Lo que hacemos

Software libre y acceso abierto

Lo que hacemos: software *libre y acceso abierto*

Esta publicación fue realizada para el Festival Latinoamericano de Instalación de *Software* Libre celebrado en Colima Hacklab en abril del 2019.

Prólogo

Esta publicación fue elaborada para el Festival Latinoamericano de Instalación de *Software* Libre realizado en Colima Hacklab en abril del 2019. El motivo de esta obra es ofrecerte de primera mano algunos de los escritos «seminales» de los movimientos del *software* libre y del acceso abierto.

Se trata de una serie de textos publicados en contextos y años distintos pero con un mismo objetivo: eliminar las barreras al conocimiento. El primero de ellos es «El manifiesto de GNU», publicado en 1985. Entre otras cosas, este manifiesto es la semilla del discurso y acción del movimiento del *software* libre, por lo que es una de las piedras angulares para empezar a indagar sobre esta otra manera de desarrollar, distribuir, estudiar y compartir *software*.

En los noventas el movimiento del *software* libre tuvo una bifurcación: la iniciativa del código abierto. «La catedral y el bazar», publicado en 1997, sería su piedra de toque. En esta publicación se incluye para recalcar que el movimiento del *software* libre no es homogéneo, hasta el punto de llegar a desacuerdos o rupturas.

Pocos años después la búsqueda por la libertad y la apertura del *software* se expandería a la producción

del conocimiento científico. En 2002 se publica la «Iniciativa de Budapest para el acceso abierto» la cual hace pantente este llamado a lo que ahora se conoce como «acceso abierto».

El llamado de esta iniciativa ha causado diversas reacciones debido a la financiación que está detrás o por su carácter institucional. Por eso al unísono empezaron a existir otros movimientos en pos del «acceso abierto» pero en vertientes más radicales. El «Manifiesto de la guerrilla por el acceso abierto», publicado en 2008, es referencia primordial para entender otras maneras de abrir el conocimiento como lo es la piratería y la «violación» de los derechos de autor.

Esta radicalización del movimiento llevó a la gestación de diversas bibliotecas «piratas» o plataformas que permiten saltar muros de pago de repositorios académicos como Library Genesis y Sci-Hub. Esto ha llevado a grandes monopolios editoriales a emprender acciones legales. En este contexto, la carta «En solidaridad con Library Genesis y Sci-Hub» es un llamado a que como individuos o colectivos hagamos algo al respecto.

Por cuestión de tiempos, esta publicación tiene dos grandes obras ausentes: «Por una cultura libre» y el «Manifiesto telecomunista». La primera es el punto de partida para comprender lo que ahora llamamos «cultura libre» y su brazo más conocido: las licencias Creative Commons. Mientras que la segunda es una crítica y una radicalización de los fundamentos del *software* y la cultura libres, así como del acceso abierto, para combatir a la «hidra capi-

talista» en los territorios de las nuevas tecnologías de la información y la comunicación.

Pero como es común dentro de estos movimientos, ya tienes aquí las referencias para que puedas indagar más al respecto. Esta publicación es solo el comienzo de una odisea.

Con la compra de este libro diste un gesto de apoyo a espacios alternativos o autogestionados como es Colima Hacklab. Te damos gracias, no por el dinero, sino porque esta mínima acción nos da combustible para seguir con la lucha.

Un perro,
abril del 2019, Colima, México

El manifiesto de GNU

Richard Stallman

¿Qué es GNU? ¡GNU no es Unix!

GNU, que significa «GNU no es Unix», es el nombre del sistema de *software* completamente compatible con Unix que estoy escribiendo para entregarlo libremente a todas las personas que puedan utilizarlo.[1] Algunos voluntarios me están ayudando. Las aportaciones de tiempo, dinero, programas y equipos son muy necesarias.

Hasta el momento tenemos un editor de texto, Emacs con Lisp, para escribir comandos de edición, un depu-

[1] Esta expresión resultó poco precisa. La intención era decir que nadie tendría que pagar por el permiso para usar el sistema GNU. Pero la expresión no es del todo clara y a menudo se interpreta que las copias de GNU deberían distribuirse siempre a un costo bajo o sin costo. Esta nunca fue la intención. Más adelante, el manifiesto menciona la posibilidad de que las empresas provean servicios de distribución para obtener ganancias. A partir de entonces, aprendí a distinguir cuidadosamente entre «*free*» (libre) en el sentido de libertad y «*free*» (gratis) en referencia al precio [en inglés, el término «free» puede significar libertad o gratuidad; N. del T.]. El *software* libre es aquel que ofrece a los usuarios la libertad de distribuirlo y modificarlo. Algunos pueden obtener copias sin pagar, mientras que otros pagan para obtenerlas, y si los fondos ayudan a mejorar el *software* es mucho mejor. Lo importante es que todos los que posean una copia tengan la libertad de colaborar con los demás al usar el programa.

rador de código fuente, un generador *parser* compatible con Yacc, un enlazador y alrededor de treinta y cinco utilidades. Un *shell* (intérprete de comandos) que está casi terminado. Un nuevo compilador portable y optimizador de C se autocompiló y posiblemente lo publicaremos este año. Existe un núcleo inicial, pero se necesitan muchas más características para emular a Unix. Cuando el núcleo y el compilador estén completos, será posible distribuir un sistema GNU apropiado para el desarrollo de programas. Usaremos el formateador de documentos TeX, pero también estamos trabajando en una versión de Nroff. Usaremos también el Sistema de Ventanas X, libre y portable. Después de esto agregaremos un Common Lisp portable, un juego *Empire*, una hoja de cálculo y cientos de otras cosas, además de la documentación en línea. Esperamos proporcionar, con el tiempo, todas las utilidades que vienen normalmente con un sistema Unix y más.

GNU podrá ejecutar programas de Unix, pero no será idéntico a Unix. Haremos todas las mejoras que sean convenientes, con base en nuestra experiencia con otros sistemas operativos. Concretamente, planeamos tener nombres de archivos más largos, números para las versiones de los archivos, un sistema de archivos a prueba de fallas y tal vez incorporemos un sistema para completar los nombres de archivos, un soporte de visualización independiente de la terminal y, quizá en el futuro, un sistema de ventanas basado en Lisp a través del cual varios programas Lisp y programas

comunes de Unix podrán compartir una pantalla. Tanto C como Lisp estarán disponibles como lenguajes de programación del sistema. Intentaremos también dar soporte a UUCP, MIT Chaosnet y protocolos de internet para las comunicaciones.

GNU está orientado inicialmente a las máquinas de la clase 68000/16000 con memoria virtual, porque son las máquinas donde es más sencilla su ejecución. El esfuerzo adicional para hacerlo funcionar en máquinas más pequeñas se lo dejaremos a quienes quieran utilizarlo en ellas.

Para evitar una horrible confusión, por favor pronuncie la «g» en la palabra «GNU» cuando se refiera al nombre de este proyecto.[2]

Por qué debo escribir GNU

Considero que la regla de oro me exige que si me gusta un programa lo debo compartir con otras personas a quienes también les guste. Los vendedores de *software* quieren dividir a los usuarios y dominarlos para que nieguen el intercambio de su *software* con los demás. Me rehúso a romper la solidaridad con otros usuarios de esta manera. Mi conciencia me impide firmar un acuerdo de confidencialidad o un acuerdo de licencia de *software*. Durante años trabajé en el Laboratorio de Inteligencia Artificial oponiéndome a estas tendencias y

[2] GNU se pronuncia en inglés de forma muy similar a «*new*», que significa «nuevo». [N. del T.].

otras descortesías, pero al final fueron demasiado lejos: no podía permanecer en una institución donde tales cosas se hicieran en mi nombre y en contra de mi voluntad.

Para poder seguir utilizando las computadoras sin deshonra, he decidido agrupar un conjunto suficiente de *software* libre para poder vivir sin usar ningún *software* que no sea libre. He renunciado al Laboratorio de Inteligencia Artificial para evitar que el MIT pueda usar alguna excusa legal que me impida regalar *software* de GNU.[3]

Por qué GNU será compatible con Unix

Unix no es mi sistema ideal, pero no es tan malo. Las características esenciales de Unix parecen ser buenas y pienso que puedo añadir lo que le falta sin echarlas a perder. Y un sistema compatible con Unix facilitará su adopción por parte de muchas otras personas.

Cómo estará disponible GNU

GNU no está en el dominio público. Todos tendrán permiso para modificarlo y distribuirlo, pero a nadie se le permitirá restringir su redistribución. Es decir, no se autorizarán modificaciones privativas. Quiero asegurarme de que todas las versiones de GNU permanezcan libres.

[3] La expresión «regalar» es otro indicio de que yo todavía no había separado claramente la cuestión de la gratuidad de la cuestión de la libertad. Ahora recomendamos no usar esta expresión al hablar acerca del *software* libre. Para una explicación más detallada, véase el artículo «Palabras y frases confusas que vale la pena evitar».

Por qué muchos programadores quieren colaborar

He encontrado muchos programadores que están entusiasmados con GNU y quieren colaborar.

Muchos programadores están descontentos con la comercialización del *software* de sistema. Puede permitirles ganar más dinero, pero los hace sentirse en conflicto con otros programadores en lugar de sentirse como compañeros. El fundamento de la amistad entre programadores es el intercambio de programas, pero los acuerdos de mercadotecnia que los programadores suelen utilizar básicamente prohíben tratar a los demás como amigos. El comprador de *software* debe escoger entre la amistad y la obediencia a la ley. Naturalmente, muchos deciden que la amistad es más importante. Pero aquellos que creen en la ley a menudo no se sienten a gusto con ninguna de las opciones. Se vuelven cínicos y piensan que la programación es solo una manera de ganar dinero.

Al desarrollar y utilizar GNU en lugar de programas privativos, podemos ser hospitalarios con todos y obedecer la ley. Además, GNU sirve como ejemplo de inspiración y como bandera para animar a otros a unirse a nosotros en el intercambio. Esto puede darnos una sensación de armonía que es imposible obtener cuando utilizamos *software* que no es libre. Porque, para cerca de la mitad de los programadores con quienes hablo, esto es un importante motivo de felicidad que el dinero no puede reemplazar.

Cómo colaborar

Para conocer las tareas en las que puedes colaborar en el ámbito del software, *consulta la lista de proyectos prioritarios y la lista de ayuda requerida que indica las tareas en general para paquetes de* software *de* GNU. *Para ayudar de otras formas, consulta la guía para colaborar con el proyecto* GNU.

Pido a los fabricantes de ordenadores que donen máquinas y dinero. A los individuos les pido donaciones en forma de programas y trabajo.

Una de las consecuencias que puedes esperar si donas máquinas es que GNU se ejecutará en ellas con anticipación. Las máquinas deben estar completas, listas para utilizar sistemas, aprobadas para su uso en zonas residenciales y no requerir ventilación o fuentes de energía sofisticadas.

He encontrado muchos programadores ansiosos por contribuir para GNU mediante trabajo de medio tiempo. Para la mayoría de los proyectos, tal trabajo distribuido a medio tiempo sería muy difícil de coordinar: las partes escritas de forma independiente no funcionarían correctamente unidas. Pero para la tarea particular de reemplazar Unix, este problema no existe. Un sistema completo Unix contiene cientos de programas de utilidades, cada uno de los cuales se documenta por separado. La mayoría de las especificaciones de interfaz se fijan por compatibilidad con Unix. Si cada colaborador puede escribir un reemplazo compatible para una sola utilidad Unix y hace que funcione correctamente en lugar del original

en un sistema Unix, entonces estas utilidades funcionarán correctamente cuando se ensamblen. Aun teniendo en cuenta las leyes de Murphy acerca de algunos problemas inesperados, el montaje de estos componentes será una tarea factible (el núcleo requerirá una comunicación más estrecha y deberá trabajarse en un grupo pequeño y compacto).

Si obtengo más donaciones, podría contratar a algunas personas de tiempo completo o medio tiempo. El sueldo no será alto para los estándares de los programadores, pero estoy buscando a gente para la cual la construcción de un espíritu comunitario es tan importante como ganar dinero. Lo veo como una forma de permitir que estas personas se dediquen con todas sus energías a trabajar en GNU, ahorrándoles la necesidad de ganarse la vida de otra manera.

Por qué se beneficiarán todos los usuarios de computadoras

Cuando GNU esté terminado, todo el mundo podrá obtener un buen sistema de *software* tan libre como el aire.[4]

Esto significa mucho más que ahorrarse el dinero para pagar una licencia Unix. Significa evitar el derroche

[4] Aquí también omití distinguir cuidadosamente entre los dos diferentes significados de «*free*» [que en inglés puede significar «gratis» o «libre»; N. del T.]. La afirmación tal como está escrita no es falsa, se pueden obtener copias gratuitas de *software* de GNU —de los amigos o a través de internet–, pero sugiere una idea errónea.

inútil de la duplicación de esfuerzos en la programación de sistemas. En su lugar, este esfuerzo se puede invertir en el avance de la tecnología.

El código fuente del sistema completo estará disponible para todos. Como resultado, un usuario que necesite cambios en el sistema siempre será libre de hacerlo él mismo, o contratar a cualquier programador o empresa disponible para que los haga. Los usuarios ya no estarán a merced de un programador o de empresas propietarias de las fuentes, quienes son los únicos que pueden realizar modificaciones.

Las escuelas podrán ofrecer un entorno mucho más educativo y alentar a todos los alumnos a estudiar y mejorar el código. El laboratorio de computación de Harvard solía tener la política de que ningún programa podía ser instalado en el sistema si no se publicaba previamente su código fuente, llegando al punto de negarse a instalar ciertos programas. Yo me inspiré mucho en esa política.

Por último, el lastre de considerar quién es dueño de qué sistema de *software* y de lo que está o no está permitido hacer con él, habrá desaparecido.

Los acuerdos que obligan a la gente a pagar por usar un programa, incluyendo el licenciamiento de las copias, siempre incurren en un costo enorme para la sociedad a través de los mecanismos engorrosos necesarios para calcular la cantidad que debe pagar una persona (es decir, qué programas). Y solo un estado policial puede forzar a todos a obedecer. Considérese la posibilidad de una estación espacial en donde el aire debe fabricarse con

un gran costo: cobrar a cada persona por litro de aire puede ser justo, pero usar una máscara para medir el aire durante todo el día y toda la noche es insoportable, incluso si todo el mundo puede permitirse el lujo de pagar la factura por el aire. Y las cámaras de video en todas partes, para ver si alguna vez alguien se quita la máscara, son indignantes. Lo mejor es apoyar a la planta de aire con un impuesto y desechar las máscaras.

Copiar todo o parte de un programa es tan natural para un programador como respirar, además de productivo. Debería ser igual de libre.

