Evolución de la Arquitectura del Sistema Unix: Un estudio de caso exploratorio

Diomidis Spinellis, Miembro sénior, IEEE y Paris Avgeriou, Miembro sénior, IEEE

Resumen: Unix ha evolucionado durante casi cinco décadas, moldeando sistemas operativos modernos, tecnologías de software clave y prácticas de desarrollo. Estudiar la evolución de este notable sistema desde una perspectiva arquitectónica puede proporcionar información sobre cómo administrar el crecimiento de sistemas de software grandes, complejos y de larga duración. A lo largo de los principales lanzamientos de Unix que conducen al linaje FreeBSD, examinamos las decisiones de diseño arquitectónico central, la cantidad de funciones y la complejidad del código, en función del análisis del código fuente, la documentación de referencia y las publicaciones relacionadas. Informamos que el crecimiento en tamaño ha sido uniforme, con algunos valores atípicos notables, mientras que la complejidad ciclomática se ha salvaguardado religiosamente. Una gran cantidad de decisiones de diseño que definen a Unix se implementaron desde el principio, y la mayoría de ellas aún juegan un papel importante. Unix continúa evolucionando desde una perspectiva arquitectónica, pero la tasa de innovación arquitectónica se ha ralentizado durante la vida útil del sistema. La deuda técnica arquitectónica se ha acumulado en forma de duplicación de funciones e instalaciones no utilizadas, pero en términos de complejidad ciclomática se está pagando sistemáticamente a través de lo que parece ser un proceso de autocorrección. Algunas fuerzas arquitectónicas no reconocidas que dieron forma a Unix son el énfasis en las convenciones sobre la aplicación rígida, el impulso por la portabilidad, un ecosistema sofisticado de otros sistemas operativos y organizaciones de desarrollo, y el surgimiento de una arquitectura federada, a menudo a través de la adopción de subsistemas de terceros. Estos hallazgos nos han llevado a formar una teoría inicial sobre la evolución de la arquitectura del software de sistemas operativos grandes y complejos.

Términos del índice: Unix, arquitectura de software, evolución del software, decisiones de diseño de arquitectura, sistemas operativos

C

1 INTRODUCCIÓN

UNIX[1] tiene una larga y célebre historia. Su evolución, abarca cinco décadas y es el resultado del trabajo de miles de desarrolladores, incluidos varios pioneros distinguidos. Como sistema operativo, ha dejado una huella innegable en la historia de la informática, al mismo tiempo que ha influido enormemente en el estado actual de la ingeniería de software, redes y hardware.

Estudiar la evolución del software del sistema operativo no solo es significativo desde una perspectiva histórica; puede proporcionar información valiosa sobre las mejores prácticas y antipatrones de capacidad de evolución para sistemas grandes, complejos y de larga duración. Unix es un caso único entre todos los sistemas operativos, tanto por su longevidad como por su impacto en los sistemas operativos posteriores. La evolución de un sistema de este tamaño, complejidad y antigüedad puede arrojar luz sobre cómo sistemas similares pueden crecer de forma sostenible sin los peligros del envejecimiento del software, como una deuda técnica vertiginosa o un deterioro arquitectónico descontrolado.

En este artículo estudiamos la evolución de Unix a lo largo del linaje FreeBSD desde una perspectiva de arquitectura de software. Si bien ha habido estudios sobre cómo evolucionó Unix (consulte la Sección 2), estos se han centrado principalmente en el nivel del código fuente y se limitaron al núcleo. Por el contrario, dirigimos nuestra atención a la arquitectura del sistema y estudiamos a) las decisiones de diseño de la arquitectura central en las versiones principales, y b) la evolución en el número de características del sistema (obtenidas de la documentación de referencia de Unix) y en el código. complejidad. El primero implica un análisis cualitativo, mientras que el segundo, cuantitativo. Estos análisis conducen posteriormente a formar una teoría inicial sobre la evolución de la arquitectura de sistemas operativos grandes y complejos, en cuanto a su forma, ritmo, fuerzas motrices, así como la acumulación de deuda técnica arquitectónica.

El resto del documento está estructurado de la siguiente manera: en la Sección 2 presentamos el trabajo relacionado, mientras que en la Sección 3 elaboramos el diseño del estudio de caso. En las Secciones 4 y 5 presentamos los resultados cualitativos (principales decisiones de diseño arquitectónico) y los resultados cuantitativos (evolución del tamaño y complejidad) respectivamente. A continuación, en la Sección 6 discutimos los principales hallazgos y en la Sección 7 las amenazas a la validez de este estudio. Finalmente, en la Sección 8 concluimos el artículo con un resumen y una discusión de nuestros hallazgos.

2 TRABAJO RELACIONADO

El trabajo informado aquí cubre principalmente dos áreas: a) la evolución del software en general, que ha sido intensamente estudiada, yb) la evolución de Unix en particular, donde el trabajo relacionado es más escaso sobre el terreno.

1. UNIX es una marca registrada de The Open Group. En aras de la simplicidad, en este documento usamos la palabra "Unix" para referirnos tanto a los sistemas UNIX desarrollados en Bell Labs como a los sistemas similares a Unix, como FreeBSD, que descienden de ellos.

D. Spinellis trabaja en la Universidad de Economía y Negocios de Atenas, Athina 104 34, Grecia. Correo electrónico: dds@aueb.gr.
P. Avgeriou trabaja en la Universidad de Groningen, Groningen 9712, Países Bajos. Correo electrónico: paris@cs.rug.nl.

Manuscrito recibido el 19 de mayo de 2018; revisado el 18 de diciembre de 2018; aceptado el 28 de diciembre 2018. Fecha de publicación 2 de mayo de 2019; fecha de la versión actual 14 de junio de 2021.
(Autor para correspondencia: Diomidis Spinellis.)
Recomendado para su aceptación por R. Mirandola.
Identificador de objeto digital no. 10.1109/TSE.2019.2892149

2.1 Evolución del software

Se han realizado varios estudios sobre la evolución longitudinal de los grandes sistemas. El trabajo seminal de Lehman [1] y sus subsecuentes mejoras intentaron establecer leyes de evolución del software, no muy diferentes a las de la evolución biológica. Esas leyes han sido objeto de mucha discusión y trabajo de investigación [2]: su validez ha sido debatida durante mucho tiempo, su naturaleza y alcance han sido refinados iterativamente por muchos investigadores, mientras que varios estudios han examinado si las leyes se cumplen para casos particulares. El fenómeno de la evolución del software también ha sido estudiado bajo diferentes términos, como Software Aging [3], Software Decay [4] y, más recientemente, Technical Debt [5].

Una de las formas más populares de estudiar la evolución del software se centra en el crecimiento del código fuente. Hatton et al. llevó a cabo el estudio más grande hasta la fecha sobre la tasa de crecimiento del software; específicamente, estudiaron la tasa de crecimiento de más de 404 millones de líneas tanto de código abierto como de software propietario y concluyeron que el código se duplica aproximadamente cada 42 meses [6]. De manera similar, un gran estudio sobre 6000 sistemas de código abierto realizado por Koch [7] reveló que, si bien el crecimiento medio es lineal, existe un porcentaje significativo de sistemas con un crecimiento superlineal.

Varios artículos examinan la evolución del software de código abierto desde diversos ángulos [8]. Muchos adoptan un enfoque cuantitativo, utilizando estadísticas para determinar las relaciones entre varios atributos, como la modularidad y la complejidad [9], la tasa de crecimiento y cambio [10], la complejidad y el cambio acumulativo [11], o incluso las contribuciones y colaboraciones de la comunidad de usuarios. a través del análisis de redes sociales [12].

Algunos artículos examinan la evolución de los sistemas escritos en C en términos de modularidad y complejidad y, por lo tanto, son directamente relevantes para este trabajo. Un estudio inicial del crecimiento del kernel de Linux realizado por Godfrey y Tu [13] argumentó que la tasa de crecimiento superlineal del kernel podría atribuirse al crecimiento lineal de varios subsistemas; esto está relacionado con nuestro hallazgo (Sección 6.3) de que la acumulación de grandes subsistemas juega un papel importante en la evolución moderna de Unix. Un estudio posterior sobre el mismo tema [14] también analizó el problema de la complejidad del código y descubrió que "la complejidad promedio por función y la distribución de complejidades de las diferentes funciones están mejorando con el tiempo". Tendencias más o menos similares, junto con lo que parece ser un proceso de autocorrección, se muestran en un estudio de las prácticas de programación de Unix [15].

También ha habido un número significativo de estudios sobre la evolución de los sistemas operativos, particularmente Linux. Mac Cor mack et al. [16] estudió Linux en términos de su estructura y lo comparó con la primera versión evolucionada de Mozilla; los resultados enfatizan la modularidad de Linux y cómo Mozilla evolucionó de una estructura menos a una más modular (en comparación con Linux) en cuestión de años. Además del estudio de Linux antes mencionado de Godfrey y Tu [13], que midió principalmente líneas de código del sistema operativo y sus principales subsistemas, un estudio posterior del kernel de Linux, realizado por Israeli y Feitelson [17], tuvo como objetivo caracterizar ing el sistema operativo de acuerdo con las leyes de evolución de Lehman. Utilizaron una serie de métricas cuantitativas además de las líneas de código, como el número de llamadas al sistema y la complejidad ciclomática [18]. Pudieron confirmar varias de las leyes de Lehman, mientras que uno de los hallazgos interesantes es que la complejidad disminuye con el tiempo. En un estudio de seguimiento, Feitelson estudió el ciclo de vida de la evolución del kernel de Linux [19], resumiéndolo como un modelo lineal por partes con pendientes crecientes.

Además de estudiar la evolución del software a nivel de código fuente, varios estudios se han centrado en el nivel de arquitectura. Behnamghader et al. [20] propuso un método para la recuperación de la arquitectura y posteriormente utilizó este método para estudiar 23 sistemas de código abierto examinando los cambios arquitectónicos durante largos períodos de evolución del sistema. Otros enfoques también han analizado la evolución de la arquitectura, pero utilizando artefactos de código fuente, como clases y paquetes, como entidades de primera clase. Por ejemplo, D'Ambros et al. [21] propusieron métricas arquitectónicas derivadas del análisis del código fuente y, posteriormente, visualizaron esas métricas para ilustrar diferentes aspectos de la evolución tanto del código como de la arquitectura.

De manera similar, Wettel y Lanza [22] se centraron en la visualización de las características "de grano grueso" de la evolución del software (paquetes y clases), así como las de "grano fino" (métodos).
Un último ejemplo es el trabajo de Bouwers et al. [23], quien propuso una métrica de arquitectura para la partición de la arquitectura en componentes basada en la evolución de numerosos sistemas de código abierto y propietarios.

Comparado con el trabajo relacionado discutido, nuestro trabajo tiene las siguientes diferencias: a) nos enfocamos en Unix; b) analizamos la evolución de la arquitectura no a nivel de componentes sino a nivel de decisiones de arquitectura, los siete tipos de características clave de Unix (comandos de usuario, llamadas al sistema, bibliotecas, etc.), así como la forma y el ritmo de la evolución de la arquitectura, arquitectura deuda técnica, y notables características arquitectónicas; c) utilizamos fuentes de datos que abarcan 48 años y 30 versiones del sistema.

2.2 Trabajo sobre el diseño y la evolución de Unix La importancia de Unix y su pedigrí, arraigado primero en la investigación industrial (AT&T Bell Labs) y luego en la académica (Universidad de California en Berkeley), lo ha dotado de numerosas publicaciones que detallan la estructura del sistema. y evolución. Estos cubren instantáneas, subsistemas o períodos específicos.

El personal de Bell Labs publicó decenas de artículos sobre Unix y sus aplicaciones como informes técnicos [24], [25], [26], [27], [28], [29], [30], [31], [32] , [33]. La mayoría de estos también se distribuyeron con cada versión de Unix como Volumen 2—Documentos complementarios, del Manual del programador de Unix adjunto. Dos números de The Bell System Technical Journal, que aparecieron en 1978 y 1984, estaban enteramente dedicados a Unix; más tarde también se publicaron en forma de libro [34], [35]. El personal de Bell Labs también publicó en medios que cubren temas más diversos [36], [37], [38], [39], [40], [41]. Esta tradición de publicación abierta fue continuada por el personal y los ex alumnos del Grupo de Investigación en Ciencias de la Computación (CSRG) de Berkeley, así como por otros investigadores y desarrolladores de sistemas [42], [43], [44], [45], [46], [47], [48], [49], [50]. Estos documentos y muchos otros brindan información valiosa sobre la funcionalidad y la evolución de instalaciones específicas, así como de todo el sistema.

De particular importancia para este estudio son: el documento CACM que presenta las características, ideas y diseño de Unix [36]; la retrospectiva de Ritchie, que detalla los puntos fuertes y débiles del sistema [51]; la visión general de Thompson sobre la implementación de Unix [52]; el artículo de Rosler sobre la evolución de C [53]; el estudio de la portabilidad como fuerza moldeadora del diseño [54]; y un informe posterior de Ritchie sobre la evolución de Unix, centrándose en el sistema de archivos, el control de procesos, la redirección de E/S,

y lenguajes de alto nivel [55]. Artículos más recientes han cubierto la restauración y conservación de artefactos históricos, como las primeras ediciones de Unix [56], [57] y los repositorios de ellos [58], o su estudio posterior [15], [59].

Otra categoría de material relacionado con este estudio son los libros que detallan el funcionamiento interno de Unix y, por lo tanto, también partes de su arquitectura. Lo que empezó todo es un pequeño conjunto de dos volúmenes preparado en 1977 por John Lions como material didáctico para su curso de sistemas operativos en la Universidad de Nueva Gales del Sur. El primer volumen contiene una lista línea por línea del núcleo Unix de la sexta edición, mientras que el segundo volumen es un comentario del código fuente que explica la funcionalidad de cada elemento enumerado. La confusión con respecto a los derechos de propiedad intelectual asociados hizo que circulara durante dos décadas en fotocopias samizdat o escaneos digitales, antes de que se levantaran los obstáculos legales para permitir su publicación formal [60].

Una década después, Maurice Bach publicó un libro que cubría en términos abstractos, sin referencia a elementos específicos del código fuente, el diseño del kernel de Unix, con énfasis en System V Release 2 [61]. El libro, basado en material que el autor preparó para un curso que impartió en AT&T Bell Laboratories, cubre las estructuras de datos y los algoritmos más importantes. Mientras tanto, en la costa oeste, los investigadores que habían trabajado en las versiones de Berkeley de Unix, publicaron otro libro que detalla el diseño de BSD Unix [62]. Este trabajo fue ampliado y actualizado a intervalos regulares para cubrir nuevas ediciones de BSD Unix [63] y luego su descendiente FreeBSD [64], [65].

En esta área también mencionamos el análisis de arquitectura de alto nivel de Organick del sistema operativo MULTICS [66], un sistema mucho más grande y considerablemente más ambicioso que varias versiones anteriores de Unix. Esto es relevante porque AT&T Bell Labs estaba desarrollando el sistema junto con MIT y General Electric. Cuando AT&T se retiró del desarrollo de MULTICS, el equipo de Bell Labs se quedó sin un sistema en el que experimentar con el diseño del sistema operativo y, además, con valiosas lecciones aprendidas del proyecto MULTICS.

Este documento no es directamente comparable con el trabajo resumido aquí, pero se basa en él (consulte la Sección 3.3) y en datos empíricos para estudiar la evolución de Unix durante un período de medio siglo.

3 DISEÑO DE ESTUDIO DE CASO

El estudio de caso como método empírico se utiliza para investigar un fenómeno en su contexto de vida real [67]. La razón principal para seleccionar realizar un estudio de caso en lugar de otros tipos de estudios empíricos es que queremos una comprensión profunda de cómo y por qué se produjeron los fenómenos de evolución de la arquitectura dentro del ecosistema Unix. Este estudio de caso ha sido diseñado y se presenta de acuerdo con las pautas de Runeson et al. [67].

3.1 Objetivos y preguntas de investigación

El objetivo de este estudio, establecido aquí utilizando el enfoque Goal Question-Metric (GQM) [68], es “analizar el sistema operativo Unix con el fin de evaluar y caracterizar la evolución de su arquitectura con respecto a sus principales decisiones de diseño de arquitectura, tamaño y complejidad desde el punto de vista de los desarrolladores de software en el contexto de el ecosistema Unix”. El objetivo antes mencionado se puede lograr respondiendo a las siguientes preguntas de investigación.

RQ1 ¿Cuáles son las principales decisiones de diseño arquitectónico a lo largo de las principales versiones del sistema?

RQ2 ¿Cómo evolucionó la complejidad y la cantidad de funciones a lo largo de las principales versiones del sistema?

La primera pregunta pretende investigar la evolución de la arquitectura desde una perspectiva cualitativa. Una arquitectura es el conjunto de decisiones de diseño principales [69], [70]. Por lo tanto, estudiamos la evolución de la arquitectura mediante la identificación de las principales decisiones de diseño que se introdujeron a lo largo de varios de los lanzamientos más significativos (consulte la Sección 4). Tales decisiones de diseño son principalmente: (a) componentes de la arquitectura, incluidas sus interfaces, como el núcleo, los shells y las bibliotecas; (b) conectores de arquitectura como tuberías y archivos de encabezado C; (c) patrones de arquitectura [71] que se aplicaron en el sistema, como estratificación y reflexión; y (d) los principios que guían la arquitectura del sistema, como la modularidad y la separación de intereses. Los componentes y conectores de la arquitectura, los patrones y los principios constituyen algunas de las decisiones clave de la arquitectura de los sistemas de software [69], [72]. También reportamos otros tipos de decisiones que no pueden clasificarse en estas categorías, por ejemplo, convenciones de nomenclatura.
Cada decisión de diseño va acompañada de una justificación, que es la sección más importante en la documentación de la decisión [70].

La segunda pregunta también analiza la evolución de la arquitectura Unix, pero desde un punto de vista cuantitativo. Específicamente, observamos cómo evolucionan las métricas de tamaño y complejidad con el tiempo; estas métricas se refieren a características del sistema (p. ej., número de comandos de usuario o llamadas al sistema), según lo determinado por la documentación de referencia de Unix (para obtener más detalles, consulte la Sección 5.1). Esto nos brinda una perspectiva complementaria a los resultados cualitativos, ya que podemos discernir tendencias generales a lo largo de décadas en lugar de cambios de arquitectura notables en versiones individuales. Eventualmente, combinamos los resultados cuantitativos y cualitativos durante nuestra discusión (ver Sección 6) para derivar hallazgos y conclusiones.

Observamos que en tales análisis cuantitativos, es común medir también la cohesión y el acoplamiento. Sin embargo, en el caso de Unix, esto requeriría mucho más trabajo manual para cada revisión de Unix. Es decir, implicaría: a) el desarrollo de herramientas personalizadas para analizar el ensamblado de PDP-7 y PDP-11, así como los primeros dialectos de C; b) la configuración de herramientas de análisis de las políticas de diseño y enlace de archivos de cada revisión.
Por lo tanto, esto se considera fuera del alcance de este trabajo, pero constituye un trabajo futuro atractivo.

Las respuestas a ambas preguntas de investigación son interesantes más allá del caso de Unix. Por lo tanto, se utilizarán como datos sin procesar para formar una teoría inicial sobre la arquitectura de sistemas operativos grandes y complejos (consulte la Sección 6).

3.2 Selección de casos y unidades de análisis

El caso de estudio de este artículo se caracteriza por ser de caso único e integrado [67]: el sistema operativo Unix es el caso, mientras que las diferentes versiones son las unidades de análisis. Nuestro estudio comienza con la versión sin nombre de 1970 PDP-7 que se convirtió en Unix, seguida de las llamadas ediciones de "Investigación" que surgieron de Bell Labs, luego continúa con Berkeley Software Distributions (BSD) y termina con versiones de FreeBSD. distribución del sistema operativo que continúa su desarrollo hasta el día de hoy (ver Fig. 7 en la Sección 6). No pudimos estudiar Unix

TABLA 1

Unidades de análisis: lanzamientos clave, fechas y métricas de tamaño

Liberar	Fecha	Líneas de código		
		Programas de la biblioteca del kernel		

Investigación PDP7 1970 Investigación V1 3 de noviembre de 1971 Investigación V2 12 de junio de 1972
Investigación V3 15 de febrero de 1973
Investigación V4 30 de noviembre de 1973
Investigación V5 de junio de 1974 Investigación V6 de mayo de 1975 Investigación V7 de enero de 1979 Bell
32V 28 de agosto de 1979 BSD 3 22 de marzo de 1980 BSD 4
16 de noviembre de 1980 BSD 4.1c/2 2 abril 1983 BSD 4.2
1 enero 1985 BSD 4.3 4 marzo 1987 BSD 4.3/ Tahoe 6 enero 1990
BSD 4.3/Reno 2 enero 1991 BSD 4.3/Net 2 20 agosto 1991

2.489 0 4.768 ? ? 9,095? 1,075? ? 7,141 ?
8,778 5,634 53,428 16,968? ?
12,347 7,092
137,723 19,710
14,251 290,142 26,572 14,224 297,688
25,096 4,637 545,492 25,613 1,082,174
91,309 31,296 40,265 3,401 2,076,218 8,383
2,552,789 357,956 72,265 3,869,558 218
0.0 777,114 386 BSD 871,209,385 BSD
p/k 210,828 176,213 2,894,037 FreeBSD
FreeBSD 2.0 38 BSD 2,064,252,930 2,932,865
6,362,709 BSD 4.4 Lite2 5,245 FreeBSD
3.0.0 21 957,625 FreeBSD 4.0.0 14 Mar
2000 1,371,122 450,225 6,442,184
FreeBSD 5.0.0 16 Jan 2003 2,180,639
613,034 7,919,274 FreeBSD 6.0.0 3 Nov
2005 2,796,311 567,130 9,183,131 FreeBSD 7.0.0 24 Feb 2008 3,561,595 632,643 10,238,166
8.0.0 20 Nov 2009 4,099,256 746,689 10,747,631 FreeBSD 9.0.0 Jan 2012 5,371,628 761,459
6,599,640 699 317 17 780 699 FreeBSD 11.0.0 22 de septiembre de 2016 8 518 968 733 620
4 de marzo de 1992 92.365 176.680 FreeBSD
julio de 1992 15,135,803 FreeBSD 10.0.0 16 Jan 2014
20 de junio de 1993 21,529,326
28 de octubre de 1993
22 de noviembre de 1994
julio de 1995
julio de 1995
enero de 1999

2

versiones que se derivan de las ediciones Research a través de AT&T System V, como Solaris, AIX y HPUX, porque la mayor parte del código correspondiente sigue siendo propietario e inaccesible.

Decidimos no estudiar la evolución de las ediciones de Investigación en Plan 9 [41], debido a la adopción limitada del sistema y la falta de distribuciones de lanzamiento empaquetadas. Otros sistemas derivados de la base de código fuente de BSD son NetBSD, que se centra en la portabilidad de arquitectura de amplia difusión, especialmente entre dispositivos integrados, y OpenBSD, que se centra en la seguridad. Aunque estos proyectos difieren en términos de visión y tecnologías, todos intercambian frecuentemente entre ellos código e ideas. Este artículo examina la evolución de la arquitectura en la popular línea FreeBSD, para capitalizar el conocimiento interno de FreeBSD del primer autor y la excelente documentación de diseño publicada del sistema [64], [65].

En la Tabla 1.L1,2 aparece una descripción general de las emisiones principales que comprenden las unidades de análisis de este estudio.

Ediciones de investigación, la fecha de publicación se deriva de la fecha del manual correspondiente; en los casos restantes desde la marca de tiempo del archivo más reciente. Se marcan los casos en los que el código asociado a versiones específicas no se ha conservado.

2. Las notas al pie con el prefijo I (IN) documentan la derivación de números y tablas a través de listados correspondientemente numerados que aparecen en el material complementario en línea, que se puede encontrar junto con este documento en https://doi.org/10.1109/TSE.2019.2892149.

con un signo de interrogación. Los detalles sobre ese lanzamiento se obtuvieron estudiando su manual, que, afortunadamente, está disponible para todas las versiones de Unix. Más datos cuantitativos para cada una de estas versiones aparecen en la Fig. 3 en la Sección 5.1.

3.3 Recopilación de datos Para

responder a las preguntas de investigación, recopilamos datos tanto cualitativos como cuantitativos. Más específicamente, tanto para RQ1 como para RQ2 utilizamos dos técnicas de recopilación de datos [73]: análisis de documentación en varios documentos (datos cualitativos) y análisis estático del código fuente (datos cuantitativos). Para este último, examinamos el código fuente de cada una de las versiones de Unix, obtenido del repositorio histórico de Unix [58].3 Para el primero, utilizamos los siguientes documentos.

La documentación (páginas del Manual de referencia de Unix) asociada con cada versión [74]. En los casos en que no estaba disponible, se reconstruyó a partir de la

marcado del código fuente.4

Libros y trabajos de investigación descritos en la Sección 2.2.

Recuerdos de los pioneros de Unix [75], [76], [77], [78], [79], [80], [81].

El uso de múltiples fuentes de datos nos permitió realizar una triangulación de fuentes de datos, es decir, pudimos confirmar los hallazgos de diferentes tipos de fuentes de datos. Se dan más detalles en la Sección 7.

Gran parte de nuestro estudio se basa en un conjunto de datos de la documentación de referencia de Unix y su visualización en forma de líneas de tiempo [74]. Esta documentación está disponible desde la Primera Edición de Investigación en adelante en lo que se conoce como "Volumen I" del Manual del Programador de Unix [82]. Tenga en cuenta que el Volumen II [83] contiene documentos complementarios, que brindan un tratamiento en profundidad de herramientas y temas específicos, como el shell [84], el lenguaje de programación C [85], el verificador de programas lint [86], la tabla tbl formateador [87], y así sucesivamente. Afortunadamente, la documentación de Unix se mantiene en formato electrónico (como archivos troff [33]) junto con el código fuente del sistema. Para las versiones en las que se ha perdido el código fuente (indicado con un signo de interrogación en la Tabla 1), aún están disponibles copias escaneadas del manual.

Para responder a RQ1, se creó un conjunto de datos de todas las decisiones de diseño arquitectónico del sistema para cada versión disponible, basado en la documentación. El formato de los datos y su proceso de recopilación se describen en la referencia [74]. Los datos correspondientes y los scripts de generación están disponibles en línea.5

Para responder a RQ2, recopilamos datos principalmente a través del análisis del código fuente (para medir la complejidad) y el análisis de documentos en los manuales de referencia de Unix (para medir el tamaño del conjunto de funciones).

3.4 Análisis de datos Los

datos cuantitativos se analizan mediante estadísticas descriptivas simples y se ilustran mediante histogramas y diagramas de dispersión.

Para calcular la complejidad ciclomática [18] a nivel de componente, observamos el valor medio de todas las funciones que comprenden el componente correspondiente. Esto sigue a los resultados publicados recientemente que indican que la media y la mediana en lugar de la suma son mejores predictores de defectos [88].

