Introducción a la creación y mantenimiento de paquetes Debian

Antonio García Domínguez
Universidad de Cádiz
nyoescape@gmail.com

4 de marzo de 2008

Copyright © 2008 Antonio García Domínguez.

Legal notice

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

Me gustaría dar las gracias a mi profesor Francisco Palomo Lozano, del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Cádiz, por sus constantes sugerencias y comentarios.

Índice general

Índice general 5

1. Introducción 7
- 1.1. ¿Qué es un paquete Debian? 7
- 1.2. ¿Por qué se desarrollan paquetes? 7
- 1.3. ¿Quién desarrolla paquetes? 8
- 1.4. Notas acerca de esta guía 9

2. Preparativos 11
- 2.1. Cómo instalar las herramientas básicas 11
- 2.2. Cómo conseguir paquetes limpios 12
- 2.3. Cómo simplificar la distribución 13
- 2.4. Control de versiones 14
- 2.5. Autenticación 16
- 2.6. Distribución a través de Internet 18
  - 2.6.1. Instalación del servidor 18
  - 2.6.2. Instrucciones para un usuario de nuestro paquete 20

3. Creación y mantenimiento del paquete 23
- 3.1. Adaptaciones previas al uso de Subversion 23
  - 3.1.1. Creación de un esqueleto 23
  - 3.1.2. Edición 24
  - 3.1.3. Construcción preliminar 35
  - 3.1.4. Inyección en el repositorio 35
- 3.2. Preparación de una primera versión 36
  - 3.2.1. Construcción definitiva 36
  - 3.2.2. Verificación 36
  - 3.2.3. Envío al repositorio Debian 41
- 3.3. Actualización del paquete a una nueva versión del programa 41

4. Otros aspectos de interés 47
- 4.1. Integración de repositorio propio con la jaula *chroot* 47
- 4.2. Sincronización del repositorio local con un repositorio en Internet 47
- 4.3. Adaptación de aplicaciones Java 48
- 4.4. Actualización del escritorio 50
  - 4.4.1. Accesos directos e iconos 50
  - 4.4.2. Actualización de los tipos MIME 52

4.5. Generación automática de paquetes 53
4.5.1. Módulos Perl 53
4.5.2. Adaptación de guiones de instalación 54
4.6. Otros formatos de paquete 54
4.6.1. Comparativa con otros formatos 54
4.6.2. Conversión desde paquetes Debian 56

Bibliografía **57**

1 Introducción

1.1. ¿Qué es un paquete Debian?

Un paquete Debian, en principio, no es más que un fichero comprimido que contiene los ficheros de la aplicación, cierta información adicional y un par de guiones específicos de apoyo. A través de un paquete Debian, podemos instalar de forma muy sencilla cualquier software que necesitemos, sin tener que entrar en detalles de cómo está hecha o cómo se configura inicialmente la aplicación.

Además, la información adicional contenida en dicho paquete permite al sistema que se ocupa de gestionarlos (*dpkg*) mantener un control de las dependencias. Así, si instalamos una determinada aplicación, todas las bibliotecas requeridas por ésta serán automáticamente instaladas. El robusto control de dependencias del sistema de empaquetado de Debian es precisamente uno de sus puntos más fuertes frente a otros sistemas de empaquetado, como *RPM* (Red Hat Package Manager).

1.2. ¿Por qué se desarrollan paquetes?

Uno podría preguntarse las razones detrás del desarrollo de un paquete: ¿no podríamos simplemente compilar nosotros mismos el código? Efectivamente, podríamos, pero existen una serie de factores que hacen poco factible dicho enfoque:

- *Compilar lleva tiempo, y esfuerzo*: además del considerable tiempo de CPU requerido para compilar cualquier aplicación con un nivel mínimo de complejidad, hay que pensar en sus dependencias. Éstas son transitivas, con lo que tendremos que compilar una biblioteca que sea usada por otra biblioteca que sí use directamente el programa que queremos.

  Un ejemplo extremo puede ser el conocido reproductor multimedia *mplayer*, cuyas dependencias se extienden de forma recursiva a lo largo de decenas de bibliotecas, en una versión con toda la funcionalidad activada.

  También hay que pensar en los sistemas «exóticos» de compilación que algunos sistemas usan: no siempre tenemos la suerte de contar con sistemas estándar como el de las autotools, donde sabemos de antemano que basta con hacer esto en el 80 % de los casos:

  ```
  ./configure
  make
  make install
  ```

- *Falta de uniformidad*: Supongamos que ya hemos compilado todo. ¿Cómo instalamos el programa? No existe un enfoque uniforme a lo largo del gran número y variedad de aplicaciones existentes: cada proyecto es un mundo. Tampoco sabemos en principio cómo configurarlo.

  Además, es muy probable que el programa en su estado actual no esté pensado para ser instalado de dicha forma: puede que haya rutas escritas directamente en el código, o que se hagan ciertas suposiciones poco estándares.

  Más aún: ¿y las asociaciones de fichero? ¿Y la entrada del menú? ¿Estarán bien hechas, y funcionarán bajo todos los entornos de escritorio? ¿Se podrá desinstalar el programa?

En resumen, podría decirse que la utilidad de un paquete Debian se halla en garantizar una experiencia de instalación cómoda y uniforme a lo largo de todo el software disponible, independientemente de cómo esté hecho. Así, se pone algo de orden al problema que se presenta ante la gran configurabilidad y heterogeneidad que presentan los sistemas GNU/Linux.

De todas formas, para no perder la libertad que tenemos al descargar el código fuente, podemos instalar no un paquete con los ejecutables y bibliotecas compilados, sino con su código fuente adaptado para nuestra distribución. Son los paquetes de código fuente (*source packages*): normalmente se usan cuando queremos compilar partiendo de la misma base que el paquete binario de nuestra distribución.

Por ejemplo: el paquete `kernel-source-2.4.27` contiene el código fuente de la versión del kernel de Linux que emplea Debian. Este código trae de por sí una serie de modificaciones que lo diferencian del kernel que podríamos bajarnos de la página oficial, y se halla ya configurado con las mismas opciones que nuestro propio kernel.

1.3. ¿Quién desarrolla paquetes?

Así, vemos que hacer un paquete no es algo trivial, pudiendo requerir todo tipo de cambios a un programa. Sin embargo, su conveniencia les hace imprescindibles: no podemos esperar obtener un número aceptable de usuarios a menos que nuestro software sea fácil de instalar, configurar y usar.

Por ello, tendremos que distinguir entre dos roles, que podrán ser cubiertos por las mismas o distintas personas:

- Equipo de desarrollo del programa en sí (equipo original de desarrollo o *upstream*): son los que realmente se ocupan de añadir funcionalidad y corregir errores. El vocablo inglés, cuyo significado es «río arriba», hace referencia al hecho de que las nuevas funciones «fluyen» de este equipo hacia nuevas revisiones del paquete.

- Desarrolladores del paquete: son los que deben adaptar el programa al sistema de empaquetado, controlar las dependencias, y en resumen, hacer que el programa se integre bien con el resto del sistema. Sobre todo en las primeras versiones, es

posible que tenga que hacer cambios o mejoras al código de la aplicación, que tendrá que discutir con el equipo original de desarrollo.

Es frecuente que un paquete vaya pasando por distintos responsables, y que haya todo un equipo detrás y no sólo un responsable (como en el caso del Ubuntu MOTU Team[1], y se halla formado sobre todo por usuarios de dicha distribución).

1.4. Notas acerca de esta guía

Escribí esta guía para recoger cierta información que anda dispersa por la Red, y parte de mi (poca) experiencia como la combinación de un repositorio *reprepro* con sistemas de versionado y demás.

Existen otras guías, fantásticas sin duda, como [Wik08] y [Tea07a]. No intento reemplazarlas, sino complementarlas con más información de otros tipos que podría ser útil, como, por ejemplo, como conseguir tener fácilmente nuestro propio repositorio.

[1] Es el responsable de mantener los paquetes de los repositorios *universe* y *multiverse* de Ubuntu.

2 Preparativos

Antes de empezar a crear nuestro primer paquete, vamos a preparar el entorno con todas las herramientas que necesitaremos. Por supuesto, si no deseamos distribuir nuestro paquete a través de Internet, o gestionar varios paquetes dependientes entre sí, no nos será realmente necesario crear un repositorio local y publicarlo bajo un servidor Web.

> NOTA
>
> A lo largo de esta guía, supongo que el usuario empleará Ubuntu Linux «Gutsy Gibbon» 7.10. Las órdenes exactas podrían variar ligeramente, pero por lo general todo debería funcionar en cualquier sistema basado en Debian.

2.1. Cómo instalar las herramientas básicas

En primer lugar, requeriremos ciertas herramientas para empezar a desarrollar paquetes Debian. Para ello, ejecutaremos (como haremos a lo largo de esta guía) bajo una terminal de texto, como la disponible desde el menú principal en *Aplicaciones* → *Accesorios* → *Terminal*, la siguiente orden:

```
sudo aptitude install cdbs debhelper autotools-dev fakeroot \
    desktop-file-utils linda lintian devscripts dh-make debian-policy \
    automake-1.9 autoconf
```

Posteriormente a lo largo de este manual iremos describiendo para qué sirven algunos de estos paquetes. Adelantaremos algunos usos útiles de *dpkg*, que después nos serán de mucha ayuda:

`dpkg -l` *`patrón`* Permite buscar paquetes que cumplan un cierto patrón. Así, `dpkg -l '*sdl*'` nos buscará todos los paquetes relacionados con la biblioteca SDL. Es importante usar comillas simples, para evitar que una posible sustitución automática del shell interfiera con la orden.

`dpkg -L` *`paquete`* Lista todos los ficheros de un paquete determinado.

`dpkg -S` *`fichero`* Esta orden es muy útil: lista el paquete al cual pertenece un determinado fichero del sistema.