Algunas objeciones fácilmente rebatibles a los objetivos de GNU

«Nadie lo usará si es libre, porque eso significa que no cuenta con ningún tipo de asistencia». «Hay que cobrar por el programa para pagar por el servicio de asistencia».

Si la gente prefiere pagar por el soporte de GNU en lugar de recibir GNU sin servicio, debe ser rentable una empresa que preste solamente esta clase de asistencia.[5]

Debemos distinguir entre el soporte en forma de trabajo de programación real y lo que es simplemente una guía al usuario. El primero es algo que uno no puede confiar a un proveedor de *software*. Si tu problema no es compartido por bastante gente, el vendedor no se preocupará por solucionarlo.

[5] Ya existen varias compañías de este tipo.

Si tu empresa necesita poder contar con soporte, la única manera es tener todo el código fuente y las herramientas necesarias. Entonces puedes contratar a cualquier persona disponible para corregir el problema, evitando estar a merced de algún individuo. Con Unix, el precio del código fuente deja fuera de consideración a la mayoría de las empresas. Con GNU esto será sencillo. Puede ser que no esté disponible ninguna persona competente, pero este problema no sería culpa de los acuerdos de distribución. GNU no elimina todos los problemas del mundo, solo algunos de ellos.

Mientras tanto, los usuarios que no saben nada acerca de las computadoras necesitan que los guíen: hacer cosas que fácilmente podrían hacer por sí mismos, pero que no saben cómo.

Estos servicios podrán ser prestados por empresas que vendan solamente el servicio de asesoría y de reparación. Si bien es cierto que los usuarios prefieren gastar dinero y obtener un producto con el servicio, también estarán dispuestos a adquirir el servicio al obtener el producto de forma gratuita. Las empresas de servicios competirán en calidad y precio, los usuarios no estarán atados a ninguna en particular. Mientras tanto, aquellos de nosotros que no necesitamos el servicio deberíamos tener la posibilidad de utilizar el programa sin tener que pagar por ello.

«No se puede llegar a muchas personas sin publicidad, y para financiarla es necesario cobrar por el programa». «No tiene sentido publicitar un programa que la gente puede obtener gratuitamente».

Hay diversas formas de publicidad gratuita o muy barata que se puede utilizar para informar a los usuarios de computadoras acerca de algo como GNU. Pero quizás sea cierto que uno puede llegar a más usuarios de microcomputadoras con publicidad. Si esto es realmente así, un negocio que publicite la contratación de un servicio de copiado y envío por correo del *software* de GNU debería ser lo suficientemente exitoso como para pagar por su publicidad y mucho más. De esta manera, solo los usuarios que se benefician de esta publicidad la pagarán.

Por otro lado, si mucha gente consigue GNU de sus amigos y esas empresas no tienen éxito, esto demostrará que la publicidad no era realmente necesaria para difundir GNU. ¿Por qué los defensores del libre mercado no quieren dejar que el libre mercado lo decida?[6]

«Mi compañía necesita un sistema operativo privativo para tener una ventaja competitiva».

GNU quitará el *software* de sistema operativo del entorno de la competencia. No podrá obtener una ventaja en esta

[6] Aunque es una organización sin ánimo de lucro más que una empresa, la Free Software Foundation (FSF) durante diez años ha obtenido la mayoría de los fondos a partir de su servicio de distribución. Puedes comprar artículos de la FSF para apoyar su actividad.

área, pero tampoco la competencia podrá tenerla frente a usted. Ambos competirán en otras áreas, mientras se benefician mutuamente en esta. Si su negocio consiste en vender un sistema operativo, no le gustará GNU, pero ese es su problema. Si su negocio es de otro ámbito, GNU puede salvarlo de ser empujado dentro del costoso negocio de la venta de sistemas operativos.

Me gustaría ver que el desarrollo de GNU se mantuviera gracias a donaciones de algunos fabricantes y usuarios, reduciendo el coste para todos.[7]

«¿No merecen los programadores una recompensa por su creatividad?».

Si hay algo que merece una recompensa, es la contribución social. La creatividad puede ser una contribución social, pero solo en la medida en que la sociedad sea libre de aprovechar los resultados. Si los programadores merecen ser recompensados por la creación de programas innovadores; entonces, por la misma razón merecen ser castigados si restringen el uso de estos programas.

«¿No debería un programador poder pedir una recompensa por su creatividad?».

No hay nada malo en querer un pago por el trabajo o en buscar maximizar los ingresos personales, siempre y cuando no se utilicen medios que sean destructivos. Pero

[7] Alrededor de 1991 un grupo de empresas de informática reunió fondos para apoyar el mantenimiento del compilador de C de GNU.

los medios habituales en el campo del *software* hoy en día se basan en la destrucción.

Extraer dinero de los usuarios de un programa limitando su uso es destructivo porque las restricciones reducen la cantidad y las formas en que el programa puede ser utilizado. Esto reduce la cantidad de beneficios que la humanidad obtiene del programa. Cuando hay una elección deliberada de restricción, las consecuencias dañinas son una destrucción deliberada.

La razón por la que un buen ciudadano no utiliza estos medios destructivos para volverse más rico es debido a que, si todos lo hicieran, nos empobreceríamos por mutua destrucción. Esto es ética kantiana o la regla de oro. Como no me gustan las consecuencias que resultarían si todos acapararan información, debo considerar como erróneo que alguien lo haga. Específicamente, el deseo de ser recompensado por la creatividad de uno no justifica privar al mundo de toda o parte de esa creatividad.

«¿No se morirán de hambre los programadores?».

Podría responder que nadie está obligado a ser programador. La mayoría de nosotros no puede conseguir dinero por hacer muecas en la calle. No estamos, por consiguiente, condenados a pasar nuestras vidas en la calle haciendo muecas y muriéndonos de hambre. Podemos dedicarnos a otra cosa.

Sin embargo, esta es una respuesta errónea porque acepta la suposición implícita del interrogador: que sin

la propiedad del *software* a los programadores no se les puede pagar un centavo. En este supuesto es todo o nada.

La verdadera razón por la que los programadores no se morirán de hambre es porque aún es posible que se les pague por programar, solo que no se les pagará tanto como en la actualidad.

Restringir la copia no es la única forma de hacer negocios con el *software*. Es la forma más común[8] porque es con la que se obtiene más dinero. Si se prohibiera o fuese rechazada por el comprador, el negocio del *software* se desplazaría hacia otras formas de organización que actualmente no se usan tan a menudo. Siempre existen muchos modos para organizar cualquier tipo de negocio.

Probablemente la programación no será tan lucrativa bajo esta nueva forma como lo es actualmente. Pero esto no es un argumento en contra del cambio. No se considera una injusticia que los empleados en los comercios obtengan los salarios que ganan actualmente. Si los programadores ganaran lo mismo, no sería tampoco una injusticia (en la práctica ganarán considerablemente más).

[8] Creo que me equivoqué al decir que el *software* privativo era la base más común para ganar dinero en el campo del *software*. Parece ser que en realidad el modelo de negocio más común era y es el desarrollo de *software* a medida, que no ofrece la posibilidad de percibir una renta, por lo que la empresa tiene que seguir haciendo el trabajo real para seguir recibiendo ingresos. El negocio del *software* a medida podrá seguir existiendo, más o menos igual, en un mundo de *software* libre. Por lo tanto, ya no supongo que los programadores ganarían menos en un mundo de *software* libre.

«¿La gente no tiene derecho a controlar cómo se usa su creatividad?».

El «control del uso de las ideas de alguien» realmente constituye el control de las vidas de otras personas y por lo general se utiliza para hacerles la vida más difícil.

Las personas que han estudiado cuidadosamente el tema de los derechos de propiedad intelectual[9] (por ejemplo, los abogados) dicen que no existe un derecho intrínseco a la propiedad intelectual. Los supuestos tipos de derechos de propiedad intelectual que reconoce el gobierno fueron creados mediante actos legislativos específicos con fines determinados.

Por ejemplo, el sistema de patentes se estableció para animar a los inventores a revelar los detalles de sus inventos. El objetivo era ayudar a la sociedad más que a los inventores. El periodo de validez de diecisiete años para una patente era corto comparado con el ritmo de desarrollo de la técnica. Dado que las patentes solo son relevantes para los fabricantes, para quienes el costo y el esfuerzo de un acuerdo de licencia son pequeños comparados con la

[9] En la década de los ochenta todavía no me había dado cuenta de lo confuso que era hablar de la «cuestión» de la «propiedad intelectual». Esa expresión es obviamente prejuiciosa, más sutil es el hecho de que agrupa leyes dispares que plantean cuestiones muy diferentes. Hoy en día insto a la gente a rechazar completamente el término «propiedad intelectual», para no inducir a otros a pensar que esas leyes forman un tema coherente. Para hablar con claridad, hay que referirse a las patentes, el *copyright* y las marcas registradas por separado. Véase el artículo «¿Ha dicho "propiedad intelectual"? Es solo un espejismo seductor» para ver cómo esta expresión genera confusión y prejuicios.

puesta en marcha de la producción, las patentes a menudo no hacen mucho daño. No representan un obstáculo para la mayoría de los individuos que usan productos patentados.

La idea del *copyright* no existía en tiempos antiguos, cuando los autores frecuentemente copiaban extensivamente a otros autores en obras de no ficción. Esta práctica era útil, y ha sido la única forma de que las obras de muchos autores, aunque solo sea en parte, hayan sobrevivido. El sistema de *copyright* se creó expresamente con el propósito de promover la autoría. En el ámbito para el que se inventó —libros, que solo podían copiarse de forma económica en una imprenta— hacía muy poco daño y no obstruía a la mayor parte de los individuos que leían los libros.

Todos los derechos de propiedad intelectual son solamente licencias otorgadas por la sociedad porque se pensaba, con razón o sin ella, que la sociedad en su conjunto se beneficiaría de su concesión. Pero, en cada situación particular, tenemos que preguntarnos: ¿nos beneficia realmente otorgar esta licencia?, ¿qué tipo de acto le estamos permitiendo hacer a una persona?

El caso de los actuales programas es muy diferente al de los libros de hace cien años. El hecho de que la forma más sencilla de copiar un programa sea de un vecino a otro, el hecho de que un programa esté formado tanto por el código fuente como el código objeto, siempre distintos, y el hecho de que el programa se use en lugar de leerlo y disfrutarlo, se combinan para crear una situación en la

que una persona que hace valer el *copyright* está dañando a la sociedad en su conjunto tanto materialmente como espiritualmente; nadie debería hacerlo a pesar de que la ley se lo permita.

«La competencia hace que las cosas se hagan mejor».

El paradigma de la competencia es una carrera: al premiar al ganador, estamos alentando a todos a correr más rápido. Cuando el capitalismo realmente funciona de esta manera, hace un buen trabajo; pero sus partidarios están equivocados al suponer que siempre funciona así. Si los corredores olvidan por qué se otorga el premio y se centran en ganar sin importar cómo, pueden encontrar otras estrategias, como atacar a los otros corredores. Si los corredores se enredan en una pelea a puñetazos, todos llegarán tarde a la meta.

El *software* privativo y secreto es el equivalente moral a los corredores en una pelea a puñetazos. Es triste decirlo, pero el único árbitro que tenemos no parece objetar las peleas, solo las regula («por cada diez metros que corras, puedes realizar un disparo»). Lo que debería hacer es separar y penalizar a los corredores, incluso por tratar de enredarse en una pelea.

«¿No dejarán todos de programar si no hay un incentivo económico?».

De hecho, mucha gente programará sin absolutamente ningún incentivo económico. La programación ejerce una

atracción irresistible en algunas personas, generalmente en quienes son los mejores en ese ámbito. No hay escasez de músicos profesionales que sigan en lo suyo aunque no tengan esperanzas de ganarse la vida de esa forma.

En realidad esta pregunta, aunque se formula muchas veces, no es adecuada para la situación. El pago a los programadores no va a desaparecer, solo se va a reducir. La pregunta correcta es: ¿alguien programará si se reduce el incentivo económico? Mi experiencia muestra que sí lo harán.

Por más de diez años, muchos de los mejores programadores del mundo trabajaron en el Laboratorio de Inteligencia Artificial por mucho menos dinero de lo que podrían haber obtenido en otro sitio. Tenían muchos tipos de recompensas que no eran económicas: fama y aprecio, por ejemplo. Y la creatividad también es divertida, es una recompensa en sí misma.

Luego, la mayoría se fue cuando se les ofreció la oportunidad de hacer ese mismo trabajo interesante por mucho dinero.

Lo que muestran los hechos es que la gente programa por razones distintas a la riqueza; pero si se les da la oportunidad de ganar también mucho dinero, eso los llenará de expectativas y lo van a exigir. Las organizaciones que pagan poco no podrán competir con las que pagan mucho, pero no tendría que irles tan mal si las que pagan mucho fueran prohibidas.

«Necesitamos a los programadores desesperadamente. Si ellos nos pidieran que dejemos de ayudar a nuestro prójimo, tendríamos que obedecer».

Uno nunca está tan desesperado como para tener que obedecer este tipo de exigencia. Recuerda: millones para nuestra defensa, pero ¡ni un centavo para tributos![10]

«Los programadores necesitan tener alguna forma de ganarse la vida».

A corto plazo, esto es verdad. Sin embargo, hay bastantes maneras en que los programadores pueden ganarse la vida sin vender el derecho a usar un programa. Esta manera actualmente es frecuente porque es la que les da a los programadores y hombres de negocios más dinero, no porque sea la única forma de ganarse la vida. Es fácil encontrar otras formas, si quieres encontrarlas. He aquí unos cuantos ejemplos:

Un fabricante que introduce una nueva computadora pagará por adecuar los sistemas operativos al nuevo *hardware*.

La enseñanza, así como los servicios de asistencia y de mantenimiento también pueden dar trabajo a programadores.

La gente con nuevas ideas podría distribuir programas como *freeware*,[11] pidiendo donaciones a los usuarios

[10] Véase el caso XYZ para más información sobre el contexto de esta sentencia. [N. del T.].

[11] Posteriormente aprendimos a distinguir entre «*software* libre»

satisfechos o vendiendo servicios de asistencia. Yo he conocido a personas que ya trabajan así y con mucho éxito.

Los usuarios que tengan las mismas necesidades pueden formar un grupo de usuarios y pagar sumas de dinero. Un grupo contratará a empresas de programación para escribir programas que a los miembros del grupo les gustaría utilizar.

Todo tipo de desarrollo puede ser financiado con un impuesto al *software*.

Supongamos que todos los que compren una computadora tengan que pagar un porcentaje de su precio como impuesto de *software*. El gobierno entrega este dinero a una agencia como la Fundación Nacional de las Ciencias (NSF, por sus siglas en inglés) para que lo empleé en el desarrollo de *software*.