3. DOI: 10.5281/zenodo.2525587 4. DOI: 10.5281/zenodo.2525571 DOI: 10.5281/zenodo.2525574 5. DOI: 10.5281/zenodo.2525613 DOI: 10.5281/zenodo.2525612

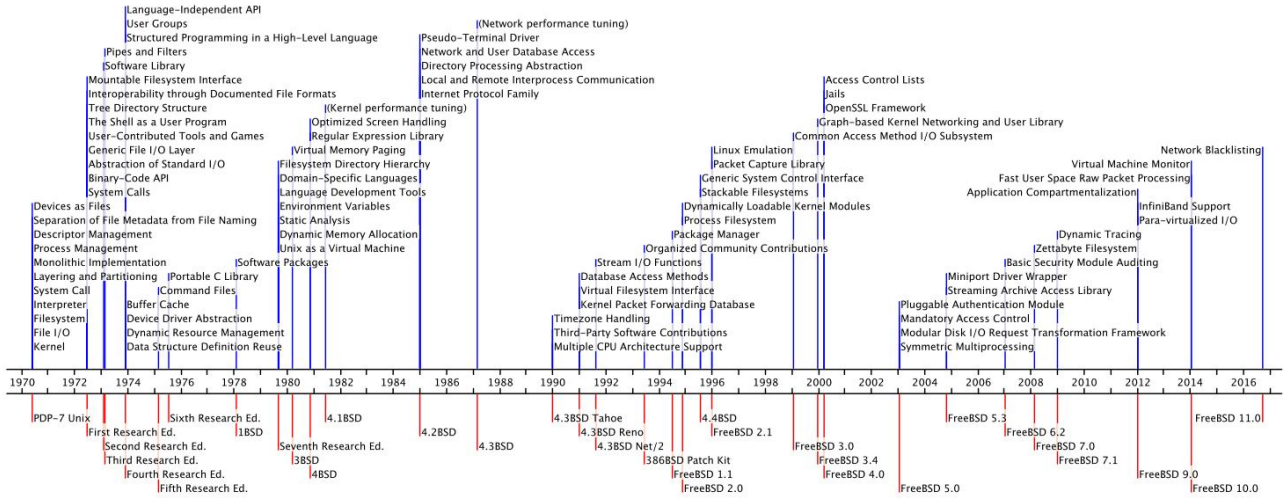

Fig. 1. Cronología de los principales lanzamientos de Unix y decisiones de diseño arquitectónico.

Con el fin de analizar los datos cualitativos, hemos realizado la codificación mediante Comparación constante [89]. Específicamente, realizamos la Comparación constante de forma iterativa, refinando los códigos con sus relaciones en cada iteración. Los códigos corresponden a las decisiones de diseño de arquitectura que se consideró que valía la pena informar por versión principal del sistema; en ese sentido, los códigos no fueron preformados sino posformados (es decir, fueron creados durante el proceso de codificación). Como es común en la Comparación constante (ver referencia [89]), nos enfocamos en unificar las explicaciones para las diversas decisiones estudiadas, en particular por qué se tomaron esas decisiones y cómo.

En las siguientes secciones intentamos responder las preguntas de investigación: la Sección 4 se enfoca en RQ1 al mostrar la evolución de las decisiones de diseño de arquitectura en las principales versiones de Unix; La Sección 5 aborda RQ2 presentando la evolución del tamaño y la complejidad de Unix.

4 RESULTADOS CUALITATIVOS

Para responder a la primera pregunta de investigación, examinamos las principales decisiones de diseño arquitectónico en las principales versiones del sistema (consulte la Tabla 1). Cada subsección primero presenta la versión de Unix y luego proporciona una breve discusión de las principales decisiones de diseño para esa versión, como componentes (p. ej., comandos, rutinas, etc.), conectores (p. ej., llamadas al sistema, sockets, etc.), patrones (p. ej., capas, conductos y filtros, reflexión, etc.) y principios (p. ej., modularidad, virtualización, bajo acoplamiento, etc.).

Usamos las líneas de tiempo interactivas detalladas descritas en la referencia [74] para anotar cuándo apareció cada característica y cuándo desapareció. Los hipervínculos de las líneas de tiempo a la documentación nos permitieron evaluar el tipo y la importancia de cada característica nueva. La figura 1 resume los nueve diagramas de línea de tiempo en línea en una sola línea de tiempo de las principales decisiones de diseño de Unix.

En el siguiente texto, cuando la documentación relacionada con una decisión de diseño arquitectónico en particular apareció en una versión determinada de Unix, se hace referencia a la "página del manual" correspondiente utilizando el formato de nombre convencional (SECCIÓN). por ejemplo, ls(l) se refiere al comando ls en la Sección l del manual de referencia de Unix.6 En

6. Tenga en cuenta que la numeración de la sección romana (I–VIII) se empleó desde la primera hasta la sexta edición de investigación. Seguimos la misma convención en nuestras referencias a estas ediciones.

En otros casos, nuestro texto puede hacer referencia al código fuente de Unix, utilizando una nota al pie como esta .

4.1 PDP-7 Unix

Unix se escribió originalmente (como un sistema sin nombre) en lenguaje ensamblador PDP-7. Un artefacto recientemente encontrado y restaurado de mediados de la década de 1970 [59], nos permite examinar su estructura y las técnicas empleadas en su construcción. Se destacan las siguientes decisiones de diseño. Muchos de estos sobreviven hasta hoy.

Kernel A pesar del diminuto tamaño del sistema de 13.691 L2 líneas, existe una clara separación entre un kernel de sistema operativo que ofrece unas pocas decenas de servicios y comandos a nivel de usuario. El núcleo carga y ejecuta comandos a nivel de usuario, proporciona la abstracción de archivos, virtualiza las interfaces de hardware y establece la propiedad de los archivos.

Capas y particiones El sistema está estructurado en dos capas: el kernel y los comandos. Siguiendo el patrón Layers [71, p. 33], los comandos llaman al kernel, pero el kernel no depende de los comandos.

Además, los comandos se adhieren al principio de bajo acoplamiento: el código de cada comando no está acoplado al código de otros comandos. Esta partición se establece a través de una convención de nomenclatura de archivos: los nombres de archivos que comienzan con la misma secuencia (por ejemplo, ed para ed1.sy ed2.s) pertenecen a la misma partición.

Llamada al sistema La transferencia de control entre los programas de usuario y el núcleo se implementa a través de conectores especiales: llamadas al sistema. Los archivos de código fuente del kernel definen etiquetas 35L3 cuyo nombre comienza con un punto.
Estos son los nombres de los puntos de entrada de llamadas al sistema. Un subconjunto de etiquetas 28L4 está agrupado en una tabla, S2 que permite llamarlas desde programas de usuario utilizando la instrucción sys (algunas etiquetas, como las de acceso a disco de bajo nivel, no se exportan). En el manual de la segunda edición, encuentre las llamadas al sistema documentadas en una sección dedicada (II) del manual.

Listado 1. La definición de inodo en PDP-7 Unix

```
ii: .=.+1
inodo:
  i.flags: .=.+1
  i.dskps: .=.+7 i.uid: .=.
  +1 i.nlks: .=.+1
  i.size: .=.+1 i.uniq: .
  =.+1 .= inodo+12
```

Intérprete Al menos dos comandos del sistema ind (sangría) y lcase (conversión a minúsculas) están escritos en un lenguaje (relativamente) de alto nivel, a saber, B. Esto se implementa con un intérprete de código de subprocesos [90].

Implementación monolítica El kernel está estructurado como nueve archivos en lenguaje ensamblador (s1.s – s9.s) que carecen de descomposición y particionamiento fácilmente perceptibles. Lo mismo se observa para el editor ed, que consta de dos archivos con nombres similares (ed1.s y ed2.s).

Gestión de procesos El kernel puede crear una copia programada de forma independiente de un proceso en ejecución a través de la llamada al sistema de bifurcación fácil de implementar, S3, que lleva el nombre y el modelo de la bifurcación de Melvin Conway y se une a las primitivas multitarea propuestas [91]. El reemplazo de la copia del proceso en ejecución con otro programa a través de la llamada al sistema exec no se implementó en ese momento. En cambio, el shell superpone el código en ejecución al leer el código del otro proceso del disco y luego transfiere la ejecución a su punto de entrada con una instrucción de salto [55], [59].

Para la Primera edición de investigación, la interfaz de administración de procesos se había convertido en cuatro llamadas al sistema que definen la forma en que los procesos: se crean: bifurcación (II), se carga su código: exec (II), se terminan: exit (II) y son monitoreados para terminación—espera(II). Este modelo básico ha sido estandarizado bajo POSIX [92] y sobrevive hasta el día de hoy. La separación de la creación de un nuevo proceso a partir de la carga del código correspondiente puede parecer una elección arquitectónica peculiar, porque sus beneficios (la capacidad de crear un hermano idéntico de un proceso existente) son pequeños; por lo general, una llamada a la bifurcación es seguida inmediatamente por una a exec. La razón detrás de esta elección parece ser histórica. Dada la existencia de la llamada al sistema de bifurcación, era más fácil agregar una llamada ejecutiva que crear desde cero una llamada que combinara las dos.

Gestión de descriptores El kernel proporciona funcionalidad de E/S, como lectura y escritura, a través de conectores especiales que maneja el descriptor de archivos; estos son números enteros pequeños que asignan llamadas de E/S al archivo o dispositivo subyacente. Las rutinas del kernel fgetS4 y fputS5 proporcionan una interfaz básica para obtener y deshabilitar presentar descriptores de archivo a otras llamadas al sistema del kernel (por [S6] ejemplo, creat, S7 seekS8) open. Separación de los metadatos del archivo del nombre del archivo El kernel PDP-7 separa los metadatos de un archivo (identificación de usuario, tamaño, ubicaciones de bloque de disco, número de enlaces) del nombre del directorio del archivo mediante la introducción del concepto de un nodo de información de archivo (inode; ver Listado 1). Una función (nameiS9) puede obtener el inodo asociado con un nombre de ruta, mientras que otras funciones (igetS10 e iputS11) se ocupan de los archivos abiertos a través de sus inodos. Este elegante conector simplifica muchas tareas de administración de archivos.

Dispositivos como archivos El núcleo sigue el principio de virtualización al abstraer dispositivos, como la consola, la segunda terminal y la unidad de cinta de papel, en archivos a los que se puede acceder a través del directorio del sistema del sistema de archivos S12 (/dev en versiones posteriores) . Este tipo de vinculación permite que programas arbitrarios se comuniquen con cualquier dispositivo.

File I/O Una interfaz simple pero poderosa, basada en y el sistema llama abierto, [S13] leer, [S14] escribir, [S15] buscar, [S16] decir, [S17] cerrar, permite que los programas accedan a los [S18] archivos como una secuencia plana de bytes, tanto en forma de acceso secuencial como aleatorio. Esta interfaz ha sobrevivido hasta hoy, tanto como llamadas al sistema Unix como API de E/S en lenguajes de programación populares.

Sistema de archivos Cuatro llamadas al sistema permiten la manipulación de archivos dentro del sistema de archivos: creat y [S19] [S20] cambiar el nombre, [S21] Desvincular [S22] . enlace, Todos han sobrevivido hasta hoy. La funcionalidad de la llamada al sistema creat ha sido usurpada de forma generalizada por open. [S23] Además, las llamadas al sistema renombrar, vincular y desvincular se ampliaron en 2008 con hermanos que trabajan en descriptores de archivos para evitar condiciones de carrera. .

4.2 Primera edición de investigación La

Primera edición de investigación (3 de noviembre de 19717) fue una reescritura del PDP-7 Unix dirigida al procesador PDP-11.
Las siguientes decisiones de diseño arquitectónico son visibles en esta edición. Tenga en cuenta que las decisiones de diseño relacionadas con el caparazón también pueden haber estado disponibles en la edición PDP-7, pero el caparazón correspondiente no parece haber sobrevivido para estudiar su implementación.

Llamadas al sistema Aunque la primera edición de Unix fue una reescritura completa de PDP-7 Unix, retuvo una gran cantidad de las llamadas al sistema definidas, estableciendo así la arquitectura central de la interfaz de llamadas al sistema de Unix. En concreto, de las llamadas al sistema 28L6 implementadas en la versión S25 del PDP-7 y las llamadas 34L7 implementadas en la Primera Edición, S26 18L8 son comunes entre las dos: L9 chdir, chmod, chown, close, creat, exit, fork, getuid, enlazar, abrir, leer, rele, buscar, setuid, contar, tiempo, desvincular, escribir. Más impresionante aún, de las llamadas al sistema 34L10 implementadas en la Primera edición, 18L11 también han sobrevivido en la versión moderna de FreeBSD-11.0.1: S27, L12 chdir, chmod, chown, close, creat, fork, fstat, getuid, link, mkdir, montar, abrir, leer, setuid, stat, desvincular, esperar, escribir.

API de código binario A nivel de la CPU, las llamadas al sistema normalmente se envían a través de un vector de dirección de memoria que contiene la ubicación del código que implementa cada llamada. En el nivel de programación, las llamadas al sistema se denominan por nombres, como open o exec. En lugar de asignar dinámicamente los nombres de las llamadas al sistema a las entradas de esta tabla, la Primera Edición estableció un esquema de numeración para colocar las llamadas al sistema en posiciones estables dentro de la tabla. Esto permite que los sistemas Unix mantengan la compatibilidad API binaria de los programas compilados entre versiones sucesivas e incluso entre diferentes implementaciones, como Linux, sin necesidad de costosas capas de adaptación. Como se puede ver en las llamadas al sistema definidas en 1971 First EditionS28 (Listado 2L13) y las llamadas correspondientes definidas en FreeBSD-11.0.1S29 de 2016 (Listado 3L14), el esquema de numeración establecido persiste hasta hoy.

7. Las fechas proporcionadas aquí son dadas por Salus [77, p. 43].

Listado 2. Llamadas al sistema 0–10 definidas en la primera edición de Unix de 1971

```
sysrele / 0 sysexit /
1 sysfork / 2
sysread / 3
syswrite / 4
sysopen / 5
sysclose / 6 syswait /
7 syscreat / 8
syslink / 9 sysunlink /
10
```

Listado 3. Llamadas al sistema 0–10 definidas en 2016 FreeBSD 11.0.1

```
0   { int nosys(vacío); } syscall nosys_args int { void sys_exit(int rval); } exit
1   sys_exit_args void { int fork(void); } { ssize_t read(int fd, void *buf, size_t
        nbyte); } { ssize_t write(int fd, const void *buf, size_t nbyte); } { int
2   open(char *path, int flags, int mode); } { int cerrar(int fd); } { int wait4(int pid,
3   int *status, int options, struct rusage *rusage); }
4
5
6
7
8   { int creat(char *ruta, modo int); } { int link(char *ruta, char
9   *enlace); } { int unlink(char *ruta); }
10
```

Abstracción de E/S estándar El shell First Edition ofrece la capacidad de asociar archivos especificados por el usuario en lugar de la entrada estándar y la salida estándar del programa, a través de los símbolos de redirección de E/S correspondientes (< y >). Esto sigue el principio de virtualización al abstraer la E/S estándar de un programa del terminal, lo que permite que los programas operen en archivos arbitrarios. La decisión de diseño se implementa cerrando el descriptor de archivo de entrada o salida predeterminado (normalmente asociado con el terminal) y abriéndolo de nuevo para asociarlo con el archivo especificado.S30

Capa de E/S de archivos genéricos A lo largo de varias versiones sucesivas, vemos la evolución de una capa entre las llamadas al sistema de lectura y escritura y los controladores de dispositivos [50]. Este maneja la lectura (readiS31) y la escritura (writeiS32) a través de un inodo, la funcionalidad de lectura/escritura común a ambos (rdwrS33), así como la asignación de datos a bloques de disco (bmapS34).

Herramientas y juegos aportados por el usuario El manual de la primera edición contiene una sección (VI) que documenta los "Programas mantenidos por el usuario". Sorprendentemente, esto sucedió décadas antes de que las distribuciones de sistemas operativos de código abierto, como Debian y FreeBSD, comenzaran a organizar las contribuciones de código de terceros en forma de los llamados "paquetes" o "puertos". Los sistemas operativos por definición alojan código escrito por el usuario. La importancia arquitectónica de esta decisión de diseño de la Primera Edición es que los componentes mantenidos por el usuario están documentados en el manual del sistema y están instalados en un directorio visible en todo el sistema (normalmente /usr/ bin, archivos binarios del usuario) en lugar de en la casa de los autores

TABLA 2
Formatos de archivos documentados y sus usuarios en la Primera y Segunda (*) Edición de Investigación

Formato	Descripción	Clientela
a.fuera	Ensamblador y enlazador producción	as, ld, tira, nm, onu
Archivo	Bibliotecas de código objeto ar, ld	
Centro	Kernel de imagen de programa bloqueado, db	
Directorio	Directorios del sistema de archivos	du, find, ls, ln, mkdir, rmdir check, dump,*
sistema de archivos	formato del sistema de archivos	mkfs, restor* chown, find, getpw,* login,*
Contraseña	Cuentas de usuario y contraseñas	ls, passwd* mt,* tap* init, login,* who,* write * cuenta, fecha,
Cinta*	formato de archivo DECtape	inicio, inicio de sesión,
utmp	Usuarios Conectados	tacto, quién
wtmp*	Historial de inicio de sesión de los usuarios	

directorios. Este método admite un método ligero para que los usuarios contribuyan con código al sistema, que luego puede madurar para convertirse en una parte oficialmente compatible.

Los programas aportados por los usuarios de la primera edición incluían lenguajes de programación (básico), juegos (bj: black jack, chess, moo, ttt: tic-tac-toe), herramientas (das: desensamblador), interfaces periféricas (dli, dpt: cargar cintas de papel DEC), y utilidades conocidas hoy en día (cal, sort). La documentación del software aportado por el usuario se hizo cumplir a través de una medida técnica interesante: un trabajo programado (cron) eliminaría el software que carecía de páginas de manual actualizadas [80]. Actualmente, la sección VI del manual de Unix documenta los juegos, mientras que algunas herramientas documentadas en la primera edición ahora son comandos de usuario de Unix estandarizados. Los terceros aún pueden aportar código a las distribuciones de Unix a través de sus puertos o mecanismos de paquetes.

El shell como programa de usuario La documentación del archivo de contraseñas, passwd(V), detalla que el quinto campo de cada registro contiene el programa que se utilizará como shell. Esto permite especificar componentes arbitrarios como aquellos con los que interactuará un usuario que haya iniciado sesión; se dieron como ejemplos un editor para el personal de oficina y juegos [36].

Interoperabilidad a través de formatos de archivo documentados La Sección V del manual de la Primera Edición documenta nueve formatos de archivo. Éstos actúan como conectores, permitiendo que diversos programas interoperen a través de un mecanismo de acoplamiento externo leyendo y escribiendo los archivos correspondientes. Se agregaron dos más en la segunda edición y se siguieron agregando más en ediciones futuras. Los formatos de archivo utilizados por más de un programa se enumeran en la Tabla 2. Los archivos demuestran dos de los principios arquitectónicos del sistema: usar archivos planos en lugar de estructuras de archivos elaboradas y adherirse a las convenciones (uso de formatos documentados) en lugar de implementar una aplicación compleja. mecanismos (por ejemplo, API).

Estructura de directorios en árbol Dos llamadas al sistema, mkdir(II) y chdir(II), proporcionan la interfaz utilizada para crear un nuevo directorio de archivos y para establecer un directorio como el actual.
Otros elementos necesarios para crear una estructura de directorios en árbol se establecen por convención, lo que minimiza la complejidad arquitectónica. Específicamente, los directorios son archivos sin formato que contienen entradas de archivos en un formato documentado conocido: directorio (V). Además, dos entradas de directorio con nombres especiales, “.” y

“..”, apunta al directorio actual y principal respectivamente.
Varios comandos brindan el soporte requerido a nivel de usuario: chdir(I), find(I), ln(I), ls(I), stat(I), mkdir(I), mv(I), rm(I) y rmdir(I). Realizan tareas administrativas y hacen cumplir las restricciones al operar directamente en los datos del directorio.

Interfaz de sistema de archivos montable Dos llamadas al sistema, mount(I) y umount(I), y dos programas de administrador con el mismo nombre proporcionan una interfaz para conectar unidades de almacenamiento que contienen sistemas de archivos a puntos arbitrarios de la estructura de directorios. Su existencia admite un único espacio de nombres con estructura de árbol para todos los archivos, ocultando a los usuarios y programas la complejidad y fealdad de las "unidades" o "dispositivos". También guió con el ejemplo la filosofía de utilizar un único esquema de nomenclatura coherente para todos los archivos, lo que demostró ser importante a medida que el sistema evolucionaba.

4.3 Segunda edición de investigación

El código fuente de la segunda edición (12 de junio de 1972) solo ha sobrevivido como unos pocos fragmentos del programa de utilidad del sistema, que se recuperaron de un subconjunto de DECtapes de un volcado de disco. Afortunadamente, el manual de esta edición sobrevivió como documento impreso y proporcionó la base para las observaciones de la evolución arquitectónica de esta sección.

Biblioteca de software El manual de la Segunda Edición contiene una sección (III) que documenta 23 “subrutinas”, con un alcance considerablemente más amplio que las pocas documentadas en la Primera Edición. Estos componentes consisten principalmente en un emulador matemático de coma flotante, funciones matemáticas trigonométricas, logarítmicas y de conversión, E/S almacenadas en búfer, administración de memoria, clasificación y procesamiento de cadenas. Más de la mitad (14) de ellas han sobrevivido como funciones con el mismo nombre y funcionalidad en la biblioteca C moderna: atan(III), atof(III), atoi(III), ctime(III), cos(III),exp (III), getc(III), hipot (III), itoa(III), log(III), putc(III), qsort(III),sin(III) y sqrt(III). Su supervivencia muestra el poder de las abstracciones bien elegidas.

4.4 Tercera edición de investigación

La tercera edición (febrero de 1973) está disponible a través de las páginas del manual (14 982 L15 líneas de código troff) y el compilador C (2751 L16 líneas de código C).

Tuberías y filtros Este patrón se introdujo en la Tercera Edición [77, p. 50], pero el código ensamblador del núcleo correspondiente no ha sobrevivido. Incluso en el código fuente C del núcleo de la Cuarta Edición, la llamada al sistema de tuberías es solo un stub que redirige al punto de entrada de la llamada al sistema nosys. Parece probable que la llamada al sistema correspondiente se implementara en la versión de ensamblaje del núcleo, que coexistía con él, y la versión C no se había puesto al día. Sin embargo, el manual de la tercera edición documenta el sistema de tuberías llamado S35 y la construcción de tuberías a través del shell.S36 (la sintaxis utilizada para las tuberías era en ese momento diferente de la actual). Además, la interfaz de diversos comandos se cambió de la noche a la mañana para permitir para que se ejecuten como filtros, es decir, reciban entradas de otro proceso a través de su flujo de entrada estándar y proporcionen su salida a otro proceso a través de su flujo de salida estándar [75]. Por ejemplo, los comandos cat, od, pr y sort están documentados en el manual de la segunda edición con un argumento de archivo de entrada obligatorio. En las páginas correspondientes del manual de la Tercera Edición, el argumento del archivo es opcional—cuando faltan los comandos procesan su entrada estándar. Además, la documentación de numerosos comandos —crypt(I), hyphen(I), od(I), opr(I), ov(I), pr(I), sort(I)— establece explícitamente que pueden ser utilizado como filtro.

NAME

pipe – create a pipe

SYNOPSIS

(pipe = 42.)
sys pipe
(read file descriptor in r0)
(write file descriptor in r1)

pipe(fildes)
int fildes[2];

Fig. 2. La interfaz de llamada al sistema pipe(II) documentada tanto para lenguaje ensamblador (usando los registros r0 y r1) como para llamadores C.

4.5 Cuarta Edición de Investigación La

Cuarta Edición (noviembre de 1973) está disponible a través de las páginas de su manual—18,975L17 líneas de código troff—y el kernel—7,141L18 líneas de las cuales sólo 768L19 están escritas en ensamblador PDP-11 y el resto están escritas en C. Curiosamente, el kernel exhibe una división de esfuerzos en los límites arquitectónicos: Ken Thompson (ken) parece haber trabajado más en la parte principal del kernel, S37, mientras que Dennis Ritchie (dmr) parece haber trabajado principalmente en los controladores de dispositivos. S38

Programación estructurada en un lenguaje de alto nivel La reescritura del núcleo del sistema del lenguaje ensamblador PDP-11 en un lenguaje de alto nivel que luego se convirtió en C (en ese momento se conocía como "nueva B") impuso disciplina en el alcance de identificadores. Esto aumentó la modularidad del kernel al permitir la definición de funciones pequeñas (en promedio, alrededor de 17.9L20 líneas de largo).
Por lo tanto, el núcleo de la Cuarta Edición define funciones C 105L21 y símbolos de lenguaje ensamblador 50L22 . Compare estos números con los símbolos 200L23 (globales) definidos en el kernel PDP-7 y los símbolos 248L24 definidos en el kernel First Edition (PDP-11).

Grupos de usuarios El núcleo introduce grupos de usuarios y dos llamadas al sistema para gestionarlos: getgid(II) y setgid(II). Algunos comandos como chmod(I) y ls(I) se ajustan correspondientemente, y los permisos de archivo se amplían para incluir los de grupo además de la configuración existente de 'propietario' y 'otros'. A pesar de su interfaz espartana, el concepto es extremadamente poderoso. Junto con la propiedad grupal de archivos (que incluye dispositivos asignados al espacio de nombres del sistema de archivos), permisos asociados con un grupo de archivos y la capacidad de hacer que los programas asuman la identidad de un grupo específico, permite el control administrativo del acceso a los recursos de acuerdo con el grupo y la acción de un usuario. Por ejemplo, los permisos de grupo apropiados pueden proporcionar a todos los operadores acceso a la unidad de disco y cinta con fines de copia de seguridad, sin necesidad de asociar una lista de control de acceso compleja con cada dispositivo correspondiente. El concepto es un caso elegante de resolver un problema agregando otro nivel de direccionamiento indirecto.

API independiente del lenguaje La implementación gradual del sistema en un lenguaje de alto nivel requería una API que fuera compatible tanto con el código de lenguaje ensamblador como con el código escrito en C. En consecuencia, las llamadas al sistema se proporcionan y documentan a través de una API a la que se puede llamar desde ambos lenguajes; se puede ver un ejemplo en la Fig. 2. Dichos mecanismos que respaldan la coexistencia de lenguajes bajo el mismo techo se ampliaron más tarde para cubrir Fortran y Pascal, y hoy en día sirven diversos lenguajes que van desde Java y Go hasta JavaScript y Python.