`dpkg-deb -c` *`fichero`* Lista los contenidos de un paquete Debian. Útil para comprobar que todo está en su lugar.

`dpkg-deb -x` *fichero ruta* Extrae los contenidos del paquete Debian a la ruta que le indiquemos.

`dpkg-deb -e` *fichero* Extrae el directorio de control `DEBIAN` de un paquete. Así podemos ver los guiones de postinstalación y postdesinstalación, por ejemplo, y otra información de interés.

2.2. Cómo conseguir paquetes limpios

Normalmente, desarrollamos el paquete Debian dentro de nuestro sistema habitual. Esto quiere decir que en nuestro sistema ya habrá un gran número de paquetes instalados por nosotros mismos. ¿Cómo podemos asegurarnos de que no hemos olvidado alguna dependencia? Peor aún: ¿se está compilando el programa con las bibliotecas que debiera, o está usando alguna versión que nosotros olvidamos haber instalado hace ya tiempo?

En el sistema de empaquetado Debian, estos problemas se resuelven creando e instalando el paquete no dentro del propio sistema, sino de una imagen de un sistema «limpio», sin otros paquetes que los esenciales. Dicha imagen se conoce como una *jaula* chroot, a partir de la aplicación del mismo nombre, que nos «encierra» dentro de un árbol de directorios, haciéndonos pensar de forma temporal que se trata del sistema de ficheros completo.

Adicionalmente, el uso de esta jaula nos permite probar el proceso de instalación completo una y otra vez sin peligro de dañar a nuestro sistema, y evitar posibles conflictos que puedan surgir.

Aunque nosotros mismos podríamos (con el suficiente tiempo y esfuerzo) hacernos una jaula y ejecutar las órdenes de construcción en ella manualmente, la mejor opción es hacer uso de alguna serie de guiones escritos específicamente para la construcción de paquetes Debian en éstas. Básicamente, lo que hacen es añadir el proceso de entrada y salida de la jaula sobre lo que ya hacen otros guiones como *dpkg-buildpackage* y *debuild*.

En principio, con *pbuilder* nos bastaría. Sin embargo, tiene un problema: la imagen que crea es simplemente un gran fichero `tar.gz`, que descomprime cada vez que construimos el paquete. Esto hace que sea mucho más lento que una construcción habitual. Lo combinaremos con *cowdancer*, que usa un mecanismo de *copy-on-write*[1] para evitarlo, reduciendo en gran medida dicha sobrecarga.

Así, los pasos a seguir son:

1. Instalamos los dos paquetes que necesitamos, junto con sus dependencias:

   ```
   sudo aptitude install pbuilder cowdancer
   ```

[1] Este mecanismo usa el árbol de directorios original directamente en un principio, y realiza todas las posteriores escrituras sobre copias de los originales, para mantener el árbol de directorios original limpio, y no tener que realizar una lenta copia completa por otro lado.

2. Generamos de forma automatizada la imagen del sistema Ubuntu con la siguiente orden. Este proceso puede llevar bastante tiempo: tiene que descargar un Ubuntu mínimo a través de la red.

```
sudo cowbuilder --create \
 --othermirror \
 "deb http://archive.ubuntu.com/ubuntu gutsy universe multiverse"
```

3. Tras un buen rato, nuestra imagen estará lista. Añadimos las siguientes líneas al fichero ~{}/.pbuilderrc:

```
# Seleccionamos los componentes de Ubuntu
# (son distintos a los de Debian, que son
#  main, contrib y non-free)
COMPONENTS="main universe multiverse"

# Uso de cowbuilder en vez de pbuilder normal (más rápido)
export PDEBUILD_PBUILDER=cowbuilder

# Aqui pondremos nuestro correo como desarrollador de paquetes
export DEBEMAIL="tudireccion@decorreo"
```

2.3. Cómo simplificar la distribución

Supongamos que alguien quiere instalar nuestro paquete. ¿Cómo lo consigue? ¿Se lo descarga de alguna web, teniendo que buscar una y otra vez cada vez que necesite algo? ¿Tendrá el cliente que comprobar una y otra vez dicha página web, o tendrá el desarrollador de la aplicación que implementar algún mecanismo de actualización automática, aunque sólo se trate de un guión Perl? No olvidemos que, además, en el caso de una distribución GNU/Linux, podemos tener miles de paquetes instalados (1646 en mi caso, retirando los paquetes esenciales). No parece factible simplemente descargarlo de alguna página web.

En realidad lo que se hace es agrupar los paquetes en repositorios. Se conoce como *repositorio Debian* a un árbol de directorios con una cierta estructura, desde el que el sistema de empaquetado de software de Debian, *dpkg*, pueda instalar software. Dicho repositorio puede estar disponible de forma local en nuestro disco duro, como el que haremos aquí, o en otra máquina que aloje un servidor HTTP (web) o FTP.

Este paso sólo es estrictamente necesario cuando tengamos que construir nosotros mismos varios paquetes que dependan entre sí. De todas formas, recomiendo seguirlo, ya que nos permite depurar ciertas cosas como que el paquete se halle correctamente firmado y nuestra clave pública bien distribuida. Además, podemos usarlo para actualizar a partir de él nuestra versión instalada del mismo paquete, y así reproducir exactamente lo que un usuario normal vería.

Podríamos gestionar manualmente el contenido del repositorio a través de las herramientas del paquete `apt-utils`, pero por simplicidad, usaremos el sistema *reprepro* (anteriormente *mirrorer*) para gestionar todo el proceso.

Seguiremos estos pasos:

1. Instalamos el paquete correspondiente con:

   ```
   sudo aptitude install reprepro
   ```

2. Creamos el repositorio dentro de `/var/packages/ubuntu`:

   ```
   sudo mkdir -p /var/packages/ubuntu/conf
   ```

3. Situamos en el fichero `/var/packages/ubuntu/conf/distributions` las siguientes líneas:

   ```
   Origin: Mi Nombre
   Label:  Mi Nombre
   Codename: gutsy
   Architectures: i386 source
   Components: main
   Description: Mi propio repositorio
   ```

   Como vemos, tenemos una distribución, *gutsy*, que contiene paquetes binarios para la arquitectura IA-32 (Intel 80386 o superior) y paquetes de código fuente repartidos en un único componente *main*.

4. Ahora ya podemos manipular el repositorio de distintas formas. Posteriormente veremos cómo obtener los ficheros `.deb` y `.dsc`. Por ejemplo:

   - Añadimos paquetes binarios con:

     `sudo reprepro -b /var/packages/ubuntu includedeb gutsy (fichero .deb)`

   - Añadimos paquetes de fuentes con:

     `sudo reprepro -b /var/packages/ubuntu includedsc gutsy (fichero .dsc)`

   - Retiramos un paquete mediante:

     `sudo reprepro -b /var/packages/ubuntu remove gutsy (nombre)`

2.4. Control de versiones

Todo desarrollo de software se puede beneficiar del uso de un sistema de control de versiones. A través de éste, podremos siempre acceder a todas las versiones realizadas en el pasado de cualquier fichero, y mantener múltiples copias de trabajo siempre

sincronizadas. Son especialmente importantes en proyectos de software libre, donde puede haber potencialmente muchos participantes.

Además, el uso explícito de un sistema de control de versiones elimina una molestia a la hora de desarrollar paquetes Debian: la avalancha de ficheros que tendríamos que gestionar de otra forma, con ficheros `tar.gz` (más conocidos como *tarballs*), parches y demás que habrá que mantener versión a versión.

Como siempre, existe una gran variedad, pero en este caso elegí Subversion, descendiente a nivel conceptual del conocido CVS, por su excelente documentación, estabilidad y simplicidad. Adicionalmente, podremos hacer uso de un conjunto de guiones de ayuda desarrollados por la comunidad Debian, que nos ayudan a estandarizar un poco la estructura del repositorio.

> NOTA
>
> Para los preocupados por su espacio en disco: el enfoque seguido por Subversion, al igual que en la mayoría de los sistemas de control de versiones, es mucho más eficiente que lo que nosotros podríamos hacer simplemente guardando todas las versiones. Sólo guarda, comprimidos, los cambios de una versión a otra.

El proceso de preparación de nuestro propio repositorio de Subversion y una copia de trabajo local es sencillo:

1. Instalamos Subversion y el paquete de desarrollo de paquetes (valga la redunancia) bajo repositorios Subversion:

   ```
   sudo aptitude install subversion svn-buildpackage
   ```

2. Creamos el repositorio central en `~{}/.svnDebian`:

   ```
   svnadmin create .svnDebian
   ```

3. Ahora creamos una copia de trabajo, sobre la que crearemos y modificaremos ficheros, que luego enviaremos al repositorio central para que otras personas puedan propagar los cambios a sus propias copias de trabajo:

   ```
   svn checkout file:///home/minombredeusuario/.svnDebian packages
   ```

Por lo pronto, no haremos nada más: la estructura de nuestro repositorio será generada de forma automática, y diferirá ligeramente de la estándar que el libro de Subversion describe.

A título de referencia para la fase de creación, he aquí algunas de las órdenes que podemos usar dentro de una copia de trabajo. Para más información, referirse al libro electrónico [CSFP07].

`svn add` *`fichero`* Marca un fichero o directorio para añadir al repositorio.

`svn rm` *`fichero`* Marca un fichero o directorio para eliminar del repositorio.

`svn mv` *`fichero ruta`* Mueve un fichero o directorio a otra ruta. El fichero o directorio no debe haber sufrido modificaciones desde su último envío.

`svn cp` *`fichero ruta`* Copia un fichero o directorio a otra ruta. El fichero o directorio no debe haber sufrido modificaciones desde su último envío.

`svn mkdir` *`directorio`* Crea un nuevo directorio y lo añade al repositorio.