Pero si el comprador de la computadora hace por sí mismo un donativo para el desarrollo de *software* puede verse exento de este impuesto. Puede donar al proyecto de su elección —a menudo, porque espera utilizar los resultados tan pronto como se haya completado–. Puede tomar crédito por cierta cantidad donada hasta la totalidad del impuesto que tendría que pagar.

La tasa total de impuesto podría decidirse mediante el voto de los contribuyentes, sopesada de acuerdo con la cantidad sobre la que se aplicará el impuesto.

---

y «*freeware*». El término «*freeware*» significa que el *software* se puede redistribuir libremente, pero por lo general no ofrece la libertad para estudiar y modificar el código fuente, así que la mayoría de esos programas no son *software* libre. Véase el artículo «Palabras y frases confusas que vale la pena evitar» para más detalles.

Las consecuencias:

1. La comunidad usuaria de computadoras apoyará el desarrollo de *software*.
2. Esta comunidad decidirá qué nivel de apoyo será necesario.
3. Los usuarios a quienes les importa a qué proyectos se destinará su parte así podrán escogerlos.

A largo plazo, hacer programas libres es un paso hacia el mundo sin escasez, donde nadie tendrá que trabajar demasiado duro solo para ganarse la vida. La gente será libre para dedicarse a actividades entretenidas, como la programación, después de haber dedicado diez horas obligatorias a la semana a las tareas requeridas como lo es la legislación, el asesoramiento familiar, la reparación de robots o la exploración de asteroides. No habrá necesidad de ganarse la vida mediante la programación.

Hemos alcanzado ya una gran reducción de la cantidad de trabajo que la sociedad en su conjunto debe realizar para mantener su productividad actual, pero solo un poco de esta reducción se ha traducido en descanso para los trabajadores, dado que hay mucha actividad no productiva que se requiere para acompañar a la actividad productiva. Las causas principales de esto son la burocracia y la fuerza isométrica contra la competencia. El *software* libre reducirá en gran medida estos drenajes en el campo de producción de *software*. Debemos hacerlo, para que los avances técnicos en la productividad se traduzcan en menos trabajo para nosotros.

La catedral y el bazar

Eric Steven Raymond

Analizo un exitoso proyecto de *software* libre (Fetchmail), que fue realizado para probar deliberadamente algunas sorprendentes ideas sobre la ingeniería de *software* sugeridas por la historia de Linux. Discuto estas teorías en términos de dos estilos de desarrollo fundamentalmente opuestos: el modelo catedral, de la mayoría de los fabricantes de *software* comercial, contra el modelo bazar, del mundo Linux. Demuestro que estos modelos parten de puntos de vista contrapuestos acerca de la naturaleza de la tarea de depuración de *software*. Posteriormente hago una argumentación a partir de la experiencia de Linux de la siguiente sentencia: «si se tienen las miradas suficientes, todas las pulgas saltarán a la vista». Al final, sugiero algunas fructíferas analogías con otros sistemas autorregulados de agentes individuales y concluyo con una somera exploración de las implicaciones que pude tener este enfoque en el futuro del *software*.

La catedral y el bazar

Linux es subversivo. ¿Quién hubiera pensado hace apenas cinco años que un sistema operativo de talla mundial sur-

giría, como por arte de magia, gracias a la actividad *hacker* desplegada en ratos libres por varios programadores diseminados en todo el planeta, conectados solamente por los tenues hilos del internet?

Lo que sí es seguro es que yo no. Cuando Linux apareció en mi camino, a principios de 1993, yo tenía invertidos en Unix y en el desarrollo de *software* libre alrededor de diez años. Fui uno de los primeros en contribuir con GNU a mediados de los ochenta y he estado aportando una buena cantidad de *software* libre a la red, desarrollando o colaborando en varios programas (NetHack, los modos Version Control y Grand Unified Debugger de Emacs, Xlife, entre otros) que todavía son ampliamente usados. Creí que sabía cómo debían hacerse las cosas.

Linux vino a trastocar buena parte de lo que pensaba que sabía. Había estado predicando durante años el evangelio Unix de las herramientas pequeñas, de la creación rápida de prototipos y de la programación evolutiva. Pero también creía que existía determinada complejidad crítica, por encima de la cual se requería un enfoque más planeado y centralizado. Yo pensaba que el *software* de mayor envergadura (sistemas operativos y herramientas realmente grandes, tales como Emacs) requería construirse como las catedrales; es decir, que debía ser cuidadosamente elaborado por genios o pequeñas bandas de magos trabajando encerrados a piedra y lodo, sin liberar versiones beta antes de tiempo.

El estilo de desarrollo de Linus Torvalds («libere rápido y a menudo, delegue todo lo que pueda, sea abierto

hasta el punto de la promiscuidad») me cayó de sorpresa. No se trataba de ninguna forma reverente de construir la catedral. Al contrario, la comunidad Linux se asemejaba más a un bullicioso bazar de Babel, colmado de individuos con propósitos y enfoques dispares (fielmente representados por los repositorios de archivos de Linux, que pueden aceptar aportaciones de quien sea), de donde surgiría un sistema estable y coherente únicamente a partir de una serie de artilugios.

El hecho de que este estilo de bazar pareciera funcionar, y hacerlo bien, realmente me dejó sorprendido. A medida que iba aprendiendo a moverme, no solo trabajé arduamente en proyectos individuales: también traté de comprender por qué el mundo Linux no naufragaba en el mar de la confusión, pues se fortalecía con una rapidez inimaginable para los constructores de catedrales.

A mediados de 1996 creí empezar a comprender. El destino me dio un medio perfecto para demostrar mi teoría, en forma de un proyecto de *software* libre que trataría de realizar siguiendo el estilo bazar de manera consciente. Así lo hice y resultó un éxito digno de consideración.

En el resto de este artículo relataré la historia de este proyecto y la usaré para proponer algunos aforismos sobre el desarrollo real de *software* libre. No todas estas cosas fueron aprendidas del mundo Linux, pero veremos cómo fue que este les vino a otorgar un sentido particular. Si estoy en lo cierto, te servirán para comprender mejor qué es lo que hace a la comunidad linuxera tan buena fuente de *software* y te ayudarán a ser más productivo.

El correo tenía que llegar

Desde 1993 he estado encargado de la parte técnica de un pequeño proveedor de servicios de internet (ISP, por sus siglas en inglés) de acceso gratuito llamado Chester County InterLink (CCIL) en West Chester, Pensilvania (fui uno de los fundadores de CCIL y escribí su original *software* BBS multiusuario que actualmente soporta más de tres mil usuarios en diecinueve líneas). Este empleo me permitió tener acceso a la red las veinticuatro horas del día a través de la línea de 56k de CCIL; de hecho, ¡el trabajo prácticamente me lo demandaba!

Para ese entonces ya me había habituado al correo electrónico. Por diversas razones fue difícil obtener un Serial Line Internet Protocol (SLIP) para enlazar mi máquina en casa con CCIL. Cuando finalmente lo logré, encontré que era particularmente molesto tener que entrar desde Telnet a Locke continuamente para revisar mi correo. Lo que quería era que se reenviara a Snark para recibir notificaciones cuando me llegara y poder manejarlo usando mis herramientas locales.

Un simple redireccionamiento con Sendmail no iba a funcionar porque Snark no siempre está en línea y no tiene una IP estática. Necesitaba un programa que saliera por mi conexión SLIP y trajera el correo hasta mi máquina. Yo sabía que tales programas ya existían y que la mayoría usaba un protocolo simple llamado Protocolo de Oficina de Correos (POP), así que me cercioré de que el servidor POP3 estuviera en el sistema operativo BSD/OS de Locke.

Necesitaba un cliente POP3, de tal manera que lo busqué en la red y encontré uno. En realidad hallé tres o cuatro. Usé POP Perl durante un tiempo, pero le faltaba una característica a todas luces evidente: la capacidad de identificar las direcciones de los correos recuperados para poder contestarlos correctamente.

El problema era este: supongamos que un tal Monty en Locke me enviaba un correo. Si yo lo hacía llegar desde Snark y luego intentaba responder, entonces mi programa de correos dirigía la respuesta a un Monty inexistente en Snark. En poco tiempo, la edición manual de las direcciones de respuesta para pegarles el @ccil.org se volvió algo muy molesto.

Era evidente que la computadora tenía que hacer esto por mí. (De hecho, de acuerdo con RFC 1123, sección 5.2.18, Sendmail tenía que hacerlo). Sin embargo, ¡ninguno de los clientes POP lo hacía realmente! Esto nos lleva a la primera lección:

> **1. Todo buen trabajo de *software* comienza a partir de las necesidades personales de quien programa. (Todo buen trabajo empieza cuando uno tiene que rascarse su propia comezón).**

Esto podría sonar muy obvio: el viejo proverbio dice que «la necesidad es la madre de todos los inventos». Empero, hay muchos programadores de *software* que gastan sus días a cambio de un salario en programas que ni necesitan ni quieren. No ocurre lo mismo en el mundo Linux,

lo cual sirve para explicar por qué la calidad promedio de *software* es tan alta en esa comunidad.

Por todo esto, ¿pensarán que me lancé inmediatamente a la vorágine de escribir a partir de cero el programa de un nuevo cliente POP3 que compitiese con los existentes? ¡Nunca en la vida! Revisé cuidadosamente las herramientas POP que tenía al alcance, preguntándome «¿cuál se aproxima más a lo que yo necesito?», porque:

> **2. Los buenos programadores saben qué escribir; mientras que los mejores, qué rescribir (y reutilizar).**

Aunque no presumo de ser un gran programador, trato de imitarlos. Una importante característica de los grandes programadores es la meticulosidad con la que construyen. Saben que les pondrán diez, no por el esfuerzo sino por los resultados, y que casi siempre será más fácil partir de una buena solución parcial que desde cero.

Linus, por ejemplo, no intentó escribir Linux desde cero. En vez de eso, comenzó por reutilizar el código y las ideas de Minix, un pequeño sistema operativo tipo Unix, hecho para máquinas 386. Eventualmente terminó desechando o rescribiendo todo el código de Minix, pero mientras contó con él le sirvió como una importante plataforma de lanzamiento para el proyecto en gestación que posteriormente se convertiría en Linux.

Con ese espíritu comencé a buscar una herramienta POP que estuviese razonablemente bien escrita, para usarla como plataforma inicial de desarrollo.

La tradición del mundo Unix de compartir las fuentes siempre se ha prestado a la reutilización del código (esta es la razón por la que el proyecto GNU escogió a Unix como su sistema operativo base, pese a las serias reservas que se tenían). El mundo Linux ha asumido esta tradición hasta llevarla muy cerca de su límite tecnológico; posee terabits de código fuente que están generalmente disponibles. Por eso es que la búsqueda de algo bueno tiene mayores probabilidades de éxito en el mundo Linux que en ningún otro lado.

Así sucedió en mi caso. Además de los que había encontrado antes, en mi segunda búsqueda conseguí un total de nueve candidatos: Fetchpop, PopTart, Getmail, Gwpop, Pimp, POP Perl, Popc, Popmail y Upop. El primero que elegí fue Fetchpop, un programa de Seung-Hong Oh. Le agregué mi código para que tuviera la capacidad de rescribir los encabezados y varias mejoras más, las cuales fueron incorporadas por el propio autor en la versión 1.9.

Sin embargo, unas semanas después me topé con el código fuente de Popclient, escrito por Carl Harris, y descubrí que tenía un problema. Pese a que Fetchpop poseía algunas ideas originales (como su modo *daemon*), solo podía manejar POP3 y fue escrito de manera *amateur* (Seung-Hong era un brillante programador pero no tenía experiencia, y ambas características eran palpables). El código de Carl era mejor, bastante profesional y robusto, pero su programa carecía de varias de las características importantes del Fetchpop que eran difíciles de implementar (incluyendo las que yo mismo había agregado).

¿Seguía o cambiaba? Cambiar significaba desechar el código que había añadido a cambio de una mejor base de desarrollo.

Un motivo práctico para cambiar fue la necesidad de contar con soporte de múltiples protocolos. POP3 es el protocolo de servidor de correos que más se utiliza, pero no es el único. Ni Fetchpop ni otros manejaban POP2, RPOP o APOP, y yo tenía ya la idea vaga de añadir el Protocolo de Acceso a Mensajes por Internet (IMAP) solo por entretenimiento.

Pero había una razón más teórica para pensar que el cambio podía ser una buena idea, algo que aprendí mucho antes de Linux:

> **3. «Contempla desecharlo; de todos modos tendrás que hacerlo».**

Diciéndolo de otro modo: no se entiende cabalmente un problema hasta que se implementa la primera solución. La siguiente ocasión quizá uno ya sepa lo suficiente para solucionarlo. Así que, si quieres resolverlo, disponte a empezar de nuevo al menos una vez.

«Bien —me dije— los cambios a Fetchpop fueron un primer intento, así que cambio».

Después de enviarle mi primera serie de mejoras a Carl Harris el 25 de junio de 1996, me enteré de que él había perdido el interés por Popclient desde hacía ya un tiempo. El programa estaba un poco abandonado, polvoriento y con algunas pulgas menores colgando. Como se le tenían que hacer varias correcciones, pronto

acordamos que lo más lógico era que yo asumiera el control del proyecto.

Sin darme cuenta, el proyecto había alcanzado otras dimensiones. Ya no estaba intentando hacerle unos cuantos cambios menores a un cliente POP, sino que me había hecho responsable de uno y las ideas que bullían en mi cabeza me conducirían probablemente a cambios mayores.

En una cultura del *software* que estimula a compartir el código fuente, esta era la forma natural de que el proyecto evolucionara. Yo actuaba de acuerdo con lo siguiente:

> **4. Si tienes la actitud adecuada, encontrarás problemas interesantes.**

Pero la actitud de Carl Harris fue aún más importante. Él entendió que:

> **5. Cuando se pierde el interés en un programa, el último deber es heredarlo a un sucesor competente.**

Sin siquiera discutirlo, Carl y yo sabíamos que el objetivo común era obtener la mejor solución. La única duda entre nosostros era si yo podía probar que el proyecto iba a quedar en buenas manos. Una vez que lo hice, él actuó de buena gana y con diligencia. Espero comportarme igual cuando llegue mi turno.

La importancia de contar con usuarios

Así es como heredé Popclient. Además, recibí su base de usuarios, lo cual fue igual o más importante. Tener usuarios es maravilloso. No solo porque prueban que uno está satisfaciendo una necesidad o que se ha hecho algo bien, sino porque, cultivados adecuadamente, pueden convertirse en magníficos asistentes.

Otro aspecto importante de la tradición Unix, que Linux nuevamente lleva al límite, es que muchos de los usuarios son también *hackers*, y al estar disponible el código fuente se vuelven *hackers* muy efectivos. Esto puede resultar tremendamente útil para reducir el tiempo de depuración de los programas. Con un buen estímulo, los usuarios diagnosticarán problemas, sugerirán correcciones y ayudarán a mejorar los programas mucho más rápido de lo que uno lo haría sin ayuda.

