfg` 21
2.10 Salida del comando `ps -p 1` para *SysVinit* 22
2.11 Ejemplo de archivo `/etc/fstab` para *SysVinit* 23
2.12 Salida del comando `ps -p 1` para *systemd* 24
2.13 Contenido del fichero `sshd.service` para la gestión del servicio *SSH* en *Systemd* 25
2.14 Ejemplo de archivo `/etc/fstab` para *Systemd* 25
2.15 Ejemplo de archivo `/etc/fstab` para *Systemd* 30

3.1 Ejemplo de entidades XML 44
3.2 Contenido parcial del fichero `gmp.xml` 45
3.3 Comandos para la instalación del servicio `opensshd` con la herramienta *LFSBuilder* 48
3.4 Contenido de la receta `openssh.yaml` para el programa *OpenSSH* 48
3.5 Uso del fichero `functions.py` para sobreescribir la función `run_previous_steps` en el componente *kernel* 49
3.6 Contenido de la receta `provider.yaml` para el constructor `provider` 50
3.7 Receta para el componente Redis 51
3.8 Contenido del fichero `script.tpl` 51
3.9 Comandos para ejecutar los tests unitarios de la herramienta *LFSBuilder* 53
3.10 Estructura de la interfaz de línea de comandos para la herramienta *LFSBuilder* 54
3.11 Función `check_lfs_builders_order` como solución de un Mapa de Karnaugh 56
3.12 Receta en formato YAML para el componente *kernel* del libro *Linux from Scratch* 57
3.13 Comandos para la descarga de los ficheros XML y el código fuente de los componentes del libro *Beyond Linux from Scratch* 57
3.14 Comando para la construcción de los componentes del libro *Beyond Linux from Scratch* 58
3.15 Comandos extraídos del libro *Linux from Scratch* para el componente *kernel* 61
3.16 Función `run_previous_steps` sobreescrita por el componente *kernel* 61
3.17 Contenido del *script* `compile.sh` para la construcción del componente *kernel* 62
3.18 Contenido del *script* `post.sh` para la construcción del componente *kernel* 63

3.19 Comando para la instalación del paquete `PyYAML` usando el gestor `pip` 64

4.1 Mensaje de error para el *kernel panic* referido a un sistema de archivos desconocido 73

Bibliografía

[1] University Information Technology Services at Indiana University, *Tips for staying safe online*, *https://kb.iu.edu/d/akln*.

[2] GNU, *"Free GNU/Linux distributions"*, *https://www.gnu.org/distros/free-distros.html*.

[3] ———, *"Free Non-GNU distributions"*, *https://www.gnu.org/distros/free-non-gnu-distros.html*.

[4] Parminder Singh Kocher, *Microservices and Containers. ISBN 0134598385*, Addison-Wesley Professional, 2018.

[5] LFS, *"libro Linux from Scratch"*, *http://www.linuxfromscratch.org/lfs/*.

[6] The Top500 list, *"Computers list ranked by their performance on the LINPACK Benchmark"*, *https://www.top500.org/*.

[7] Mark Lutz, *Learning Python, 5th Edition. ISBN 978-1449355739*, O'Reilly Media, 2013.

[8] Gonzalo Sampascual Maicas, *Psicología de la educación, tomo I. ISBN 8436243765*, Universidad Nacional de Educación a Distancia, 2002.

[9] Jorge Vasconcelos Santillán, *Tecnologías de la información (2a. ed.). Versión eBook. ISBN 9786077442462*, Grupo Editorial Patria, 2015.

[10] Francisco Javier Payán Somet, *Principios de Comunicaciones Digitales I. Fundamentos. ISBN 9788447215430*, Sevilla : Universidad de Sevilla, Secretariado de Publicaciones, 2014.

[11] Richard Stallman, *Linux and the GNU System*, *https://www.gnu.org/gnu/linux-and-gnu.html*.

[12] AT&T Bell Laboratories Tom Duff, *Experience with viruses on UNIX Systems*, *https://www.usenix.org/legacy/publications/compsystems/1989/spr_duff.pdf*.

[13] James Kurose y Keith Ross, *Computer Networking: A Top-Down Approach, 5th Edition. ISBN 9780131365483*, Addison-Wesley, 2010.

[14] Christine Bresnahan y Richard Blum, *LPIC-1. Linux Profesional Institute Certification, cuarta edición. ISBN 9788441537477*, Ediciones Anaya Multimedia, 2016.

Glosario

ARM Advanced RISC Machines. 19

BIOS Basic Input/Output System. 19

BLFS Beyond Linux from Scratch. 9

FHS Filesystem Hierarchy Standard. 10

GPT GUID Partition Table. 20

GRUB GRand Unified Bootloader. 20

Kernel panic Error irreversible del sistema. 73

LFS Linux from Scratch. 9

LFSBuilder Linux from Scratch Builder. 33

LSB Linux Standard Base. 11

MBR Master Boot Record. 20

POST Power On Self Test. 19

UEFI Unified Extensible Firmware Interface. 19

USB Universal Serial Bus. 19

XML eXtensive Markup Language. 12