`svn commit -m` *`mensaje`* Confirma todos los cambios hechos hasta el momento, y los envía al repositorio, registrando el envío con el mensaje que hayamos proporcionado. Tiene semántica transaccional: el envío se produce por completo, o no se produce en absoluto.

`svn status` Muestra el estado de todos los ficheros bajo el árbol actual, con una letra antes de su ruta. Algunas de las que pueden aparecer son:

- **?** No pertenece al repositorio (quizás habría que añadirlo con `svn add`).
- **A** El fichero va a ser añadido en el próximo envío.
- **D** El fichero va a ser borrado en el próximo envío.
- **M** El fichero ha sido modificado desde el último envío.

`svn export` *`rutaDirOrigen rutaDestino`* Exporta el directorio origen, que forma parte de una copia de trabajo, a la ruta destino, retirando todos los ficheros usados por Subversion. Muy útil cuando queremos redistribuir código, por ejemplo, y no la copia de trabajo entera.

2.5. Autenticación

Sólo nos queda una cosa más para tener el entorno completamente listo antes de desarrollar nuestro paquete Debian: preparar nuestra firma digital personal. Adicionalmente, la integraremos con nuestro repositorio local, y usaremos un agente de claves para evitar escribir una y otra vez su contraseña a la hora de construir el paquete.

¿Por qué querríamos usar una? No olvidemos que estamos haciendo un paquete que efectivamente va a ser instalado en el sistema final bajo la propiedad del superusuario. Una persona malintencionada podría manipular los contenidos del paquete y comprometer a los sistemas que lo instalen, añadiendo *rootkits* o puertas traseras entre nuestros ficheros.

Para un usuario muy centrado en seguridad, el mayor nivel de seguridad obtenible con un nivel razonable de esfuerzo, sin llegar a tener que analizar en profundidad el código, sería directamente descargar el código de alguna fuente fiable, y comprobar su integridad mediante algún algoritmo de *hashing*. Lo más común es encontrar *checksums* MD5, que aunque no criptográficamente seguros, son rápidos de calcular, y cumplen el propósito original de evitar descargas corruptas.

No llegaremos a esos extremos. En lugar de eso, aprovecharemos una de las aplicaciones de los algoritmos de cifrado asimétrico. En ellos, todo usuario tiene dos claves:

la pública, que damos a todo el mundo, y la privada. Si ciframos algo con una clave, se puede descifrar con la otra. Así no nos hace falta transmitir la clave a la otra parte, como en los algoritmos tradicionales simétricos. En particular, si ciframos algo con nuestra clave privada, y el usuario lo descifra con nuestra clave pública, éste puede estar seguro que el paquete es auténtico.

Dado que cifrar todo un paquete puede ser potencialmente muy costoso, lo que se suele hacer es cifrar el resultado de alguna función de hash aplicada sobre el fichero, como SHA-1 o alguna variante criptográficamente más fuerte. De hecho, en los repositorios Debian, se va un paso más allá: sólo se firma el fichero central con el listado de los paquetes disponibles, sus tamaños y sus *checksums* MD5, para tener el menor impacto posible en rendimiento. En nuestra configuración, se trata de `/var/packages/ubuntu/dists/gutsy/Release`, y su firma se halla en `/var/packages/ubuntu/dists/gutsy/Release.g`

El proceso de instalación es algo más elaborado que en los anteriores casos, dado que se extiende a lo largo de unos cuantos ficheros de configuración:

1. Primero instalamos los paquetes que necesitamos: `gnupg` implementa la funcionalidad de cifrado en sí, `gnupg-agent` es el agente ocupado de almacenar temporalmente nuestra contraseña y `pinentry-gtk2` nos permite usar un diálogo gráfico para introducir la contraseña, en vez de usar la terminal:

```
sudo aptitude install gnupg gnupg-agent pinentry-gtk2
```

2. Ahora generaremos nuestro par de claves, siguiendo las instrucciones que se nos irán indicando:

```
gpg --gen-key
```

   El campo de correo electrónico es muy importante para el resto de pasos, y debe coincidir con el que usamos anteriormente en DEBEMAIL (ver sección 2.2 en la página 12).

3. Añadiremos los ajustes necesarios al agente de GnuPG, en el fichero `~{}/.gnupg/gpg-agent.conf`:

```
pinentry-program /usr/bin/pinentry-gtk-2
no-grab
default-cache-ttl 1800
```

   Así, sólo tendremos que introducir nuestra contraseña cada 30 minutos como mucho, y emplearemos una interfaz gráfica basada en GTK 2.0 (estilo GNOME).

4. Hemos de indicar a GnuPG que haga uso del agente para pedir la contraseña de la clave privada. Hay que descomentar la siguiente línea de `~{}/.gnupg/gpg.conf`:

```
use-agent
```

5. Al instalar el paquete `gnupg-agent` se nos añadió el script necesario para su arranque en `/etc/X11/Xsession.d/90gpg-agent`. Hemos de reiniciar el servidor X, saliendo de nuestra sesión y pulsando `CTRL + ALT + Retroceso`.

6. Debemos decirle a *reprepro* que use dicha clave para firmar los paquetes. Añadiremos la siguiente línea con el email que usamos en la clave GPG a `/var/packages/ubuntu/conf/dis`

   ```
   SignWith: <tudireccion@decorreo>
   ```

7. Por último, hemos de exportar nuestra clave pública a un fichero, para poder pasársela a los demás, y añadirla a la lista de claves confiables de *apt*:

   ```
   gpg -a --export tudireccion@decorreo > claveDebian.asc
   sudo apt-key add claveDebian.asc
   sudo cp claveDebian.asc /var/packages
   ```

2.6. Distribución a través de Internet

2.6.1. Instalación del servidor

Podemos aprovechar nuestro repositorio local para distribuir nuestros paquetes a través de la red, simplemente publicando el directorio `/var/packages` bajo un servidor HTTP o FTP. En particular, aquí veremos cómo hacerlo con el servidor Apache 2.

Evidentemente, hacerlo o no es completamente opcional. Los pasos a seguir son:

1. Si no tenemos una dirección IP estática, podemos en su lugar registrarnos en sitios como no-ip (`http://www.no-ip.org`) o DynDNS (`http://dyndns.com`), donde conseguiremos un nombre de dominio que actualizaremos periódicamente con nuestra IP.

2. Ahora hemos de asegurarnos de que el puerto 80 por TCP se halle accesible, cambiando los ajustes del cortafuegos y del encaminador en caso de que usemos NAT (Network Address Translation). Con Ubuntu recién instalado, el cortafuegos no tendrá ninguna regla definida, por lo que en principio no habría que hacer nada en cuanto a la configuración del servidor.

3. Instalamos el servidor Apache mediante:

   ```
   sudo aptitude install apache2
   ```

4. Crearemos el fichero `/etc/apache2/sites-available/repoDebian`, con el siguiente contenido, donde sustituiremos la dirección de nuestro servidor y el correo que empleamos en la firma GPG:

```
<VirtualHost *:80>
        ServerAdmin tudireccion@decorreo

        DocumentRoot /var/packages
        ServerName direccion.delservidor.org:80
        ErrorLog /var/log/apache2/error.log

        LogLevel warn

        CustomLog /var/log/apache2/access.log combined
        ServerSignature On

        # Permite que la gente navegue por el directorio
        <Directory "/var/packages">
                Options Indexes FollowSymLinks MultiViews
                DirectoryIndex index.html
                AllowOverride Options
                Order allow,deny
                allow from all
        </Directory>

        # Oculta el directorio conf
        <Directory "/var/packages/*/conf">
                Order allow,deny
                Deny from all
                Satisfy all
        </Directory>

        # Oculta el directorio db
        <Directory "/var/packages/*/db">
                Order allow,deny
                Deny from all
                Satisfy all
        </Directory>
</VirtualHost>
```

5. Activaremos dicha página web y desactivaremos la página por defecto:

```
cd /etc/apache2/sites-enabled
sudo a2dissite default
sudo a2ensite repoDebian
```

6. Reiniciamos el servidor mediante:

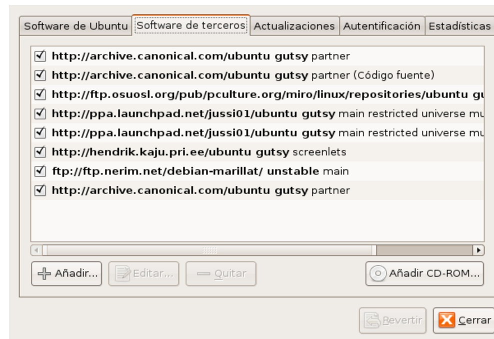

Figura 2.1: Pestaña de Software de Terceros

```
sudo /etc/init.d/apache2 restart
```

7. ¡Listo! Habría que probar a cargar en un navegador la dirección `http://127.0.0.1/` y ver que todo coincide.

2.6.2. Instrucciones para un usuario de nuestro paquete

Ahora nos pondremos por un momento en la piel de un usuario de nuestro paquete. ¿Cómo podría tener nuestro paquete siempre a la última, con la seguridad de que es la versión genuina?

En este caso, vamos a intentar hacerlo todo a través de interfaces gráficas, más amigables para un usuario medio. Recomiendo añadirnos a nosotros mismos como un cliente más, para facilitar la depuración. Sólo tendríamos que sustituir nuestro nombre de host real con `127.0.0.1` en el resto de instrucciones, o mejor aún, añadir la siguiente línea a `/etc/hosts`, reemplazando la que tuviera la misma dirección si hubiera alguna:

```
127.0.0.1 localhost direccion.delservidor.org
```

En primer lugar, el usuario debe de haber descargado el fichero `http://`*`direccion.delservidor.org`* que exportamos antes, con nuestra clave pública.

Accederemos, a partir de la barra de programas en la parte superior de la pantalla, al elemento *Sistema → Administración → Orígenes de software*.