Reutilización de definición de estructura de datos El kernel contiene en su directorio de nivel superior 12L25 C archivos de encabezado que se utilizan en

Instancias 165L26 por archivos de código fuente del kernel 35L27 . (Con respecto a la adopción de encabezados por parte de los programas de espacio de usuario, el código fuente de la quinta edición, el más cercano disponible, tiene 17L28 instancias de uso de archivos de encabezado en archivos 13L29). Los archivos de encabezado proporcionan un mecanismo compartido para comunicarse a través de estructuras de datos reutilizadas, algo que en el pasado se realizó simplemente copiando la definición de la estructura de datos de un manual en el código de cada programa. El uso de archivos de encabezado permite la evolución de las estructuras de datos mediante la adición de campos y cambios en sus tipos. Esto, a su vez, se puede utilizar para promover la portabilidad mediante el uso de tipos apropiados para cada arquitectura de CPU.

Gestión dinámica de recursos Se introducen dos rutinas, mallocS39 y mfree, para gestionar la asignación dinámica y la liberación de bloques de S40 área de intercambio de disco continuo para procesos intercambiados. A través de estas rutinas, ambas asignaciones reutilizan la misma estructura de datos subyacente, un mapa. Cada uno de los dos mapas (coremapS41 y swapmapS42) es una matriz de estructuras que contienen la posición y el tamaño de cada bloque asignado [60, p. 5–1].

Abstracción del controlador del dispositivo Los documentos del manual en la sección IV 16 "archivos especiales", que se encuentran en el directorio /dev. Estos corresponden a diversos dispositivos, incluyendo la interfaz de fotocompositor cat(IV), la unidad de respuesta de voz da(IV), la interfaz de teléfono de datos dc(IV), la máquina de escribir de consola kl(IV), el lector de cinta de papel pc(IV) / punch, la interfaz de cinta magnética tm(IV) y varios tipos de unidades de disco. Estos archivos son implementados por controladores de dispositivos. S43 A nivel del kernel, cada controlador de dispositivo de caracteres proporciona a través de la

tabla cdevsw lo que hoy llamaríamos una interfaz orientada a objetos con cinco métodos: S44 d_open, d_close, d_read, d_write y d_sgtyy. Los dispositivos de bloque proporcionan tres funciones a través de la interfaz bdevsw similar: d_open, d_close y d_strategy. Estas funciones tienen una semántica en su mayoría obvia, transformando las solicitudes de E/S independientes del hardware en el protocolo requerido por los dispositivos correspondientes. La función d_strategy es responsable de poner en cola las solicitudes de lectura y escritura y la función d_sgtty de obtener y configurar la velocidad de un terminal y los indicadores de procesamiento. Esta interfaz estandarizada oculta las complejidades del hardware específico del dispositivo del resto del núcleo y de los programas de nivel de usuario, virtualizando así los dispositivos subyacentes. A diferencia de esta interfaz modular, las funciones de interrupción asociadas con los dispositivos están codificadas directamente en la tabla de interrupciones. S45

Sorprendentemente, tanto la interfaz cdevsw como la bdevsw (renombrada como devsw), ampliada con algunas funciones más, aún existen en las versiones modernas de Unix, lo que demuestra la perdurable relevancia y utilidad del diseño. S46, S47 Buffer Cache El buffer cacheS48

almacena en la memoria principal copia de los datos leídos o escritos en el almacenamiento secundario.
Esto cierra la brecha de rendimiento entre el almacenamiento secundario de alta latencia y la memoria principal de baja latencia.
Ofrecido como un servicio para todas las E/S de dispositivos de bloque, mejora el rendimiento de la E/S de disco de proceso de usuario y kernel, a expensas de complicar el mantenimiento de estructuras de disco coherentes.

La memoria caché del búfer es otro patrón que ha persistido a lo largo del tiempo hasta la versión actual de FreeBSD-11, incluso hasta los nombres de tres indicadores de estructura de búfer. S49, S50

4.6 Quinta edición de investigación

A la quinta edición sobreviviente (junio de 1974) solo le falta el marcado de origen de las páginas del manual. Esta edición se puso oficialmente a disposición de las universidades para uso educativo [93, p. 8].

Archivos de comandos Ya desde la segunda edición, el shell documenta su capacidad para ejecutarse con el nombre de un archivo que contiene comandos como argumento. En la Quinta Edición vemos cuatro archivos que contienen tales secuencias de comandos. Estos se utilizan para configurar el sistema en el momento del arranque, S51 para actualizar el archivo del compilador C que contiene tablas de plantillas de expresiones en idiomas diferentes, S52 para compilar, vincular e instalar diversos archivos del sistema, S53, S54 y para crear la tabla de instrucciones del manual. content and index.S55 Con solo 69L30 líneas, la cantidad de código incrustado en estos archivos es muy modesta. Sin embargo, este uso marca el comienzo de las secuencias de comandos en Unix, que luego se convertirá en un paradigma dominante.

4.7 Sexta edición de investigación

La sexta edición (mayo de 1975) es la primera que estuvo ampliamente disponible fuera de Bell Labs a través de licencias para usuarios comerciales y gubernamentales. John Lions estudió y documentó la estructura del núcleo como material para la enseñanza de dos cursos de sistemas operativos en la Universidad de Nueva Gales del Sur en Australia en 1977 [60].

Biblioteca C portátil Se proporciona una biblioteca S56 de rutinas implementadas en el lenguaje de programación C con el objetivo explícito de mejorar la portabilidad entre los tres sistemas operativos en los que se puso a disposición el lenguaje: PDP 11 Unix, Honeywell 6000 GCOS e IBM 370 OS. La biblioteca implementa en C, funcionalidad que en ese momento estaba codificada en lenguaje ensamblador, como la impresión formateada S57, S58 y la asignación dinámica de memoria. S59, S60 En el lanzamiento de la sexta edición, parece que tanto la biblioteca portátil como las rutinas originales coexistían, y que las herramientas de Unix dependían de las rutinas del lenguaje ensamblador. Algunas rutinas, por ejemplo, printf(III), se ofrecieron como alternativas compatibles con plug, mientras que otras funciones (por ejemplo, asignación de memoria) se proporcionaron mediante diferentes interfaces.
Con el tiempo, la biblioteca C portátil influyó en el diseño y la implementación de la biblioteca Unix C. La biblioteca C estándar moderna define tanto las rutinas derivadas de la implementación del lenguaje ensamblador original como aquellas, como system (3), que fueron introducidas por la versión de la biblioteca portátil.

4.8 Séptima edición de investigación La

séptima edición (enero de 1979) incluye muchos comandos nuevos influyentes y es la versión que se adaptó ampliamente a otras arquitecturas de procesador.

Unix como máquina virtual Primeros problemas al portar programas escritos en C entre diversos sistemas operativos [54, p. 2025] convenció a Dennis Ritchie de que sería más fácil portar el sistema operativo entre diversos hosts que portar los programas de aplicación entre sistemas operativos [81]. Se compró y utilizó una computadora Interdata 8/32 para probar este punto. El proyecto involucrado

- la implementación de un compilador de C cuya parte de generación de código podría adaptarse a varias arquitecturas de CPU [94]; la extensión del lenguaje de programación C para ayudar a la portabilidad del código escrito en él;

> la abstracción a través de bibliotecas y definiciones de archivos de encabezado de elementos que variaban entre diferentes máquinas; y la identificación, revisión y aislamiento de las partes dependientes de la máquina del núcleo del grueso (95 por ciento) del código que podría permanecer igual en todos los sistemas [54].

Asignación dinámica de memoria Un asignador de memoria principal, malloc(3) se ofrece como parte de la biblioteca C. Permite que los programas asignen dinámicamente espacio de memoria para almacenar datos, en lugar de reservar cantidades fijas de espacio. El vacío que llena, es evidente por su adopción rápida y generalizada.
Es utilizado directamente por 26 programas en modo usuario (de aproximadamente 160) y también en la implementación de otras funciones de biblioteca, concretamente por la E/S estándar y por las bibliotecas aritméticas de precisión múltiple.

Análisis estático Se ofrece un programa dedicado, lint(1) [25], para verificar el código C en busca de problemas que no detecte el compilador de C. Realiza una verificación estricta de tipos, detecta posibles problemas de portabilidad e identifica construcciones propensas a errores o derrochadoras. El análisis estático de fallas del programa era, y sigue siendo, una tarea imprecisa y que demanda recursos. Implementarlo como un programa separado libera al compilador de sus demandas y también proporciona un lugar de experimentación aislado que no puede interrumpir fácilmente el desarrollo diario del código de producción.

Variables de entorno El núcleo, S61 el shell, S62 y la biblioteca C S63 actúan en conjunto para admitir variables de entorno: environ(5). Éstos permiten que una serie de cadenas arbitrarias (por convención, pares clave-valor) pasen por el árbol de invocación del proceso, estableciendo así un conector de paso de parámetros simple, de baja sobrecarga, abierto, unidireccional.
Las variables de entorno aparecen en el shell como variables ordinarias y se puede acceder a ellas en código C con una única llamada de función: getenv(3). Al ser parte de los datos de contexto del sistema operativo de un proceso, se heredan hasta niveles arbitrarios de invocación de procesos, sin necesidad de coordinación con capas intermedias.

Una variable de entorno importante es PATH, que especifica una lista de rutas de directorio donde el shell busca programas ejecutables. Cambiar estas rutas permite a los usuarios finales ampliar o sustituir los programas proporcionados por el sistema operativo.
La configuración del sistema operativo del usuario final se amplió posteriormente a otras áreas, incluida la ubicación de las páginas del manual (MAN PATH) y las bibliotecas vinculadas dinámicamente (ID_RUN_PATH), así como la arquitectura de alquiler del sistema de archivos: chroot(2). Esta línea de evolución culminó en los modernos sistemas de virtualización a nivel de sistema operativo, como los espacios de nombres y grupos de control del kernel de Linux, FreeBSD Jails y Solaris Zones.

Herramientas de desarrollo del lenguaje El generador de analizadores léxicos, lex(1) [29], presentado en la séptima edición, complementa el generador de analizadores sintácticos, yacc(1) [28], ya presente en la sexta edición. Juntos, estos dos ofrecen la base para construir interfaces de lenguaje de programación [95]. Esto simplifica significativamente la implementación de un analizador de lenguaje de programación a una tarea que puede realizar un programador competente en lugar de un experto en teoría de autómatas.
La utilidad de este enfoque se ejemplifica con la existencia de doce herramientas cuya gramática está escrita en yacc: awk(1), bc(1), cpp(1), egrep(1), eqn(1), lex(1), m4(1), make(1), pcc(1),S64 neqn(1) y struct(1). A través de la disponibilidad del compilador herramientas, la implementación de muchas instalaciones complejas se abstrae en el desarrollo de un lenguaje de dominio específico que actúa como una plataforma para resolver el problema correspondiente.

Lenguajes específicos de dominio Con la ayuda de la disponibilidad de herramientas de compilación, varias herramientas basadas en lenguajes pequeños o específicos de dominio [96], [97], [98] admiten una variedad de tareas de procesamiento genéricas de una manera que permite a los usuarios finales escribir código especializado para lograr sus objetivos particulares. Las herramientas introducidas en la séptima edición incluyen Bourne shell [84], [99]—sh(1), awk(1) para procesar registros orientados a campos [100], sed(1) para manipular archivos de texto sin formato [101], find(1) para recorridos de la jerarquía del sistema de archivos, expr(1) y bc(1) para evaluar expresiones, egrep(1) para encontrar líneas que coincidan con una expresión regular extendida, m4(1) para realizar el procesamiento de macros [102] y make(1) para mantener las dependencias del programa [103]. Algunos de los lenguajes, como los empleados por find, expr y egrep, son bastante básicos y el código escrito en ellos rara vez ocupa más de una sola línea. El resto son más sofisticados, y algunos se han usado ocasionalmente (mal) para construir aplicaciones grandes.

Jerarquía de directorios del sistema de archivos El diseño documentado (hier(7)) para la jerarquía del sistema de archivos especifica la función y el contenido de los directorios 51L31 . La estructura se ha mantenido mayormente estable a lo largo de los años. Incluso ha sido adoptado y estandarizado por la comunidad de Linux, en forma de Estándar de jerarquía del sistema de archivos. La estructura documentada ofrece otro ejemplo del establecimiento de convenciones flexibles sobre la implementación de mecanismos de aplicación rígidos. También demuestra la formalización de una estructura que evolucionó orgánicamente a lo largo de los años. Aunque la jerarquía de directorios cambió mucho antes de la séptima edición, ya que el equipo de Unix experimentó con varios diseños, se estabilizó después de documentarse y evolucionó solo gradualmente. Un ejemplo de un cambio temprano es que el directorio / usr se usó inicialmente para programas de usuario, pero luego se reutilizó para denotar un directorio de uso general, que normalmente reside en un gran sistema de archivos montado. Los desarrollos significativos posteriores a la séptima edición incluyen la adición del directorio /home para los archivos de usuario y el directorio /var para los archivos del sistema que cambian mientras el sistema se está ejecutando. Es importante destacar que la jerarquía documentada contiene todas las partes de un sistema autohospedado: código fuente, herramientas de desarrollo, bibliotecas y documentación.

4.9 Primer y segundo software de Berkeley Distribuciones

La primera distribución de software de Berkeley (BSD), lanzada a principios de 1978, contenía el sistema Pascal de Berkeley [104], el editor de línea ex [105] y varias herramientas. La segunda distribución de software de Berkeley (2BSD), incluía el editor de pantalla completa vi [106], la base de datos de capacidad de terminal asociada y la biblioteca de administración termcap, y muchas más herramientas, como csh shell [107].

Paquetes de software Las dos distribuciones de Berkeley presentaron a la comunidad de usuarios paquetes de software de terceros destinados a Unix. A lo largo de los años, los paquetes proliferaron y se distribuyeron, inicialmente a través de grupos de noticias de USENET [108, pp. 958–959] y luego a través de Internet en forma de puertos para una distribución específica del sistema operativo. La jerarquía de directorios del sistema de archivos establecida proporcionó una plantilla para diseñar el código fuente, la documentación y el manual.

paginas Además, el uso de make(1) proporcionó una forma común de expresar las reglas de compilación y despliegue. En total, 2BSD vino con archivos MAKE 32L32 .

4.10 3BSD El

lanzamiento de 3BSD, que salió a fines de 1979, extendió Unix 32V, un puerto directo de la séptima edición de Unix a la arquitectura DEC/VAX, con soporte para memoria virtual y las adiciones de 2BSD.

Paginación de memoria virtual El subsistema de memoria virtual (VM) [42] es un componente principal introducido en el kernel 3BSD, que comprende el 17 por ciento del código principal del kernel (2808 de 16039 líneas de código fuente C). S65 Está delimitado del resto de el código del kernel, al hacer que sus once archivos de código fuente comiencen con un prefijo único (vm), un método seguido por otros elementos en versiones futuras.

El sistema de VM admite principalmente la asignación de memoria a procesos cuando no hay memoria principal disponible mediante el intercambio de páginas de VM adecuadas al disco (paginación). Además, se proporcionan nuevas formas de llamadas al sistema read(2), write(2) y fork(2), que utilizan VM para realizar la E/S (vread(2) y vwrite(2)) y para reutilizar el espacio de memoria de un proceso—vfork(2). Esta violación de la abstracción resultó ser un experimento de corta duración. Las versiones posteriores abstrajeron el uso de VM por las rutinas de E/S comunes (lectura (2) y escritura (2)), eliminando la necesidad de llamar a rutinas separadas para beneficiarse de las capacidades de VM. Además, se desaconseja el uso de vfork (2) en las versiones modernas de FreeBSD.

4.11 4BSD

El lanzamiento de 4BSD (octubre de 1980) fue desarrollado por el recién establecido Grupo de Investigación de Sistemas Informáticos que trabaja en un contrato con la Agencia de Proyectos de Investigación Avanzada de Defensa (DARPA). El contrato pretendía estandarizar, a nivel de sistema operativo mediante la adopción de Unix, el entorno informático utilizado por los centros de investigación de DARPA [77, p. 159–160]. El lanzamiento incluía un sistema de archivos de bloques de 1k, compatibilidad con VAX-11/750, correo electrónico mejorado, control de trabajos y nueva semántica de señales que abordaba las condiciones de carrera existentes: las llamadas señales confiables [109, pp. 270–283].

Biblioteca de expresiones regulares Las expresiones regulares ocupan un lugar destacado en numerosas herramientas de Unix, como ed(1), grep(1), egrep(1), awk(1), sed(1) y expr(1). En consecuencia, admitir la funcionalidad correspondiente como parte de la biblioteca C (regex(3)) es una decisión de diseño obvia. La provisión de la biblioteca de expresiones regulares en 4BSD es una mejora importante, presagiando el soporte generalizado para expresiones regulares en la mayoría de los lenguajes de programación modernos.

Sin embargo, el desarrollo de la biblioteca de expresiones regulares tomó tiempo y su adopción fue mediocre. Inicialmente, cada herramienta tenía su propio motor de expresiones regulares. S66, S67, S68, S69, S70 La razón de esto puede haber sido incompatibilidades entre las expresiones regulares procesadas por diversas herramientas, soporte primitivo para bibliotecas o propietarios de herramientas demasiado aficionados. su propia implementación de expresiones regulares para exigir una biblioteca común [110].
Incluso cuando se proporcionó la biblioteca, pocos programas la adoptaron. En 4BSD, solo un programa, more(1), hacía uso de la biblioteca. S71 Para 4.3BSD (1986), solo dos programas más (nuevos) la usaban: dbx(1) y rdist(1). La razón de esta adopción lenta puede ser que los desarrolladores de BSD Unix No se sienten dueños de las herramientas y el código desarrollado en Bell Labs. Para el lanzamiento de FreeBSD 11.0 (2016), la situación había cambiado y cuatro de las herramientas mencionadas en el párrafo anterior, ed(1), grep(1), sed(1) y expr(1), se reescribieron como software de código abierto y usó la versión contemporánea de la biblioteca de expresiones regulares.

Manejo de pantalla optimizado La biblioteca curses(3) aborda el problema de colocar caracteres en posiciones arbitrarias de diversas pantallas de terminales incompatibles en una conexión de ancho de banda bajo. Las pantallas direccionables por cursor solían requerir secuencias de escape diferentes e incompatibles para realizar tareas como limpiar la pantalla, usar una fuente resaltada o colocar el cursor en una posición específica de la pantalla. Además, actualizar una pantalla completa con un tamaño de 80 24 caracteres a través de una conexión de terminal serie de 300 a 1200 baudios puede llevar mucho tiempo. En consecuencia, en aras de la eficiencia, el contenido de la pantalla utilizable debe conservarse y solo las diferencias de contenido deben enviarse a la línea. Para resolver estos problemas, la biblioteca curses abstrae las secuencias de escape de caracteres requeridas para manipular terminales direccionables por cursor en un conjunto de rutinas de biblioteca C y una base de datos, termcap(5), que almacena las secuencias asociadas con cada tipo de terminal. La biblioteca también optimiza las actualizaciones de la pantalla reflejando su contenido en estructuras de datos internas y utilizando esos datos para minimizar los datos transmitidos.

Las modernas interfaces de línea de comandos funcionan en emuladores de terminal que se ejecutan en pantallas de mapa de bits con conexiones de gran ancho de banda, y casi todos los emuladores están estandarizados para seguir las secuencias de escape xterm del sistema X Window. Por lo tanto, ninguno de los requisitos originales asociados con la biblioteca de curses se cumple en la actualidad. Sin embargo, la biblioteca todavía se usa para mantener la funcionalidad de los programas diseñados para hardware completamente diferente.

4.12 4.2BSD

La versión 4.2BSD (septiembre de 1983) se basó en un diseño descrito en un manual de arquitectura escrito por Bill Joy y sus colegas [111]. Incluía muchas características importantes entregadas en 4.1BSD y tres versiones provisionales informales más [78]: 4.1BSD (mejoras de rendimiento); 4.1aBSD (herramientas de redes y redes TCP/IP); 4.1bBSD (Berkeley Fast Filesystem [43] y enlaces simbólicos); y 4.1cBSD (nuevo código de señal). En comparación con los lanzamientos preliminares, el lanzamiento final mejoró el soporte de red y agregó nuevas instalaciones de señal y cuotas de disco.

Familia de protocolos de Internet El soporte para la familia de protocolos de Internet fue posiblemente una de las decisiones de diseño de Unix más influyentes que apareció en la segunda década de vida del sistema. Con 6586 L33 líneas de código que implementan cinco protocolos (ARP, IP, TCP, UDP e ICMP), el esfuerzo parece muy modesto según los estándares actuales. Esta pila de protocolos se usó ampliamente como implementación de referencia en enrutadores y otros sistemas operativos. Desde una perspectiva arquitectónica, la decisión de implementar esta funcionalidad en el kernel en lugar de como un programa de espacio de usuario (como fue, por ejemplo, el caso de la implementación de KA9Q [112]) puede haber contribuido al rendimiento, la estandarización y la adopción generalizada de estos protocolos y la implementación correspondiente.

Comunicación entre procesos local y remota La comunicación entre procesos (IPS) bidireccional tanto local como remota entre procesos arbitrarios se establece a través del, ahora

TABLA 3
Usos de la API de socket en 4.2BSD

Llamada al sistema	Usos
unir	23
conectar	15
aceptar	13
seleccionar	12
escuchar	11
enviar a	10
apagar recvfrom	9
	8
recibe el nombre del calcetín	6
recibir	2
enviar	2
sendmsg	1
getsockopt	0
recvmsg	0
par de enchufes	0

ubicuo, socket(2) API para configurar y aceptar conexiones de red. En versiones anteriores, IPS se implementaba principalmente a través de la realización de llamadas al sistema pipe(2) del patrón correspondiente. Esto establece solo una ruta de comunicación unidireccional entre procesos de un común
antepasado.

En contraste con la parsimonia de las adiciones anteriores de llamadas al sistema Unix, la nueva API se basa en una plétora de nuevas llamadas al sistema (Tabla 3L34). Hay argumentos a favor y en contra de calzar nuevas funciones en las llamadas al sistema existentes, como podría hacerse en este caso reutilizando la API open(2), read(2), write(2) y close(2). Reutilizar o mejorar una API existente reduce la superficie de la API del sistema y la curva de aprendizaje asociada, pero también puede afectar negativamente la compatibilidad del código existente, el rendimiento del tiempo de ejecución y la facilidad de uso de la API. Ciertamente, sin embargo, el despilfarro exhibido marca una desviación del estilo arquitectónico de la parsimonia de las ediciones anteriores de Unix.

La API de sockets se usa en la versión 4.2BSD por dos funciones de biblioteca: rcmd(3X) y rexec(3X); once demonios del sistema: comsat(8C), ftpd(8C), gettable(8C), implogd(8C), rexecd(8C), rlogind(8C), af(8C), rshd(8C), rwhod(8C), telnetd (8C) y tftpd(8C); y ocho programas en modo usuario: ftp(1C), rlogin(1C), rsh(1C), talk(1C), telnet(1C), tftp(1C), whois(1C) y sendmail(1C). Según el número, a veces pequeño, de usos de llamadas al sistema proporcionados en el código del cliente (Tabla 3), se podría afirmar que la API proporcionada fue sobrediseñada.

Además, en retrospectiva, la abstracción del protocolo TCP a los sockets de flujo y del protocolo UDP a los sockets de datagramas fue otro ejemplo de sobreingeniería, porque durante décadas los sistemas convencionales mantuvieron una relación uno a uno entre los dos protocolos y los correspondientes. abstracciones. Sin embargo, la proliferación y evolución de los protocolos de red en ese momento obligó a los diseñadores de la pila de redes a adoptar la abstracción como precaución para otras vías evolutivas. Esto se expresa en una sección de Advertencia en la página del manual de inet(4F).

> "El soporte del protocolo de Internet está sujeto a cambios a medida que se desarrollan los protocolos de Internet. Los usuarios no deben depender de los detalles de la implementación actual, sino de los servicios exportados".

Abstracción de procesamiento de directorios Tres nuevas llamadas al sistema—mkdir(2), rename(2), rmdir(2)—y la biblioteca de acceso al directorio(3) que comprende opendir(3), readdir(3), telldir(3), seekdir (3) y las funciones closedir(3), documentadas individualmente en 4.3BSD, abstraen el procesamiento de las entradas del directorio. Antes de la introducción de esta función, las operaciones de directorio se realizaban accediendo y manipulando directamente el contenido de las estructuras de disco correspondientes. Esta abstracción promueve la innovación en el diseño de sistemas de archivos, como los nombres largos de archivos introducidos con el sistema de archivos rápido de Berkeley [43].

Acceso a la red y a la base de datos de usuarios Una serie de funciones de biblioteca proporcionan una interfaz para acceder a las entradas almacenadas en la tabla del sistema de archivos—getfsent(3X), el archivo de grupo de usuarios—getgrent(3), la base de datos de hosts—gethostent(3N), la base de datos de redes — getnetent(3N), la base de datos de protocolos—getprotoent(3N), el archivo de detalles del usuario—getpwent(3) y la base de datos de servicios de red—getservent(3N). La abstracción de esta funcionalidad en bibliotecas reutilizables reduce la duplicación de código, los errores y las incompatibilidades, y también hace que los programas que utilizan esta funcionalidad sean fáciles de adaptar en el futuro. Por ejemplo, en FreeBSD moderno se pueden configurar las mismas rutinas (mediante el archivo nsswitch.conf(5)) para proporcionar datos: desde archivos locales (como era el caso en las implementaciones originales), desde una base de datos local de claves/valores, desde el sistema de nombres de dominio de Internet, desde servidores NIS/YP o desde un demonio de almacenamiento en cache

Controlador de pseudo-terminal El controlador de pseudo-terminal, pty (4), permite la creación de dispositivos similares a terminales controlados por software. Estos aparecen como un par de dispositivos maestro-esclavo. Un proceso, como un shell o un editor, adjunto al extremo esclavo tiene la ilusión de trabajar en una terminal física.
Sin embargo, su E/S no proviene de un terminal real, sino de otro proceso que controla el extremo maestro. A través de este mecanismo conector, los procesos de usuario arbitrarios pueden crear terminales virtuales que pueden ser utilizados por otros procesos sin ningún arreglo o ajuste previo. La función es utilizada en 4.2BSD por el demonio de inicio de sesión remoto: rlogind(8), el demonio (similar) de telnet: telnetd(8) y el programa TypeScript de la sesión de terminal: script(1).

4.13 4.3BSD Tahoe

La versión 4.3BSD Tahoe (junio de 1988) admitió la minicomputadora CCI Power 6/32 (nombre en clave Tahoe) y algoritmos TCP mejorados.

Compatibilidad con múltiples arquitecturas de CPU El núcleo se divide en partes independientes y dependientes de la máquina. Las piezas que dependen de la máquina son compatibles con la arquitectura VAXS72 original y la nueva arquitectura TahoeS73. La división coloca el código para las interfaces, la configuración del sistema S74 , S75 y bootingS76 en directorios separados. En total, de las 218 783 L35 líneas del código fuente del kernel del sistema, 104 279 L36 residen en los directorios vax y 42 112 L37 residen en los directorios tahoe. Por lo tanto, una parte significativa del código del núcleo (72.392L38 líneas) parece ser transferible entre diferentes arquitecturas de procesador.