> **6. Tratar a los usuarios como colaboradores es la forma más apropiada de mejorar el código, y la más efectiva de depurarlo.**

Suele ser fácil subestimar el poder de este efecto. De hecho, muchos infravalorábamos la capacidad multiplicadora que se adquiere con el número de usuarios y que reduce la complejidad de los sistemas, hasta que Linus demostró lo contrario.

En realidad considero que la genialidad de Linus no radica en la construcción misma del *kernel* de Linux, sino en la invención del modelo de desarrollo de Linux.

Cuando en una ocasión expresé esta opinión delante de él, sonrió y repitió quedito una frase que ha dicho muchas veces: «Básicamente soy una persona muy floja a quien le gusta obtener el crédito por lo que realmente hacen los demás». Flojo como un zorro. O, como diría Robert Heinlein, demasiado flojo para fallar.

En retrospectiva, un precedente de los métodos y el éxito que tiene Linux podría encontrarse en el desarrollo de las bibliotecas del Emacs GNU, así como los archivos del código de Lisp. En contraste con el estilo catedral de construcción del núcleo del Emacs escrito en C, y de muchas otras herramientas de la Free Software Foundation (FSF), la evolución del código de Lisp fue bastante fluida y, en general, dirigida por los propios usuarios. Las ideas y los prototipos de los modos se rescribían tres o cuatro veces antes de alcanzar su forma estable final, mientras que las frecuentes colaboraciones informales se hacían posibles gracias al internet, al estilo Linux.

Es más, uno de mis programas con mayor éxito antes de Fetchmail fue probablemente el modo Version Control (VC) para Emacs, una colaboración tipo Linux, que realicé por correo electrónico conjuntamente con otras tres personas, de las cuales solamente he conocido a una (Richard Stallman) hasta la fecha. VC era una *frontend* para Source Code Control System (SCCS), Revision Control System (RCS) y posteriormente Concurrent Versions System (CVS), que ofrecía operaciones de control de versiones de manera directa desde Emacs. Era el desarrollo de un pequeño y hasta cierto punto rudimentario modo sccs.el

que alguien más había escrito. El desarrollo de VC tuvo éxito porque, a diferencia del Emacs mismo, el código de Emacs en Lisp podía pasar por el ciclo de publicación, prueba y depuración muy rápidamente.

(Uno de los efectos colaterales de la política de la FSF de atar legalmente el código a la General Public License [GPL] fue su dificultad para usar el modo bazar, debido a la idea de que se debían de asignar derechos de autor por cada contribución individual de más de veinte líneas, con la finalidad de inmunizar el código protegido por la GPL de cualquier problema legal surgido de la ley de derechos de autor. Los usuarios de las licencias BSD o MIT no tienen este problema, debido a que no intentan reservarse derechos que difícilmente alguien más intentaría poner en duda).

Libera rápido y a menudo

Las publicaciones rápidas y frecuentes del código constituyen una parte crítica del modelo Linux de desarrollo. La mayoría de los programadores (incluyéndome), creía antes que esta era una mala práctica para proyectos que no fueran triviales, debido a que las versiones de prueba, casi por definición, suelen estar plagadas de errores y a nadie le gusta agotar la paciencia de los usuarios.

Esta idea reafirmaba la preferencia de los programadores por el estilo catedral de desarrollo. Si el objetivo principal era que los usuarios vieran la menor cantidad de errores, entonces solo había que liberar una vez cada

seis meses (o aun con menos frecuencia) y trabajar como perro en la depuración de las versiones que salieran a la luz. El núcleo del Emacs escrito en C se desarrolló de esta forma. No así la biblioteca de Lisp, ya que los repositorios de sus archivos donde se podían conseguir versiones nuevas y en desarrollo del código, independientemente del ciclo de desarrollo del Emacs, estaban fuera del control de la FSF.

El más importante de estos archivos fue el Elisp de la Universidad Estatal de Ohio, el cual se anticipó al espíritu y a muchas de las características de los grandes archivos actuales de Linux. Pero solamente algunos de nosotros reflexionamos realmente acerca de lo que estábamos haciendo, o de lo que la simple existencia del archivo sugería sobre los problemas implícitos en el modelo catedral de la FSF. Yo realicé un intento serio, alrededor de 1992, de unir formalmente buena parte del código de Ohio con la biblioteca Lisp oficial del Emacs. Me metí en problemas políticos muy serios y no tuve éxito.

Pero un año después, a medida que Linux se agigantaba, quedó claro que estaba pasando algo distinto y mucho más sano. La política abierta de desarrollo de Linus era lo más opuesto a la construcción estilo catedral. Los repositorios de archivos en SunSITE y TSX-11 mostraban una intensa actividad y muchas distribuciones de Linux circulaban. Y todo esto se manejaba en la publicación de programas con una frecuencia que no tenía precedentes.

Linus estaba tratando a sus usuarios como colaboradores de la forma más efectiva posible:

7. Libera rápido. Libera a menudo. Y escucha a tus clientes.

La innovación de Linus no consistió tanto en esto (algo parecido había venido sucediendo en la tradición del mundo Unix desde hacía tiempo), sino en llevarlo a un nivel de intensidad acorde a la complejidad de lo que estaba desarrollando. En ese entonces no era raro que liberara una nueva versión del *kernel* ¡más de una vez al día! Y, debido a que cultivó su base de desarrolladores y buscó colaboración en internet más intensamente que ningún otro, funcionó.

¿Pero cómo fue que funcionó? ¿Era algo que yo podía emular o se debía a la genialidad única de Linus?

No lo considero así. Está bien, Linus es un *hacker* endiabladamente astuto (¿cuántos de nosotros podríamos diseñar un *kernel* de alta calidad?). Pero Linux en sí no representa ningún salto conceptual sorprendente. Linus no es (al menos no hasta ahora) un genio innovador del diseño como lo son Richard Stallman o James Gosling. En realidad, para mí Linus es un genio de la ingeniería; tiene un sexto sentido para evitar los callejones sin salida en el desarrollo o la depuración, y es muy sagaz para encontrar el camino con el mínimo esfuerzo desde el punto A hasta el punto B. De hecho, todo el diseño de Linux transpira esta calidad y refleja un Linus conservador que simplifica el enfoque en el diseño.

Por lo tanto, si las publicaciones frecuentes del código y la búsqueda de asistencia en internet no son accidentes,

sino partes integrales del ingenio de Linus para ver la ruta crítica del mínimo esfuerzo, ¿qué era lo que estaba maximizando? ¿Qué era lo que estaba exprimiendo de la maquinaria?

Planteada de esta forma, las pregunta se responde por sí sola. Linus estaba manteniendo a sus usuarios-*hackers*-colaboradores constantemente estimulados y recompensados por la perspectiva de tomar parte en la acción y satisfacer su ego, premiado con la exhibición y mejora constante, casi diaria, de su trabajo.

Linus apostaba claramente a maximizar el número de horas por persona invertidas en la depuración y el desarrollo, a pesar del riesgo que corría de volver inestable el código y agotar a la base de usuarios si un error serio resultaba insondable. Linus se portaba como si creyera en algo como esto:

> **8. Dada una base suficiente de colaboradores y *beta testers*, casi cualquier problema puede ser identificado rápidamente y su solución será obvia al menos para alguien.**

O, dicho de manera menos formal, «con muchas miradas, todos los errores saltarán a la vista». Esto lo he bautizado como la Ley de Linus.

Mi formulación original rezaba que todo problema deberá ser transparente para alguien. Linus descubrió que las personas que entendían y las que resolvían un problema no eran necesariamente las mismas, ni siquiera en la mayoría de los casos. Decía que «alguien encuentra

el problema y otro lo resuelve». Pero el punto está en que ambas cosas suelen suceder con gran rapidez.

Aquí, pienso, subyace una diferencia esencial entre el estilo bazar y el catedral. En el enfoque estilo catedral de la programación, los errores y problemas de desarrollo son fenómenos truculentos, insidiosos y profundos. Generalmente toma meses de revisión exhaustiva para unos cuantos el alcanzar la seguridad de que han sido eliminados del todo. Por eso se dan los intervalos tan largos entre cada versión que se libera, al igual que la inevitable desmoralización cuando estas versiones, largamente esperadas, no resultan perfectas.

En el enfoque de programación estilo bazar, por otro lado, se asume que los errores son fenómenos relativamente evidentes o, por lo menos, que pueden volverse relativamente evidentes cuando se exhiben a miles de desarrolladores entusiastas que colaboran en cada una de las versiones. En consecuencia, se libera con frecuencia para poder obtener una mayor cantidad de correcciones, logrando como efecto colateral benéfico el perder menos cuando un obstáculo se atraviesa.

Y eso es todo. Con eso basta. Si la Ley de Linus fuera falsa, entonces cualquier sistema que sea lo suficientemente complejo, como el *kernel* de Linux que está siendo manipulado por tantos, debería haber colapsado en algún punto bajo el peso de ciertas interacciones imprevistas y errores «muy profundos» inadvertidos. Pero si es cierta, bastaría para explicar la relativa ausencia de errores en el código de Linux.

Después de todo, esto no debe parecernos tan sorpresivo. Hace algunos años los sociólogos descubrieron que la opinión promedio de un numero grande de observadores igualmente expertos (o igualmente ignorantes) es más confiable de predecir que la de uno de los observadores seleccionado al azar. A esto se le conoce como el método Delphi. Al parecer, lo que Linus ha demostrado es que esto también es valedero en el ámbito de la depuración de un sistema operativo: que el método Delphi puede abatir la complejidad implícita en el desarrollo, incluso al nivel asociado al núcleo de un sistema operativo.

Estoy en deuda con Jeff Dutky, quien me sugirió que la Ley de Linus puede replantearse diciendo que «la depuración puede hacerse en paralelo». Jeff señala que a pesar de que la depuración requiere que los participantes se comuniquen con un programador que coordina el trabajo, no demanda ninguna coordinación significativa entre ellos. Por lo tanto, no cae víctima de la asombrosa complejidad cuadrática ni de los costos de maniobra que ocasionan que la incorporación de desarrolladores resulte problemática.

En la práctica, la pérdida teórica de eficiencia debido a la duplicación del trabajo por parte de los programadores casi nunca es un tema que revista importancia en el mundo Linux. Un efecto de la «política de liberar rápido y a menudo» es que esta clase de duplicaciones se minimizan al propagarse las correcciones rápidamente.

Brooks hizo una observación relacionada con la de Jeff: «El costo total del mantenimiento de un programa

muy usado es típicamente alrededor del cuarenta por ciento o más del costo del desarrollo. Sorpresivamente, este costo está fuertemente influenciado por el número de usuarios. *Más usuarios detectan una mayor cantidad de errores*». (El énfasis es mío).

Una mayor cantidad de usuarios detecta más errores debido a que tienen diferentes maneras de evaluar el programa. Este efecto se incrementa cuando los usuarios son colaboradores. Cada uno se enfoca a la tarea de la caracterización de los errores con un bagaje conceptual y con instrumentos analíticos distintos, desde un ángulo diferente. El método Delphi parece funcionar precisamente debido a estas diferencias. En el contexto específico de la depuración, dichas diferencias también tienden a reducir la duplicación del trabajo.

Por lo tanto, el agregar más *beta testers* podría no contribuir a reducir la complejidad del «más profundo» de los errores actuales, desde el punto de vista del desarrollador, pero aumenta la probabilidad de que la caja de herramientas de alguno de ellos se equipare al problema, de manera que esa persona vea claramente el error.

Linus también dobla sus apuestas. En el caso de que realmente existan errores serios, las versiones del *kernel* de Linux son enumeradas de tal manera que los usuarios potenciales puedan escoger la última versión considerada como «estable» o ponerse al filo de la navaja y arriesgarse a los errores con tal de aprovechar las nuevas características. Esta táctica no ha sido formalmente imitada por la mayoría de los *hackers* de Linux, pero quizá

deberían hacerlo. El hecho de contar con ambas opciones lo vuelve aun más atractivo.

¿Cuándo una rosa no es una rosa?

Después de estudiar la forma en que actuó Linus y de haber formulado una teoría sobre por qué tuvo éxito, tomé la decisión consciente de probarla en mi nuevo proyecto (el cual, debo admitirlo, es mucho menos complejo y ambicioso).

Lo primero que hice fue reorganizar y simplificar Popclient. El trabajo de Carl Harris era muy bueno, pero exhibía una complejidad innecesaria, típica de muchos de los programadores en C. Él trataba el código como la parte central y las estructuras de datos como un apoyo para este. Como resultado, el código resultó muy elegante, pero el diseño de las estructuras de datos quedó descuidado y feo (por lo menos con respecto a los estándares exigentes de este viejo *hacker* de Lisp).

Sin embargo, tenía otro motivo para rescribir, además de mejorar el diseño de la estructura de datos y el código: el proyecto debía evolucionar en algo que yo entendiera cabalmente. No es nada divertido ser el responsable de corregir los errores en un programa que no se entiende.

Por lo tanto, durante el primer mes, o algo así, simplemente fui siguiendo los pormenores del diseño básico de Carl. El primer cambio serio que realicé fue agregar el soporte de IMAP. Lo hice reorganizando los administradores de protocolos en un administrador genérico con tres

tablas de métodos (para POP2, POP3 e IMAP). Este y algunos cambios anteriores muestran un principio general que es bueno que los programadores tengan en mente, especialmente los que programan en lenguajes tipo C y no manejan estructuras de datos dinámicas:

> **9. Las estructuras de datos inteligentes y el código burdo funcionan mucho mejor que en el caso inverso.**

De nuevo Fred Brooks: «Muéstrame tu código y esconde tus estructuras de datos, y continuaré intrigado. Muéstrame tus estructuras de datos y generalmente no necesitaré ver tu código: resultará evidente».

En realidad, él hablaba de «diagramas de flujo» y «tablas». Pero con treinta años de cambios terminológicos y culturales resulta prácticamente la misma idea.

En este momento (a principios de septiembre de 1996, aproximadamente seis semanas después de haber comenzado) empecé a pensar que un cambio de nombre podría ser apropiado. Después de todo, ya no se trataba de un simple cliente POP. Pero todavía vacilé, debido a que no había nada nuevo y genuinamente mío en el diseño. Mi versión de Popclient aún tenía que desarrollar una identidad propia.