En el diálogo que aparece, pasamos a la pestaña *Software de terceros*, cuyo aspecto será similar al de la figura .

Pulsaremos en el botón `Añadir` una primera vez, y pegaremos la siguiente línea:

```
deb http://tunombre.deservidor.org/ubuntu gutsy main
```

Volvemos a pulsarlo, y ahora introducimos ésta, correspondiente a un repositorio de paquetes de código fuente (de ahí el *deb-src*):

```
deb-src http://tunombre.deservidor.org/ubuntu gutsy main
```

Figura 2.2: Pestaña de Autentificación

Cambiamos a la pestaña *Autentificación*, que será parecida a la de la figura , y pulsamos `Importar clave`. Seleccionaremos el fichero `claveDebian.asc` antes descargado.

Guardamos los cambios realizados pulsando `Cerrar`, y finalmente pulsamos el botón `Recargar` de la barra de herramientas de la ventana principal. Cuando se cierre automáticamente el diálogo que aparece, ya podremos usar como siempre *Synaptic*, empleando el botón `Buscar` para hallar el paquete que deseamos, marcarlo con un doble clic e instalarlo mediante `Aplicar`. Además, a partir de ahora, el sistema *apt* se ocupará de monitorizar dicho repositorio y mantener actualizado el sistema.

3 Creación y mantenimiento del paquete

En este capítulo, daré un ejemplo de un paquete razonablemente sencillo, pero completo: un emulador de SNES, conocido como ZSNES. Veremos todas las fases, desde que nos descargamos el código fuente hasta que tenemos el paquete instalado en nuestro sistema y funcionando. Existen muchas guías de creación de paquetes, pero en mi opinión la información se halla bastante fragmentada. De todas formas, el Ubuntu MOTU Team tiene excelentes introducciones acerca del tema en su wiki [Tea07a], y también hay una guía por parte de la comunidad Debian [RA07], aunque en mi opinión la de Ubuntu está más actualizada.

Para simplificar, describiré el proceso de forma secuencial. Sin embargo, lo normal es que sea iterativo, teniendo muchas revisiones intermedias del paquete hasta dejarlo listo para su distribución. Se ven aspectos más avanzados en el siguiente capítulo.

3.1. Adaptaciones previas al uso de Subversion

3.1.1. Creación de un esqueleto

Antes de poder introducir los ficheros fuente de nuestro paquete en el repositorio Subversion que previamente preparamos, hemos de crear una primera versión de nuestro paquete.

Tras descargar el código fuente de la página oficial (`http://www.zsnes.com/index.php?page=files`) a `/tmp/packages/zsnes151src.tar.bz2`, crearemos el esqueleto básico del paquete mediante `dh_make`, una de las muchas herramientas del paquete *debhelper* de ayuda. Ejecutaremos las siguientes órdenes en una terminal dentro del directorio `/tmp/packages`:

```
tar -xjf zsnes151src.tar.bz2
mv zsnes_1_51 zsnes-1.510
cd zsnes-1.510
dh_make -e tudireccion@decorreo -c GPL -s --createorig
```

La carpeta que hemos creado (`zsnes-1.510`) obedece al convenio seguido por Debian `nombrepaquete-versionUpstream`, donde la versión del programa original se entiende como una serie de números separados por puntos: así, `zsnes-1.510` es más reciente que `zsnes-1.6`, y menos que `zsnes-1.600`.

Por otro lado, las opciones pasadas a `dh_make` son:

-e tudireccion@decorreo Especifica nuestra dirección email como desarrollador del paquete. Se usa en el registro de cambios (de ahora en adelante el *Changelog*), y para realizar firmas digitales.

-c GPL Indica que el código original sigue la General Public License. Otras opciones incluyen la LGPL, BSD o la licencia artística. En otro caso, se nos dejará un hueco (posteriormente veremos dónde) para que lo rellenemos con el texto de la licencia en cuestión.

-s Existen varios tipos de paquete Debian: de un solo binario, de varios, bibliotecas, o paquetes que emplean CDBS (el resto usan únicamente *debhelper*). Aquí hemos decidido hacer un paquete de un solo binario, mediante debhelper. Después veremos también cómo hacer un paquete con CDBS.

–createorig Creamos en el directorio padre un fichero `zsnes_1.510.orig.tar.gz` con el código fuente original, para poder comparar con la versión que usemos al construir el paquete y volcar las diferencias a un fichero `diff`.

3.1.2. Edición

Ya tenemos el esqueleto del paquete. Todos los ficheros específicos de él se hallan bajo el directorio `/tmp/packages/zsnes-1.510/debian`. Si examinamos dicho directorio, veremos que hay un gran número de ficheros. No utilizaremos los ejemplos incluidos, indicados por la extensión `.ex`, así que los retiraremos, junto con el fichero `README.Debian`, dado que no hay nada especial acerca de nuestro paquete:

```
rm debian/*.ex,EX debian/README.Debian
```

Iremos rellenando cada fichero de control en `debian` con los datos necesarios. Iremos detallando su sintaxis y semántica a lo largo de esta sección.

changelog

Éste es el registro de cambios de nuestro paquete. Aquí iremos indicando los cambios realizados a lo largo de cada versión del paquete, no del software original. Este fichero es el que nuestros usuarios leerán para ver qué hay de nuevo en cada versión del paquete.

Escribiremos nuestra primera entrada:

```
zsnes (1.510-0ubuntu1) gutsy; urgency=low

  * Versión inicial del paquete

 -- Antonio Garcia <nyoescape@gmail.com>  Fri, 19 Feb 2008 13:49:12 +0100
```

Vemos cómo la versión actual del paquete junto con su última fecha y autor del cambio se hallan codificados en el registro. También se tiene en cuenta la distribución (en nuestro caso *gutsy*, de Ubuntu), y la urgencia del cambio (por lo general baja, a menos que se trata de una vulnerabilidad de seguridad o algo del estilo).

Para una misma versión del software original, tendremos distintas versiones del paquete, separadas del número de versión original por un guión, como vemos aquí. En particular, los paquetes de Ubuntu [Duc98] usan el esquema `-XubuntuY`, indicando que se trata de la Y-ésima versión del paquete de Ubuntu originado de la X-ésima versión del paquete Debian (0 si no proviene de un paquete Debian). Los números de versión de paquete comienzan por 1.

> NOTA
>
> La razón de este esquema de versionado es para permitir una fácil integración con los paquetes Debian. Normalmente, la política de Ubuntu es sólo crear nuevos paquetes o versiones de éstos si el paquete Debian está anticuado o tiene algún problema. Así, si los de Debian sacan una nueva versión, como la `-3`, a partir de `-2ubuntu3`, se reflejará dicha información de forma correcta.

Así, para la próxima versión del paquete, sólo tendremos que añadir la entrada en cuestión al registro, y nuestros guiones de ayuda harán el resto del trabajo.

> NOTA
>
> Mucho cuidado con el formato del registro, es muy rígido. El espaciado debe ser exactamente el mismo que en el ejemplo, como los dos espacios entre la dirección de correo y la fecha, o el espacio inicial al inicio de la misma línea.

compat

Para este fichero no hay que hacer nada: sólo contiene un número entero, indicando qué versión del paquete *debhelper* estamos usando.

control

Este fichero es muy importante: describe todos los paquetes que estamos definiendo y enuncia sus dependencias. El formato es también bastante rígido, pero muy simple. Utiliza una serie de campos delimitados por ':' y saltos de línea.

Por supuesto, no existe ninguna receta mágica que nos diga las dependencias de un programa cualquiera. Para ello, normalmente tendremos que examinar la documentación del desarrollador original, y/o el guión de compilación que utilice: como aquí usan las *autotools*, podríamos consultar `src/configure.in`. Una buena referencia respecto a las *autotools* es el Autobook [VETT06].

Por suerte, los desarrolladores de ZSNES han incluido dichas dependencias en `docs/install.txt`, con lo que no tendremos que ir buscando en los guiones de compilación.

El fichero que usaremos será éste:

```
Source: zsnes
Section: games
Priority: optional
Maintainer: Antonio Garcia <nyoescape@gmail.com>
Build-Depends: cdbs, debhelper (>= 4.1.0), autotools-dev, fakeroot,
  desktop-file-utils, g++ (>= 4), libsdl1.2-dev, nasm (>= 0.98),
  zlib1g-dev (>= 1.2.3), libpng12-dev (>= 1.2), libncurses5-dev,
  libgl1-mesa-dev
Standards-Version: 3.7.2

Package: zsnes
Architecture: i386
Depends: $shlibs:Depends
Description: Emulador de Super Nintendo
 Emulador de la consola Super Nintendo con más funciones disponibles.
 Permite guardar y cargar estados, grabar demostraciones, y aplicar
 diversos filtros. Tiene una compatiblidad inmejorable.
```

> NOTA
>
> En éste y en cualquier otro fichero de control de Debian, no debemos olvidar poner un salto de línea justo al final del fichero.

Examinando el fichero de campo a campo, tenemos:

Source: zsnes Indica que el paquete fuente del que derivan todos se llama "zsnes".

Section: games Por la política de Debian [JS98], todo paquete se halla en alguna sección de las disponibles. Así indicamos qué tipo de aplicación es: un juego, un editor, etc.

Priority: optional Indica la importancia del paquete: desde imprescindibles (*required*), pasando por importantes (*important*), estándar (*standard*), opcionales (*optional*), y extra (tienen conflicto con alguno de más prioridad).

Maintainer: Antonio Garcia <nyoescape@gmail.com> Nombre y dirección de contacto del desarrollador del paquete.

Build-Depends: ... Paquetes requeridos para poder compilar este paquete. Incluye las herramientas para paquetes Debian y las dependencias del propio programa.

Standards-Version: 3.7.2 Versión de la política de Debian que este documento sigue. Realmente se halla compuesta por varios documentos, todos situados bajo el directorio `/usr/share/doc/debian-policy`.