Contribuciones de software de terceros El sistema contiene archivos 59L39 que comprenden 25,739L40 líneas que están marcadas como "software contribuido a Berkeley". Estos provienen de individuos (Arthur Olson, Chris Torek y Rick Adams), así como de corporaciones (Computer Consoles Inc, Excelan Inc, Harris Corp, Sun Microsystems, Inc y Symmetric Computer Systems). Aunque el tamaño de la

contribuciones es modesto, el fenómeno es importante, porque marca el comienzo del crecimiento del sistema a través de lo que evolucionó para convertirse en una comunidad de software de código abierto. Para la próxima versión (4.3BSD Reno), las contribuciones de software de terceros se habían incrementado a 896L41 archivos, 218,455L42 líneas, de alrededor de 70L43 entidades.

Gestión de zonas horarias La versión incorpora un paquete de gestión de zonas horarias de dominio público desarrollado fuera de Berkeley [64, p. 9].S77 El paquete almacena las reglas de cambio de zona horaria en una base de datos, lo que permite que se actualice independientemente del código que interpreta esas reglas. Esto permite a los usuarios finales configurar individualmente su zona horaria local y a los administradores actualizar fácilmente la base de datos a medida que entran en vigor nuevas reglas. Este es el enfoque seguido ahora por la mayoría de los sistemas Unix.

4.14 4.3BSD Reno La versión

4.3BSD Reno (junio de 1990) admitía implementaciones de sistemas de archivos virtuales a través de la interfaz vnode, estaciones de trabajo Hewlett-Packard 9000/300 y redes OSI. También incorporó un nuevo sistema de memoria virtual adaptado del sistema operativo de micronúcleo MACH de Carnegie-Mellon, una implementación de Network File System (NFS) realizada en la Universidad de Guelph, S78 y un demonio automontador.

Berkeley lanzó una cantidad considerable de código con una licencia que permite su fácil redistribución y reutilización.

Base de datos de reenvío de paquetes del kernel El kernel proporciona un dominio de socket de red especial, PF_ROUTE, que los programas de nivel de usuario pueden usar para consultar y manipular su base de datos de enrutamiento de paquetes de red: ruta (4). El núcleo utiliza esta base de datos para actuar como un enrutador al reenviar paquetes entre interfaces de red, mientras que los programas de nivel de usuario, como enrutado(8) y XNSrouted(8), implementan protocolos de enrutamiento al comunicarse con otros hosts a través de la red. Siguiendo elegantemente el principio de separación de preocupaciones, esto minimiza la cantidad de código complejo que debe mantenerse dentro del núcleo, al tiempo que evita la sobrecarga de cambio de contexto de un programa de enrutamiento en modo usuario.

Interfaz de sistema de archivos virtual Las operaciones de disco que se realizaron en inodos se virtualizan a través de una interfaz orientada a objetos de operaciones de vnode (vnodeops). S79 Una estructura de punteros de función proporciona acceso al almacenamiento, con funciones como abrir, leer y escribir, en cuanto a la denominación de archivos con funciones como mkdir, rename y readdir.
Los dos grupos se dividieron en 4.4BSD para simplificar la implementación de diferentes métodos de almacenamiento, como un sistema de archivos con estructura de registro [64, p. 244]. Esta interfaz se utiliza para implementar tres sistemas de archivos: UFS, el sistema de archivos Unix original; MFS, un sistema de archivos que almacena archivos en la memoria virtual; y NFS, un sistema de archivos que opera a través de conexiones de red.

Métodos de acceso a bases de datos La biblioteca db(3) y la API permiten que los programas almacenen y recuperen pares clave-valor en un archivo o base de datos ligera residente en memoria [113]. Los elementos se pueden almacenar utilizando estructuras de datos btree, hash o de archivo plano. Una interfaz orientada a objetos, implementada a través de punteros de función, proporciona métodos de acceso secuencial, get, put y delete. En contraste con la adopción letárgica de la biblioteca de expresiones regulares agregada en 4BSD, la funcionalidad provista es reutilizada por decenas de programas, con el archivo de encabezado db.h correspondiente incluido 105L44 veces en FreeBSD 11.1.

4.15 4.3BSD Neto/2

La versión 2 de redes 4.3BSD (junio de 1991) vino con una (lo que ahora se llama) reimplementación de código abierto de casi todas las utilidades y bibliotecas importantes que solían requerir una licencia de AT&T. También incluía un kernel que se había limpiado del código fuente de AT&T, lo que requería solo seis archivos adicionales para hacer un sistema completamente funcional. Esta fue la versión utilizada por Bill Jolitz para crear un sistema Unix de arranque compilado para PC basadas en Intel 386.

Funciones de flujo de E/S La familia de funciones funopen(3) permite que los programas C accedan a datos arbitrarios a través de la interfaz stdio(3), ampliamente utilizada. Las funciones similares a constructoras orientadas a objetos toman como argumentos punteros de función de lectura, escritura, búsqueda y cierre y devuelven un puntero de ARCHIVO opaco que admite todas las operaciones habituales, como getchar(3) e printf(3). Esta elegante interfaz se puede utilizar para proporcionar acceso similar a un flujo de datos comprimidos (zopen(3), protocolo de aplicación, fetch(3) o datos cifrados. Sin embargo, la interfaz específica, las pocas funciones de biblioteca que se basan en ella y su biblioteca GNU equivalente, funopencookie(3), que se agregó en FreeBSD 11, no han visto una adopción significativa.

4.16 4.4BSD El

lanzamiento de 4.4BSD (junio de 1994) salió después de dos años de litigios y conversaciones de acuerdos sobre el presunto uso de material propietario de AT&T. Como resultado de la negociación, esta versión eliminó tres archivos que estaban incluidos en la versión Net/2, agregó los derechos de autor de Unix System Laboratories (USL) a aproximadamente 70 archivos y realizó cambios menores en algunos otros. El lanzamiento incluyó trabajo adicional realizado en el sistema, como soporte para el sistema de archivos

Sistemas de archivos apilables La creación de una pila de vnode permite que un nuevo tipo de sistema de archivos use las operaciones de uno existente. El uso más simple de este concepto es el sistema de archivos nulo—mount_-null(8), que permite que un subárbol de un sistema de archivos existente aparezca en un lugar arbitrario del espacio de nombres del sistema de archivos global. Este concepto se amplió en 4.4BSD/Lite1 con el sistema de archivos de unión—mount_union(8), que permite la adición translúcida de un sistema de archivos (p. ej., grabable) encima de otro (p. ej., un CD-ROM).

Interfaz genérica de control del sistema La interfaz genérica de control del sistema—sysctl(3,8)—proporciona una función de biblioteca—sysctl(3)—y una utilidad de administrador—sysctl(8)—para examinar o configurar el estado del kernel, documentado más tarde a través de su interfaz interna—sysctl(9). Esta interfaz reemplaza el método original que implicaba acceder directamente al espacio de memoria del kernel a través de un archivo especial: /dev/kmem.

La función sysctl ofrece importantes beneficios de portabilidad, eficiencia, seguridad y mantenimiento en comparación con el método de acceso /dev/kmem al que reemplaza [64, pp. 612–614].

Sin embargo, al ofrecer una vista jerárquica independiente del espacio del kernel a través de la abstracción de la "base de información de administración" (MIB) comúnmente adoptada, se encuentra en desacuerdo y compite con conceptos arquitectónicos alternativos, a saber, la provisión de interfaces a través del sistema de archivos jerárquico de Unix, y la interfaz del kernel a través de llamadas al sistema, como podría hacerse a través de los sistemas de archivos kernfs, procfs y fdesc [63, p. 238].
Kirk McKusick, diseñador y desarrollador principal de BSD, en un correo electrónico a los autores de este artículo, explicó esta elección afirmando que los usuarios de BSD encontraron rotundamente que la instalación de sysctl era una forma mucho más conveniente para la administración remota del sistema en comparación con

el método de acceso al sistema de archivos jerárquico. Agregó que la interfaz sysctl también es considerablemente más eficiente.

4.17 Juego de parches 386BSD

386BSD fue un derivado de BSD Networking 2 Release dirigido a la arquitectura Intel 386 desarrollada por Lynne y William Jolitz [45]. El kit de parches de 386BSD contiene 171 confirmaciones asociadas con parches realizados en 386BSD 0.1 por un grupo de voluntarios desde mediados de 1992 hasta mediados de 1993.

Contribuciones de la comunidad organizada La funcionalidad del kit de parches agrega a la arquitectura de Unix un mecanismo para aceptar y distribuir contribuciones provenientes de un equipo descentralizado de personas. Unix se distribuyó por primera vez con una licencia de código abierto a través de la versión 1 de red 4.3BSD (Net/1) en noviembre de 1988. Este era un subconjunto del código que no incluía material que requería una licencia de AT&T. Se lanzó para ayudar a los proveedores a crear productos de red independientes sin incurrir en los costos de licencia binaria de AT&T, pero no incluía todo el material necesario para ejecutar el sistema. Esto fue abordado más tarde por la versión 386BSD. Sin embargo, ninguno de los dos lanzamientos ofreció una forma de administrar las contribuciones de terceros. Esta característica esencial de un proyecto de código abierto (a diferencia del software de código abierto) se formó más de cuatro años después del lanzamiento de 4.3BSD Net/1. Los elementos del kit de parches contienen sus cambios en el formato de "diferencia de contexto" de Unix y, por lo tanto, se pueden aplicar automáticamente a la distribución 386BSD. Cada parche va acompañado de un archivo de metadatos que enumera su título, autor, descripción y requisitos previos.

4.18 Descripción general de las versiones de FreeBSD

El Proyecto FreeBSD comenzó a principios de 1993 con el lanzamiento de FreeBSD 1.0 para abordar las dificultades de mantener 386/BSD a través de parches y trabajar con su autor para asegurar el futuro de 386/BSD [114]. El enfoque del proyecto era soportar la arquitectura de la PC, atrayendo a una audiencia grande, no necesariamente técnicamente muy sofisticada [64, p. 11]. Por razones legales asociadas con la resolución del caso USL, mientras que las versiones de FreeBSD hasta 1.1.5.1 se derivaron de BSD Networking 2 Release, las posteriores se derivaron de 4.4BSD-Lite Release 2 con adiciones 386/BSD.

4.19 FreeBSD 1.1

Administrador de paquetes La instalación de puertos de software S80 proporciona un mecanismo para compilar e instalar paquetes de terceros y sus dependencias. Se documentó por primera vez (puertos (7)) en FreeBSD 2.2.6 y está disponible y creciendo en los sistemas FreeBSD modernos. Maneja las modificaciones (parches) requeridas para hacer que un paquete de software funcione bajo FreeBSD, la instalación de las dependencias requeridas y la instalación y desinstalación del paquete correspondiente. El acoplamiento flexible de los paquetes al sistema operativo y el manejo automático de las dependencias permiten que el sistema FreeBSD crezca en funcionalidad en diversas direcciones sin sobrecargar excesivamente su núcleo.

4.20 Sistema de archivos

de proceso de FreeBSD 2.0 El sistema de archivos /proc—procfs(5)—proporciona una vista de dos niveles de los procesos en ejecución en forma de archivos que aparecen en la jerarquía del sistema de archivos [115]. fue originalmente introducido en una forma diferente en 4.4BSD/Lite1.S81 En su parte superior hay una lista de directorios correspondientes a los procesos en ejecución. Cada directorio de proceso contiene archivos que permiten monitorear y controlar el estado y estado del proceso, como sus registros de CPU, memoria y uso de recursos. La importancia arquitectónica del directorio de procesos es que proporciona una interfaz alternativa a la funcionalidad que normalmente se proporciona a través de llamadas al sistema como ptrace(2) y (para aplicaciones dentro de un contexto de proceso) getrlimit(2) y getrusage(2).

Módulos del kernel cargables dinámicamente La función del módulo del kernel cargable, lkm(4), permite la carga y descarga dinámicas del código del kernel en tiempo de ejecución. Se reemplazó en FreeBSD 3.0.0 con la función similar del enlazador dinámico del kernel (kldload(8), kldstat(8), kldunload(8)—para admitir el enlace dinámico del código del kernel en el momento del arranque [116, pp. 636– 637]. Los módulos de kernel cargables permiten la provisión de una funcionalidad significativa para el kernel, como controladores de dispositivos, sistemas de archivos, emuladores y llamadas al sistema. Esto reduce la huella de memoria predeterminada del núcleo y la superficie de ataque. La versión reciente (11.1) de FreeBSD proporciona un kernel cargable 992L45
módulos.

4.21 FreeBSD 2.1

Emulación de Linux Aunque el kernel de Linux se desarrolló independientemente de los sistemas Unix examinados aquí, sigue la API de llamada del sistema Unix hasta su esquema de numeración.
Sin embargo, FreeBSD no admite directamente algunas de sus llamadas al sistema, mientras que otras tienen diferencias sutiles en la especificación de su interfaz. Además, su formato de archivo ejecutable difiere del de FreeBSD. Un conjunto de archivos kernel S82 permite que FreeBSD cargue y ejecute programas ejecutables compilados para el sistema operativo GNU/Linux. Esto se logra marcando adecuadamente el proceso correspondiente para emular el comportamiento de las llamadas al sistema específicas de Linux.

Biblioteca de captura de paquetes La captura y el control eficientes de los paquetes de red es una función de diagnóstico importante. La librería de captura de paquetes pcapS83—documentada en FreeBSD 8.0 como pcap(3)—junto con el programa tcpdump(1) permiten capturar la especificación de paquetes, la compilación del filtro correspondiente en un programa de máquina virtual que puede ser dinámicamente inyectado para su ejecución en el kernel del sistema operativo, y la recuperación y análisis de los paquetes capturados.

Desarrollada por un grupo independiente, la biblioteca abstrae diversos mecanismos de captura de paquetes en una interfaz portátil.

4.22 Subsistema de E/S

del método de acceso común de FreeBSD 3.0 El subsistema de E/S del método de acceso común (CAM) abstrae las operaciones a los dispositivos de almacenamiento basándose en un estándar ANSI (borrador). Comenzó admitiendo discos SCSI y CD-ROM, pero con el lanzamiento de FreeBSD 9.0 evolucionó para cubrir también las unidades de disco ATA y SATA de uso común [65, Sección 8.1]. Sus tres capas comprenden (desde el núcleo hasta el dispositivo) el acceso periférico específico del dispositivo (p. ej., unidad SATA), la programación y envío de comandos de E/S y el enrutamiento de comandos a dispositivos a través del adaptador de bus host.

4.23 FreeBSD 3.4 Biblioteca

de usuarios y redes de kernel basadas en gráficos El subsistema netgraph(4) permite la implementación de protocolos de trabajo en red sofisticados siguiendo un modelo de flujo de datos. Diverso

Los nodos de procesamiento de paquetes de red están conectados a través de funciones de enlace en una estructura de datos de gráficos por medio de una interfaz orientada a objetos. Los nodos de Netgraph pueden implementar protocolos, como el protocolo punto a punto (PPP, ng_ppp(8)) o proporcionar funciones de utilidad, como el filtro de paquetes de Berkeley (BPF, ng_bpf(8). El subsistema de netgraph S84 de FreeBSD 11.1 contiene archivos 177L46 que ofrecen funcionalidad de netgraph a través de 90,471L47 líneas de código.

4.24 FreeBSD 4.0

OpenSSL Framework OpenSSL (capa de sockets seguros (SSL) y capa de seguridad de transporte (TLS) frameworkS85) proporciona dos bibliotecas C y un comando de usuario, openssl, que permiten a los programas y usuarios trabajar con estos protocolos. Además, los elementos del marco exponen una variedad de algoritmos de encriptación de claves públicas y simétricas, funciones de resumen de mensajes y operaciones de manejo de certificados. El tamaño del framework es considerable, comprende 1,127L48 archivos y 227,118L49 líneas de código. Desde una perspectiva de arquitectura, la incorporación del marco es notable debido a su tamaño, su método de desarrollo (fue implementado por un equipo separado) y el hecho de que incorpora su propio método de interfaz de comando, a saber, la provisión de 22L50 sub - com comandos accesibles desde el comando openssl(1).

Jails La llamada al sistema y el comando jail (2,8) permiten al administrador del sistema aislar un conjunto de procesos en un entorno confinado, restringiendo las operaciones que los procesos pueden realizar [117]. Esto amplía la llamada al sistema chroot(2), que puede ofrecer a un proceso una vista restringida del espacio de nombres del sistema de archivos, para cubrir la virtualización de las redes, la comunicación entre procesos y el montaje del sistema de archivos. Por lo tanto, las cárceles permiten a los administradores ejecutar procesos con requisitos complejos o frágiles en entornos de contenedores virtualizados independientes, como los proporcionados por Docker [118]. Las cárceles también permiten que los proveedores de servicios en la nube alojen muchos clientes con acceso administrativo completo a su host (virtual) en el mismo servidor. Dichos clientes no pueden ejecutar su propio sistema operativo, como podrían hacerlo con un hipervisor completo, pero el servicio es muy eficiente en términos de utilización de recursos. En términos de arquitectura, las cárceles proporcionan un nivel ligero adicional de virtualización además del sistema operativo.

Listas de control de acceso Una biblioteca—acl(3)—proporciona una interfaz para extender el modelo tradicional de control de acceso discrecional de usuarios/grupos/todas las lecturas/escrituras/ejecuciones de Unix con listas de control de acceso (ACLS). Las versiones posteriores agregan soporte para ACL en el sistema de archivos UFS S86 y para los permisos más detallados de NFSv4.S87. En la versión actual de FreeBSD 11.1, ACLS permite la especificación de más de una docena de permisos para un número arbitrario de principales (usuarios o grupos).

4.25 FreeBSD 5.0

Multiprocesamiento simétrico El código del núcleo puede ejecutarse en múltiples procesadores o núcleos de CPU al sincronizar el acceso a los recursos compartidos a través de un conjunto ordenado jerárquicamente de primitivos de bloqueo [64, p. 93]. Una gran parte de este gran cambio implica la adición de 3764L51 llamadas de sincronización de subprocesos basadas en mutex, que existen en el 7,3L52 por ciento de los 4873L53 archivos de código fuente del kernel.

Marco de transformación de solicitudes de E/S de disco modular El marco de transformación de solicitudes de E/S de disco modular GEOM (geom(3, 4, 8)) permite que las solicitudes del subsistema de almacenamiento sean transformado para admitir la partición del disco, la agregación, el cifrado, el registro en diario y la recopilación de estadísticas de E/S. Está diseñado en torno a un esquema en el que cada funcionalidad (por ejemplo, creación de bandas) se implementa en una clase separada. Las instancias de objetos con interfaces de proveedor y consumidor están conectadas en un gráfico acíclico dirigido, que forma las capas de transformación.

Control de acceso obligatorio El marco de control de acceso obligatorio, mac(4), admite un control detallado de las políticas de seguridad de un sistema a través de diversos módulos de políticas conectables. Los ejemplos de políticas admitidas incluyen seguridad multinivel [119], marca de agua baja, Biba [120] y partición de procesos. El kernel asocia etiquetas independientes de políticas con objetos del sistema de archivos, interfaces de red, terminales y usuarios.
Esto luego permite que los subsistemas del kernel relevantes (sistema de archivos, red, IPS, gestión de procesos, VM) obtengan permisos de control de acceso del marco e informen sobre los eventos del ciclo de vida de los objetos [65, Sección 5.10].

Módulo de autenticación conectable La arquitectura del módulo de autenticación conectable (PAM) proporciona una forma de implementar y abstraer diversos métodos de autenticación de usuario de bajo nivel, al tiempo que presenta programas cliente con una única API de alto nivel: pam(3). Además de la autenticación, la biblioteca admite la administración de cuentas, sesiones y contraseñas. Adaptar el sistema de autenticación de Unix con PAM simplifica la introducción de métodos de autenticación sofisticados, como los que utilizan contraseñas de un solo uso y acceso a directorios, mediante la instalación de los módulos correspondientes.

4.26 FreeBSD 5.3

Streaming Archive Access Library Más de una docena de formas de empaquetar varios archivos en uno solo se han generalizado durante el último medio siglo. Por lo general, cada formato está asociado con los programas de empaquetado y compresión correspondientes, como ar(1), tar(1), cpio(1), gzip(1), compress(1) o bzip2(1). La biblioteca de acceso archive(3) consolida estos formatos dispares. Permite que los programas que lo utilizan lean y escriban los formatos de archivo más comunes, interactuando entre los archivos de un archivo y los que residen en un sistema de archivos.

Envoltorio de controlador de minipuerto Una instalación del núcleo y un programa de aplicación permiten el uso de controladores de dispositivos de hardware de interfaz de red que se ajustan a la API de minipuerto de especificación de interfaz de controlador de red (NDIS) de Microsoft Windows para ser utilizados en FreeBSD. Por lo tanto, los componentes binarios (código compilado) desarrollados para un sistema operativo radicalmente diferente pueden convertirse en controladores de dispositivos FreeBSD. Este mecanismo reduce la dificultad de construir u obtener controladores de dispositivos específicos de FreeBSD para interfaces de red.

4.27 FreeBSD 6.2

Auditoría básica del módulo de seguridad La función de Auditoría básica del módulo de seguridad (BSM) comprende cambios en el kernel, llamadas al sistema: audit(2), una biblioteca libbsm(3), archivos de configuración, por ejemplo, audit_control(5), un formato de archivo binario: audit.log (5) y programas de soporte, praudit(1), auditreduce(1), audit(8), auditd(8), para generar y procesar flujos de registros que se requieren para la auditoría de seguridad. Los eventos auditados incluyen tanto eventos a nivel de kernel, como sistemas de archivos o accesos a la red, como eventos a nivel de aplicación, como la autenticación de un usuario [65, Sección 5.11].

4.28 FreeBSD 7.0

Sistema de archivos Zettabyte El sistema de archivos Zettabyte (ZFS) es una evolución del sistema de archivos con estructura de registro 4.4BSD derivado del código base OpenSolaris de Sun. Se basa en el concepto de puntos de control, que permiten que el sistema de archivos se mueva de un estado consistente a otro [48]. Además, al utilizar la disponibilidad de abundantes recursos de memoria y potencia de procesamiento en los servidores modernos, garantiza la integridad de los datos a través de sumas de verificación de extremo a extremo, ofrece varios niveles de software RAID y proporciona escalabilidad masiva (del tamaño de zettabytes) a través de (potencialmente híbrido) agrupación de dispositivos [65, Capítulo 10]. El código del sistema de archivos está organizado en torno a una arquitectura en capas de tamaño considerable, comenzando con 80 107 L54 líneas en FreeBSD 7.0 y creciendo hasta 205 899 L55 líneas en FreeBSD 11.1.

4.29 FreeBSD 7.1

Seguimiento dinámico La función DTrace, traída de Solaris de Sun, se basa en el patrón arquitectónico de reflexión [71, p. 193] para permitir el seguimiento del funcionamiento del sistema a través de miles de sondas. Las sondas se pueden configurar y monitorear a través de programas escritos en el lenguaje específico del dominio D [49]. El comando dtrace(1) ejecuta estos programas para habilitar las sondas especificadas e informar los detalles recopilados. DTrace tiene dos ventajas importantes sobre los enfoques alternativos, como el seguimiento de llamadas del sistema, las estadísticas del contador del kernel o la creación de perfiles. En primer lugar, puede monitorear toda la pila de aplicaciones, incluidos los límites de las funciones, la creación de redes, la programación, los sistemas de archivos, las llamadas al sistema y el código de la aplicación. En segundo lugar, al instalar solo las sondas requeridas a través de parches de código dinámico, su impacto en el rendimiento de los sistemas de producción es insignificante cuando no se recopilan datos.

4.30 FreeBSD 9.0

Un conjunto de dispositivos de E/S paravirtualizados que se ajustan a la especificación VirtIO (virtio(4)) permiten una E/S eficiente en los casos en que FreeBSD se ejecuta sobre un hipervisor. Las interfaces de red, almacenamiento y memoria provistas pueden eliminar el costo de emular hardware heredado y de copiar memoria entre el hipervisor y el sistema operativo huésped.

Compatibilidad con InfiniBand InfiniBand es un estándar de comunicaciones de redes informáticas que ofrece alta velocidad (10–300 Gb/s) y baja latencia (140–2600 ns). Estas decisiones de diseño lo hacen atractivo en aplicaciones informáticas de alto rendimiento, así como en otras áreas que requieren interconexiones rápidas entre computadoras o entre computadoras y sistemas de almacenamiento. La tecnología es compleja y exigente. Por lo tanto, un grupo llamado OpenFabrics Alliance está desarrollando una pila InfiniBand multiplataforma para diversos sistemas operativos y distribuyéndola como software de código abierto. El soporte InfiniBand de FreeBSD incorpora este gran código base (325,818L56 líneas). Compartimentación de aplicaciones S88 La capacidad del sistema operativo capsicum(4) y el marco sandbox permiten que las aplicaciones y bibliotecas se compartimenten en componentes aislados para reducir el impacto de las capacidades de vulnerabilidad de seguridad. Funciona permitiendo que las aplicaciones entren en un modo de capacidad reducida y ofreciendo una API de llamada al sistema para restringir el acceso de una aplicación a espacios de nombres globales, como el sistema de archivos. Por ejemplo, la parte de procesamiento de una aplicación potencialmente vulnerable puede recibir solo el derecho de escribir en un archivo abierto previamente por otra parte de la aplicación que ha conservado la autoridad ambiental.

4.31 FreeBSD 10.0

Monitor de máquina virtual El monitor de máquina virtual bhyve(4,8) permite que un sistema FreeBSD aloje instancias de otros sistemas operativos no modificados que se ejecutan sobre él. Los sistemas operativos compatibles incluyen FreeBSD, NetBSD, OpenBSD, GNU/Linux, Windows y SmartOS. Con 30 391 L57 líneas de código (aumentando a 62 906 L58 líneas en FreeBSD 11.1), es un esfuerzo de implementación modesto, que depende en gran medida del soporte de virtualización que ofrecen las CPU modernas y el hardware de soporte.

Procesamiento rápido de paquetes sin procesar en el espacio del usuario El marco netmap(4) [121] proporciona una API a través de la cual las aplicaciones del espacio del usuario pueden acceder e inyectar paquetes asociados con las interfaces de red, la pila de red del sistema o el conmutador virtual vale(4). A través del acceso sincronizado directo a los búferes de anillo correspondientes del kernel, las aplicaciones evitan la sobrecarga de la copia de datos y, por lo tanto, pueden procesar millones de paquetes por segundo. Esto permite que los sistemas FreeBSD implementen dispositivos de red como enrutadores, conmutadores y cortafuegos [6

4.32 Lista negra de red

de FreeBSD 11.0 El demonio blacklistd(8) escucha notificaciones de otros demonios de red sobre intentos de conexión exitosos o fallidos. El archivo de configuración blacklistd.conf(5) especifica las condiciones bajo las cuales el demonio de la lista negra configurará el filtro de paquetes del sistema para bloquear las conexiones asociadas con los puertos en los que se ha detectado un comportamiento abusivo. La biblioteca libblacklist(3) implementa el protocolo para la comunicación entre los demonios.