Esto cambió radicalmente cuando Fetchmail aprendió a remitir el correo recibido al puerto del protocolo para transferencia simple de correo (SMTP). Volveré a este punto en un momento. Primero quiero decir lo siguiente: yo afirmé anteriormente que decidí utilizar este proyecto

para probar mi teoría sobre qué había hecho bien Linus Torvalds. ¿Cómo lo hice?, podrían ustedes preguntar. Fue de la siguiente manera:

1. Liberaba rápido y a menudo (casi nunca dejé de hacerlo en periodos menores a diez días; durante las etapas de desarrollo intenso, una vez al día).
2. Ampliaba mi lista de analistas de versiones beta o *beta testers*, incorporando a toda persona que me contactara para saber sobre Fetchmail.
3. Efectuaba anuncios espectaculares a esta lista cada vez que liberaba una nueva versión, estimulando a la gente a participar.
4. Escuchaba a mis *beta testers*, consultándoles decisiones referentes al diseño y tomándolos en cuenta cuando me mandaban sus mejoras o retroalimentación.

La recompensa por estas simples medidas fue inmediata. Desde el principio del proyecto obtuve reportes de errores de calidad, frecuentemente con buenas soluciones anexas que envidiarían la mayoría de los desarrolladores. Obtuve crítica constructiva, mensajes de admiradores e inteligentes sugerencias. Lo que lleva a la siguiente lección:

> **10. Si tratas a tus analistas (*beta testers*) como si fueran tu recurso más valioso, ellos te responderán convirtiéndose en tu recurso más valioso.**

Una medida interesante del éxito de Fetchmail fue el tamaño de la lista de analistas beta del proyecto, los amigos de Fetchmail. Cuando escribí esto tenía 249 miembros y se sumaban entre dos o tres semanalmente.

Revisándola hoy, a finales de mayo de 1997, la lista ha comenzando a perder miembros debido a una razón sumamente interesante. Varias personas me han pedido que las dé de baja debido a que Fetchmail les está funcionando tan bien ¡que ya no necesitan ver todo el tráfico de la lista! A lo mejor esto es parte del ciclo vital normal de un proyecto maduro, realizado por el método de construcción estilo bazar.

Popclient se convierte en Fetchmail

El momento crucial fue cuando Harry Hochheiser me mandó su código fuente para incorporar la remisión del correo recibido a la máquina cliente a través del puerto SMTP. Comprendí casi inmediatamente que una implementación adecuada de esta característica iba a dejar todos los demás métodos a un paso de ser obsoletos.

Durante muchas semanas había estado perfeccionando Fetchmail, agregándole características a pesar de que sentía que el diseño de la interfaz era útil pero algo burdo, poco elegante y con demasiadas opciones insignificantes colgando fuera de lugar. La facilidad de vaciar el correo recibido a un buzón de correos o la salida estándar me incomodaba de cierta manera, pero no alcanzaba a comprender por qué.

Lo que advertí cuando me puse a pensar sobre la expedición del correo por el SMTP fue que Popclient estaba intentando hacer demasiadas cosas juntas. Había sido diseñado para funcionar al mismo tiempo como un agente de transporte (MTA) y un agente de entrega (MDA). Con la remisión del correo por el SMTP podría abandonar la función de MDA y centrarme solamente en la de MTA, mandando el correo a otros programas para su entrega local, justo como lo hace Sendmail.

¿Por qué sufrir con toda la complejidad de configurar el agente de entrega o realizar un bloqueo y luego añadirlo al final del buzón de correos, cuando el puerto 25 está garantizado casi en toda plataforma con soporte TCP/IP? Especialmente cuando esto significa que el correo obtenido de esta manera tiene garantizado verse como un correo que ha sido transferido de manera normal por el SMTP, que es lo que realmente queremos.

De aquí se extraen varias lecciones. Primero, la idea de enviar por el puerto SMTP fue la mayor recompensa individual que obtuve al tratar de emular los métodos de Linus. Un usuario me proporcionó una fabulosa idea y lo único que restaba era comprender sus implicaciones.

> **11. Lo más grande, después de tener buenas ideas, es reconocer las buenas ideas de tus usuarios. Esto último es a veces lo mejor.**

Lo que resulta muy interesante es que uno rápidamente encontrará que, cuando está absolutamente convencido y seguro de lo que le debe a los demás, entonces el mundo

lo tratará como si hubiera realizado cada parte de la invención por sí mismo, y esto le hará apreciar con modestia su ingenio natural. ¡Todos podemos ver lo bien que funcionó esto para el propio Linus!

(Cuando leía este documento en la Conferencia de Perl de agosto de 1997, Larry Wall estaba en la fila del frente. Al llegar a lo que acabo de decir, Larry dijo con voz alta: «¡Anda, di eso, díselos, hermano!». Todos los presentes rieron porque sabían que eso también le había funcionado muy bien al inventor de Perl).

Y a unas cuantas semanas de haber echado a andar el proyecto con el mismo espíritu comencé a recibir adulaciones similares; no solo de parte de mis usuarios, sino de otras personas que se habían enterado por terceros. He puesto a buen recaudo parte de esos correos. Los volveré a leer en alguna ocasión, si es que me llego a preguntar si mi vida ha valido la pena. :-)

Pero hay otras dos lecciones más fundamentales, que no tienen que ver con las políticas, que son generales para todos los tipos de diseño:

> **12. Frecuentemente, las soluciones más innovadoras y espectaculares surgen al darte cuenta de que la concepción del problema era errónea.**

Había estado intentando resolver el problema equivocado al continuar desarrollando Popclient como un agente de entrega y de transporte, con toda clase de modos raros de entrega local. El diseño de Fetchmail requería ser

repensado de arriba abajo como un agente de transporte puro: como eslabón, si se habla de SMTP, de la ruta normal que sigue el correo en internet.

Cuando te topas con un muro durante el desarrollo —cuando te resulta difícil pensar mas allá de la siguiente corrección— es, a menudo, la hora de preguntarse no tanto si realmente se tiene la respuesta correcta, sino si se está planteando la pregunta correcta. Quizá el problema requiere ser replanteado.

Bien, yo ya había replanteado mi problema. Evidentemente, lo que tenía que hacer ahora era: 1) programar el soporte de envío por SMTP en el controlador genérico, 2) convertirlo en el modo por omisión y 3) eliminar eventualmente todas las demás modalidades de entrega, especialmente las de envío a buzón y a la salida estándar.

Estuve, durante algún tiempo, titubeando para dar el tercer paso; temiendo trastornar a los viejos usuarios de Popclient, quienes dependían de estos mecanismos alternativos de entrega. En teoría, ellos podían cambiar inmediatamente a archivos .forward o sus equivalentes en otro esquema que no fuera Sendmail para obtener los mismos resultados. Pero, en la práctica, la transición podría complicarse demasiado.

Cuando por fin lo hice, los beneficios fueron inmensos. Las partes más intrincadas del código del controlador desaparecieron. La configuración se volvió radicalmente más simple: al no tratar con el MDA del sistema ni con el buzón del usuario, ya no había que preocuparse de que el sistema operativo soportara el bloqueo de archivos.

Asimismo, el único riesgo de extraviar correo también se había desvanecido. Antes, si especificabas el envío a un buzón y el disco estaba lleno, entonces el correo se perdía irremediablemente. Esto no pasa con el envío vía SMTP, debido a que el SMTP del receptor no devolverá un OK mientras el mensaje no haya sido entregado con éxito o al menos mandado a la cola para su entrega ulterior.

Además, el desempeño mejoró mucho (aunque uno no lo notara en la primera corrida). Otro beneficio nada despreciable fue la simplificación de la página del manual.

Más adelante hubo que agregar la entrega a un agente local especificado por el usuario con el fin de manejar algunas situaciones oscuras involucradas con la asignación dinámica de direcciones en SLIP. Sin embargo, encontré una forma mucho más simple de hacerlo.

¿Cuál era la moraleja? No hay que vacilar en desechar alguna característica superflua si puedes hacerlo sin pérdida de efectividad. Antoine de Saint-Exupéry (aviador y diseñador aeronáutico, cuando no se dedicaba a escribir libros clásicos para niños) afirmó que:

> **13. «La perfección —en diseño— se alcanza no cuando ya no hay nada que agregar, sino cuando ya no hay algo que quitar».**

Cuando el código va mejorando y se va simplificando es cuando sabes que estás en lo correcto. Así, en este proceso, el diseño de Fetchmail adquirió una identidad propia, diferente de su ancestro, Popclient.

Había llegado la hora de cambiar de nombre. El nuevo diseño parecía más un doble del Sendmail que del viejo Popclient; ambos eran MTA, agentes de transporte, pero mientras que Sendmail empuja y luego entrega, el nuevo Popclient acarrea y después entrega. Así que, después de dos arduos meses, lo bauticé de nuevo con el nombre de Fetchmail.

El crecimiento de Fetchmail

Allí me encontraba con un bonito e innovador diseño, un programa cuyo funcionamiento tenía asegurado gracias al uso diario y al equipo de *beta testers*. Esta gradualmente me hizo ver que ya no estaba involucrado en un *hackeo* personal y trivial que podía resultar útil para unas cuantas personas más. Tenía en mis manos un programa que cualquier *hacker*, con una caja Unix y una conexión SLIP o PPP, realmente necesita.

Cuando el método de expedición por SMTP se puso delante de la competencia, se convirtió en un «matón profesional», uno de esos programas clásicos que ocupa tan bien su lugar que las otras alternativas no solo son descartadas, sino olvidadas.

Pienso que uno realmente no podría imaginar o planear un resultado como este. Tienes que meterte a manejar conceptos de diseño tan poderosos que posteriormente los resultados parezcan inevitables, naturales o incluso predestinados. La única manera de hacerse de estas ideas es jugar con un montón de propuestas o tener

una visión de la ingeniería lo suficientemente competente como para poder llevar las buenas ideas de otras personas más allá de lo que estas pensaban que podían llegar.

Andrew Stuart Tanenbaum tuvo una buena idea original con la construcción de un Unix nativo y simple que sirviera como herramienta de enseñanza para computadoras con microprocesador 386. Linus Torvalds llevó el concepto de Minix más allá de lo que Andrew imaginó que pudiera llegar y se transformó en algo maravilloso. De la misma manera (aunque en una escala menor), tomé algunas ideas de Carl Harris y Harry Hochheiser y las impulsé fuertemente. Ninguno de nosotros era «original» en el sentido romántico de la idea que se tiene de un genio. Pero la mayor parte del desarrollo de la ciencia, la ingeniería y el *software* no se debe a un genio original, sino a la mitología del *hacker*, por el contrario.

Los resultados fueron siempre un tanto complicados: de hecho, ¡justo el tipo de reto para el que vive un *hacker*! Y esto implicaba que tenía que fijar aún más alto mis propios estándares. Para lograr que Fetchmail fuese tan bueno como ahora veía que podía ser, tenía que escribir no solo para satisfacer mis propias necesidades, sino también incluir y dar el soporte a otros que estuvieran fuera de mi órbita. Y esto lo tenía que hacer manteniendo el programa sencillo y robusto.

La primera característica más importante y contundente que escribí después de hacer eso fue el soporte para recabado múltiple; esto es, la capacidad de recoger el correo de los buzones que habían acumulado todo el correo

de un grupo de usuarios y luego trasladar cada mensaje al recipiente individual del respectivo destinatario.

En parte, decidí agregar el soporte de recabado múltiple debido a que algunos usuarios lo reclamaban, pero sobre todo porque evidenciaría los errores de un código de recabado individual, al forzarme a abordar el direccionamiento con generalidad. Tal como ocurrió. Poner el RFC 822 a que funcionara correctamente me tomó bastante tiempo, no solo porque cada una de las partes que lo componen son difíciles, sino porque involucraban un montón de detalles confusos e interdependientes entre sí.

Así, el direccionamiento del recabado múltiple se volvió una excelente decisión de diseño. De esta forma supe que:

> **14. Toda herramienta debe resultar útil en la forma prevista, pero una gran herramienta te permite usarla de la manera menos esperada.**

El uso inesperado del recabado múltiple de Fetchmail fue el trabajar las listas de correo con la lista guardada y realizar la expansión del alias en el lado del cliente de la conexión SLIP o PPP. Esto significa que alguien que cuenta con una computadora y una cuenta ISP puede manejar una lista de correos sin que tenga que continuar entrando a los archivos del alias del ISP.

Otro cambio importante reclamado por mis *beta testers* era el soporte para las extensiones multipropósito de correo de internet (MIME) de 8 bits. Esto se podía obtener

fácilmente, ya que había sido cuidadoso en mantener el código de 8 bits limpio. No es que yo me hubiera anticipado a la exigencia de esta característica, sino que obedecía a otra regla:

> **15. Cuando se escribe *software* para una puerta de enlace de cualquier tipo, hay que tomar la precaución de alterar el flujo de datos lo menos posible, y ¡nunca eliminar información a menos que los receptores obliguen a hacerlo!**

Si no hubiera obedecido esta regla, entonces el soporte MIME de 8 bits habría sido difícil y lleno de errores. De tal modo, lo que tuve que hacer fue leer el RFC 1652 y agregar algo de lógica trivial en la generación de encabezados.

Algunos usuarios europeos me presionaron para que introdujera una opción que limitase el número de mensajes acarreados por sesión (de manera que pudieran controlar los costos de sus caras redes telefónicas). Me opuse a dicho cambio durante mucho tiempo y aun no estoy totalmente conforme con él. Pero si escribes para el mundo debes escuchar a tus clientes: esto no debe cambiar en nada solo porque no te están dando dinero.

Algunas lecciones más extraídas de Fetchmail

Antes de volver a los temas generales de ingeniería de *software*, hay que ponderar otras dos lecciones específicas sacadas de la experiencia de Fetchmail.

La sintaxis de los archivos RC incluye una serie de palabras clave que pueden ser consideradas como «ruido» y son ignoradas por el analizador. La sintaxis inglesa que estas permiten es considerablemente más legible que la secuencia de los pares clave-valor tradicionales que obtienes cuando las quitas.

Estas comenzaron como un experimento de madrugada, cuando noté que muchas de las declaraciones de los archivos RC se asemejaban un poco a un minilenguaje imperativo. (Esta también fue la razón por la cual cambié la palabra clave original de Popclient de «servidor» a «poll»).

Me parecía en ese entonces que aproximar ese minilenguaje imperativo al inglés lo podía hacer más fácil de usar. Ahora, a pesar de que soy un partidario convencido de la escuela de diseño «hágalo un lenguaje», ejemplificada en Emacs, HTML y muchas bases de datos, no soy normalmente un fanático de la sintaxis inglesa.

Los programadores han tendido a favorecer tradicionalmente la sintaxis de control debido a que es muy precisa y compacta, además de no tener redundancia alguna. Esto es una herencia cultural de la época en que los recursos de cómputo eran muy caros, por lo que la etapa de análisis tenía que ser la más sencilla y económica posible. El inglés, con un cincuenta por ciento de redundancia, parecía ser un modelo muy inapropiado en ese entonces.