Package: zsnes Nombre de uno de los paquetes binarios generados a partir del fuente. Aquí sólo hay uno y tiene el mismo nombre.

Architecture: i386 Arquitectura a la que va dirigida el paquete. Existe una gran variedad de valores, pero nos interesan sobre todo *i386* (la IA-32 habitual), *source* (código fuente) y *all* (código sin una arquitectura definida, como programas Java, o guiones de algún lenguaje interpretado como Perl o Python).

Depends Paquetes requeridos para que éste se instale y funcione correctamente. La variable `${shlib:Depends}` incluye las dependencias deducidas de forma automática en cuanto a bibliotecas dinámicas se refiere.

Description Incluye una descripción corta de una sola línea y otra más larga de varias líneas del contenido del paquete. Al igual que siempre, su formato es muy rígido: toda línea de la descripción larga comienza por un espacio, y líneas vacías únicamente añaden un punto ('.'). El campo termina tras la primera línea sin dicho espacio inicial.

> NOTA
>
> Mucho cuidado con los acentos y demás en el nombre del desarrollador del paquete y otros campos: podrían causar problemas en el interior de la jaula *chroot*, que sólo tiene soporte para los caracteres ASCII de 7 bits.

copyright

Contiene la información relativa a la licencia del paquete y del programa original, junto con datos acerca de los autores originales, su copyright y de dónde descargamos el código fuente.

En este caso sólo tenemos que rellenar sin más los campos. No se fuerza ningún formato particular sobre el fichero. Es importante sustituir Üpstream Author(s)"por Üpstream Authorsz fijarnos en la información en `docs/authors.txt` del código fuente, o los verificadores de paquetes que veremos después darán avisos al respecto.

dirs

En este fichero listamos los directorios en que vamos a instalar algún fichero. Si no lo listamos aquí, dicho directorio no va a hallarse disponible durante la construcción del paquete, así que hay que tener cuidado. Las rutas deben de seguir el Filesystem Hierarchy Standard (FHS), disponible a través de la orden `man hier` desde cualquier terminal.

Algunas rutas importantes y sus contenidos son:

`/bin` Ejecutables usados en modo monousuario. Normalmente realizan tareas de mantenimiento a bajo nivel, entre otras cosas. Instalados a través de paquetes Debian.

`/boot` Configuración de GRUB, ficheros de imagen de los *kernels* disponibles, etc.

`/dev` Árbol de directorios donde cada dispositivo conectado al sistema es un fichero.

`/etc` Ficheros de configuración global (para todos los usuarios).

`/home` Directorios de casa de cada usuario, con espacio para cada uno de ellos.

`/mnt` Dispositivos externos montados temporalmente: particiones de Windows, CD, DVD, pendrives, etc.

`/proc` Árbol de directorios con información del *kernel* en cada fichero: procesos en ejecución, dispositivos disponibles, etc.

`/root` Directorio de casa del superusuario.

`/sbin` Ejecutables para uso del superusuario.

`/usr` Datos, programas y bibliotecas compartidos por todos los usuarios.

`/usr/bin` Ejecutables para todos los usuarios, instalados a través de paquetes Debian.

`/usr/lib` Bibliotecas para todos los usuarios, instalados a través de paquetes Debian.

`/usr/local` Similar a `/usr`, pero para uso del administrador.

`/usr/share` Datos compartidos por todos los usuarios.

`/usr/share/man` Páginas de *man* disponibles. Toda página se halla dentro de una sección. En particular, la de *zsnes* estaría en la 1, tras ver las instrucciones disponibles a través de la orden `man man`.

Dado que tenemos que instalar el ejecutable para todos los usuarios y una página *man*, nuestro fichero `dirs` contendrá:

```
usr/bin
usr/share/man/man1
```

> NOTA
>
> En este fichero, las rutas *no* incluyen una barra inicial, como suelen hacer. Para ser más exactos, son rutas a crear dentro del área temporal de construcción del directorio `debian/zsnes`, donde colocaremos todos los ficheros tal y como se descomprimirán después bajo el directorio raíz, `/`.

`rules`

Aquí está el fichero más importante de todos. Es el que decide qué hay que hacer exactamente para compilar e instalar el paquete completo: documentación, binarios, guiones y ficheros de datos.

De todas formas, en términos generales, no es más que un *makefile*, si bien uno que puede hacerse muy complejo: el objetivo *build* compila, *install* instala dentro del área de construcción del paquete, y *clean* retira los ficheros generados durante la construcción. Esta última se halla bajo la ruta relativa `debian/zsnes` respecto del directorio principal del paquete.

Dado que escribir una y otra vez un *makefile* completo para muchas aplicaciones parecidas era una pérdida de tiempo, se han desarrollado diversos paquetes que factorizan cierta funcionalidad común, como instalar páginas *man*, tipos MIME, entradas de menú, y cosas del estilo: son los guiones del paquete `debhelper`. Prácticamente nadie hoy en día desarrolla sus paquetes sin estos guiones.

Algunos desarrolladores han decidido ir un paso más allá, y factorizar reglas para perfiles completos de aplicaciones. Así, si sabemos que se trata de una aplicación desarrollada a través de las autotools, sólo tendremos que aplicar dicho perfil, añadiendo las opciones oportunas que pasar a `configure`, por ejemplo. Esto es el Common Debian Build System (CDBS) [DP07], que usaremos en esta guía. Existen perfiles para aplicaciones Python, Perl, GNOME, KDE, Java (basadas en Ant), o incluso para aquellas con un simple *makefile*.

Por supuesto, para paquetes complicados, este sistema se queda corto, pero son minoría comparados con los demás. Además, no es sólo cuestión de simplicidad: factorizando la mayor proporción posible de reglas, blindaremos nuestro paquete ante cambios en la política de Debian en el futuro.

De todas formas, en general, la comunidad de desarrolladores se halla muy dividida entre usar o no CDBS: aunque factoriza mucha complejidad, resulta difícil de comprender y aprovechar en casos difíciles, a menos que seamos capaces de leer complejos ficheros *makefile* por nosotros mismos, dado que no existe mucha documentación detallada al respecto: para CDBS, el código es la mejor documentación.

Dado que resulta imposible entender bien CDBS si no se comprende antes el sistema tradicional, en esta sección explicaremos las dos alternativas. Comenzaremos por el sistema "tradicionalçon los guiones de debhelper, y luego veremos cómo CDBS factoriza la mayor parte de estas reglas.

Reglas con *debhelper* Partiendo del esqueleto que automáticamente nos ha creado `dh_make`, lo retocamos para este paquete en particular. Vamos a ver qué tal ha quedado, y luego explicaremos qué partes exactamente hemos cambiado, y por qué:

```
#!/usr/bin/make -f
# -*- makefile -*-

# This file was originally written by Joey Hess and Craig Small. As a
# special exception, when this file is copied by dh-make into a dh-make
```

```
# output file, you may use that output file without restriction.  This
# special exception was added by Craig Small in version 0.37 of dh-make.

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

# El código se halla en un subdirectorio, no en la raíz
SRCDIR = src
# Opciones a pasar a configure (--enable-release activa optimizaciones)
CONFIGURE_FLAGS = --disable-cpucheck --enable-release --with-x --with-opengl
# Opciones a usar en el compilador
CFLAGS = -Wall -g

ifneq (,$(findstring noopt,$(DEB_BUILD_OPTIONS)))
CFLAGS += -O0
else
CFLAGS += -O2
endif

configure: configure-stamp
configure-stamp:
dh_testdir
touch configure-stamp

build: build-stamp

build-stamp: configure-stamp
dh_testdir

cd $(SRCDIR) && \
          force_arch=i586 ./configure $(CONFIGURE_FLAGS) --prefix=/usr
$(MAKE) -C $(SRCDIR)

touch $@

clean:
dh_testdir
dh_testroot
rm -f build-stamp configure-stamp

# Limpiamos también las herramientas internas usadas por ZSNES en su
# compilación
-$(MAKE) -C $(SRCDIR) clean tclean
# Borramos los ficheros temporales generados por la compilación
$(RM) $(SRCDIR)/tools/depbuild $(SRCDIR)/config.log,status,h $(SRCDIR)/Makefile
```

```
dh_clean

install: build
dh_testdir
dh_testroot
dh_clean -k
dh_installdirs

$(MAKE) -C $(SRCDIR) install DESTDIR=$(CURDIR)/debian/zsnes

# Build architecture-independent files here.
binary-indep: build install
# We have nothing to do by default.

# Build architecture-dependent files here.
binary-arch: build install
dh_testdir
dh_testroot
dh_installchangelogs
dh_installdocs
dh_installexamples
# dh_install
# dh_installmenu
# dh_installdebconf
# dh_installlogrotate
# dh_installemacsen
# dh_installpam
# dh_installmime
# dh_python
# dh_installinit
# dh_installcron
# dh_installinfo
dh_installman $(SRCDIR)/linux/zsnes.1
dh_link
dh_strip
dh_compress
dh_fixperms
# dh_perl
# dh_makeshlibs
dh_installdeb
dh_shlibdeps
dh_gencontrol
dh_md5sums
dh_builddeb
```

```
binary: binary-indep binary-arch
.PHONY: build clean binary-indep binary-arch binary install configure
```

Aunque es bastante largo, conceptualmente es sencillo, gracias al uso de *debhelper*. Un par de cosas a destacar:

1. Dado que nuestro código fuente se halla bajo un subdirectorio y no en el directorio raíz del paquete, añadimos una variable `SRCDIR`, cuyo valor tendremos en cuenta para realizar un cambio de directorio antes de cada orden de compilación.

2. Por otro lado, `CURDIR` contiene la ruta del directorio raíz actual desde el cual se está construyendo el paquete. La ruta `$(CURDIR)/debian/zsnes` contiene un árbol de directorios que sigue el FHS, y se corresponde con los ficheros contenidos en el paquete `zsnes`.