5 RESULTADOS CUANTITATIVOS

Desde 1970 hasta hoy, el código fuente del sistema creció en tres órdenes de magnitud, de 13 mil a más de diez millones de líneas de código. ¿Se refleja esta tasa de crecimiento en términos del número de características? ¿Qué tipos de características son responsables del crecimiento principal y cuál es su tasa de crecimiento? ¿Cuáles son los valores atípicos y cómo pueden explicarse potencialmente? ¿El crecimiento del tamaño va acompañado de un crecimiento de la complejidad del código? Para responder a estas preguntas, como se mencionó anteriormente en la Sección 3.3, utilizamos la documentación de referencia del sistema, así como el análisis del código fuente.

5.1 Crecimiento de

características Analizamos la documentación de referencia del sistema en lugar de otras categorías de características (por ejemplo, las particiones del sistema como se muestra en la Fig. 5 o 6), porque su estructura se ha mantenido esencialmente sin cambios. Específicamente, a lo largo de su vida, la documentación de referencia de Unix se divide en nueve secciones. En este estudio ignoramos dos: la Sección 6, que ha evolucionado para documentar algunos juegos, y la Sección 7, que documenta elementos misceláneos que van desde el conjunto de caracteres ASCII hasta direcciones de correo electrónico, macros de formato y la jerarquía de directorios del sistema. Las siete secciones restantes se enumeran en las dos primeras columnas de la Tabla 4. La evolución en el número de sus características se

Como podemos ver en la figura correspondiente, durante el último medio siglo el sistema Unix creció en una proporción similar en el número de todos los tipos de características. Algunos valores atípicos pueden explicarse por restricciones o elecciones asociadas con versiones particulares. Por ejemplo, la disminución en el número de comandos en 386BSD probablemente se deba al hecho de que esta versión

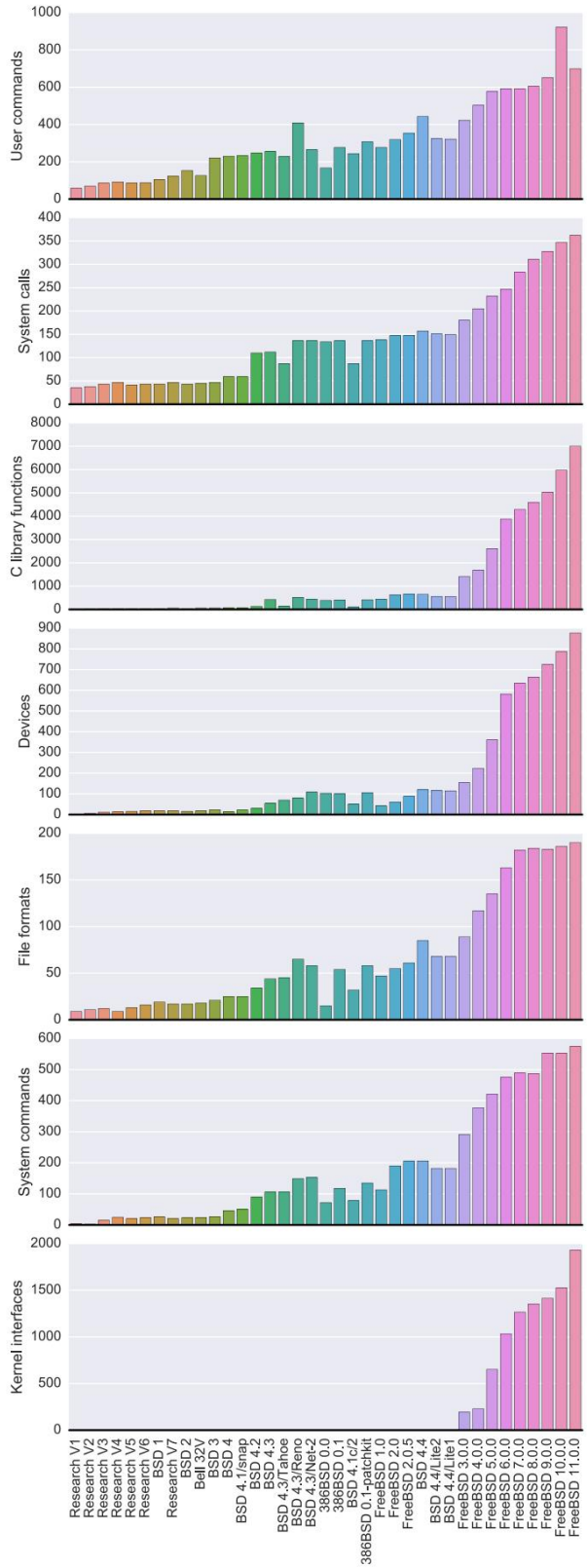

Fig. 3. Evolución en la cantidad de tipos de funciones en los lanzamientos clave.

calzó un sistema que se distribuía a través de cintas para ejecutarse en minicomputadoras VAX en uno que se distribuía a través de disquetes para ejecutarse en PCS. Durante el mismo período, el sistema se lanzó como software de código abierto, lo que resultó en lanzamientos que eliminaron elementos que contenían código propietario.
Estos fueron luego gradualmente reimplementados o reemplazados por alternativas de código abierto. El alto número temporal de comandos de usuario en 4.3BSD Reno se deriva de la inclusión de diversos programas aportados por los usuarios, como Emacs, USENET News y X-Window System. Estos fueron distribuidos más tarde como paquetes separados.

Al centrar nuestra atención en tipos de características específicas, vemos que el crecimiento no ha sido uniforme en todas ellas; hay evidencia de tendencias interesantes por las cuales podemos hipotetizar razones específicas. El crecimiento de los comandos de administración de usuarios y sistemas, así como de los formatos de archivo, ha sido relativamente uniforme. Esto es de esperar, si consideramos un sistema operativo como una plataforma que aloja (un conjunto en expansión) de programas y archivos.

La evolución en el número de llamadas al sistema cuenta una historia más interesante. Hay dos períodos de relativa estabilidad.
Uno sobre las ediciones de Research Unix, lo cual se puede entender si se considera que sus desarrolladores se enorgullecieron de demostrar "que un sistema operativo poderoso para uso interactivo no tiene por qué ser costoso ni en equipo ni en esfuerzo humano" [36]. En consecuencia, evitaron inflar el núcleo con una funcionalidad de utilidad marginal. Se puede ver un aumento posterior en el número de llamadas al sistema seguidas de estabilidad en los lanzamientos de Berkeley. El aumento se puede atribuir a la investigación dirigida a áreas específicas: redes, sistemas de archivos y uso interactivo. La estabilidad posterior marca una fase de consolidación en la que las interfaces desarrolladas son utilizadas por un número cada vez mayor de comandos de administración de usuarios y sistemas. El aumento continuo en el número de llamadas al sistema sobre las versiones de FreeBSD se puede atribuir a una comunidad interesada en la innovación del sistema operativo y, tal vez, una en la que un gran número de desarrolladores voluntarios están ansiosos por dejar su huella en el kernel.

La evolución de las funciones de la biblioteca C cuenta una historia similar. Una timidez restringida sobre las ediciones de Research Unix dio como resultado un conjunto básico de funciones, la mayoría de las cuales se estandarizaron más tarde como la biblioteca del lenguaje de programación C. Los lanzamientos de Berkeley rompieron esa tradición al introducir muchas funciones nuevas para acomodar la funcionalidad recientemente proporcionada. La actitud de ese período parece ser que si alguna funcionalidad es generalmente útil, entonces debería estar disponible como biblioteca. Esto estableció una tradición para proporcionar interfaces de biblioteca para acceder a los archivos del sistema y empaquetar en bibliotecas funcionalidades complejas, como coincidencia de expresiones regulares y compatibilidad con bases de datos integradas. Habiendo roto el tabú de limitar la biblioteca C a un conjunto básico de funciones portátiles, el aumento de la funcionalidad provista continuó con las versiones de FreeBSD, lo que resultó en una cantidad sustancialmente mayor de funciones de biblioteca. Los marcos modernos, como .NET, Jakarta EE y Python, han seguido este ejemplo al brindar un amplio soporte para diversas funcio

Los cambios en la cantidad de dispositivos admitidos probablemente se debieron a factores externos, a saber, la disponibilidad de dichos dispositivos, la demanda para usarlos y los recursos para implementar su código de controlador. La caída en el número de dispositivos de 386BSD a FreeBSD 1.0 proviene de la limpieza de los controladores de dispositivos obsoletos que no funcionan: desde la interfaz DARPA IMP del host local/ distante acc(4) hasta el multiplexor de comunicaciones de acceso telefónico vx(4).

La documentación para el kernel APIS (Manual de referencia de Unix, sección 9) se introdujo recién a fines de la década de 1990, por lo que hay menos que observar en la figura correspondiente.
El aumento inicial probablemente se deba a un vigoroso esfuerzo por documentar las interfaces existentes, mientras que el crecimiento posterior puede haber sido orgánico.

TABLA 4
Número de funciones documentadas en los sistemas operativos actuales

Section	Description	FreeBSD	macOS	OpenBSD	Solaris	Ubuntu	Windows
1	User commands	700[L59]	1,352 [L60]	404[L61]	2,169[L62]	1,916 [L63]	N/A
2	System calls	370[L64]	252 [L65]	270[L66]	236[L67]	462 [L68]	} 6,001[L69]
3	C library functions	7,280[L70]	10,186 [L71]	5,094[L72]	7,693[L73]	3,726 [L74]	
4	Supported devices	907[L75]	46 [L76]	964[L77]	402[L78]	4,693 [L79]	N/A
5	File formats	191[L80]	192 [L81]	124[L82]	245[L83]	234 [L84]	N/A
8	Administration commands	579[L85]	661 [L86]	373[L87]	894[L88]	735 [L89]	461[L90]
9	Kernel interfaces	1,936[L91]	N/A	928[L92]	1,534[L93]	7,434 [L94]	N/A

Para juzgar en contexto la evolución de las funciones admitidas, las columnas restantes de la Tabla 4 enumeran el número de tipos de funciones documentadas en diversos sistemas operativos actuales: FreeBSD 11.1.0, Apple macOS 10.13.3, OpenBSD 6.3, Oracle Solaris 11.3, Ubuntu Linux 16.04.5 LTS y Microsoft Windows 10 (compilación 16.299). Los números se obtuvieron de la siguiente manera: para FreeBSD y OpenBSD al procesar el código fuente y Makefiles; L95 para Solaris al procesar los índices de la biblioteca de referencia en línea de Oracle ; 8, L96, L97 para Windows al procesar el código fuente de Windows Server documentación9 y el marcado HTML del índice de la biblioteca general UAP de Windows;10, L98 para Ubuntu y macOS contando el número de archivos de página del manual o procesando el código fuente del kernel en servidores ofrecidos por la plataforma de integración continua Travis CI11, L99 a través de un pequeño proyecto construido para este propósito.12 Para mantener las cifras comparables, tratamos de proporcionar números que reflejen las instalaciones del servidor en lugar de las de escritorio.

Como se desprende de la tabla, la cantidad de tipos de características es similar en magnitud entre sistemas con diferentes historias, arquitecturas o caminos evolutivos. Cuando existen marcadas diferencias, estas pueden explicarse fácilmente. Por ejemplo, en el caso de los controladores de dispositivos, las diferencias se derivan del uso de hardware estandarizado (macOS) o de la adopción generalizada (Ubuntu Linux). Además, debido a que la API de Windows no distingue claramente entre las interfaces del kernel y las funciones de utilidad a nivel de usuario, los puntos de entrada de su API aparecen en la tabla que abarca las filas para las llamadas al sistema y para las funciones de la biblioteca C. Ninguno de los sistemas exhibe la economía evidente en, digamos, la Séptima Edición de Unix de 1979. Interpretamos esto como una señal de que los requisitos de un sistema operativo moderno impulsan la complejidad esencial correspondiente (y, a veces, la complejidad accidental). El aumento cuantitativo observado en los tipos de características admitidos no es una coincidencia, sino una respuesta a las presiones ambientales.

5.2 Complejidad ciclomática

También analizamos la evolución de la complejidad ciclomática de las dos particiones principales del sistema: el código del espacio del kernel (archivos de código fuente C, aquellos con un sufijo .c, que actualmente residen en el directorio sys) y el código del espacio del usuario. Para este último, distinguimos además entre las bibliotecas compartidas entre varios programas (archivos C en lib) y los comandos de usuario, administrador y sistema (todos los demás archivos C). El motivo de esta distinción es que las bibliotecas son reutilizadas por otros programas y, por lo tanto, deben ser más fáciles de mantener (tener una menor complejidad).

La Fig. 4 muestra las tendencias de evolución de la complejidad ciclomática a lo largo del tiempo para los tres tipos mencionados. En términos generales, esto sigue a un fuerte aumento seguido de una disminución gr Una posible explicación del aumento podría ser que la tecnología mejorada (por ejemplo, terminales de vidrio de 9600 baudios que reemplazan a los teletipos de 110 baudios) podría permitir la adopción de estructuras de programas más complejas [15]. La inclinación de la curva podría explicarse por la rápida introducción de estas tecnologías, que disfrutan de los beneficios de crecimiento exponencial asociados con la Ley de Moore [122]. La caída gradual podría atribuirse correspondientemente a las correcciones que abordan la complejidad excesiva, implementadas agregando un código nuevo mejor o refactorizando el código existente. La razón detrás de tales cambios podría ser satisfacer los requisitos de calidad implícitos asociados con la construcción de un artefacto de software grande y sofisticado [123]. Esta hipótesis se ve corroborada por el hecho de que la complejidad ciclomática de las tres áreas obedece a su criticidad e importancia relativas. Es menor para el kernel donde una falla puede derribar un sistema completo, así como para las bibliotecas donde un problema puede afectar a muchos programas. De hecho, las curvas para el núcleo y las bibliotecas son sorprendentemente similares, especialmente después de mediados de la década de 1990. Por el contrario, es más alto para los comandos de usuario, administrador y sistema donde el código está aislado en procesos separados y donde los problemas normalmente afectan solo a un comando. Sin embargo, en los tres casos, la complejidad ciclomática media al final del período estudiado es de alrededor de 6, lo que se considera en general bastante bajo [124, pp. 342–344] para un sistema tan complejo y de larga duración.

Un factor habilitador para luchar contra la complejidad ciclomática pueden ser los avances en las velocidades de reloj de la CPU y en la tecnología del compilador, como el análisis entre procedimientos ofrecido por GCC y posteriormente LLVM [125]. Estos han permitido a los desarrolladores que metían el código en una sola función, para evitar la penalización del rendimiento de las llamadas a funciones, escribir funciones más pequeñas y modulares.

Para poner en perspectiva la evolución de la complejidad ciclomática, la parte inferior de la figura 4 ilustra la correspondiente evolución de la complejidad de las coreutils de GNU, la biblioteca C de GNU y el kernel de Linux, yuxtapuestas con las de los comandos, la biblioteca y el kernel de Unix. núcleo respectivamente. Observamos que los períodos medidos no son idénticos, ya que la parte superior comienza a mediados de los años 70, mientras que la parte inferior comienza a partir de 1995 o un poco antes; por lo tanto, omitimos las dos primeras décadas de Unix en nuestra comparación. La semejanza en cada

8. https://docs.oracle.com/cd/E53394_01/
9. https://github.com/MicrosoftDocs/windowsserverdocs/
10. https://docs.microsoft.com/en-gb/windows/desktop/apiindex/windows-umbrella-libraries _11. https://travis-ci.org/dspinellis/documented-facilities/builds/459375741
12. https://github.com/dspinellis/documented-facilities

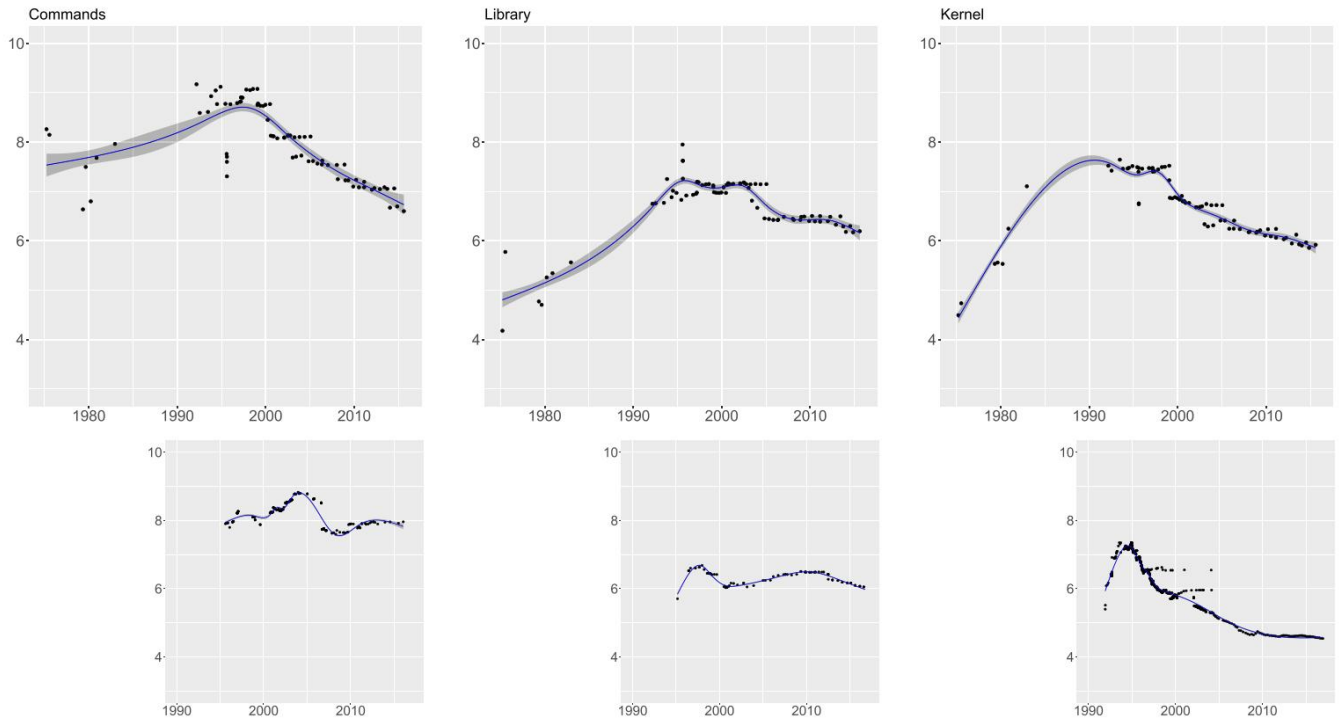

Fig. 4. Complejidad ciclomática media del código a lo largo del tiempo. Arriba: para los comandos de espacio de usuario, bibliotecas C y kernel de los sistemas Unix de este estudio. Abajo: para los comandos de espacio de usuario GNU coreutils, la biblioteca GNU C y el kernel de Linux.

par de curvas es llamativa: se observa la misma inclinación inicial y posterior descenso.

Conjeturamos que la curva en U invertida en el caso de GNU/Linux es causada por razones similares a las de Unix: la constante mejora de las capacidades del hardware a lo largo de los años 80 y 90 condujo a la inclinación, seguida de acciones correctivas para mejorar la calidad, a medida que la complejidad comenzaba a volverse abrumador. Parece que la comunidad GNU/Linux exhibe una madurez similar a la de FreeBSD [123], esforzándose por la calidad del código mediante la reelaboración y refactorización del código. La complejidad ciclomática real también fluctúa alrededor de las mismas cifras: 7 a 9 para los comandos, 6 a 7 para las bibliotecas (después de 1995) y 4,5 a 7,5 para el kernel.

Sin embargo, también hay algunas diferencias pronunciadas. Mientras que la complejidad de los comandos de Unix se redujo gradualmente hasta el final de nuestras mediciones, alcanzando 6,5, los comandos del espacio de usuario de GNU se estabilizaron después de 2010 en aprox. 8. La razón detrás de esto puede ser una menor estabilidad y requisitos de mantenimiento con respecto a los comandos individuales en comparación con el kernel monolítico. Además, el pico de las dos curvas difiere en aproximadamente una década, lo que indica que la comunidad GNU/Linux comenzó a incorporar pautas y prácticas de mejora de la calidad más tarde que la comunidad Unix. Además, la biblioteca GNU C tuvo un segundo período de creciente complejidad, aunque mucho más moderado.

Esto puede indicar nuevamente una progresiva falta de atención con respecto a la calidad del diseño a medida que se olvidaron las lecciones del impulso anterior, o ser un efecto secundario del esfuerzo por adaptarse a una nueva versión de C (C11). Finalmente, con respecto a la complejidad del kernel, si bien alcanzó su clímax a mediados de los 90 en ambos casos, la complejidad del kernel de Linux mejoró a un ritmo más rápido, cayendo incluso más bajo que su punto de partida. Esto indica un fuerte impulso en la comunidad de Linux para refactorizar y eliminar la deuda técnica, probablemente liderado por miembros clave del equipo de desarrollo del kernel.

6 HACIA UNA TEORÍA INICIAL DEL OPERAR EVOLUCIÓN DE LA ARQUITECTURA DEL SISTEMA

Nuestros hallazgos del análisis cualitativo y cuantitativo son interesantes no solo para el caso de Unix, sino también para sistemas operativos similares. Así, pueden formar la base para establecer una teoría inicial sobre cómo evoluciona la arquitectura de los sistemas operativos. La construcción de teorías en Ingeniería del Software se ha argumentado, entre otros, como un medio necesario para generalizar analíticamente los resultados, yendo más allá de los hallazgos individuales [126]. Para construir esta teoría, seguimos los primeros cuatro pasos, según lo prescrito por Sjøberg et al. [127]. Así derivamos: constructos, que son las entidades principales de la teoría; proposiciones, que establecen relaciones entre los constructos; explicaciones, que arrojan más luz sobre las proposiciones; y el alcance que determina dónde se aplica la teoría. El quinto paso, que implica probar la teoría a través de más estudios empíricos, se considera trabajo futuro.

Las construcciones en nuestro caso incluyen los conceptos principales de las preguntas de investigación, es decir, decisiones de arquitectura, evolución, vida útil del sistema, características, tamaño y complejidad. También se extienden a deuda técnica, convenciones, portabilidad, ecosistemas de software y sistemas de terceros. Este conjunto de construcciones se basa en los datos recopilados, como se describe en la Sección 3.3, y comprende los conceptos principales que se derivaron durante el análisis de datos (en particular, Comparación constante; consulte la Sección 3.4). En consecuencia, el alcance incluye sistemas operativos grandes, complejos y de larga duración.

Las proposiciones y explicaciones derivadas se elaboran en los siguientes subapartados, agrupados en los referentes a: a) la forma y el ritmo de la evolución arquitectónica, b) la acumulación de la deuda técnica arquitectónica, y c) las fuerzas de la evolución arquitectónica. Cada proposición se formula como una oración (en cursiva) y se elabora brevemente, seguida de un párrafo con la explicación.

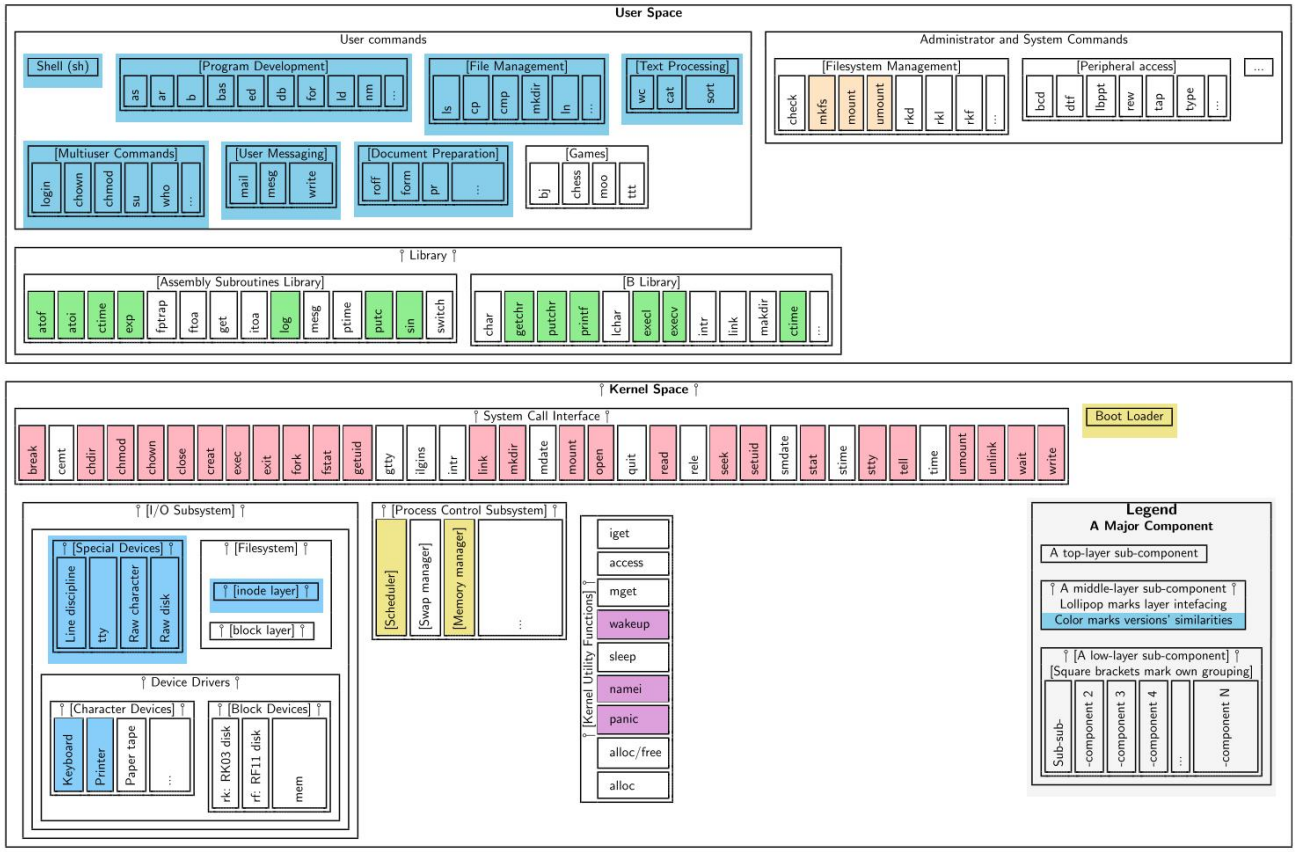

Fig. 5. Arquitectura de alto nivel de la primera edición de investigación (1972).

6.1 Forma y ritmo de evolución arquitectónica

Proposición 1. Muchas decisiones de arquitectura central se toman en el comienzo de la vida útil del sistema.