Esta no es la razón por la cual yo dudo de la sintaxis inglesa; solo la menciono aquí para negarla. Con los ciclos baratos, la fluidez no debe ser un fin por sí misma. Ahora es más importante para un lenguaje el ser conveniente

para los humanos que ser económico en términos de recursos computacionales.

Sin embargo, hay razones suficientes para andar con cuidado. Una es el costo de la complejidad de la etapa de análisis: nadie quiere incrementarlo a un punto tal que se vuelva una fuente importante de errores y de confusión para el usuario. Otra radica en que al implementar una sintaxis inglesa para el lenguaje se exige con frecuencia que se deforme considerablemente el «inglés» inicial, por lo que la semejanza superficial con un lenguaje natural es tan confusa como podría haberlo sido la sintaxis tradicional. (Puedes ver mucho de esto en los lenguajes de programación de cuarta generación o 4GL y en los lenguajes de búsqueda en bancos de datos comerciales).

La sintaxis de control de Fetchmail parece esquivar estos problemas debido a que el dominio de su lenguaje es extremadamente restringido. Está muy lejos de ser un lenguaje de uso amplio; las cosas que dice no son muy complicadas, por lo que hay pocas posibilidades de confusión al moverse de un reducido subconjunto del inglés y el lenguaje de control real. Creo que se puede extraer una lección más general de esto:

> **16. Cuando tu lenguaje está lejos de un Turing completo, entonces puedes endulzar tu sintaxis.**

Otra lección trata de la seguridad por obscuridad: recurrir al secreto para proteger ciertos datos. Algunos usuarios de Fetchmail me solicitaron cambiar el *software* para

poder guardar las claves de acceso encriptadas en su archivo RC, de tal manera que los *crackers* no pudieran verlas por pura casualidad.

No lo hice debido a que esto prácticamente no proporcionaría ninguna protección adicional. Cualquiera que adquiera los permisos necesarios para leer el archivo RC respectivo sería de todos modos capaz de correr Fetchmail y, si por su *password* fuera, podría sacar el decodificador necesario del mismo código de Fetchmail para obtenerlo.

Todo lo que la encriptación de *password* en el archivo `.fetchmailrc` podría haber conseguido era una falsa sensación de seguridad para la gente que no está muy metida en este medio. La regla general es la siguiente:

> **17. Un sistema de seguridad es tan seguro como secreto. Cuídate de los secretos a medias.**

Condiciones necesarias para el modelo bazar

Los primeros que leyeron este documento, y las primeras versiones inacabadas que se hicieron públicas, preguntaban constantemente sobre los requisitos necesarios para un desarrollo exitoso dentro del modelo bazar, incluyendo la calificación del líder del proyecto, así como la del estado del código cuando uno va a hacerlo público y a comenzar a construir una comunidad de codesarrolladores.

Está claro que uno no puede partir de cero en el modelo bazar. Con él, uno puede probar, buscar errores, poner

a punto y mejorar algo, pero sería muy difícil originar un proyecto de un modo semejante al bazar. Linus no lo intentó de esta manera. Yo tampoco lo hice así. Nuestra naciente comunidad de desarrolladores necesita algo que ya corra para jugar.

Cuando uno comienza la construcción del edificio comunal, lo que debe ser capaz de hacer es presentar una promesa plausible. El programa no necesita ser particularmente bueno. Puede ser burdo, tener muchos errores, estar incompleto y pobremente documentado. Pero en lo que no se puede fallar es en convencer a los potenciales codesarrolladores de que el programa puede evolucionar hacia algo elegante en el futuro.

Linux y Fetchmail se hicieron públicos con diseños básicos, fuertes y atractivos. Mucha gente piensa que el modelo bazar ha considerado correctamente esto como crítico, para después saltar a la conclusión de que es indispensable que el líder del proyecto tenga un mayor nivel de intuición para el diseño y mucha capacidad.

Sin embargo, Linus obtuvo su diseño a partir de Unix. Yo inicialmente conseguí el mío del antiguo Popmail (a pesar de que cambiaría mucho posteriormente, mucho más, guardando las proporciones, de lo que lo ha hecho Linux). Entonces, ¿es necesario que el líder o coordinador posea realmente un talento extraordinario en el modelo bazar o basta con que aproveche el talento de otros para el diseño?

Creo que no es indispensable que quien coordine sea capaz de originar diseños de calidad excepcional, pero lo

que sí es absolutamente esencial es que él o ella sea capaz de reconocer las buenas ideas de los demás sobre diseño.

Tanto el proyecto de Linux como el de Fetchmail dan evidencias de esto. A pesar de que Linus no es un diseñador original espectacular (como lo discutimos anteriormente), ha mostrado tener una poderosa habilidad para reconocer un buen diseño e integrarlo al *kernel* de Linux. Ya he descrito cómo la idea de diseño de mayor envergadura para Fetchmail (reenvío por SMTP) provino de otro.

Los primeros lectores de este artículo me halagaron al sugerir que soy propenso a subestimar la originalidad del diseño en los proyectos bazar porque yo tengo mucha, y en consecuencia la tomo por sentada. En parte puede ser verdad: el diseño es ciertamente mi fuerte (comparado con la programación o la depuración).

Pero el problema de ser listo y original en el diseño de *software* es que se tiende a convertir en hábito: uno hace las cosas como por reflejo, de manera tal que parezcan elegantes y complicadas, cuando debería mantenerlas simples y robustas. Ya he sufrido tropiezos en proyectos debido a esta equivocación, pero me las ingenié para que no sucediera lo mismo con Fetchmail.

Así, pues, considero que el proyecto de Fetchmail tuvo éxito en parte debido a que contuve mi propensión a ser astuto; este es un argumento que va (por lo menos) contra la originalidad en el diseño como algo esencial para que los proyectos bazar sean exitosos. Consideremos de nuevo Linux. Supóngase que Linus Torvalds hubiera

estado tratando de desechar innovaciones fundamentales en el diseño del sistema operativo durante la etapa de desarrollo; ¿podría acaso ser tan estable y exitoso como el *kernel* que tenemos?

Por supuesto, se necesita un cierto nivel mínimo de habilidad para el diseño y la escritura de programas, pero es de esperar que cualquiera que quiera seriamente lanzar un esfuerzo al estilo bazar ya esté por encima de este nivel. El mercado interno de la comunidad de *software* libre, por reputación, ejerce una presión sutil sobre la gente para que no inicie esfuerzos de desarrollo que no sea capaz de mantener. Hasta ahora, esto parece estar funcionando bastante bien.

Existe otro tipo de habilidad que no está asociada normalmente con el desarrollo de *software*, la cual yo considero igual de importante que el ingenio en el diseño para los proyectos bazar y a veces hasta más. Un coordinador o líder de proyecto estilo bazar debe tener buena capacidad de comunicación.

Esto podría parecer obvio. Para poder construir una comunidad de desarrollo se necesita atraer gente, interesarla en lo que se está haciendo y mantenerla a gusto con el trabajo que se está desarrollando. El entusiasmo técnico constituye una buena parte para poder lograr esto, pero está muy lejos de ser definitivo. Además, es importante la personalidad que uno proyecta.

No es una coincidencia que Linus sea un tipo que hace que la gente lo aprecie y desee ayudarle. Tampoco es una coincidencia que yo sea un extrovertido incansable que

disfruta de trabajar con una muchedumbre o que tenga un poco de porte e instintos de cómico improvisado. Para hacer que el modelo bazar funcione ayuda mucho tener al menos un poco de capacidad para las relaciones sociales.

El contexto social del *software* libre

Bien se ha dicho: los mejores *hackeos* comienzan como soluciones personales a los problemas cotidianos del autor y se vuelven populares debido a que el problema es común para un buen grupo de usuarios. Esto nos hace regresar al tema del aforismo 1, que quizá puede replantearse de una manera más útil:

> **18. Para resolver un problema interesante, comienza por encontrar un problema que te resulte interesante.**

Así ocurrió con Carl Harris y el antiguo Popclient, y así sucede conmigo y Fetchmail. Esto, sin embargo, se ha entendido desde hace mucho. El punto interesante, que las historias de Linux y Fetchmail nos piden enfocar, está en la siguiente etapa: en la de la evolución del *software* en presencia de una amplia y activa comunidad de usuarios y codesarrolladores.

En *The mythical man-month*, Fred Brooks observó que el tiempo del programador no es un consumible más; que el agregar desarrolladores a un proyecto maduro de *software* lo vuelve tardío. Expuso que la complejidad y los costos de comunicación de un proyecto aumentan al cuadrado el

número de desarrolladores, mientras que el trabajo crece solo linealmente. A este planteamiento se le conoce como la Ley de Brooks y es generalmente aceptado como algo cierto. Pero si la Ley de Brooks fuese general, entonces Linux sería imposible.

Unos años después, el clásico de Gerald Weinberg, *The psychology of computer programming*, plantea, visto en retrospectiva, una corrección esencial a Brooks. En su discusión sobre la «programación sin ego», Weinberg señala que los lugares donde los desarrolladores no tienen propiedad sobre su código, estimulando a otras personas a buscar errores y posibles mejoras, son los lugares donde el avance es dramáticamente más rápido que en cualquier otro lado.

La terminología empleada por Weinberg ha evitado quizá que su análisis gane la aceptación que merece: uno tiene que sonreír al escuchar que los *hackers* de internet no tienen ego. Creo, no obstante, que su argumentación parece más válida ahora que nunca.

La historia de Unix debió habernos preparado para lo que hemos aprendido de Linux (y lo que he verificado experimentalmente en una escala más reducida al copiar deliberadamente los métodos de Linus). Esto es: mientras que la creación de programas sigue siendo esencialmente una actividad solitaria, los desarrollos realmente grandes surgen de la atención y la capacidad de pensamiento de comunidades enteras. El desarrollador que usa solamente su cerebro sobre un proyecto cerrado se está quedando atrás del que sabe crear en un contexto abierto y evolu-

tivo, en el que la búsqueda de errores y las mejoras son realizadas por cientos de personas.

Pero el mundo tradicional de Unix no pudo llevar este enfoque hasta sus últimas consecuencias debido a varios factores. Uno era el conjunto de limitaciones legales producidas por varias licencias, secretos e intereses comerciales. Otra (en retrospectiva) era que el internet no había madurado lo suficiente para lograrlo.

Antes de que el internet fuera tan accesible, había comunidades geográficamente compactas en las cuales la cultura estimulaba la «programación sin ego» de Weinberg y el desarrollador podía atraer fácilmente a muchos desarrolladores y usuarios capacitados. El Bell Labs, el MIT AI Lab y la Universidad de California en Berkeley son lugares donde se originaron innovaciones que son legendarias y aún poderosas.

Linux fue el primer proyecto que se esforzó de forma consciente y exitosa en usar el mundo entero como un nido de talento. No creo que sea coincidencia que el periodo de gestación de Linux haya coincidido con el nacimiento de la World Wide Web o que Linux haya dejado su infancia durante el mismo periodo (1993-1994) en el que se vio el despegue de la industria ISP y la explosión del interés masivo por el internet. Linus fue el primero que aprendió a jugar con las nuevas reglas que ese internet penetrante hace posibles.

A pesar de que el internet barato era una condición necesaria para que evolucionara el modelo de Linux, no creo que fuera en sí misma una condición suficiente.

Otros factores vitales fueron el desarrollo de un estilo de liderazgo y el arraigo de hábitos cooperativos, que permiten a los programadores atraer más codesarrolladores y obtener el máximo provecho del medio.

Pero ¿cómo son ese estilo de liderazgo y esos hábitos? No pueden estar basados en relaciones de poder; aunque lo estuvieran, el liderazgo por coerción no produciría los resultados que estamos viendo. Weinberg cita un pasaje de la autobiografía del anarquista ruso del siglo XIX, Kropotkin: *Memorias de un revolucionario*, que está muy acorde con este tema:

> Habiendo sido criado en una familia que tenía siervos, me incorporé a la vida activa, como todos los jóvenes de mi época, con una gran confianza en la necesidad de mandar, ordenar, regañar, castigar y cosas semejantes. Pero cuando en una etapa temprana tuve que manejar empresas serias y tratar con personas libres, cuando cada error podría acarrear serias consecuencias, comencé a apreciar la diferencia entre actuar con base en el principio de orden y disciplina, y actuar con base en el principio del entendimiento. El primero funciona admirablemente en un desfile militar pero no sirve en la vida real, cuando el objetivo solo puede lograrse mediante el esfuerzo serio de muchas voluntades convergentes.

El «esfuerzo serio de muchas voluntades convergentes» es precisamente lo que todo proyecto estilo Linux requiere, mientras que el «principio de orden y disciplina» es efectivamente imposible de aplicar a los voluntarios del paraíso anarquista que llamamos internet. Para poder trabajar y competir de manera efectiva, los *hackers* que quieran encabezar proyectos de colaboración deben aprender a reclutar y entusiasmar a las comunidades de un modo vagamente sugerido por el «principio del entendimiento mutuo» de Kropotkin. Deben aprender a usar la Ley de Linus.

Anteriormente me referí al método Delphi como una posible explicación de la Ley de Linus. Pero existen analogías más fuertes con sistemas adaptativos en biología y economía que se sugieren irresistiblemente. El mundo de Linux se comporta en muchos aspectos como el libre mercado o un sistema ecológico, donde un grupo de agentes individualistas buscan maximizar la utilidad en la que los procesos generan un orden espontáneo autocorrectivo más desarrollado y eficiente que lo que podría lograr cualquier tipo de planeación centralizada. Esta es entonces la manera de ver el «principio del entendimiento mutuo».

La «función de utilidad» que los *hackers* de Linux están maximizando no es económica en el sentido clásico, sino algo intangible como la satisfacción de su ego y su reputación entre otros *hackers*. (Uno podría hablar de su «motivación altruista», pero ignoraríamos el hecho de que el altruismo en sí mismo es una forma de satisfacción del ego). Los grupos voluntarios que realmente funcionan de

esta manera no son escasos; uno en el que he participado es el de aficionados a la ciencia ficción que, a diferencia del mundo de los *hackers*, reconoce explícitamente el *egoboo* (*ego boosting*, el realce de la reputación de uno entre los demás) como la motivación básica que está detrás de la actividad voluntaria.

Linus, al ponerse exitosamente como vigía de un proyecto en el que el desarrollo es realizado por otros y al alimentar el interés en él hasta que se hizo autosustentable, ha mostrado el largo alcance del «principio del entendimiento mutuo» de Kropotkin. Este enfoque cuasieconómico del mundo de Linux nos permite ver cuál es la función de tal entendimiento.

Podemos ver el método de Linus como la forma de crear un mercado eficiente en torno al *egoboo*, que liga el individualismo de los *hackers* a objetivos difíciles que solo se pueden lograr con la cooperación sostenida. Con el proyecto de Fetchmail he demostrado (en una escala mucho menor, claro) que sus métodos pueden copiarse con buenos resultados. Posiblemente lo mío fue realizado de una forma un poco más consciente y sistemática que la de él.