3. Bajo el objetivo de compilación `build-stamp` añadimos las órdenes requeridas para compilar: invocamos al guión `configure` con las opciones necesarias e iniciamos la compilación. La opción `-C` pasada a `make` hace el cambio de directorio antes de comenzar, justo como `cd` hace para las demás. Hay que hacerlo para cada orden y no al principio debido al hecho de que tras cada orden volvemos al directorio original, al restaurarse el estado anterior del shell.

4. Repetimos el cambio en la invocación a `make` para los otros objetivos `clean` e `install`. Puede verse cómo se pasa la variable de entorno `DESTDIR` con la ruta al área de construcción para la instalación: evidentemente, el *makefile* debe de estar hecho para tener esto en cuenta. Tenemos la suerte para este paquete de que ya sea así: de lo contrario, tendríamos que adaptar dicho fichero, ¡y posiblemente el resto del programa!

5. Por último, en el objetivo `binary-arch` tenemos una serie de llamadas a distintos guiones de *debhelper*. Comentaremos y descomentaremos según nos haga falta: así, por ejemplo, un programa escrito en Perl no necesita `dh_link` ni `dh_strip`, al no generar ejecutables.

   Hemos añadido un argumento a `dh_installman` con la página *man* que queremos que se instale. Al igual que con todo lo demás, si no hubiera una, tendríamos que crearla nosotros. Lo más usual en este caso es escribir un fichero SGML o XML DocBook y transformarlo a *nroff* (el formato de las páginas *man*) mediante *docbook-to-man* o una hoja de estilos XSLT, por ejemplo. Podríamos partir del ejemplo creado antes por `dh_make`>, `zsnes.sgml.ex`.

Un detalle importante: la mayoría de los guiones suponen que los ficheros bajo nuestro directorio `debian` siguen una serie de convenciones. Por ejemplo, `dh_installdocs` supone que existe algún fichero `debian/zsnes.docs` o `debian/docs` que liste la documentación a instalar. En este caso, aprovechamos la documentación que ya trae ZSNES, con lo que tendríamos esto en `debian/docs`:

```
docs/srcinfo.txt
docs/README.SVN
docs/opengl.txt
docs/stdards.txt
docs/authors.txt
docs/todo.txt
docs/install.txt
docs/thanks.txt
docs/support.txt
docs/README.LINUX
docs/readme.txt/about.txt
docs/readme.txt/faq.txt
docs/readme.txt/history.txt
docs/readme.txt/gui.txt
docs/readme.txt/advanced.txt
docs/readme.txt/index.txt
docs/readme.txt/games.txt
docs/readme.txt/netplay.txt
docs/readme.txt/readme.txt
docs/readme.txt/support.txt
docs/readme.htm/styles/release.css
docs/readme.htm/styles/print.css
docs/readme.htm/styles/jipcy.css
docs/readme.htm/styles/radio.css
docs/readme.htm/styles/corner.png
docs/readme.htm/styles/plaintxt.css
docs/readme.htm/styles/shared.css
docs/readme.htm/images/zsneslogo.png
docs/readme.htm/images/netplay.png
docs/readme.htm/images/quick.png
docs/readme.htm/images/saveslot.png
docs/readme.htm/images/cheat.png
docs/readme.htm/images/gui.png
docs/readme.htm/images/config.png
docs/readme.htm/images/game.png
docs/readme.htm/images/f1_menu.png
docs/readme.htm/images/misc.png
docs/readme.htm/netplay.htm
docs/readme.htm/about.htm
docs/readme.htm/games.htm
docs/readme.htm/advanced.htm
docs/readme.htm/gui.htm
docs/readme.htm/support.htm
docs/readme.htm/readme.htm
docs/readme.htm/history.htm
```

```
docs/readme.htm/license.htm
docs/readme.htm/faq.htm
docs/readme.htm/index.htm
docs/readme.1st
```

> NOTA
>
> Hemos usado ya otro fichero más del mismo estilo: `dirs` es realmente el fichero que `dh_installdirs` utiliza, por ejemplo.

Reglas con CDBS Ahora que ya sabemos cuál es la estructura real de un fichero de reglas, lo reescribiremos empleando CDBS:

```
#!/usr/bin/make -f
# -*- makefile -*-

DEB_SRCDIR = src

include /usr/share/cdbs/1/rules/debhelper.mk
include /usr/share/cdbs/1/class/autotools.mk

DEB_CONFIGURE_SCRIPT_ENV   += force_arch=i586
DEB_CONFIGURE_EXTRA_FLAGS   = --disable-cpucheck --with-x \
  --enable-release --with-opengl
DEB_INSTALL_MANPAGES_zsnes += $(DEB_SRCDIR)/linux/zsnes.1
```

Sorprendentemente, esto es todo: ocho líneas. Los dos `include` se ocupan de importar los conjuntos de reglas de apoyo para el uso interno de *debhelper* e implementar el soporte para el perfil de las autotools, respectivamente.

A continuación, pasamos las mismas opciones a `configure` que antes: pedimos que compile para Pentium o superior, que emplee aceleración 3D y un interfaz gráfico, optimice algo más de lo normal (

```
--enable-release
```

) y no intente autodetectar nuestra CPU. Finalmente, indicamos que instale la página man `src/linux/zsnes.1` que incluye ZSNES.

Usaremos esta versión de `debian/rules` para realizar el resto del documento, aprovechando algunas funcionalidades adicionales que aporta. Sin embargo, internamente, es exactamente lo mismo de antes.

3.1.3. Construcción preliminar

Con todo listo, ya podemos construir el paquete. Situándonos en el directorio principal del paquete, `/tmp/packages/zsnes-1.510`, ejecutaremos:

```
dpkg-buildpackage -rfakeroot
```

Tras un cierto tiempo, nos preguntará la contraseña de nuestra clave privada para firmar el paquete de forma automática. Poco después, tendremos en `/tmp/packages` nuestra primera versión del paquete Debian, `zsnes_1.510-0ubuntu1_i386.deb`, junto con un fichero `.dsc` que describe el paquete fuente que también hemos construido, y un `diff.gz` con las diferencias respecto a las fuentes originales.

3.1.4. Inyección en el repositorio

Con nuestra primera versión del paquete lista, sólo nos queda inyectar el paquete en el repositorio Subversion. Nos situaremos en `~{}/packages` y ejecutaremos:

```
svn-inject -c2 -o /tmp/packages/zsnes_1.510-0ubuntu1.dsc \
  file:///home/tunombredeusuario/.svnDebian
```

Tras un cierto tiempo, ya tendremos enviado al repositorio central nuestro paquete, y nuestra copia de trabajo habrá sido creada, con lo que no necesitaremos más `/tmp/packages`.

La opción `-o` evita que se guarde el código fuente en el repositorio, usando únicamente archivos `tar.gz` en el subdirectorio `tarballs` del directorio principal del repositorio, que no se hallará bajo control de versiones.

El repositorio creado tiene la siguiente estructura:

`~{}/packages/tarballs` Contiene los ficheros `tar.gz` con las fuentes originales de nuestros paquetes.

`~{}/packages/zsnes/build-area` Se trata del directorio destino en el que se depositarán todos los paquetes y demás ficheros que vayamos produciendo.

`~{}/packages/zsnes/branches` Almacena las distintas ramas de desarrollo. Normalmente contendría las distintas versiones del código original, pero al usar tarballs, es prácticamente inútil en nuestro caso.

`~{}/packages/zsnes/tags` Permite asociar números de versión con determinadas revisiones del repositorio. Posteriormente veremos cómo se usa.

`~{}/packages/zsnes/trunk` Aquí se almacena la versión actual del paquete. Sólo almacenamos el directorio `debian`, para ahorrar la complejidad y tiempo de descarga necesario de otra forma.

3.2. Preparación de una primera versión

3.2.1. Construcción definitiva

Es el momento de reconstruir el paquete, pero esta vez haciendo uso de la jaula *chroot*, para asegurar que efectivamente hemos listado todas las dependencias correctamente.

Dado que la orden es bastante larga, lo mejor es añadir un alias a dicha orden, o mejor aún, declarar una función de *dash* (la versión reducida del shell *bash* de Ubuntu) que haga dicha tarea, en `~{}/.bashrc`. Añadiremos estas líneas:

```
function svn-b ()
    buildarea="‘pwd‘/../build-area";
    sudo cowbuilder --update
    svn-buildpackage  \
        --svn-builder="pdebuild --auto-debsign --buildresult $buildarea" \
        --svn-postbuild="rm $buildarea/*_source.changes";


function svn-tag ()
    svn-buildpackage --svn-only-tag
```

Nos situaremos en `~{}/packages/zsnes/trunk` y ejecutaremos:

```
svn-b
```

Tras un tiempo prudencial, e introducir nuestras contraseñas de GPG y superusuario, tendremos una primera versión del paquete, comprobada dentro de una jaula *chroot*.

3.2.2. Verificación

Sin embargo, no basta con que se compile e instale correctamente. Además, el paquete debe de cumplir todas las políticas de Debian, como pertenecer a un determinado conjunto de secciones, tener todos sus ficheros de control bien escritos, respetar el estándar FHS que antes mencionamos, etc.

De hecho, si queremos que nos lleguen a aceptar cualquier paquete en un repositorio Debian oficial, como la sección *universe* o *multiverse* de Ubuntu, o la sección *unstable* o *testing* de Debian, lo mejor que podemos hacer para evitar hacer perder tiempo a nuestro supervisor o supervisores (que serán los que efectivamente suban al repositorio el paquete las primeras veces) es pasar antes nuestro paquete por los dos verificadores disponibles: Linda y Lintian, y retirar todos los avisos.