Un hallazgo sorprendente de nuestro estudio fue la gran cantidad de decisiones de diseño definitorias de Unix que se implementaron desde el principio. Esto se puede ver claramente en la línea de tiempo de la evolución (Fig. 1). A pesar del tamaño diminuto del PDP-7 y las ediciones First Research, incluían las llamadas al sistema más importantes que todavía se usan en la actualidad, la noción de dispositivos como archivos, la abstracción de E/S estándar y una estructura de directorios en árbol.

La influencia de las primeras decisiones arquitectónicas también es evidente si se compara la arquitectura de alto nivel (vista de módulo) del diagrama arquitectónico de la Primera Edición (Fig. 5) con la arquitectura actual del sistema (Fig. 6) . la descomposición del nivel se ha mantenido esencialmente igual. La permanencia de muchas de las primeras decisiones de diseño se ilustra a través de elementos resaltados en los dos diagramas.

Tenga en cuenta que, dado que el diagrama de la arquitectura actual se dibuja a una escala mucho más gruesa, muchas de las características de la Primera Edición aparecen en la arquitectura actual agrupadas bajo una entidad con el mismo color. Por ejemplo, el cuadro de llamada del sistema de E/S de archivo en la figura 6 incluye las llamadas al sistema de abrir, leer, escribir y cerrar representadas individualmente en la figura 5, mientras que las partes matemáticas, stdio, stdlib y tiempo de la biblioteca C Standard en La Fig. 6 contiene, entre otras, las funciones de biblioteca coloreadas de la Fig. 5.

13. Alentamos a los lectores a que se concentren en la estructura general, porque muchos pueden encontrar las etiquetas de texto ilegibles debido a su pequeño tamaño de fuente. Se pueden obtener más detalles fácilmente haciendo zoom en la versión digital del manuscrito.

Explicación Los desarrolladores de los primeros Unix buscaron “destilar y simplificar” [59] tres poderosos e influyentes sistemas operativos: Multics, Project Genie y CTSS [76], algunos de los cuales ya habían sufrido el “síndrome del segundo sistema” [128], [79, pág. 463]. En consecuencia, la experiencia de los desarrolladores de Unix los guió para implementar el sistema en torno a algunas ideas clave con un valor perdurable.

Proposición 2. Las decisiones arquitectónicas más importantes sobreviven durante la vida útil del sistema.

La cantidad de decisiones de diseño arquitectónico de larga duración en Unix es impresionante. De los 15.596L100 elementos documentados durante el último medio siglo, 12.043L101 (más del 75 por ciento)
todavía están documentados en la edición actual de FreeBSD. La mayoría de los comandos en desuso ofrecen una funcionalidad que hoy en día está disponible a través de paquetes complementarios (factorización de números, generación de formularios, síntesis de voz, guiones, compilación Fortran) o tratan con tecnología obsoleta (comunicación GCOS y UUCP, manejo de DECtape). Por el lado de las llamadas al sistema, las pocas eliminadas son principalmente aquellas que han sido reemplazadas por mecanismos más generales. Por ejemplo, la funcionalidad de las llamadas de manejo de señales de la Tercera Edición—cemt(II), fpe(II), ilgins(II) e intr(II)—es provista hoy en día por la única llamada sigaction(2). Por el contrario, los controladores de dispositivos han visto una tasa de abandono muy alta. Esto es de esperar debido a los grandes y visibles cambios en las tecnologías de los dispositivos de hardware; nadie hoy en día utiliza lectores de tarjetas perforadas, perforadoras de cintas de papel, datáfonos o unidades de disco RA80 de 121 MB del tamaño de una lavadora.

Además, observamos la longevidad no solo de las decisiones de diseño explícitas, sino también de las implícitas. En concreto, vimos

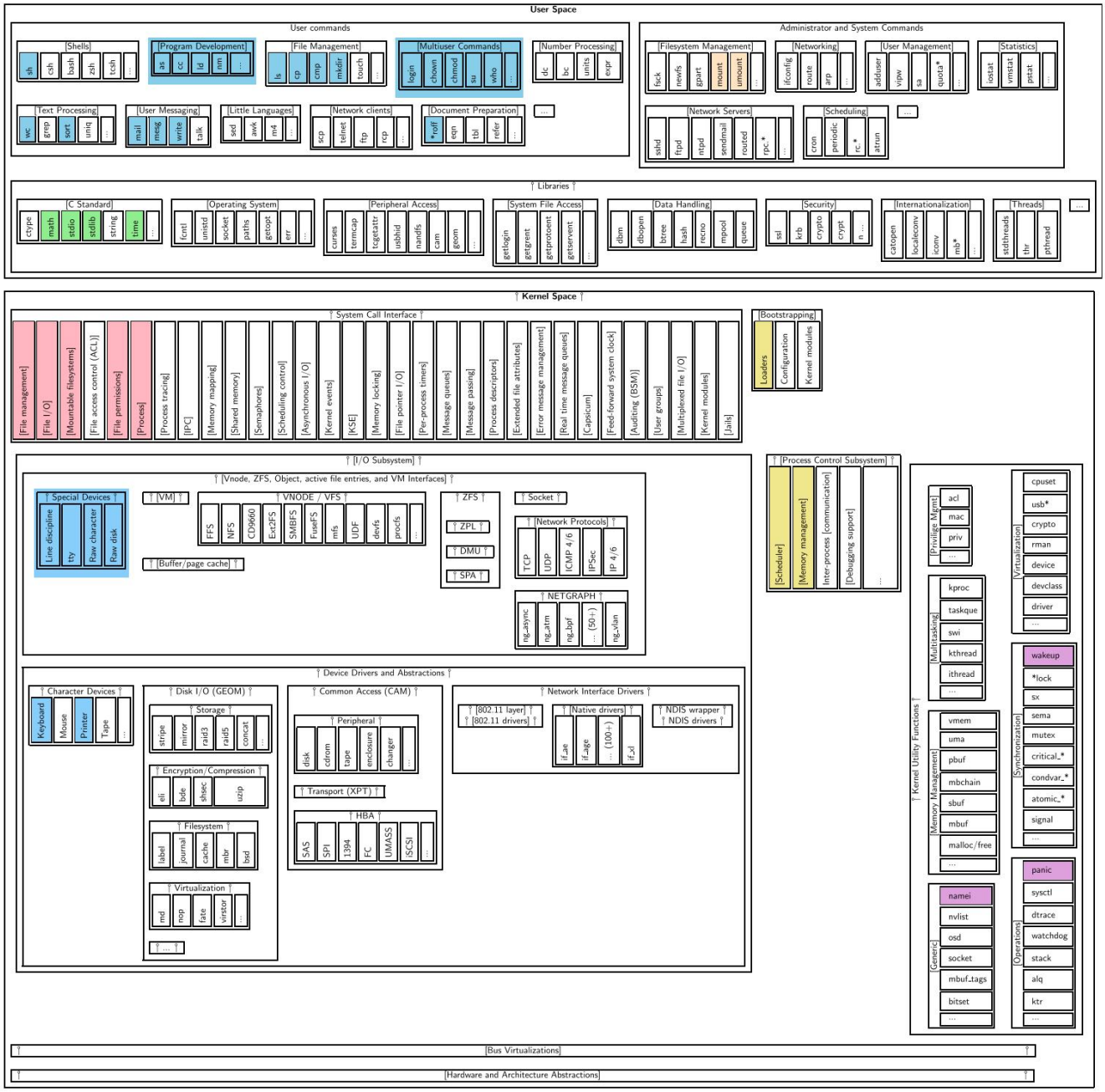

Fig. 6. Arquitectura de alto nivel de FreeBSD 11.0 (2017).

que las decisiones de diseño implícitas que no forman parte de una API documentada también pueden sobrevivir durante décadas e incluso influir en el diseño de otros sistemas. Por ejemplo, la interfaz del sistema de archivos virtual (Sección 4.14) ha sido adoptada por el kernel de Linux [129, Capítulo 13], mientras que la denominada rutina de estrategia del controlador de dispositivo (Sección 4.5) también podría encontrarse en el diseño de controladores de dispositivo de Linux [130 , Sección 14.4.3].

Explicación La longevidad de las decisiones arquitectónicas se debe principalmente al deseo de mantener la compatibilidad con versiones anteriores y los beneficios derivados de ella. Desde 1977, esto se instituyó a través de la estandarización, inicialmente informal y luego formal. Primero, un comité patrocinado por el Área de Tecnologías Informáticas de AT&T Bell Laboratories supervisó y promovió la portabilidad y evolución del lenguaje de programación C y las bibliotecas asociadas [53, p. 1687]. Más tarde, la estandarización de Unix se formalizó a través de esfuerzos como POSIX [92], [131], [132] y los estándares del lenguaje C [133], [134], [135].

Proposición 3. Las decisiones de nueva arquitectura son continuamente hecho, alimentando aún más la evolución de la arquitectura.

A pesar de la influencia y la permanencia de la arquitectura inicial, el estudio también demuestra que la arquitectura Unix continúa evolucionando significativamente muchos años después de que los cimientos del sistema se hayan grabado en piedra.
Por ejemplo, muchas decisiones importantes del diseño arquitectónico de Unix, como la portabilidad del sistema, la asignación de memoria dinámica, las variables de entorno, las herramientas de desarrollo de lenguajes, los pequeños lenguajes y el análisis de programas estáticos, aparecieron por primera vez en la séptima edición; diez años después de la implementación del prototipo PDP-7. En décadas más recientes, Unix ha continuado creciendo significativamente en tamaño y

complejidad a través de la adición de grandes subsistemas de terceros (consulte la Tabla 6) integrados a las características principales del sistema.

Explicación La razón de la evolución continua es, como era de esperar, los nuevos requisitos. Estos surgen de la necesidad de acomodar programas de usuario más sofisticados, lo que parece reflejarse en el aumento de las funciones de la biblioteca C admitidas y las llamadas al sistema que se ven en la Fig. 3, o admitir nuevo hardware, que se puede observar a través del número creciente de dispositivos compatibles. representada en la misma figura.

Los requisitos también pueden surgir de los avances realizados por otros sistemas operativos, trabajo destinado a mantenerse al día con los Jones, por así decirlo.

Proposición 4. La tasa de decisiones de arquitectura disminuye durante la vida útil del sistema.

A pesar de la evidencia de evolución arquitectónica continua, al observar los elementos enumerados en la línea de tiempo de evolución (Fig. 1), también es evidente que la tasa se ha ralentizado durante la vida útil del sistema en términos de nuevas decisiones de diseño importantes introducidas. Se pueden observar tres 'olas' principales: la primera comprende las ediciones de investigación, que presentaban un número significativo de importantes decisiones de diseño; el segundo y el tercero en las décadas de 1990 y 2000 respectivamente, que presentaban cada vez menos decisiones de diseño tan importantes.

Explicación Se pueden dar dos explicaciones plausibles. En primer lugar, los cambios de arquitectura se vuelven más difíciles a medida que el sistema envejece, debido al aumento del volumen y la complejidad del sistema. Por ejemplo, cuando se introdujeron las tuberías en la Tercera edición de investigación, los pocos miembros del equipo de Unix trabajaron durante la noche para convertir la mayoría de las utilidades del sistema en filtros (consulte la Sección 4.4). Introducir tal cambio en un sistema moderno sería mucho más difícil y complejo. En segundo lugar, debido a la madurez del sistema, rara vez se requieren nuevas características importantes o incluso disruptivas; en gran medida, estamos presenciando una saturación funcional.

6.2 Acumulación de Deuda Técnica Arquitectónica

Proposición 5. Una fuente importante de deuda técnica de arquitectura son las decisiones de arquitectura que ofrecen características que son similares a las existentes o que permanecen infrautilizadas.

Como era de esperar en un sistema desarrollado durante medio siglo, nuestro estudio también reveló síntomas de deuda técnica arquitectónica. Usamos la definición de deuda técnica de un seminario reciente de Dagstuhl [136].

> "La deuda técnica consiste en construcciones de diseño o implementación que son convenientes a corto plazo, pero establecen un contexto técnico que puede hacer que un cambio futuro sea más costoso o imposible. La deuda técnica es un pasivo contingente cuyo impacto se limita a las cualidades del sistema interno, principalmente la mantenibilidad y la capacidad de evolución".

La deuda técnica viene en muchos sabores; nuestra propuesta se refiere a dos tipos diferentes de deuda técnica que se observaron predominantemente en Unix.

El primer tipo se refiere a agregar una funcionalidad que es igual o similar a la funcionalidad existente, sin eliminar la existente ni fusionarlas en una sola fuente. Retener dos o más instalaciones competidoras que brindan servicios análogos la funcionalidad perjudica la comprensión y la mantenibilidad. Ejemplos incluyen:

- la proliferación de llamadas al sistema que realizan funciones ligeramente diferentes, como las nueve variantes para leer datos: read(2), pread(2), readv(2), preadv(2), recv(2), recvfrom(2), recvmmsg(2), recvmsg(2), sctp_ge neric_recvmsg(2)—y un número similar para escribir datos, o el 14 ...at hermanos de llamadas al sistema existentes— bindat(2), connectat(2), fstatat(2), faccessat(2), linkat(2), mkdirat(2), mkfifoat(2), mknodat(2), openat(2), readlinkat (2), symlinkat(2), unlinkat(2), renombrar(2)) [137]; el soporte de múltiples mecanismos de registro: escritura en archivos sin formato en /var/log, registro a través de syslogd(8), contabilidad de procesos a través de act(2) y auditoría de BSM a través de auditd(8) (Sección 4.27); la coexistencia de la configuración tradicional de permisos de archivo de usuario-grupo-otros, con listas de control de acceso (Sección 4.24), y un marco de control de acceso obligatorio separado (Sección 4.25); y la coexistencia de dos primitivas multitarea: hilos y procesos.

Encontramos un ejemplo sorprendente de este tipo de deuda técnica que se relaciona con la pérdida de integridad conceptual. Una innovación importante del sistema operativo Unix es el mapeo de dispositivos de almacenamiento, terminales, enlaces de comunicación y memoria en archivos especiales. Según los creadores del sistema, este tratamiento homogéneo tiene tres ventajas: hace que la API de E/S del dispositivo sea similar a la API de archivos; permite utilizar programas ordinarios en archivos especiales proporcionando sus nombres de archivo correspondientes; y reutiliza el mecanismo de protección de archivos existente en archivos especiales [138, pp. 1909–1910]. Con el tiempo, los enfoques de la competencia han violado la integridad conceptual de este enfoque al no usar archivos especiales y, por lo tanto, perder las ventajas anteriores. Por ejemplo, la supervisión y el control del sistema y sus procesos se pueden lograr siguiendo el enfoque de archivos especiales, a través del sistema de archivos procfs(5) (Sección 4.20). Sin embargo, dicha funcionalidad también se proporciona mediante llamadas al sistema: ptrace(2), getrusage(2), getrlimit(2) y mediante el sistema dtrace(1). De manera similar, se pueden usar archivos especiales para controlar la configuración del sistema operativo, como se hace, por ejemplo, a través del sistema de archivos sysfs de Linux [129, 355–361]. Sin embargo, la mayor parte de esta funcionalidad se implementa a través de numerosas llamadas al sistema, por ejemplo, acct(2), adjtime(2), auditctl(2), getfsstat(2), gettimeofday(2), kenv(2), mincore(2) , modfind(2), procctl(2), quotactl(2), settimeofday(2), y también a través de la interfaz sysctl jerárquica pero distinta (Sección 4.16).

Explicación El principal impulsor detrás de la acumulación de características competidoras similares es la falta de propiedad con respecto a la integridad conceptual de todo el sistema [79, p. 460]. A medida que la evolución de Unix se mueve entre grupos e individuos, estos pueden estar más interesados en dejar su huella a través de una nueva funcionalidad que en consolidar el trabajo existente y refactorizar el código antiguo para que funcione con funciones mejoradas gradualmente. Esto se puede ver en la Fig. 3, donde el cambio de Bell Labs a Berkeley y luego a FreeBSD está marcado por un aumento en el número de llamadas al sistema. Además, cada generación de administradores de código puede dudar en cambiar radicalmente el código de sus predecesores. Además, a medida que el sistema se construye cada vez más mediante la combinación de código desarrollado por diversos equipos para servir a múltiples proyectos,

se vuelve muy difícil coordinar cambios de refactorización extensos.

El segundo tipo de deuda técnica tiene que ver con una funcionalidad complicada que se ofreció pero que nunca se usó del todo.
Esto viola el principio YAGNI ("no lo necesitará") e incurre en un esfuerzo de mantenimiento adicional para la funcionalidad que en realidad no está en uso. Eliminar esta redundancia y limpiar el sistema remediaría la deuda técnica. Un ejemplo típico de esto es el IPS basado en socket con su gran cantidad de llamadas al sistema (consulte la Tabla 3).

Explicación Este tipo de deuda técnica casi siempre es inadvertida: ciertas decisiones arquitectónicas parecen acertadas en un momento dado, pero luego se vuelven problemáticas debido a cambios en la tecnología o el dominio de la aplicación. Por ejemplo, las elaboradas abstracciones de flujo de sockets y datagramas que se diseñaron como parte de la API del protocolo de red en 4.2BSD (Sección 4.12) se volvieron irrelevantes por la adopción universal de los protocolos de Internet y el eclipse de las tecnologías competidoras [139, p. 87]. Sin embargo, la complejidad que lo acompaña sigue siendo una carga para la API. En el lado positivo, la generalidad de la API de red permitió que se introdujera soporte para la versión 6 del Protocolo de Internet sin necesidad de nuevas llamadas al sistema.

Proposición 6. La deuda técnica de la arquitectura se paga sistemáticamente a pesar del aumento del tamaño y la complejidad del sistema.

La evidencia de la deuda técnica que encontramos en Unix es sustancial y perjudica la capacidad de mantenimiento y evolución del sistema. Sin embargo, para un sistema de su tamaño, complejidad y antigüedad, la deuda técnica de Unix es impresionantemente limitada. Por lo general, el crecimiento en tamaño y complejidad durante un largo período de tiempo da como resultado incurrir en deuda técnica a un ritmo creciente; por lo tanto, la mayoría de los sistemas de tamaño similar se han convertido en "grandes bolas de barro". Por el contrario, Unix ha mantenido una calidad interna comparativamente alta y no manifiesta muchas "soluciones rápidas" o "soluciones alternativas" arquitectónicas. La evidencia de acción correctiva luego de la acumulación de deuda técnica es visible en la Fig. 4, donde los aumentos en la complejidad ciclomática son seguidos por una disminución posterior.

Explicación Se podría argumentar que la alta calidad interna del sistema puede deberse a la dedicación y el talento excepcional de los desarrolladores que trabajaron en el sistema, junto con la falta de presión comercial para seguir atajos en aras de la conveniencia. Sin embargo, cuando se compara la calidad del kernel de FreeBSD Unix con la de otros tres sistemas (Linux, Solaris y Windows Research Kernel), todos parecen estar en niveles similares [123].

Por lo tanto, argumentamos que la razón principal de la baja deuda técnica es un proceso de selección natural: el tamaño (ocho millones de líneas en el caso de FreeBSD) y la complejidad del kernel de un sistema operativo moderno, así como los requisitos de confiabilidad [51, p. . 1960] son tales que la calidad inferior a la media se elimina o se abandona el sistema correspondiente. La forma en que los estrictos requisitos de confiabilidad obligan a una alta calidad interna se puede observar en la Fig. 4, donde la complejidad ciclomática media disminuye a medida que pasamos de los comandos de usuario independientes, a la biblioteca C utilizada por todos ellos, al núcleo monolítico grande en del que todo depende. Un contraejemplo es el caso de Multics, que Thompson ha caracterizado como sobrediseñado, sobreconstruido y casi inutilizable [79, p. 463]; nunca prosperó.

6.3 Fuerzas de la evolución arquitectónica

Una arquitectura está impulsada por requisitos, pero también por fuerzas, como la tecnología, la cultura de la organización o la filosofía del diseño. Las siguientes proposiciones se refieren a tales fuerzas.

Proposición 7. La preferencia por las convenciones en lugar de la aplicación facilita la evolución al reducir el esfuerzo y ofrecer flexibilidad.

Los desarrolladores del sistema a menudo establecieron y siguieron convenciones ligeras en lugar de implementar mecanismos de aplicación rígidos. En ediciones anteriores, tales convenciones incluían la agrupación de archivos relacionados a través de sus nombres, la configuración de espacios de nombres de identificadores a través de un prefijo (Sección 4.1), el procesamiento de directorios como archivos, la creación de una estructura de directorios de árbol navegable a través de enlaces de archivos arbitrarios, la adopción de archivos de texto simple como un formato de datos común y el uso de formatos de archivo documentados como un mecanismo de acoplamiento de programas (Sección 4.2). En la Séptima Edición de Investigación (Sección 4.8) se aplicó el mismo principio en la configuración de las variables de entorno como pares clave-valor y la documentación detallada del diseño del directorio d

Explicación La práctica de la convención sobre la aplicación minimizó el esfuerzo de implementación del sistema y promovió la experimentación. Los problemas que surgían por el comportamiento indisciplinado se abordaron cuando realmente se volvieron insuperables [76]. Esta práctica fue un factor importante que contribuyó a la funcionalidad inusualmente rica en comparación con el tamaño del código que proporcionaban los primeros sistemas Unix. La flexibilidad del enfoque también permitió la adaptación y transformación sin esfuerzo de las convenciones a las necesidades cambiantes. Argumentamos que, con buen gusto y algo de disciplina, tal enfoque puede producir mejores resultados que los que resultarían de un mecanismo de cumplimiento rígido diseñado de antemano para requisitos confusos. Una vez más, los enfoques ágiles y descriptivos prosperan sobre los prescriptivos.

Proposición 8. La portabilidad, por su inherente complejidad, es un impulsor clave de la evolución.

Otra fuerza importante que ha estado impulsando la arquitectura del software es la portabilidad. Una contribución clave de Unix fue la implementación de un sistema operativo que podía trasladarse fácilmente entre diferentes arquitecturas de máquinas. En palabras de Johnson y Ritchie [54], el sistema debería ser “fácilmente portátil sin cambios” entre diferentes hosts, pero también “fácil de cambiar” para “aprovechar al máximo máquinas mucho más potentes en muchas dimensiones posibles”. Los estrictos requisitos de portabilidad entre diversas arquitecturas de hardware y dispositivos obligaron a los diseñadores del sistema a adoptar numerosos métodos sofisticados de abstracción para domar la complejidad asociada.

Explicación La portabilidad ha impulsado la evolución de la arquitectura principalmente mediante el uso de capas para ocultar la funcionalidad no portátil detrás de abstracciones portátiles [50]. Al principio, la necesidad de portabilidad influyó en el diseño del sistema, el lenguaje de programación C, la biblioteca C portátil (Sección 4.7), así como los archivos de encabezado (Sección 4.8) y las herramientas de análisis estático (Sección 4.8) [54]. Además, un enfoque de portabilidad adoptado por los diseñadores de Unix fue definir modelos abstractos de máquinas para C y Unix [54, pp. 2041–2046]. Durante la larga evolución de Unix, se tomaron muchas decisiones de arquitectura para facilitar la portabilidad, por ejemplo, la introducción del vnode

TABLA 5
Principales influencias de terceros de FreeBSD

Fuente	confirma	LdC
Proyecto TrustedBSD	1.215	413.339
NetBSD	1.166	2.665.223
OpenBSD	726	113.195
KAME	451	163.874
Semimedio sp.	330	214.289
DragonflyBSD	179	675.906
Linux Qualcomm	151	109.600
Atheros, Inc.	139	46.608
ABT Systems Ltd	133	8.704
Juniper Networks, Inc.	125	66.971
NetApp, Inc.	120	8.044
Illumos	97	56.618
OpenSolaris	95	125.503
Wheel Systems, Inc.	81	3.552
yandex llc apple,	64	3.630
inc.	58	13.378

TABLA 6
Principales subsistemas de terceros en FreeBSD 11.1

Subsistema	kLoC	LdC %
llvm	3.413	10.81.1,9,99,99
gcc	1.576	3.52 2.76
binutils	1.111	2.39 2.09
PnT	873	1.77 1.77
Heimdal	756	1.54 1.39
subversión	661	1.28 1.25
openssl	558	1.25 1.20
gdb	488	0,90 0,98 0,7
groff	438	0,7,77 0,77
ofed	404	0,76 0,73
libstdc++	394	0,65 0,65
wpa	380	0,65 0,60
libarchive	310	0.59 0.57
sqlite3	281	0.48 0.47
maldiciones	242	0.45 0.39
netbsd-pruebas	239	0.39 0.37
zfs dtrace	230	0.36 0.34
sendmail no	205	0.32 0.332sol
enlazado gcclibs	205	en oso en
openssh byacc	189	oso en oso
libc++ tcpdump	187	en oso en
compilador-rt	179	oso en oso
ldns tcsh	150	en oso en
	142	oso en oso
	123	en oso en
	121	oso en oso
	115	en oso en
	109	oso en oso
openbsm	102	en oso en
elftoolchain	101	oso en oso en oso en oso en ando ando ando ando

interfaz para abstraer diversos sistemas de archivos (Sección 4.14) y las pilas modernas CAM (Sección 4.22), negraph (Sección 4.23) y GEOM (Sección 4.25).

Proposición 9. Un ecosistema sofisticado de otros sistemas operativos y terceros da forma constantemente a la evolución de la arquitectura.

A nivel organizacional, la evolución de la arquitectura de los sistemas Unix en general y el linaje FreeBSD estudiado aquí en particular ha sido influenciado por la tecnología desarrollada por otros sistemas y organizaciones relacionados. La Fig. 714 muestra cómo diversas variantes y versiones de Unix se cruzaron entre sí a través de la adopción de código. Además, las ideas detrás de Unix han influido en más sistemas operativos que se desarrollaron de forma independiente, incluidos Android, GNU/Linux, Microsoft Windows, Minix [140], MS DOS, QNX y Z/OS. Parte de esta influencia se aplicó a través de la estandarización formal a través del esfuerzo POSIX [92], [131], [132] y la especificación única de UNIX.

Uno de los primeros influenciadores de Unix fue DARPA, que financió a CSRG para producir 4BSD (consulte la Sección 4.11). El éxito de esta empresa provocó un mayor escrutinio, críticas sobre el rendimiento del sistema y, como respuesta, un kernel ajustado sistemáticamente lanzado como 4.1BSD [78].

Otras contribuciones reconocidas de software de terceros se remontan a 4.3BSD Tahoe (consulte la Sección 4.13). Se pueden derivar más detalles sobre las influencias de diversos sistemas y organizaciones observando las confirmaciones de código del sistema de administración de configuración individual. Desde 1994, los mensajes de confirmación en los sistemas estudiados aquí a menudo incluyen un encabezado "Obtenido de:", lo que nos permitió rastrear las influencias directas a través de la adopción del código. En total encontramos L104 7.685 confirmaciones de este tipo, de 1.283 fuentes, con un total de 7.742.678 líneas de código. Las fuentes con más de 50 confirmaciones cada una se enumeran en la Tabla 5. L102, L103 Vemos que FreeBSD ha sido influenciado principalmente por sus hermanos cercanos, como NetBSD, OpenBSD y DragonflyBSD, así como por proyectos estrechamente afiliados, como TrustedBSD y kame proyectos Además, los influencers también incluyen empresas que utilizan FreeBSD, como Semihalf, Juniper, NetApp, Yandex, Wheel Systems y Apple. Finalmente, también vemos la influencia de sistemas que están menos relacionados con FreeBSD, como Linux, Illumos y OpenSolaris. La influencia adicional de terceros proviene de los componentes mayoristas integrados (Tabla 6) que se describen en la siguiente proposición.