Muchas personas (especialmente aquellas que desconfían políticamente del libre mercado) podrían esperar que una cultura de individuos egoístas que se dirigen solos sea fragmentaria, territorial, clandestina y hostil. Pero esta idea es claramente refutada, por ejemplo, por la asombrosa variedad, calidad y profundidad de la documentación de Linux. Se da por hecho que los pro-

gramadores odian la documentación: ¿cómo entonces los *hackers* de Linux generan tanta? Evidentemente, el libre mercado de Linux basado en el *egoboo* funciona mejor para producir tal virtuosismo que los departamentos de edición, masivamente subsidiados, de los productores comerciales de *software*.

Tanto el proyecto de Fetchmail como el del *kernel* de Linux han demostrado que, con el estímulo apropiado al ego de otros *hackers*, un desarrollador o coordinador fuerte puede usar el internet para contar con un gran número de codesarrolladores, sin que se corra el peligro de desbocar el proyecto en un auténtico relajo. Por lo tanto, a la Ley de Brooks yo le contrapongo lo siguiente:

> **19. Si el coordinador de desarrollo tiene un medio al menos tan bueno como lo es el internet y sabe dirigir sin coerción, muchas cabezas serán, inevitablemente, mejor que una.**

Pienso que el futuro del *software* libre será cada vez más de la gente que sabe cómo jugar el juego de Linus, la gente que deja atrás la catedral y abraza el bazar. Esto no quiere decir que la visión y la brillantez individuales ya no importen; al contrario, creo que en la vanguardia del *software* libre estarán quienes comiencen con visión y brillantez individual, y luego las enriquezcan construyendo positivamente comunidades voluntarias de interés.

A lo mejor este no solo es el futuro del *software* libre. Ningún desarrollador comercial sería capaz de reunir el

talento que la comunidad de Linux es capaz de invertir en un problema. ¡Muy pocos podrían pagar tan solo la contratación de las más de doscientas personas que han contribuido a Fetchmail!

Es posible que a largo plazo triunfe la cultura del *software* libre, no porque la cooperación sea moralmente correcta o porque la «apropiación» del *software* sea moralmente incorrecta (suponiendo que se crea realmente en esto último, lo cual no es cierto ni para Linus ni para mí), sino simplemente por que el mundo comercial no es capaz de ganar una carrera armamentista a las comunidades de *software* libre, las cuales pueden poner más tiempo calificado en un problema que cualquier otra compañía.

Reconocimientos

Este artículo fue mejorado gracias a las conversaciones con un gran número de personas que me ayudaron a perfeccionarlo. En especial, agradezco a Jeff Dutky, quien sugirió el planteamiento de que «la búsqueda de errores pude hacerse en paralelo» y ayudó a ampliar el análisis respectivo. También agradezco a Nancy Lebovitz por su sugerencia de imitar a Weinberg al citar a Kropotkin. Asimismo, recibí críticas perspicaces de Joan Eslinger y de Marty Franz de la lista de General Technics. Paul Egger me hizo ver el conflicto entre la GPL y el modelo bazar. Agradezco también a los integrantes del Grupo de Usuarios de Linux de Filadelfia (PLUG, por sus siglas en inglés), por convertirse en el primer público para la primera

versión de este artículo. Finalmente, los comentarios de Linus Torvalds fueron de mucha ayuda y su apoyo inicial fue muy estimulante.

Otras lecturas

He citado varias partes del clásico de Frederick Brooks, *The mythical man-month*, debido a que en muchos aspectos todavía se tienen que mejorar sus puntos de vista. Yo recomiendo con cariño la edición del XXV aniversario de Addison-Wesley, que viene junto con su artículo titulado «No hay balas de plata».

La nueva edición trae una invaluable retrospectiva de veinte años, en la que Brooks admite francamente ciertas críticas al texto original que no pudieron mantenerse con el tiempo. Leí por primera vez la retrospectiva después de que estaba esencialmente terminado este artículo, y me sorprendí al encontrar que Brooks ¡le atribuye a Microsoft prácticas semejantes a las de bazar!

The psychology of computer programming, de Gerald Wienberg, introdujo el concepto de «programación sin ego». A pesar de que él estaba muy lejos de ser la primera persona en comprender la futilidad del «principio de orden», fue probablemente el primero en reconocer y argumentar el tema en relación con el desarrollo de *software*.

Richard P. Gabriel, al analizar la cultura de Unix anterior a la era de Linux, planteaba la superioridad de un primitivo modelo estilo bazar en un artículo de 1989 «*Lisp:*

good news, bad news, how to win big». Pese a estar atrasado en algunos aspectos, este ensayo todavía es muy celebrado por los admiradores de Lisp (entre quienes me incluyo). Un corresponsal me recordó que la sección titulada «Peor es mejor» predice con gran exactitud a Linux.

El trabajo de Tom DeMarco y Timothy Lister, *Peopleware: productive projects and teams*, es una joya que ha sido subestimada. Para mi fortuna, fue citada por Fred Brooks. A pesar de que poco de lo que dicen los autores es directamente aplicable a las comunidades de *software* libre o de Linux, su visión sobre las condiciones necesarias para un trabajo creativo es aguda y muy recomendable para quien intente llevar algunas de las virtudes del modelo bazar a un contexto más comercial.

Epílogo: ¡Netscape adopta el modelo bazar!

Es un extraño sentimiento el que se percibe cuando uno comprende que está ayudando a hacer historia…

El 22 de enero de 1998, aproximadamente siete meses después de que publiqué este artículo, Netscape Communications anunció planes para liberar el código fuente de Netscape Communicator. No tenía idea alguna de que esto iba a suceder antes de la fecha de anuncio.

Eric Hahn, vicepresidente ejecutivo y director de tecnología en Netscape, me mandó un correo electrónico poco después del anuncio, que dice textualmente: «De parte de todos los que integran Netscape, quiero agradecerle por habernos ayudado a llegar hasta este punto,

en primer lugar. Su pensamiento y sus escritos fueron inspiraciones fundamentales en nuestra decisión».

La siguiente semana realicé un viaje en avión a Silicon Valley como parte de la invitación para realizar una conferencia de todo un día sobre cómo crear estrategias (el 4 de febrero de 1998), con algunos de sus técnicos y ejecutivos de mayor nivel. Juntos diseñamos la estrategia de publicación del código fuente de Netscape y la licencia, y realizamos algunos otros planes de los cuales esperamos que eventualmente tengan implicaciones positivas de largo alcance sobre la comunidad de código abierto. Por el momento, mientras escribo, es demasiado pronto para ser más específico, pero se van a ir publicando los detalles en las semanas por venir.

Netscape está a punto de proporcionarnos con una prueba a gran escala, en el mundo real, del modelo bazar dentro del ámbito empresarial. La cultura del código abierto ahora enfrenta un peligro: si no funcionan las acciones de Netscape, entonces el concepto del código abierto puede llegar a desacreditarse de tal manera que el mundo empresarial no lo abordará nuevamente sino hasta en una década.

Por otro lado, esto es una oportunidad espectacular. La reacción inicial hacia este movimiento en Wall Street y en otros lados fue cautelosamente positiva. Nos están dando una oportunidad de demostrar que podemos hacerlo. Si Netscape recupera una parte significativa del mercado con este movimiento, puede desencadenar una revolución ya muy retrasada en la industria de *software*.

El siguiente año deberá ser un periodo muy interesante y de intenso aprendizaje.

Índice de aforismos

1. Todo buen trabajo de *software* comienza a partir de las necesidades personales de quien programa. (Todo buen trabajo empieza cuando uno tiene que rascarse su propia comezón).
2. Los buenos programadores saben qué escribir; mientras que los mejores, qué rescribir (y reutilizar).
3. Contempla desecharlo; de todos modos tendrás que hacerlo.
4. Si tienes la actitud adecuada, encontrarás problemas interesantes.
5. Cuando se pierde el interés en un programa, el último deber es heredarlo a un sucesor competente.
6. Tratar a los usuarios como colaboradores es la forma más apropiada de mejorar el código, y la más efectiva de depurarlo.
7. Libera rápido. Libera a menudo. Y escucha a tus clientes.
8. Dada una base suficiente de colaboradores y *beta testers*, casi cualquier problema puede ser identificado rápidamente y su solución será obvia al menos para alguien.
9. Las estructuras de datos inteligentes y el código burdo funcionan mucho mejor que en el caso inverso.
10. Si tratas a tus analistas (*beta testers*) como si fueran tu recurso más valioso, ellos te responderán convirtiéndose en tu recurso más valioso.
11. Lo más grande, después de tener buenas ideas, es reconocer las buenas ideas de tus usuarios. Esto último es a veces lo mejor.

12. Frecuentemente, las soluciones más innovadoras y espectaculares surgen al darte cuenta de que la concepción del problema era errónea.
13. La perfección —en diseño— se alcanza no cuando ya no hay nada que agregar, sino cuando ya no hay algo que quitar.
14. Toda herramienta debe resultar útil en la forma prevista, pero una gran herramienta te permite usarla de la manera menos esperada.
15. Cuando se escribe *software* para una puerta de enlace de cualquier tipo, hay que tomar la precaución de alterar el flujo de datos lo menos posible, y ¡nunca eliminar información a menos que los receptores obliguen a hacerlo!
16. Cuando tu lenguaje está lejos de un Turing completo, entonces puedes endulzar tu sintaxis.
17. Un sistema de seguridad es tan seguro como secreto. Cuídate de los secretos a medias.
18. Para resolver un problema interesante, comienza por encontrar un problema que te resulte interesante.
19. Si el coordinador de desarrollo tiene un medio al menos tan bueno como lo es el internet y sabe dirigir sin coerción, muchas cabezas serán, inevitablemente, mejor que una.

Iniciativa de Budapest para el acceso abierto

Open Society Institute[12]

Una vieja tradición y una nueva tecnología convergen para hacer posible un bien público sin precedente. La vieja tradición es el deseo de los científicos y académicos por publicar los frutos de su investigación en revistas académicas sin tener que pagar por ello, tan solo por el gusto de indagar y por el conocimiento. La nueva tecnología es internet. El bien público que hacen posible es la distribución electrónica en la red de redes de literatura periódica revisada por pares completamente gratuita y sin restricciones de acceso por todos los científicos, académicos, maestros, estudiantes y otras mentes curiosas. Retirar las barreras de acceso a esta literatura acelerará la investigación, enriquecerá la educación, compartirá el aprendizaje de los ricos con los pobres y el de los pobres con el de los ricos, hará esta literatura tan útil como sea posible y sentará los cimientos para unir a la humanidad en una conversación intelectual común y búsqueda del conocimiento.

[12] En la actualidad su nombre ha sido cambiado por Open Society Foundations. [N. del E.].

Por varias razones este tipo de disponibilidad en línea gratuita y sin restricciones, que llamaremos *libre acceso*, ha sido limitada hasta la fecha a pequeñas porciones de literatura periódica. Pero aún en estas limitadas colecciones, muchas y diversas iniciativas han demostrado que el acceso abierto es económicamente viable (posible), que le da a los lectores un poder extraordinario para encontrar y usar literatura relevante, y que ofrece a los autores y a sus trabajos una nueva visibilidad, legibilidad e impacto, vastos y medibles. Para asegurar estos beneficios para todos, hacemos un llamado a todas las instituciones e individuos interesados para que ayuden a incrementar al acceso abierto al resto de este tipo de literatura y retiren las barreras, en especial las barreras de precio que se interponen en este camino. Mientras más nos sumemos en el esfuerzo para el avance de esta causa, más rápido disfrutaremos de los beneficios del acceso abierto.

La literatura, que debería poder accederse libremente en línea, es aquella que los académicos dan al mundo sin la expectativa de recibir pago. Básicamente, es la categoría compuesta por sus artículos revisados por pares, destinados a publicaciones periódicas; pero también incluye cualquier *preprint* sin revisión que quizás les gustaría poner en línea para comentar o alertar a otros colegas sobre la importancia de hallazgos de investigación. Hay muchos grados y tipos de acceso amplio y fácil a esta literatura. Por «acceso abierto» a esta literatura queremos decir su disponibilidad gratuita en internet público, permitiendo a cualquier usuario leer, descargar, copiar, distribuir, im-

primir, buscar o usarlos con cualquier propósito legal, sin ninguna barrera financiera, legal o técnica, fuera de las que son inseparables de las que implica acceder a internet mismo. La única limitación en cuanto a reproducción y distribución y el único rol del copyright en este dominio, deberá ser dar a los autores el control sobre la integridad de sus trabajos y el derecho de ser adecuadamente reconocidos y citados.

Mientras que la literatura periódica revisada por pares debería ser accesible en línea sin costo para los lectores, no se produce sin costos. Sin embargo hay experimentos que demuestran que en costo promedio de proveer acceso abierto a esta literatura, es mucho más bajo que el costo tradicional de las formas tradicionales de diseminación. Esta oportunidad de ahorrar dinero y expandir al mismo tiempo la cobertura de la diseminación representan un fuerte incentivo para profesionales, asociaciones, universidades, bibliotecas, fundaciones, etcétera, para abrazar el acceso abierto como un medio de avanzar en sus misiones. Alcanzar el acceso abierto requiere de nuevos modelos de recuperación de costos y mecanismos de financiación, pero la significativa reducción promedio del costo de la diseminación es una razón para confiar en que el objetivo es alcanzable y no solo preferible o utópico.

Para lograr el acceso abierto a la literatura periódica académica, recomendamos dos estrategias complementarias:

1. «Autoarchivar». Primero, los académicos requieren herramientas y asistencia para depositar sus artículos referidos en archivos electrónicos abiertos, una prácticas comúnmente denominada «autoarchivo». Cuando estos archivos alcanzan los estándares creados por la iniciativa para el acceso abierto, los buscadores y otras herramientas pueden tratar los archivos separados como uno. Los usuarios no necesitan saber qué archivos existen o dónde se localizan para encontrarlos y usar su contenido.
2. Publicaciones periódicas de acceso abierto. Segundo, los académicos necesitan los medios para crear una nueva generación de publicaciones periódicas comprometidas con el acceso abierto y para ayudar a las existentes que son elegibles para hacer la transición al acceso abierto. Debido a que los artículos de estas publicaciones deberán diseminarse tan ampliamente como sea posible, las nuevas publicaciones no podrán invocar restricciones de acceso por asuntos de derechos de autor del material que publican. En cambio, usarán los derechos de autor y otras herramientas para asegurarse del permanente acceso abierto a todos los artículos que publiquen. Debido a que el precio es una barrera al acceso, estas nuevas publicaciones no tendrán cuotas de suscripción ni acceso y buscarán otras formas para cubrir sus gastos. Hay muchas fuentes alternativas de financiamiento, incluyendo fundaciones y financiamiento de la investigación por par-

> te del gobierno, las universidades y los laboratorios que emplean investigadores; las donaciones organizadas por disciplina o institución; los amigos de la causa del acceso abierto; las ganancias de las ventas de anuncios en textos básicos; la recuperación de fondos de la disminución o cancelación de suscripciones a publicaciones tradicionales o cuotas de acceso, o incluso las contribuciones de los propios investigadores. No hay necesidad de favorecer una sola de estas soluciones sobre las demás para todas las áreas del conocimiento o para todas las naciones, ni tampoco se trata de dejar de buscar otras alternativas creativas.