Nota

Según parece, Linda, escrito en Python, es más reciente que Lintian, y también más rápido. Por otro lado, Lintian está escrito en Perl. Aparte de eso, no hay muchas diferencias: simplemente ejecutaremos ambos por seguridad, ya que tampoco supone un esfuerzo adicional considerable.

Lanzaremos los verificadores sobre el `.deb`, que se hallará en `~{}/packages/zsnes/build-area`, pidiendo explicaciones de los avisos y errores con `-i`:

```
linda -i ~/packages/zsnes/build-area/zsnes_1.510-0ubuntu1_i386.deb
lintian -i ~/packages/zsnes/build-area/zsnes_1.510-0ubuntu1_i386.deb
```

Según parece, no todo está bien en nuestro paquete:

```
W: zsnes: extra-license-file usr/share/doc/zsnes/license.htm
N:
N:   All license information should be collected in the debian/copyright
N:   file. This usually makes it unnecessary for the package to install
N:   this information in other places as well.
N:
N:   Refer to Policy Manual, section 12.5 for details.
N:
E: zsnes: FSSTND-dir-in-usr usr/man/
N:
N:   As of policy version 3.0.0.0, Debian no longer follows the FSSTND.
N:
N:   Instead, the Filesystem Hierarchy Standard (FHS), version 2.3, is
N:   used. You can find it in /usr/share/doc/debian-policy/fhs/ .
N:
E: zsnes; Manual page zsnes.1.gz installed into /usr/man.
 The manual page shown is installed into the legacy location of
 /usr/man, where they should be installed into /usr/share/man.
E: zsnes; FSSTND directory /usr/man in /usr found.
 As of policy version 3.0.0.0, Debian no longer follows the FSSTND.
 Instead, the Filesystem Hierarchy Standard (FHS), version 2.1, is
 used. You can find it in /usr/share/doc/debian-policy/fhs/ .
E: zsnes; FSSTND directory /usr/man/man1 in /usr found.
W: zsnes; File /usr/share/doc/zsnes/license.htm is considered to be an
 extra license file.  The file shown above is considered to be another
 license file, where as the license for a package should be contained
 in the copyright file, which should be installed into
 /usr/share/doc/<pkg>.
```

Analizando esta salida, vemos que hay cuatro problemas con nuestro paquete según Linda, y dos según Lintian. El primer problema es fácil de tratar: hemos incluido un fichero de licencia con la documentación (`extra-license-file`), que no debería estar, ya que el archivo `copyright` debería contener toda la información. Basta en este caso con retirar la línea problemática de `debian/docs`.

El segundo problema es más complejo: el directorio `/usr/man` no es parte del estándar actual, FHS, sino de uno más antiguo, el FSSTND (FileSystem Standard). En principio, nuestras reglas están bien escritas. Veamos si efectivamente nuestro paquete pone la página man en su sitio:

```
~/packages/zsnes/trunk$ dpkg -c ../build-area/zsnes_1.510-0ubuntu1_-
i386.deb
```

Sí que lo hace: `/usr/share/man/man1/zsnes.1.gz` aparece como debería. Sin embargo, también aparece `/usr/man/man1/zsnes.1.gz`. Así, hay algo fuera de nuestras reglas que está instalando en dicho sitio la página *man*.

Vamos a descomprimir en un directorio temporal el código fuente original, y echar un vistazo con la inestimable ayuda de *grep*, empleando la opción `-R` para hacer una búsqueda recursiva por el árbol de directorios y `-l` para solamente listar los ficheros con coincidencia y no el contenido que sigue el patrón:

```
cd /tmp
tar -xzf ~/packages/tarballs/zsnes_1.510.orig.tar.gz
cd zsnes-1.510.orig
grep -Rl zsnes.1 *
docs/srcinfo.txt
docs/readme.txt/readme.txt
docs/readme.htm/readme.htm
src/Makefile.in
```

Ya lo hemos encontrado: descartando los tres ficheros que forman parte de la documentación, sólo queda `src/Makefile.in`. Tiene que ser el culpable, ya que este fichero es usado por el guión `configure` para crear el *makefile* con el que hacer la compilación e instalar el programa.

Veremos exactamente dónde está el problema, empleando la opción `-C` para ver un contexto de un determinado número de líneas alrededor de las apariciones:

```
grep -C4 zsnes.1 src/Makefile.in
install:
        @INSTALL@ -d -m 0755 $(DESTDIR)/@prefix@/bin
        @INSTALL@ -m 0755 @ZSNESEXE@ $(DESTDIR)/@prefix@/bin
        @INSTALL@ -d -m 0755 $(DESTDIR)/@prefix@/man/man1
        @INSTALL@ -m 0644 linux/zsnes.1 $(DESTDIR)/@prefix@/man/man1
```

```
uninstall:
	rm -f @prefix@/bin/$(notdir @ZSNESEXE@) @prefix@/man/man1/zsnes.1

clean:
	rm -f $(Z_OBJS) $(PSR) $(PSR_H) @ZSNESEXE@
tclean:
```

Ya hemos encontrado las dos líneas problemáticas, al final del objetivo `install`: crean el directorio e instalan la página *man*. Sólo hay que quitarlas. Sin embargo, hay un problema: no podemos tocar el código directamente, ya que no lo tenemos bajo control de versiones. Es más: no se considera buena práctica, ya que en futuras versiones los desarrolladores originales podrían arreglar ese problema, y se haría difícil ver quién ha hecho qué tras hacer unos cuantos cambios más.

Por eso, se considera buena práctica emplear un sistema de parches para hacer este tipo de modificaciones. Algunas alternativas [Tea07b] incluyen el más antiguo, *dpatch*, el más reciente (y aparentemente superior, según los desarrolladores de SUSE) *quilt* y el que usaremos aquí, *cdbs-simple-system*, que como su nombre indica, está más inclinado hacia ser sencillo que ser potente. Sin embargo, nos basta para casos sencillos como éste.

Para usarlo, añadimos la línea que instala soporte para él en `debian/rules`, justo después del primer `include`:

```
include /usr/share/cdbs/1/rules/simple-patchsys.mk
```

Pasaremos a crear el parche en sí. Para ello, tenemos que obtener el árbol de fuentes tal y como lo vería `debuild` antes de compilar, y no como lo tenemos ahora. Para ello, usaremos `svn-buildpackage`, no sin antes enviar nuestros otros cambios al repositorio (de otra forma, no nos dejará seguir):

```
~/packages/zsnes/trunk$ svn commit -m \
  "Integrado cbds-simple-patch y corregido documentación"
Enviando       trunk/debian/docs
Enviando       trunk/debian/rules
Transmitiendo contenido de archivos ..
Commit de la revisión 6.
~/packages/zsnes/trunk$ svn-buildpackage --svn-export
buildArea: /home/antonio/packages/zsnes/build-area
[...]
I: mergeWithUpstream property set, looking for upstream source tarball...
[...]
Exportación completa.
rm -rf /home/antonio/packages/zsnes/build-area/tmp-0.00194501814349124
Build directory exported to /home/antonio/packages/zsnes/build-area/zsnes-1.510
```

Iremos al directorio exportado de construcción, que contendrá todos los ficheros necesarios, y generaremos el parche usando `cdbs-edit-patch`:

```
~/packages/zsnes/trunk$ cd ../build-area/zsnes-1.510/
~/packages/zsnes/build-area/zsnes-1.510$ cdbs-edit-patch 01-man-fhs.patch
```

El programa nos indicará que nos hallamos ahora en un subshell, y que hagamos las modificaciones necesarias. Usando un editor cualquiera, retiraremos dichas líneas de `src/Makefile.in`. Hecho esto, pulsaremos `CTRL + D` o introduciremos la orden `exit` para salir del subshell y dejar que cree el parche `debian/patches/01-man-fhs.patch` con los cambios que hemos hecho.

Sólo hemos de colocar ahora el parche en el sitio correcto, y guardar los cambios en el repositorio:

```
cd ~/packages/zsnes/trunk
cp -r ../build-area/zsnes-1.510/debian/patches debian
svn add debian/patches
svn commit -m "Arreglado problema con la página man en /usr/man"
```

Ahora reconstruiremos el paquete:

```
svn-b
```

Ya, por fin, Lintian y Linda no dan ningún aviso. Probaremos a instalar el paquete, y ver qué tal funciona:

```
sudo dpkg -i ../build-area/zsnes_1.510-0ubuntu1_i386.deb
```

Tras probar un poco el emulador con alguna ROM libremente disponible, como las de PDRoms (`http://www.pdroms.com`), decidimos que el paquete está en condiciones de ser usado. Usaremos la otra función que definimos antes para marcar esta versión del paquete, copiando los contenidos de `trunk` en un nuevo subdirectorio de `tags` cuyo nombre será la versión actual del paquete, para después confirmar los cambios que se habrán realizado automáticamente en `debian/changelog`, en preparación para la siguiente versión de nuestro paquete:

```
svn-tag
svn commit -m "Copiado a tags versión 1.510-0ubuntu1 del paquete"
```

3.2.3. Envío al repositorio Debian

Después de haber verificado nuestro paquete y haber marcado la versión en el repositorio Subversion, es momento de enviarlo al repositorio Debian, para que nuestros usuarios puedan acceder fácilmente a él. Añadiremos tanto el paquete fuente como el paquete compilado:

```
sudo reprepro includedeb gutsy \
 ~/packages/zsnes/build-area/zsnes_1.510-0ubuntu1_i386.deb
sudo reprepro -S main -P low includedsc gutsy \
 ~/packages/zsnes/build-area/zsnes_1.510-0ubuntu1.dsc
```

Deberíamos probar el paquete justo como un usuario normal lo haría. Pero, primero, tenemos que desinstalar la versión que instalamos antes a partir de un fichero:

```
sudo aptitude remove zsnes
sudo aptitude update
sudo aptitude install zsnes
```

3.3. Actualización del paquete a una nueva versión del programa

Cada cierto tiempo, deberemos de actualizar nuestro repositorio con los cambios realizados en el código. Para ello, disponemos de la orden `svn-upgrade`, a la que le pasaremos un fichero `tar.gz` con el nuevo código fuente del paquete.