La evolución de la arquitectura de Unix también estuvo influenciada con el tiempo a través de muchas decisiones y desarrollos no técnicos. Los principales entre ellos fueron los asociados a) con la disponibilidad del código fuente, que inicialmente promovió las contribuciones de terceros y luego condujo a organizaciones creadas en torno al desarrollo de software de código abierto, b) el desarrollo de versiones competidoras (Fig. 7) y esfuerzos (en su mayoría sin inspiración) combinarlos, yc) el movimiento de personas entre organizaciones [79, p. 454], lo que resultó en una polinización cruzada de ideas.

Explicación Una parte significativa del desarrollo de sistemas operativos tiene lugar entre una familia de sistemas derivados de la misma base de código fuente (Fig. 7) o influenciados por las mismas ideas clave (primera columna de la Tabla 5). Para seguir siendo compatible y competitivo con otros sistemas operativos, el equipo de FreeBSD rutinariamente importa código de ellos (Tabla 5). Además, su licencia de distribución permisiva permite que los elementos de FreeBSD se reutilicen fácilmente en sistemas derivados y en esfuerzos de desarrollo relacionados, como macOS de Apple.

Los factores no técnicos influyeron en Unix desde su nacimiento. En la década de 1970, AT&T todavía operaba bajo un "decreto de consentimiento" de 1956 [141]. Según sus términos, Bell Labs

14. Basado en un diagrama de Eraserhead1, Infinity0 y Sav_vas, con licencia de Cc BY-SA 3.0, a través de Wikimedia Commons.

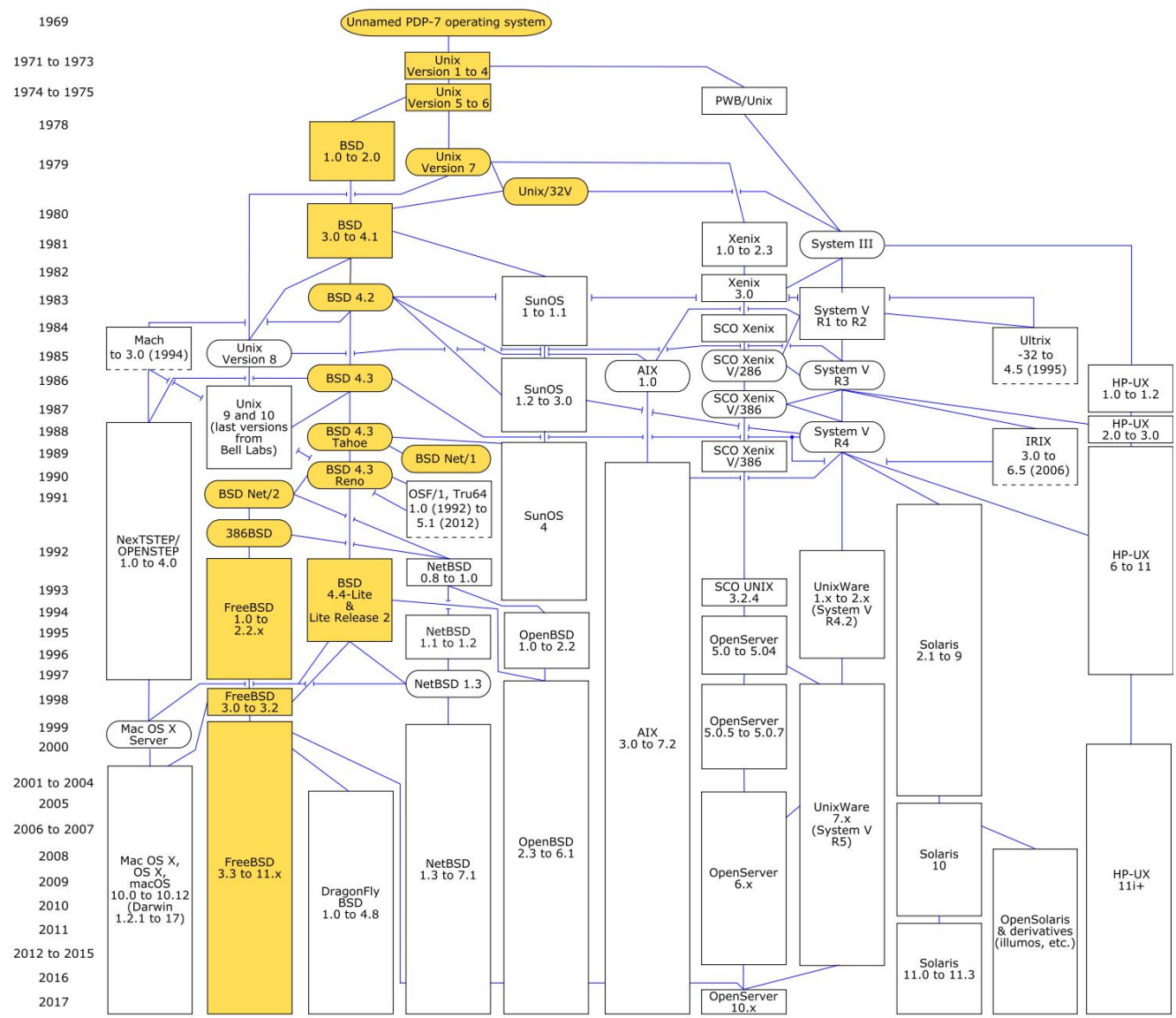

Fig. 7. Un diagrama simplificado de variantes y lanzamientos de Unix relacionados a través del código. Los elementos resaltados forman el linaje examinado en este estudio.

a los propietarios, Western Electric y AT&T, se les prohibió fabricar y ofrecer equipos y servicios que no fueran de telecomunicaciones. En consecuencia, AT&T no podía comercializar ni licenciar Unix con fines de lucro; Inicialmente, Unix tenía licencias liberales libres de regalías a través de acuerdos de cartas simples [77, p. 60], y su código fuente estuvo ampliamente disponible.

Esto permitió al personal de universidades de todo el mundo estudiar su código y contribuir con mejoras. Las restricciones legales de AT&T también dejaron un amplio espacio para el desarrollo de versiones competidoras de Unix de organizaciones como USG (Grupo de soporte de Unix de AT&T), Microsoft (XENIX), Berkeley (BSD) y decenas de proveedores de hardware [77, p. 209–210].

Muchas empresas carecían de expertos residentes que actuaran como "árbitros del gusto" [77, p. 211] en lugar de los desarrolladores originales de Unix. Como resultado, las empresas involucradas en las llamadas 'guerras de Unix' [77, p. 225] entre implementaciones competidoras a menudo acumulaban características de manera agresiva e indiscriminada, que se 'pegaban juntas' al azar [77, p. 211].

Luego, en las décadas de 1980 y 1990, los términos de licencia de AT&T se volvieron más complejos y restrictivos, lo que limitó la disponibilidad del código fuente de Unix [142], que se protegió cuidadosamente como un secreto comercial [93, pág. 20]. Estas restricciones llevaron al CSRG de Berkeley y a otros a trabajar en implementaciones de código abierto de Unix y la aparición de una estructura que conducía al desarrollo de código abierto.

Propuesta 10. La adopción de subsistemas de terceros facilita la evolución a través de la reutilización pero incurre en deuda técnica.

Otra fuerza observada ha sido la adopción de muchos subsistemas grandes, que se desarrollan mediante esfuerzos independientes y se integran periódicamente en las versiones publicadas.

La tabla 6L105 enumera los actuales cuyo tamaño supera las cien mil líneas de código (incluida la documentación y las pruebas).

Con la excepción de DTrace y ZFS, que están profundamente integrados en el árbol de código fuente de FreeBSD, los otros subsistemas 90L106 residen en dos directorios separados S89, S90 y se pueden actualizar fácilmente a medida que se lanzan nuevas versiones asc

A diferencia de los ports(7) de FreeBSD, los subsistemas de estos directorios forman parte integral del sistema operativo y, por lo general, son necesarios para su construcción y funcionamiento.

Muchos de estos subsistemas ofrecen funciones que en el pasado se desarrollaron dentro de los límites del sistema. Esta práctica

subcontrata el desarrollo de partes clave del sistema, dejando al equipo central de FreeBSD la responsabilidad de elegir entre implementaciones alternativas, como la elección entre GCC o LLVM como infraestructura de compilación.

Explicación El razonamiento detrás de la adopción de subsistemas de terceros es simple: el tamaño y la complejidad cada vez mayores de estos subsistemas implican ahorros de esfuerzo sustanciales para FreeBSD y muchas otras distribuciones de sistemas operativos, como GNU/Linux y macOS, que los reutilizan.

Por otro lado, una desventaja de este enfoque es que los subsistemas de terceros se desarrollan para utilizar solo la funcionalidad del mínimo común denominador de todos los sistemas operativos que los alojan. En consecuencia, cada sistema operativo que los adopta también hereda alguna deuda técnica: proporcionar la funcionalidad que podrían requerir algunos paquetes de terceros requiere la adición coordinada de esta función por parte de todos los sistemas operativos en los que se ejecuta el software de terceros. Esto hace que sea más probable que cada herramienta de terceros duplique alguna funcionalidad requerida (lo que resulta en redundancia) de una manera ligeramente diferente (dañando la comprensibilidad).

Proposición 11. Los grandes subsistemas forman su propia arquitectura, independientemente de la arquitectura del sistema que los engloba.

Hemos observado una fuerte fuerza hacia la federación de la arquitectura. Muchos subsistemas grandes, como el Kernel Networking basado en gráficos y la biblioteca de usuarios (netgraph—Sección 4.23), OpenSSL Framework (SSL—Sección 4.24), Control de acceso obligatorio (MAC—Sección 4.25), Módulo de autenticación conectable (PAM—Sección 4.25), El marco de transformación de solicitud de E/S de disco modular (GEOM: sección 4.25), la auditoría del módulo de seguridad básica (BSM: sección 4.27), el sistema de archivos Zettabyte (ZFS: sección 4.28) y el seguimiento dinámico (DTrace: sección 4.29) tienen su propia arquitectura. , con distintos principios, capas, componentes, mecanismos de complemento, subcomandos, patrones de diseño y convenciones. Algunas de estas estructuras constituyentes se pueden observar en la Fig. 6.

Explicación La principal razón de este fenómeno es que el tamaño y la complejidad de Unix pueden haber crecido mucho más allá del punto en el que se puede mantener como monolito (consulte la Tabla 1). Además, muchos subsistemas ahora son desarrollados de forma independiente por terceros (ver Tabla 6).
Esto dificulta la coordinación de su arquitectura con la del núcleo de FreeBSD.

7 AMENAZAS A LA VALIDEZ

Nuestro estudio está sujeto a limitaciones que pueden categorizarse en validez de constructo, validez externa y confiabilidad siguiendo las pautas de Runeson et al. [67]. La validez interna no es una preocupación para este estudio porque no examinamos las relaciones causales [67].

7.1 Validez de constructo Este

tipo de validez se refiere a hasta qué punto los ítems estudiados realmente representan lo que los investigadores pretenden de acuerdo con las preguntas de investigación [67]. En nuestro caso, las preguntas de investigación indagan sobre las principales decisiones de diseño arquitectónico de Unix a lo largo del tiempo, así como la evolución del tamaño y la complejidad del sistema. Respecto a las primeras, clasificamos como decisiones de diseño arquitectónico algunos de los componentes, conectores, patrones y principios arquitectónicos más significativos [69], [72]. Para mitigar una posible interpretación errónea de las decisiones de diseño de la arquitectura, el primer autor realizó de forma independiente la comparación constante y el segundo autor controló las decisiones de diseño codificadas en una segunda iteración. En caso de desacuerdo, los dos autores discutieron hasta llegar a un consenso; varias decisiones de diseño arquitectónico se eliminaron como resultado de este proceso.

Otro riesgo potencial se refiere a si fuimos exhaustivos durante la recopilación de datos: es decir, si nos hemos perdido alguna decisión de diseño arquitectónico significativa y, al mismo tiempo, si todas las decisiones de diseño arquitectónico informadas son significativas. Este riesgo no se puede mitigar por completo ya que la importancia de las decisiones de diseño de la arquitectura es en gran medida subjetiva. Sin embargo, nuestra triangulación de fuentes de datos ayudó a detectar aquellas decisiones de diseño arquitectónico a las que más de una fuente de datos prestó atención: se dio prioridad a las decisiones derivadas del código y la documentación de Unix que también se discutieron de manera destacada en libros y recuerdos de los pioneros de Unix. en nuestro proceso de selección.
Además, incluso si no podemos pretender exhaustividad, utilizamos una gran cantidad de fuentes de datos para aumentar las posibilidades de llegar a decisiones correctas.

Con respecto a los resultados cuantitativos, el tamaño del sistema se mide en términos de número de funciones (por ejemplo, comandos de usuario o llamadas al sistema), y la complejidad se mide en términos de complejidad ciclomática. Si bien estas pueden no ser formas únicas de medir el tamaño y la complejidad, ciertamente son válidas [124]. Además, tanto el conjunto de datos de características arquitectónicas como la herramienta utilizada para medir la complejidad ciclomática se basan en investigaciones publicadas revisadas por pares [15], [74], lo que mitiga parcialmente las amenazas asociadas con la validez del instrumento de medición.

7.2 Confiabilidad

Este tipo de validez se refiere a hasta qué punto la recolección y el análisis de datos dependen de los propios investigadores. Este riesgo se ha mitigado parcialmente ya que el primer autor realizó la codificación de forma iterativa, y el segundo autor controló los resultados. Sin embargo, debemos reconocer que el primer autor tiene décadas de experiencia en Unix. Si bien esto ha sido fundamental para comprender los detalles del objeto de estudio y, posteriormente, realizar la codificación, puede haber introducido un cierto sesgo en la selección de las decisiones de diseño arquitectónico (es posible que un experto no pueda observar el sistema objetivamente y puede estar sesgado). sobre la importancia de las diferentes decisiones de diseño). Una vez más, la triangulación de fuentes de datos ha ayudado a lidiar parcialmente con este sesgo, ya que nos aseguramos de que todas las decisiones de diseño arquitectónico seleccionadas se describieran en más de una fuente de datos, generalmente documentación y código fuente. Además, la confiabilidad del estudio se fortalece al ser abierto y explícito sobre el proceso de recopilación y análisis de datos, y al publicar en línea o en el suplemento de este documento todas las herramientas y datos utilizados.

7.3 Validez externa Este tipo

de validez se refiere a si los hallazgos pueden generalizarse a otros casos y contextos [67]. Este estudio es bastante singular en el sentido de que no tiene como objetivo proporcionar una conclusión general sobre una población (es decir, categoría de

sistemas o un dominio de aplicación). Además, la historia de Unix es excepcional, con numerosas partes interesadas y entornos que influyen en su desarrollo, por lo que la validez de extender los hallazgos a otros sistemas es discutible. En consecuencia, no afirmamos que nuestros hallazgos cualitativos o cuantitativos también deban ser válidos para otros grandes sistemas operativos. Sin embargo, Unix ha sido el sistema operativo dominante durante décadas, y su desarrollo ha influido mucho en los sistemas operativos posteriores ampliamente utilizados, como GNU/Linux, macOS y Android. En ese sentido, particularmente los resultados cualitativos con respecto a las decisiones de diseño de arquitectura de Unix son relevantes para otros sistemas operativos, porque proporcionan muchas de las decisiones de diseño significativas y la justificación que las acompaña.

8 CONCLUSIÓN

Examinamos de cerca la evolución de Unix desde una perspectiva arquitectónica examinando 30 lanzamientos principales desde PDP-7 Research Edition hasta FreeBSD 11. Triangulamos las fuentes de datos (código fuente, documentación, artículos y libros de investigación, recuerdos de los pioneros) para extraer datos válidos y actualizados. Hemos obtenido y producido una gran cantidad de datos y los hemos puesto a disposición de la comunidad [58], [74] para estudios posteriores.

Nuestro análisis arrojó resultados tanto cualitativos como cuantitativos. El examen cualitativo nos permitió establecer una línea de tiempo con los hitos más importantes que dieron forma a la arquitectura Unix; esos hitos se detallan como componentes, conectores, patrones y principios, así como otras decisiones clave de arquitectura. También discutimos el fundamento de esas decisiones y cómo afectaron los desarrollos futuros.

A través del análisis cuantitativo, mostramos las tendencias en el crecimiento del tamaño de los siete tipos principales de funciones (comandos de usuario, llamadas al sistema, bibliotecas, etc.), así como la complejidad. Encontramos un crecimiento uniforme en tamaño pero también algunos valores atípicos, para los cuales conjeturamos las explicaciones correspondientes. Descubrimos que la complejidad ciclomática creció al principio, pero luego se redujo, especialmente para la biblioteca y el kernel, donde la calidad del código es lo más importante. Finalmente, ponemos la evolución de Unix en contexto. En primer lugar, al comparar el número de características actuales de FreeBSD con el de otros cinco sistemas operativos actuales, encontramos una magnitud similar, indicativa de su complejidad esencial. En segundo lugar, al contrastar la complejidad ciclomática con GNU coreutils, la biblioteca C y el kernel de Linux, observamos en general una curva en U invertida con algunas diferencias marcadas.

A partir de los resultados, nos aventuramos a generalizarlos desarrollando una teoría inicial sobre la evolución de la arquitectura de los sistemas operativos; la teoría se compone de once proposiciones y sus correspondientes explicaciones. Numerosas decisiones iniciales de diseño sobreviven la prueba del tiempo y aún son visibles décadas después de su introducción. Sin embargo, la innovación continúa ininterrumpidamente para adaptarse a los cambios en la tecnología informática y las redes, aunque con un ritmo más lento a medida que pasan las décadas. Además, la deuda técnica arquitectónica se arrastra principalmente al retener dos o más instalaciones funcionalmente equivalentes, pero también al ofrecer una funcionalidad infrautilizada complicada que agrega esfuerzo de mantenimiento sin mucho valor real. Sin embargo, la deuda técnica arquitectónica no alcanza niveles críticos, ya que su remediación es sistemática a pesar del aumento de tamaño y complejidad. Además, la filosofía de mecanismos informales livianos en lugar de prescriptivos formales, el impulso por la portabilidad y un ecosistema intrincado de otros sistemas operativos y terceros son factores que dan forma a la evolución arquitectónica de sistemas operativos grandes y duraderos. Sin embargo, dado el tamaño actual y la complejidad de Unix, su evolución solo puede sostenerse mediante la adopción de subsistemas de terceros, mientras que muchos subsistemas grandes han formado una arquitectura propia.

De cara al futuro, el progreso en el hardware y las aplicaciones seguirá ejerciendo una presión evolutiva sobre la arquitectura de Unix en varios frentes. El almacenamiento flash y la computación de memoria universal cambian la forma en que se utiliza y se aborda el almacenamiento secundario; Las CPU con decenas de núcleos requieren soporte para un paralelismo de grano más fino; La computación GPU requiere abstracciones apropiadas de alto nivel; los métodos de aprendizaje profundo cambian la naturaleza de la computación al elevar los datos a su principal determinante; la seguridad y la privacidad exigen nuevos enfoques tanto en el centro de datos como en los perímetros; Los dispositivos móviles e IoT imponen restricciones exigentes sobre los recursos informáticos, la potencia y el rendimiento en tiempo real. Además, la gran base de código del sistema operativo y los requisitos de compatibilidad con versiones anteriores de las aplicaciones existentes dificul En resumen, los arquitectos del sistema operativo Unix tienen su trabajo separar.

EXPRESIONES DE GRATITUD

Los autores agradecen a los miembros de Unix Heritage Society15 y, en particular, a Warren Toomey y Kirk McKusick por preservar y poner a disposición muchos de los primeros artefactos importantes de Unix. También agradecen a los participantes de la lista de correo de TUHS por sus aportes y aliento con respecto a esta investigación.
Los autores están especialmente agradecidos con los revisores anónimos y con Kirk McKusick, George Neville-Neil, Warren Toomey y Alexios Zavras por sus comentarios detallados y esclarecedores sobre las versiones anteriores de este documento. La investigación descrita se ha llevado a cabo como parte del Proyecto CROSSMINER, que ha recibido financiación del Programa de Investigación e Innovación Horizonte 2020 de la Unión Europea en virtud del acuerdo de subvención n.º 732223.

REFERENCIAS

[1] MM Lehman, “Sobre la comprensión de las leyes, la evolución y la conservación en el ciclo de vida de los programas grandes”, J. Syst. Softw., vol. 1, págs. 213–221, septiembre de 1984.
[2] I. Herraiz, D. Rodríguez, G. Robles y JM Gonzalez-Barahona, “La evolución de las leyes de la evolución del software: una discusión basada en una revisión sistemática de la literatura”, ACM Comput. Surv., vol. 46, núm. 2, págs. 28:1–28:28, diciembre de 2013.
[3] DL Parnas, “Envejecimiento del software”, en Proc. 16 Int. Conf. suave Eng., 1994, págs. 279–287.
[4] SG Eick, TL Graves, AF Karr, JS Marron y A. Mockus, “¿El código se deteriora? Evaluación de la evidencia de los datos de gestión de cambios”, IEEE Trans. suave Ing., vol. 27, núm. 1, págs. 1 a 12, enero de 2001.
[5] P. Avgeriou, P. Kruchten, RL Nord, I. Ozkaya y C. Seaman, "Reduciendo la fricción en el desarrollo de software", IEEE Softw., vol. 33, núm. 1, págs. 66–73, enero de 2016.
[6] L. Hatton, D. Spinellis y M. van Genuchten, "La tasa de crecimiento a largo plazo del software en evolución: resultados empíricos e implicaciones", J. Softw.: Evolution Process, vol. 29, núm. 5, págs. e1847–n/a, 2017, e1847 smr.1847.

15. https://www.tuhs.org/

[7] S. Koch, “Evolución del software en proyectos de código abierto: una investigación a gran escala”, J. Softw. Evolución Mantenimiento: Res. Práctica, vol. 19, núm. 6, págs. 361–382, 2007.

[8] H. Breivold, M. Chauhan y M. Babar, "Una revisión sistemática de los estudios sobre la evolución del software de código abierto", en Proc. 17 Softw de Asia Pacífico. Ing. Conf., 2010, págs. 356–365.

[9] CA Conley y L. Sproull, “Es más fácil decirlo que hacerlo: una investigación empírica del diseño y la calidad del software en el desarrollo de software de código abierto”, en Proc. 42 Int. de Hawái Conf. sist. Sci., 2009, págs. 1–10.

[10] JW Paulson, G. Succi y A. Eberlein, "Un estudio empírico de productos de software de código abierto y de código cerrado", IEEE Trans. suave Ing., vol. 30, núm. 4, págs. 246 y 256, abril de 2004.

[11] A. Capiluppi, AE Faria y JF Ramil, “Explorando la relación entre el cambio acumulativo y la complejidad en un sistema de código abierto”, en Proc. 9º Euro. Conf. suave Reingeniería de mantenimiento, 2005, págs. 21–29.

[12] M. Aram, S. Koch y G. Neumann, "Análisis a largo plazo del desarrollo del marco comunitario abierto de ACS", en Proc. Conocimiento de soluciones de código abierto. Administrar. Ecosistemas tecnológicos, 2017, págs. 111–145.

[13] M. Godfrey y Q. Tu, “Evolución en el software de código abierto: un estudio de caso”, en Proc. En t. Conf. suave Mantenimiento, 2000, págs. 131–142.

[14] A. Israeli y DG Feitelson, “El kernel de Linux como caso de estudio en la evolución del software”, J. Syst. Softw., vol. 83, núm. 3, págs. 485–501, marzo de 2010.

[15] D. Spinellis, P. Louridas y M. Kechagia, "La evolución de las prácticas de programación C: un estudio del sistema operativo Unix 1973–2015", en Proc. 38 Int. Conf. suave Eng., mayo de 2016, págs. 748–759.

[16] A. MacCormack, J. Rusnak y CY Baldwin, "Explorando la estructura de diseños de software complejos: un estudio empírico de código abierto y código propietario", Manage. ciencia, vol. 57, núm. 7, págs. 1015–1030, 2006.

[17] A. Israeli y DG Feitelson, “El kernel de Linux como caso de estudio en la evolución del software”, J. Syst. Softw., vol. 83, núm. 3, págs. 485–501, 2010.

[18] TJ McCabe, "Una medida de complejidad", IEEE Trans. suave Ing., vol. SE-2, núm. 4, págs. 308–320, julio de 1976.

[19] DG Feitelson, "Desarrollo perpetuo: un modelo del ciclo de vida del kernel de Linux", J. Syst. Softw., vol. 85, núm. 4, págs. 859–875, 2012.

[20] P. Behnamghader, DM Le, J. García, D. Link, A. Shahbazian y N. Medvidovic, "Un estudio a gran escala de la evolución arquitectónica en sistemas de software de código abierto", Empirical Softw. Ing., vol. 22, núm. 3, págs. 1146–1193, junio de 2017.

[21] M. D'Ambros, H. Gall, M. Lanza y M. Pinzger, "Análisis de repositorios de software para comprender la evolución del software", en Software Evolution. Berlín, Alemania: Springer, 2008, págs. 37–67.

[22] R. Wettel y M. Lanza, "Exploración visual de la evolución del sistema a gran escala", en Proc. XV Conf. de Trabajo. Ing. inversa, octubre de 2008, págs. 219–228.

[23] E. Bouwers, JP Correia, A. v. Deursen y J. Visser, “Cuantificación de la capacidad de análisis de las arquitecturas de software”, en Proc. 9ª Conferencia de Trabajo IEEE/IFIP. suave Archit., junio de 2011, págs. 83–92.

[24] SC Johnson y BW Kernighan, "El lenguaje de programación B", Bell Laboratories, Murray Hill, NJ, EE. UU., Computer Science Tech. Rep. 8, enero de 1977. [En línea]. Disponible: http://web.archive. org/web/20180831015050/https://www.bell-labs.com/usr/ dmr/www/bintro.html [25] SC Johnson, “Lint, a C program checker”, Bell Laboratories, Murray Hill, NJ, EE. UU., Cómputo. ciencia tecnología Rep. 65, diciembre de 1977. [En línea]. Disponible: http://web.archive.org/web/20160412071448/http://files.cnblogs.com:80/files/bangerlee/10.1.1.56.1841.pdf [26] BW Kernighan y LL Cherry, “Un sistema para escribir matemáticas”, Bell Laboratories, Murray Hill, NJ, EE. UU., Comput. ciencia tecnología Rep. 17, mayo de 1974. [En línea]. Disponible: https://web.archive.org/web/20151029232442/http://tex.loria.fr/divers/ unix-eqn1.ps.gz [27] JF Maranzano y SR Bourne, “Una introducción tutorial a ADB”, Bell Laboratories, Murray Hill, NJ, EE. UU., Cómputo. ciencia tecnología Rep. 62, mayo de 1977. [En línea]. Disponible: https://web. archive.org/web/20040324013641/https://wolfram.schneider. org/bsd/7thEdManVol2/adb/adb.pdf [28] SC Johnson, “Yacc: otro compilador-compilador más”, Bell Laboratories, Murray Hill, NJ, Comput. ciencia tecnología Rep. 32, julio de 1975.