El objetivo es el acceso abierto a literatura periódica revisada por pares. El autoarchivar (1) y una nueva generación de publicaciones periódicas de acceso abierto (2) son los caminos para alcanzar este objetivo. No solo son medios directos y efectivos hacia este fin, sino que están al alcance inmediato de los propios académicos y no requieren de tiempos de espera por asuntos del mercado o de las legislaciones. Mientras avalemos estas dos estrategias recién descritas, también estaremos impulsando la experimentación con nuevas formas de hacer la transición de los métodos actuales de diseminación hacia el acceso abierto. La flexibilidad, la experimentación y la adaptación a las circunstancias locales son la mejor forma de asegurar que el progreso en regiones y ambientes diversos sea rápido, seguro y duradero.

La red original del Open Society Institute (OSI por sus siglas en inglés), fundada por el filántropo George Soros, está comprometida a proveer la ayuda inicial y el apoyo para alcanzar este objetivo. Usará sus recursos e influencia para extender y promover el autoarchivo institucional, el lanzamiento de publicaciones de acceso abierto y para ayudar a que un sistema de publicaciones de acceso abierto llegue a ser autosustentable. En tanto que el compromiso y los recursos del OSI sean sustanciales, esta iniciativa requiere del enorme apoyo de otras organizaciones que sumen su esfuerzo y sus recursos.

Invitamos a gobiernos, universidades, bibliotecas, editores de publicaciones periódicas, fundaciones, asociaciones profesionales, clubes y académicos e investigadores a que compartan nuestra visión, a que se unan a la tarea de remover las barreras que se oponen al acceso abierto y construyamos un futuro en el que la investigación y la educación, en todas partes del mundo, florezca con mucha más libertad.

14 de febrero del 2002, Budapest, Hungría

Manifiesto de la guerrilla por el acceso abierto

Aaron Swartz y anónimos

La información es poder. Pero como con todo poder, hay quienes lo quieren mantener para sí mismos. La herencia científica y cultural del mundo completa, publicada durante siglos en libros y *journals*, está siendo digitalizada y apresada en forma creciente por un manojo de corporaciones privadas. ¿Quieres leer los *papers* que presentan los más famosos resultados de las ciencias? Vas a tener que mandarle un montón de dinero a editoriales como Reed Elsevier.

Están aquellos que luchan por cambiar esto. El movimiento por el acceso abierto ha luchado valientemente para asegurarse que los científicos no cedan sus derechos de autor, sino que se aseguren que su trabajo sea publicado en internet, bajo términos que permitan el acceso a cualquiera. Pero incluso en los mejores escenarios, su trabajo solo será aplicado a las cosas que se publiquen en el futuro. Todo lo que existe hasta este momento se ha perdido.

Ese es un precio muy alto por el que pagar. ¿Forzar a los académicos a pagar para poder leer el trabajo de

sus colegas? ¿Escanear bibliotecas enteras para solo permitir leerlas a la gente de Google? ¿Proveer artículos científicos a aquellos en las universidades de élite del primer mundo, pero no a los niños del sur global? Es indignante e inaceptable.

«Estoy de acuerdo,» dicen muchos, «¿pero qué podemos hacer? Las compañías detentan los derechos de autor, hacen enormes cantidades de dinero cobrando por el acceso y es perfectamente legal —no hay nada que podamos hacer para detenerlos–.» Pero sí hay algo que podemos hacer, algo que ya está siendo hecho: podemos contraatacar.

A ustedes, con acceso a estos recursos —estudiantes, bibliotecarios, científicos— se les ha otorgado un privilegio. Ustedes pueden alimentarse en este banquete del conocimiento mientras el resto del mundo queda fuera. Pero no es necesario —de hecho, moralmente, no es posible— que se queden este privilegio para ustedes. Tienen el deber de compartirlo con el mundo. Y lo han hecho: intercambiando contraseñas con colegas, haciendo solicitudes de descarga para amigos.

Mientras tanto, aquellos de ustedes que se han quedado fuera no están cruzados de brazos. Han estado atravesando agujeros sigilosamente y trepando vallas, liberando la información encerrada por las editoriales y compartiéndola con sus amigos.

Pero todas estas acciones suceden en la oscuridad, escondidas en la clandestinidad. Se les llama robo o piratería, como si compartir la riqueza del conocimiento

fuera el equivalente moral de saquear un barco y asesinar a su tripulación. Pero compartir no es inmoral —es un imperativo moral–. Solo aquellos que están cegados por la codicia se negarían a hacerle una copia a un amigo.

Las grandes corporaciones, por supuesto, están cegadas por la codicia. Las leyes bajo las que operan lo requieren —sus accionistas se sublevarían por mucho menos–. Y los políticos que se han comprado los apoyan, aprobando leyes que les dan el poder exclusivo de decidir quién puede hacer copias.

No hay justicia alguna en obedecer leyes injustas. Es tiempo de salir a la luz y en la gran tradición de la desobediencia civil, declarar nuestra oposición a este robo privado de la cultura pública.

Necesitamos tomar la información, donde sea que esté guardada, hacer nuestras copias y compartirlas con el mundo. Necesitamos tomar las cosas que están libres de derechos de autor y agregarlas a este archivo. Necesitamos comprar bases de datos secretas y ponerlas en la *web*. Necesitamos descargar *journals* científicos y subirlos a redes de compartición de archivos. Necesitamos pelear una guerrilla por el acceso abierto.

Si somos los suficientes, alrededor del mundo, no solo enviaremos un fuerte mensaje en oposición a la privatización del conocimiento —la haremos una cosa del pasado–. ¿Vas a unírtenos?

Julio del 2008, Eremo, Italia

En solidaridad con Library Genesis y Sci-Hub

Dušan Barok, Josephine Berry, Bodó Balázs, Sean Dockray, Kenneth Goldsmith, Anthony Iles, Lawrence Liang, Sebastian Lütgert, Pauline van Mourik Broekman, Marcell Mars, spideralex, Tomislav Medak, Dubravka Sekulić, Femke Snelting y otros

En el cuento de Antoine de Saint-Exupéry, *El Principito* conoce a un hombre de negocios que acumula estrellas con el único propósito de ser capaz de comprar más estrellas. El principito está perplejo. Él solo tiene una flor, la cual riega todos los días. Tres volcanes, los que limpia cada semana. «Es de algún uso para mis volcanes, y es de algún uso para mi flor, que los posea», él dice, «pero tú no eres de uso para las estrellas que tienes».

Hay muchos hombres de negocios que poseen el conocimiento ahora. Consideren a Elsevier, el publicador más grande de trabajo académico, cuyo 37 % de margen de ganancias[13] contrasta con los pagos que incrementan su precio, las deudas del préstamo estudiantil y el pobre

[13] Vicent Larivière, Stefanie Haustein y Philippe Mongeon, «The Oligopoly of Academic Publishers in the Digital Era». PLoS ONE 10, núm. 6. y «The Obscene Profits of Commercial Scholarly Publishers».

pago para la facultad adjunta. Elsevier tiene algunas de las base de datos más grandes de material académico, las que son licenciadas a precios tan escandalosamente altos que hasta Harvard, la Universidad más rica del norte global, se ha quejado de que no puede costearlos más. Robert Darnton, el antiguo director de la Harvard Library, dice «Nosotros como facultad hacemos investigación, escribimos *papers*, revisamos *papers* de otros investigadores, servimos en juntas editoriales, todo gratuitamente… y de ahí compramos los resultados de nuestra labor a precios escandalosos».[14] A pesar de todo el trabajo que se apoya con el dinero público que beneficia a publicaciones académicas, particularmente la revisión por pares que establece su legitimación, los artículos de revistas tienen tal precio que prohíben acceso a la ciencia a muchos académicos —y a todos los no-académicos— alrededor del mundo, y lo convierten en un símbolo de privilegio.[15]

Recientemente, Elsevier ha generado una acusación de infracción de derechos de autor en Nueva Tork en contra de Sci-Hub y Library Genesis declarando millones de dólares en daños.[16] Esto ha venido como un gran golpe, no solo para los administradores de estos sitios *web*, sino también para miles de investigadores alrededor del mundo para los que estos sitios son una fuente viable de materiales académicos. Los medios sociales, las listas de

[14] Ian Sample, «Harvard University Says It Can't Afford Journal Publishers' Prices». *The Guardian*.

[15] «Academic Paywalls Mean Publish and Perish», *Al Jazeera English*.

[16] «Sci-Hub Tears Down Academia's 'Illegal' Copyright Paywalls», TorrentFreak.

correo y los canales de IRC han sido llenados con sus mensajes de ayuda, buscando desesperadamente artículos y publicaciones.

Incluso cuando el Tribunal de Distrito de Nueva York estaba dando este mandamiento judicial, se publicó la noticia de que toda la junta editorial de la estimada revista *Lingua* había renunciado de manera colectiva, citando como la razón el rechazo a Elsevier y volverse al acceso abierto para dejar los altos costos que cobra a los autores y a las instituciones académicas. Mientras escribimos estas líneas, una petición está rondando que demanda que Taylor & Francis no cierre Ashgate,[17] una editorial de humanidades que adquirió a principios del 2015. Tiene el riesgo de ser como otras publicaciones pequeñas que están siendo adquiridas por el creciente monopolio y la concentración del mercado de publicaciones. Estas son solo algunas de las señales de que el sistema está roto. Desvaloriza a los autores, a los editores y a los lectores de la misma manera. Es parásita de nuestro trabajo, frustra nuestro servicio al público y nos niega el acceso.[18]

Tenemos las maneras y los métodos de hacer que el conocimiento sea accesible para todos, sin ninguna barrera económica al acceso y a un mucho menor costo para la sociedad. Pero el monopolio del acceso cerrado sobre las publicaciones académicas, sus ganancias espectaculares y su rol central en la asignación de prestigio académico triunfa sobre el interés público. Las editoriales

[17] «Save Ashgate Publishing», Change.org.
[18] «The Cost of Knowledge».

comerciales impiden efectivamente el acceso abierto, nos criminalizan, procesan a nuestros héroes y heroínas, y destruyen nuestras bibliotecas, una y otra vez. Antes de Sci-Hub y Library Genesis estaba Library.nu o Gigapedia; antes de Gigapedia estaba textz.com; antes de textz.com había poco; y antes de poco no había nada. Eso es lo que quieren: reducirnos a nada. Y tienen el apoyo completo de las cortes y las leyes para hacer exactamente eso.[19]

En el caso de Elsevier contra Sci-Hub y Library Genesis, el juez dijo: «Va en contra del interés público el simple hecho de hacer disponible gratuitamente contenidos con derechos de autor en un sitio *web* extranjero».[20] La petición original de Alexandra Elbakyan aumentó el riesgo: «Si Elsevier logra cerrar nuestros proyectos o forzarlos a ir a la *darknet*, se demostrará una idea importante: que el público no tiene derecho al conocimiento».

Demostramos diariamente, y en una escala masiva, que el sistema está roto. Compartimos nuestros artículos secretamente detrás de las espaldas de nuestros editores, saltamos los muros de pago para acceder a artículos y publicaciones, digitalizamos y subimos publicaciones a bibliotecas. Este es el otro lado del 37 % del margen de ganancia: nuestro conocimiento común crece en las líneas

[19] De hecho, con el TPP y el TTIP siendo negociados rápidamente, ningún registrador de dominios, proveedor de ISP, servidor u organización de derechos humanos será capaz de prevenir que las industrias de los derechos de autor y las cortes criminalicen y cierren sitios *web* con toda la prontitud posible.

[20] «Court Orders Shutdown of Libgen, Bookfi and Sci-Hub», TorrentFreak.

de un sistema roto. Todos somos guardianes del conocimiento, guardianes de las mismas infraestructuras de las que dependemos para producir conocimiento, guardianes de nuestros fértiles, pero frágiles bienes comunes. Ser un guardián es, *de facto*, descargar, compartir, leer, escribir, reseñar, editar, digitalizar, archivar, mantener bibliotecas y hacerlas accesibles. Es hacer uso, en lugar de no convertir en propiedad, de nuestro conocimiento común.

Hace más de siete años, Aaron Swartz, el cual no tomó ninguna precaución al levantarse por lo que aquí les urgimos que se levanten también, escribió:

> Necesitamos tomar la información, donde sea que esté guardada, hacer nuestras copias y compartirlas con el mundo. Necesitamos tomar las cosas que están libres de derechos de autor y agregarlas a este archivo. Necesitamos comprar bases de datos secretas y ponerlas en la *web*. Necesitamos descargar *journals* científicos y subirlos a redes de compartición de archivos. Necesitamos pelear una guerrilla por el acceso abierto.
>
> Si somos los suficientes, alrededor del mundo, no solo enviaremos un fuerte mensaje en oposición a la privatización del conocimiento —la haremos una cosa del pasado—. ¿Vas a unírtenos?[21]

[21] Aaron Swartz y anónimos, «Manifiesto de la guerrilla por el

Nos encontramos en un momento decisivo. Este es el tiempo de reconocer que la mera existencia de nuestro masivo conocimiento común es un acto colectivo de desobediencia civil. Es el tiempo de emerger de nuestro escondite y poner nuestros nombres detrás de este acto de resistencia. Tal vez te sientas aislado, pero hay muchos de nosotros. La furia, la desesperación y el miedo de perder nuestra infraestructura de bibliotecas, expresados por todo el internet, nos dice eso. Este es el tiempo de que los guardianes, ya sean perros, humanos o *cyborgs* con nuestros nombres, apodos y pseudónimos alcemos nuestras voces.

Comparte esta carta, leela en público, déjala en la impresora. Comparte lo que escribes, digitaliza un libro, sube tus archivos. No dejes que nuestro conocimiento se aplastado. Cuida de nuestras bibliotecas, cuida los metadatos, cuida los respaldos. Riega las flores, limpia los volcanes.

30 de noviembre del 2015

acceso abierto». Disponible en el capítulo anterior.

Índice general

Lo que hacemos: software *libre y acceso abierto* se terminó de imprimir durante el mes de abril del 2019 en Colima Hacklab.

Documento compuesto con LaTeX.