[En línea]. Disponible: https://web.archive.org/web/20170810013946/https://www.isi.edu/pedro/Teaching/CSCI565-Fall15/Materials/Yacc.pdf

[29] ME Lesk, “Lex: un generador de analizadores léxicos”, Bell Laboratories, Murray Hill, NJ, Comput. ciencia tecnología Rep. 39, oct. 1975, [en línea]. Disponible: https://web.archive.org/web/ 20040324060316/http://wolfram.schneider.org:80/bsd/ 7thEdManVol2/lex/lex.pdf [30] BW Kernighan, “UNIX para principiantes”, Bell Laboratories, Mur ray Hill, NJ, EE. UU., Comput. ciencia tecnología Rep. 75, febrero de 1979. [En línea].

Disponible: https://web.archive.org/web/20170711222622/ http://wolfram.schneider.org/bsd/7thEdManVol2/beginners/ beginings.pdf [31] R. Morris y K. Thompson, “Password security: Una historia de caso”, Bell Laboratories, Murray Hill, NJ, EE. UU., Comput. ciencia tecnología

Rep. 71, abril de 1978. [En línea]. Disponible: https://web.archive. org/web/20180317102420/http://wolfram.schneider.org/bsd/ 7thEdManVol2/password/password.pdf [32] SI Feldman, “Make: un programa para mantener programas informáticos”, Bell Laboratories, Murray Hill, NJ, EE. UU., Cómputo.

ciencia tecnología Rep. 57, abril de 1977. [En línea]. Disponible: https://web. archive.org/web/20040805040247/http://wolfram.schneider. org:80/bsd/7thEdManVol2/make/make.pdf [33] BW Kernighan, “A typesetter-independent TROFF,” Bell Laboratories, Murray Hill, NJ, USA, Comput. ciencia tecnología Rep. 97, 1982.

[34] AT&T, Ed., Lecturas y aplicaciones del sistema UNIX, vol. I. Englewood Cliffs, NJ, EE. UU.: Prentice Hall, 1978 (Bell Syst. Tech. J., vol. 57, no. 6, julio/agosto de 1978).

[35] AT&T, Ed., Lecturas y aplicaciones del sistema UNIX, vol. II. Englewood Cliffs, NJ, EE. UU.: Prentice Hall, 1987 (AT&T Bell Laboratories Tech. J., vol. 63, no. 8, octubre de 1984).

[36] DM Ritchie y K. Thompson, “El sistema de tiempo compartido de UNIX”, Commun. ACM, vol. 17, núm. 7, págs. 365–375, julio de 1974.

[37] KL Thompson, "Reflexiones sobre confiar en la confianza", Commun. ACM, vol. 27, núm. 8, págs. 761–763, agosto de 1984.

[38] BW Kernighan, “PIC: un lenguaje para componer gráficos”, Softw.: Practice Exp., vol. 12, págs. 1 a 21, 1982.

[39] JL Bentley, LW Jelinski y BW Kernighan, “CHEM: un programa para diagramas de estructuras químicas de fotocomposición”, computar Química, vol. 11, núm. 4, págs. 281–297, 1987.

[40] R. Pike y K. Thompson, “Hello world”, en Proc. Tecnología USENIX. Conf. Proc., invierno de 1993, págs. 43–50.

[41] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey y P. Winterbottom, “Plan 9 from Bell Labs”, Comput. Syst., vol. 8, núm. 2, págs. 221–254, 1995.

[42] O. Babaoglu y W. Joy, "Conversión de un sistema basado en intercambio para hacer paginación en una arquitectura que carece de bits de referencia de página", en Proc. 8º Simposio ACM. Sistema Operativo Principios, 1981, págs. 78–86.

[43] MK McKusick, WN Joy, SJ Leffler y RS Fabry, “Un sistema de archivos rápido para UNIX”, ACM Trans. computar Syst., vol. 2, núm. 3, págs. 181–197, agosto de 1984.

[44] R. Sandberg, “El diseño y la implementación del sistema de archivos de red de Sun”, en Proc. Asociación USENIX. Conf. Proc., junio de 1985, págs. 119–130.

[45] WF Jolitz y LG Jolitz, “Porting UNIX to the 386: Un enfoque práctico. Diseño de una especificación de software”, Dr. Dobb's J., vol. 16, núm. 1, enero de 1991.

[46] WR Stevens y J.-S. Pendry, "Portales en 4.4BSD", en Proc. USE NIX 1995 Tecnología. Conf. Proc., enero de 1995, págs. 1–1.

[47] MK McKusick y GR Ganger, "Actualizaciones suaves: una técnica para eliminar la mayoría de las escrituras sincrónicas en el sistema de archivos rápido", en Proc. USENIX Anual. tecnología Conf. Freenix Track, junio de 1999, págs. 1–18.

[48] J. Bonwick, M. Ahrens, V. Henson, M. Maybee y M. Shellenbaum, “The zettabyte file system”, en Proc. 2ª Conf. Usenix. Tecnología de almacenamiento de archivos, abril de 2003.

[49] BM Cantrill, MW Shapiro y AH Leventhal, “Instrumentación dinámica de los sistemas de producción”, en Proc. USENIX Anual. tecnología Conf., junio de 2004, págs. 15–28.

[50] D. Spinellis, “Otro nivel de indirección”, en Beautiful Code: los programadores líderes explican cómo piensan, A. Oram y G. Wilson, Eds. Sebastopol, CA, EE. UU.: O'Reilly and Associates, 2007, cap. 17, págs. 279–291.

[51] DM Ritchie, “Una retrospectiva”, Bell Syst. tecnología J., vol. 56, núm. 6, págs. 1947–1969, julio/agosto. 1978.

[52] K. Thompson, “Sistema de tiempo compartido de UNIX: implementación de UNIX,” Sistema de campana tecnología J., vol. 56, núm. 6, págs. 1905–1929, julio/agosto. 1978.

[53] L. Rosler, "La evolución de C: pasado y futuro", Bell Syst. tecnología J., vol. 63, núm. 8, págs. 1685–1699, octubre de 1984.

[54] SC Johnson y DM Ritchie, “Portabilidad de los programas C y el sistema UNIX”, Bell Syst. tecnología J., vol. 57, núm. 6, págs. 2021–2048, julio/agosto. 1978.

[55] DM Ritchie, “La evolución del sistema de tiempo compartido UNIX,” Técnico de AT&T Bell Laboratories. J., vol. 63, núm. 8, págs. 1577–1593, octubre de 1984.

[56] W. Toomey, “La restauración de los primeros artefactos de UNIX”, en Proc. USENIX Anual. tecnología Conf., 2009, págs. 20–26.

[57] W. Toomey, “Primera edición de Unix: Su creación y restauración,” IEEE Ana. Computación histórica, vol. 32, núm. 3, págs. 74–82, julio-septiembre. 2010.

[58] D. Spinellis, "Un repositorio de la historia y la evolución de Unix", Software empírico. Ing., vol. 22, núm. 3, págs. 1372–1404, 2017.

[59] W. Toomey, “Unix: Creación de un entorno de desarrollo desde cero”, en Reflexiones sobre los sistemas operativos: aspectos históricos y filosóficos, L. Demol y G. Primiero, Eds. Nueva York, NY, EE. UU.: Springer, 2017.

[60] J. Lions, Lions' Commentary on Unix 6th Edition with Source Code. Comunicaciones punto a punto, San José, CA, EE. UU., 1996.

[61] MJ Bach, El diseño del sistema operativo UNIX. Englewood Cliffs, Nueva Jersey, EE. UU.: Prentice Hall, 1986.

[62] SJ Leffler, MK McKusick, MJ Karels y JS Quarterman, El diseño e implementación del sistema operativo 4.3BSD Unix. Boston, MA, EE. UU.: Addison-Wesley, 1988.

[63] MK McKusick, K. Bostic y MJ Karels, El diseño e implementación del sistema operativo 4.4BSD Unix. Reading, MA, EE. UU.: Addison-Wesley, 1996.

[64] MK McKusick y GV Neville-Neil, El diseño e implementación del sistema operativo FreeBSD. Reading, MA, EE. UU.: Addi son-Wesley, 2004.

[65] MK McKusick, G. Neville-Neil y RN Watson, El diseño y la implementación del sistema operativo FreeBSD, 2.ª ed. Reading, MA, EE. UU.: Addison-Wesley Professional, 2014.

[66] EI Organick, El Sistema Multics: Un Examen de su Estructura. Cambridge, MA, EE. UU.: MIT Press, 1972.

[67] P. Runeson, M. Host, A. Rainer y B. Regnell, Investigación de estudios de casos en ingeniería de software: pautas y ejemplos, 1.ª ed. Hoboken, Nueva Jersey, EE. UU.: Wiley Publishing, 2012.

[68] V. Basili, C. Caldiera y DH Rombach, "Paradigma métrico de preguntas de objetivos", en Encyclopedia of Software Engineering. Nueva York, NY, EE. UU.: Wiley, 1994, vol. 2, págs. 528–532.

[69] RN Taylor, N. Medvidovic y EM Dashofy, Arquitectura de software: fundamentos, teoría y práctica. Hoboken, Nueva Jersey, EE. UU.: Wiley Publishing, 2009.

[70] J. Tyree y A. Akerman, "Decisiones de arquitectura: Demy stifying architecture", IEEE Softw., vol. 22, núm. 2, págs. 19 a 27, marzo de 2005.

[71] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad y M. Stal, Arquitectura de software orientada a patrones, volumen 1: un sistema de patrones. Hoboken, Nueva Jersey, EE. UU.: Wiley, 1996.

[72] N. Harrison, P. Avgeriou y U. Zdun, "Uso de patrones para capturar decisiones arquitectónicas", IEEE Softw., vol. 24, núm. 4, págs. 38–45, julio/agosto. 2007.

[73] J. Singer, SE Sim y TC Lethbridge, "Recopilación de datos de ingeniería de software para estudios de campo", en Guide to Advanced Empirical Software Engineering, F. Shull, J. Singer y DIK Sjøberg, Eds. Londres, Reino Unido: Springer, 2008, págs. 9–34.

[74] D. Spinellis, “Instalaciones documentadas de Unix durante 48 años”, en Proc. XV Conferencia software de minería Repositorios, mayo de 2018, págs. 58–61.

[75] M. McIlroy, “Entrevista con Michael S. Mahoney”, agosto de 1989, actual diciembre de 2018. Doi archivado: 10.5281/zenodo.2525529. [En línea]. Disponible: https://www.princeton.edu/hos/mike/ transcripts/mcilroy.htm [76] K. Thompson, “Entrevista con Michael S. Mahoney”, junio de 1989, actual diciembre de 2018. Doi archivado: 10.5281/zenodo.2525529. [En línea]. Disponible: https://www.princeton.edu/~hos/mike/ transcripts/thompson.htm

[77] PH Salus, Un cuarto de siglo de UNIX. Boston, MA, EE. UU.: Addi hijo-Wesley, 1994.

[78] MK McKusick, “Veinte años de Berkeley Unix: de propiedad de AT&T a redistribución libre”, en Open Sources: Voces from the Open Source Revolution, C. DiBona, S. Ockman y M. Stone, Eds. Newton, MA, EE. UU.: O'Reilly, 1999, págs. 31–46.

[79] P. Seibel, Codificadores en el trabajo: Reflexiones sobre el oficio de la programación. Nueva York, NY, EE. UU.: Apress, 2009, cap. 12: Ken Thompson, págs. 449–483.

[80] J. Schilling, “Programas mantenidos por el usuario en la segunda edición,” TUHS: lista de correo de Unix Heritage Society, diciembre de 2016. [En línea]. Disponible : http://minnie.tuhs.org/pipermail/tuhs/ 2016-diciembre/007561.html

[81] S. Johnson, “¿Qué provocó la pelusa? [era: historias de Unix]”, lista de correo de The Unix Heritage Society, enero de 2017, consultado el: 21 de noviembre de 2017. 2017, archivado por WebCite en [en línea]. Disponible: http://www. webcitation.org/ 6v8TXa7kK [82] Bell Laboratories, Manual del programador de UNIX. Volumen 1, 7ª ed. Murray Hill, NJ, EE. UU.: Bell Telephone Laboratories, 1979.

[83] Bell Laboratories, Manual del programador de UNIX. Volumen 2: Documentos complementarios, 7.ª ed. Murray Hill, NJ, EE. UU.: Bell Telephone Laboratories, 1979.

[84] SR Bourne, “Una introducción al shell de UNIX”, en UNIX Programmer's Manual. Volumen 2—Documentos complementarios, 7ª ed. Murray Hill, NJ, EE. UU.: Bell Telephone Laboratories, 1979.

[85] DM Ritchie, “El lenguaje de programación C: manual de referencia”, en UNIX Programmer's Manual. Volumen 2—Documentos complementarios, 7ª ed. Murray Hill, NJ, EE. UU.: Bell Telephone Laboratories, 1979.

[86] SC Johnson, “Lint, a C program checker”, en UNIX Programmer's Manual. Volumen 2—Documentos complementarios, 7ª ed. Murray Hill, NJ, EE. UU.: Bell Telephone Laboratories, 1979.

[87] ME Lesk, “TBL: un programa para formatear tablas”, en UNIX Programmer's Manual. Volumen 2—Documentos complementarios, 7ª ed. Murray Hill, NJ, EE. UU.: Bell Telephone Laboratories, 1979.

[88] F. Zhang, AE Hassan, S. McIntosh e Y. Zou, "El uso de la suma para agregar métricas de software dificulta el rendimiento de los modelos de predicción de defectos", IEEE Trans. suave Ing., vol. 43, núm. 5, págs. 476–491, mayo de 2017.

[89] CB Seaman, “Qualitative method”, en Guide to Advanced Empir ical Software Engineering, F. Shull, J. Singer y DIK Sjøberg, Eds. Londres, Reino Unido: Springer, 2008, págs. 35–62.

[90] K. Thompson, “Referencia de los usuarios a B”, Memorándum técnico interno de Bell Labs. [En línea]. Disponible: https://archive.org/details/users-ref-to-b , enero de 1972, MM-72-1271-1, presentación del caso 39199-11.

[91] L. Nyman y M. Laakso, "Notas sobre la historia de la bifurcación y unión", IEEE Ana. Computación histórica, vol. 38, núm. 3, págs. 84–87, julio de 2016.

[92] Estándar IEEE para tecnología de la información: especificaciones básicas de interfaz de sistema operativo portátil (POSIX), edición 7, estándar IEEE 1003.1–2017, 2017s.

[93] D. Libes y S. Ressler, Vida con UNIX. Englewood Cliffs, Nueva Jersey, EE. UU.: Prentice Hall, 1989.

[94] SC Johnson, “Un recorrido por el compilador C portátil”, en UNIX Programmer's Manual. Volumen 2—Documentos complementarios, 7ª ed. Murray Hill, NJ, EE. UU.: Bell Telephone Laboratories, 1979.

[95] SC Johnson y ME Lesk, "Herramientas de desarrollo del lenguaje", Sistema de campana tecnología J., vol. 56, núm. 6, págs. 2155–2176, julio/agosto. 1978.

[96] JL Bentley, "Perlas de programación: pequeños lenguajes", Commun. ACM, vol. 29, núm. 8, págs. 711–721, agosto de 1986.

[97] P. Hudak, “Lenguajes específicos de dominio”, en Manual de lenguajes de programación, vol. III: Pequeños lenguajes y herramientas, PH Salus, Ed. Indianápolis, IN, EE. UU.: Macmillan Technical Publishing, 1998.

[98] M. Fowler, Idiomas específicos de dominio. Boston, MA, EE. UU.: Addi hijo-Wesley, 2010.

[99] SR Bourne, "El shell de UNIX", Bell Syst. tecnología J., vol. 56, núm. 6, págs. 1971–1990, julio-agosto. 1978.

[100] AV Aho, BW Kernighan y PJ Weinberger, “Awk: un lenguaje de procesamiento y exploración de patrones”, Softw.: Practice Exp., vol. 9, núm. 4, págs. 267–280, 1979.

[101] LE McMahon, “SED: un editor de texto no interactivo”, en UNIX Programmer's Manual. Volumen 2—Documentos complementarios, 7ª ed. Murray Hill, NJ, EE. UU.: Bell Telephone Laboratories, 1979.

[102] BW Kernighan y DM Ritchie, “El procesador de macros M4”, en Manual del programador de UNIX. Volumen 2—Documentos complementarios, 7.ª ed. Murray Hill, NJ, EE. UU.: Bell Telephone Laboratories, 1979.

[103] SI Feldman, "Make: un programa para mantener programas de computadora", Softw.: Practice Exp., vol. 9, núm. 4, págs. 255–265, 1979.

[104] WN Joy, SL Graham, CB Haley, MK McKusick y PB Kessler, “Berkeley Pascal user's manual”, en UNIX Programmer's Manual—Volume 2c—Supplementary Documents: 4.2 Berkeley Software Distribution. Berkeley, CA, EE. UU.: Grupo de Investigación de Sistemas Informáticos, Departamento de Ingeniería Eléctrica y Ciencias de la Computación, Universidad de California, agosto de 1983.

[105] WN Joy y M. Horton, "Manual de referencia de Ex", en Manual del programador de UNIX, Volumen 2c, Documentos complementarios: 4.2 Distribución de software de Berkeley. Berkeley, CA, EE. UU.: Grupo de Investigación de Sistemas Informáticos, Departamento de Ingeniería Eléctrica y Ciencias de la Computación, Universidad de California, agosto de 1983.

[106] W. Joy, “Una introducción a la edición de pantallas con vi”, en UNIX Programmer's Manual —Volumen 2c—Documentos complementarios: 4.2 Berkeley Software Distribution. Berkeley, CA, EE. UU.: Grupo de Investigación de Sistemas Informáticos, Departamento de Ingeniería Eléctrica y Ciencias de la Computación, Universidad de California, agosto de 1983.
[107] W. Joy, “Una introducción al shell C”, en UNIX Programmer's Manual—Volumen 2c—Documentos complementarios: 4.2 Distribución de software de Berkeley. Berkeley, CA, EE. UU.: Grupo de Investigación de Sistemas Informáticos, Departamento de Ingeniería Eléctrica y Ciencias de la Computación, Universidad de California, agosto de 1983.

[108] JS Quarterman y JC Hoskins, “Notable computer networks,” Commun. ACM, vol. 29, núm. 10, págs. 932–971, octubre de 1986.
[109] WR Stevens, Programación avanzada en el entorno UNIX. Reading, MA, EE. UU.: Addison-Wesley, 1992.
[110] A. Hume, "Guerras Grep: La iniciativa de búsqueda estratégica", en Proc. EUUG Spring 88 Conf., 1988, págs. 237–245.
[111] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick y D. Mosher, “4.2BSD system manual,” en UNIX Programmer's Manual—Volume 2c—Supplementary Documents: 4.2 Berkeley Software Distribución. Berkeley, CA, EE. UU.: Grupo de Investigación de Sistemas Informáticos, Departamento de Ingeniería Eléctrica y Ciencias de la Computación, Universidad de California, agosto de 1983.

[112] P. Karn, “El paquete de Internet KA9Q (TCP/IP): un informe de progreso”, en Proc. Cómputo de la 6.ª ARRL. Neto. Conf., 1987, págs. 91–94.
[113] M. Seltzer y M. Olson, “LIBTP: Transacciones portátiles y modulares para UNIX”, en Proc. Invierno de 1992 USENIX Conf., enero de 1992, págs. 9–26.
[114] Manual de FreeBSD, revisión 47376 ed., The FreeBSD Documentation Project, octubre de 2015.
[115] TS Killian, “Procesos como archivos”, en Proc. USENIX Summer 84 Conf., 1984, págs. 203–207.
[116] G. Lehey, The Complete FreeBSD, 4.ª ed. Newton, MA, EE. UU.: O'Reilly Media, 2006.

[117] P.-H. Kamp y RNM Watson, “Cárceles: Confinar la raíz omnipotente”, en Proc. 2do Int. sist. Administración. Neto. Conf., mayo de 2000.
[118] D. Merkel, “Docker: Contenedores ligeros de Linux para un desarrollo y una implementación coherentes”, Linux J., vol. 2014, núm. 239, mayo de 2014.

[119] DE Bell y LJ LaPadula, “Sistemas informáticos seguros: fundamentos matemáticos”, Mitre Corp., Bedford, MA, EE. UU., Tech. Rep. MTR-2547, vol. 1, noviembre de 1973.
[120] KJ Biba, “Consideraciones de integridad para sistemas informáticos seguros”, Mitre Corp., Bedford, MA, EE. UU., Tech. Rep. MTR 3153, Rev. 1, abril de 1977.

[121] L. Rizzo, "Mapa de red: un marco novedoso para E/S de paquetes rápidos", en Proc. USENIX Anual. tecnología Conf., 2012, págs. 101–112.
[122] GE Moore, “Cramming more components on Integrated Circuits”, Electron., vol. 38, núm. 8, págs. 114–117, abril de 1965.
[123] D. Spinellis, “A tale of four kernels”, en Proc. 30 Int. Conf. suave Eng., mayo de 2008, págs. 381–390.
[124] H. v. Vliet, Ingeniería de software: principios y práctica, 3.ª ed. Hoboken, Nueva Jersey, EE. UU.: Wiley Publishing, 2008.
[125] C. Lattner y V. Adve, "LLVM: un marco de compilación para el análisis y la transformación de programas de por vida", en Proc. En t. Síntoma Optimización de generación de código, marzo de 2004.
[126] D. Sjøberg, G. Bergersen y T. Dyba, "Por qué importa la teoría", en Perspectivas sobre la ciencia de datos para la ingeniería de software, T. Menzies, L. Williams y T. Zimmermann, Eds. Boston, MA, EE. UU.: Morgan Kaufmann, 2016, págs. 29–33.

[127] DIK Sjøberg, T. Dyba, BCD Anda y JE Hannay, "Construyendo teorías en la ingeniería de software", en Guide to Advanced Empirical Software Engineering, F. Shull, J. Singer y DIK Sjøberg, Eds. Londres, Reino Unido: Springer, 2008, págs. 312–336.
[128] FP Brooks, El mes del hombre mítico. Reading, MA, EE. UU.: Addi hijo-Wesley, 1975.
[129] R. Love, Linux Kernel Development, 3.ª ed. Río Saddle Superior, Nueva Jersey, EE. UU.: Addison-Wesley, 2010.
[130] D. Bovet, Comprender el kernel de Linux, 3.ª ed. Sebastopol, CA, EE. UU.: O'Reilly, 2006.

[131] Tecnología de la información—Interfaz de sistema operativo portátil (POSIX)—Parte 1: Interfaz de programación de aplicaciones del sistema (API) (Lenguaje C) Norma ISO ISO/IEC 9945–1:1996, 1996 (IEEE/ ANSI Std 1003.1, edición de 1996).
[132] Tecnología de la información—Interfaz de sistema operativo portátil (POSIX)—Parte 2: Estándar ISO de Shell y utilidades ISO/IEC 9945–2:1993, 1993 (IEEE/ANSI Std 1003.2-1992 e IEEE/ANSI 1003.2a-1992).
[133] Estándar Nacional Estadounidense para Sistemas de Información—lenguaje de programación—C, Estándar ANSI ANSI X3.159–1989, diciembre de 1989, (también ISO/IEC 9899:1990).
[134] Lenguajes de programación—C Norma ISO ISO/IEC 9899:1999, 1999.
[135] Lenguajes de programación—C Norma ISO ISO/IEC 9899:2018, 2018.
[136] P. Avgeriou, P. Kruchten, I. Ozkaya y C. Seaman, "Gestión de la deuda técnica en ingeniería de software (seminario Dagstuhl 16162)", Dagstuhl Rep., vol. 6, núm. 4, págs. 110–138, 2016. [En línea]. Disponible: http://drops.dagstuhl.de/opus/volltexte/2016/6693 [137] M. Bagherzadeh, N. Kahani, C.-P. Bezemer, AE Hassan, J. Dingel y JR Cordy, "Análisis de una década de llamadas al sistema Linux", Empirical Softw. Ing., vol. 23, núm. 3, págs. 1519–1551, junio de 2018.

[138] DM Ritchie y K. Thompson, “El sistema de tiempo compartido UNIX,” Sistema de campana tecnología J., vol. 57, núm. 6, págs. 1905–1929, julio/agosto. 1978.
[139] WR Stevens, Programación de redes UNIX: API de redes: Sockets y XTI, vol. 1, 2ª ed. Englewood Cliffs, Nueva Jersey, EE. UU.: Prentice Hall, 1998.

[140] AS Tanenbaum, Sistemas operativos: diseño e implementación, 2ª ed. Englewood Cliffs, Nueva Jersey, EE. UU.: Prentice Hall, 1997.
[141] A. Lewis, AT&T resuelve caso antimonopolio; Acciones Patentes. Nueva York, NY, EE. UU.: New York Times, 25 de enero de 1956, págs. 1, 16.
[142] N. Takahashi y T. Takamatsu, “La licencia de UNIX hace que Linux sea la última pieza que falta del rompecabezas”, Ann. Autobús. administrar ciencia, vol. 12, págs. 123 a 137, 2013.

Diomidis Spinellis es profesor de ingeniería de software en el Departamento de Ciencia y Tecnología de Gestión de la Universidad de Economía y Negocios de Atenas, Grecia y director del Laboratorio de análisis empresarial de la universidad.

Es autor de dos libros galardonados, Code Reading y Code Quality: The Open Source Perspective. Su libro más reciente es Depuración efectiva: 66 formas específicas de depurar software y sistemas. Ha contribuido con el código que se envía con macOS de Apple y BSD Unix, y es el desarrollador de CScout, UMLGraph, dgsh y otros paquetes, bibliotecas y herramientas de software de código abierto. Se desempeñó como editor en jefe de IEEE Software durante el período 2015–2018. Es miembro sénior del IEEE.

Paris Avgeriou es profesor de ingeniería de software en la Universidad de Groningen, Países Bajos, donde dirige el grupo de investigación de ingeniería de software desde septiembre de 2006. Sus intereses de investigación se encuentran en el área de la arquitectura de software, con un fuerte énfasis en el modelado de arquitectura, el conocimiento , evolución, patrones y deuda técnica. Es editor en jefe del Journal of Systems and Software, así como editor asociado de IEEE Software. Ha coorganizado varias conferencias internacionales (por ejemplo, ECSA e ICSA) y talleres (principalmente en ICSE). Es miembro sénior del IEEE.