Ahora probaremos a actualizar nuestro paquete con el código correspondiente a la última versión en el repositorio Subversion del equipo de ZSNES. Primero, prepararemos el tarball con el código original. Descargamos el código fuente, y lo exportamos a otro directorio, para retirar los ficheros usados por *Subversion*:

```
svn checkout https://svn.bountysource.com/zsnes/trunk zsnes-svn
svn export zsnes-svn zsnes-1.510.SVN5215
```

He decidido usar el esquema `1.510.SVN5215` para indicar que se trata de la revisión 5215 del repositorio. De esta forma, si sale una nueva versión oficial, o si empleo una revisión más reciente del repositorio Subversion, como `1.520` o `1.510.SVN5216`, éstas serán considerada más recientes, y la actualización se podrá hacer de forma correcta. Podemos comprobar que es efectivamente así mediante la siguiente orden:

```
dpkg --compare-versions "1.510-0ubuntu1" gt "1.510.SVN5215-0ubuntu1" && \
  echo "1.510-0ubuntu1 mayor que 1.510.SVN5215-0ubuntu1" || \
  echo "1.510-0ubuntu1 menor que 1.510.SVN5215-0ubuntu1"
```

Ahora que ya tenemos el código, inspeccionaremos un momento para ver si todos los ficheros importantes se hallan en su sitio. Normalmente, no se suelen almacenar ficheros generados automáticamente en el repositorio, así que pueden que falten algunas cosas importantes. Mirando en `src`, vemos que falta el guión `configure` que necesitamos para compilar. Lo generaremos ejecutando las siguientes órdenes bajo `src`, el mismo directorio donde se halla su fichero fuente, `configure.in`:

```
aclocal
autoconf
```

También faltan los directorios `docs/readme.txt` y `docs/readme.htm`, que añadiremos en su sitio. Con esto tenemos los ficheros fuente listos. Vamos a crear el tarball que necesitamos:

```
/tmp$ tar -czf zsnes-1.510.SVN5215.tar.gz zsnes-1.510.SVN5215
```

El siguiente paso es añadir dicho código a nuestra área de trabajo, e indicar que vamos comenzar a empaquetar una nueva versión del programa. Volvemos al directorio con la versión actual de nuestro paquete y ejecutamos:

```
~/packages/zsnes/trunk$ svn-upgrade /tmp/zsnes-1.510.SVN5215.tar.gz
```

Se harán los cambios necesarios en `debian/changelog` y el resto del repositorio. Cambiaremos el número de versión del paquete a `0ubuntu1` y confirmamos dichos cambios antes de volver a reconstruir el paquete:

```
~/packages/zsnes/trunk$ svn commit -m \
  "Actualizado con rev 5215 upstream"
~/packages/zsnes/trunk$ svn-b
```

Sin embargo, la reconstrucción falla. Buscando entre los distintos mensajes, podemos ver una línea del estilo:

```
Trying patch 01-man-fhs.patch at level 1 ... 0 ... 2 ... failed.
```

En resumen: el parche que antes hicimos para corregir el problema de `Makefile.in` no se ha podido aplicar. Tras inspeccionar sus contenidos, vemos que efectivamente `Makefile.in` ha cambiado demasiado:

```
install:
@INSTALL@ -d -m 0755 $(DESTDIR)/@bindir@
@INSTALL@ -m 0755 @ZSNESEXE@ $(DESTDIR)/@bindir@
@INSTALL@ -d -m 0755 $(DESTDIR)/@mandir@/man1
@INSTALL@ -m 0644 linux/zsnes.1 $(DESTDIR)/@mandir@/man1
```

Vaya, parece que ellos mismos han corregido ya el problema que teníamos antes. Ésta es precisamente la razón por la que usamos un parche, en vez de cambiar directamente el código. Sólo tenemos que eliminar el parche, confirmar los cambios y reconstruir:

```
~/packages/zsnes/trunk$ svn rm debian/patches
D  debian/patches/01-man-fhs.patch
~/packages/zsnes/trunk$ svn commit -m \
  "01-man-fhs.patch: Retirado (corregido por upstream)"
Deleting       zsnes/trunk/debian/patches/01-man-fhs.patch
Transmitting file data .
Committed revision 11.
~/packages/zsnes/trunk$ svn-b
```

Esta vez ya no da el fallo del parche, pero indica que faltan las bibliotecas de desarrollo de Qt para la nueva interfaz. Añadiremos la dependencia en `libqt4-dev` al fichero `control` y volveremos a intentarlo tras confirmar otra vez nuestros cambios.

Esta vez falla diciendo que no puede construir el fichero `ui_zsnes.h`, que hace falta para compilar. Probaremos a compilar sobre el directorio `/tmp/zsnes-1.510.SVN5215/src` tras instalar las dependencias en nuestro propio sistema:

```
./configure
make
```

Da el mismo problema, así que no es culpa de nuestro paquete. Vamos a mirar con *grep*, a ver qué puede ser:

```
grep -R ui_zsnes.h *
makefile.ms:$GUI_D/gui.cpp: $GUI_D/ui_zsnes.h
src/gui/gui.h:#include "ui_zsnes.h"
Makefile.in:GUI_QO=$(GUI_D)/moc_gui.cpp $(GUI_D)/ui_zsnes.h
Makefile:GUI_QO=$(GUI_D)/moc_gui.cpp $(GUI_D)/ui_zsnes.h
makefile.dep:gui/gui.o: gui/gui.cpp gui/gui.h ui_zsnes.h
```

Si nos fijamos, se puede ver que en `makefile.dep` la ruta no se corresponde con la de las anteriores entradas, por lo que seguramente ahí estará el fallo. Esta vez buscamos por este fichero:

```
grep -R makefile.dep *
src/configure:touch -t 198001010000 makefile.dep
src/Makefile.in:main: makefile.dep $(Z_QOBJS) $(Z_OBJS)
src/Makefile.in:include makefile.dep
src/Makefile.in:makefile.dep: $(TOOL_D)/depbuild Makefile
src/Makefile.in: $(TOOL_D)/depbuild @CC@ "@CFLAGS@" @NASMPATH@
"@NFLAGS@" $(Z_OBJS) > makefile.dep
src/Makefile.in: rm -f makefile.dep $(Z_OBJS) $(Z_QOBJS) $(PSR)
$(PSR_H) @ZSNESEXE@
src/Makefile:main: makefile.dep $(Z_QOBJS) $(Z_OBJS)
```

```
src/Makefile:include makefile.dep
src/Makefile:makefile.dep: $(TOOL_D)/depbuild Makefile
src/Makefile: $(TOOL_D)/depbuild gcc " -pipe -I. -I/usr/local/include
-I/usr/include -D__UNIXSDL__ -I/usr/include/SDL -D_GNU_SOURCE=1
-D_REENTRANT -D__OPENGL__ -DNO_DEBUGGER -DNDEBUG -march=athlon-xp -O2
-fomit-frame-pointer -s -DQT_SHARED -I/usr/include/qt4
-I/usr/include/qt4/QtCore -I/usr/include/qt4/QtGui " nasm "
-w-orphan-labels -D__UNIXSDL__ -f elf -DELF -D__OPENGL__ -DNO_DEBUGGER
-O1" $(Z_OBJS) > makefile.dep
src/Makefile: rm -f makefile.dep $(Z_OBJS) $(Z_QOBJS) $(PSR) $(PSR_H)
zsnes
src/autom4te.cache/output.0:touch -t 198001010000 makefile.dep
src/autom4te.cache/output.1:touch -t 198001010000 makefile.dep
src/configure.in:touch -t 198001010000 makefile.dep
```

Puede verse que en `src/Makefile` es donde se crea a través de una herramienta propia de los desarrolladores de ZSNES, que obtiene automáticamente las dependencias. Recordando la lista anterior de ficheros que mencionaban a `ui_zsnes.h`, se nos viene a la cabeza `gui/gui.h`. Vamos a probar a cambiar la ruta de inclusión a `gui/ui_zsnes.h`:

```
cd ~/packages/zsnes/trunk
svn-buildpackage --svn-export
cd ../build-area/zsnes-1.510.SVN5215
cdbs-edit-patch 02-fix-depbuild.patch
```

Hacemos el cambio antes mencionado con el editor *vim* y salimos del shell, tras lo cual reintentaremos la construcción del paquete:

```
vim src/gui/gui.h
exit
```

Ahora la reconstrucción ha tenido éxito. Ya sólo tendríamos que seguir los mismos pasos anteriores de verificación, validación manual, marcado en el repositorio (tras retirar el aviso «NOT RELEASED YET» añadido automáticamente a `debian/changelog`, claro), y envío. Cualquiera de nuestros usuarios será notificado eventualmente de la actualización y podrá instalar la versión más reciente del paquete.

Una última nota: si se quiere, se puede integrar la verificación y publicación en la función `svn-b` antes definida, cambiando las líneas correspondientes en `~{}/.bashrc` por:

```
function svn-b ()
    buildarea="`pwd`/../build-area";
```

```
    rutapaquete="$buildarea/\$package_\$debian_version*.deb";
    rutadsc="$buildarea/\$package_\$debian_version*.dsc";
    rutarepo="/var/packages/ubuntu";
    sudo cowbuilder --update
    svn-buildpackage  \
        --svn-builder="pdebuild --auto-debsign --buildresult $buildarea" \
--svn-postbuild="rm $buildarea/*_source.changes; \
                         lintian -i $rutapaquete; linda -i $rutapaquete; \
                         cd $rutarepo && \
                         sudo reprepro remove gutsy \$package && \
                         sudo reprepro includedeb gutsy $rutapaquete && \
                         sudo reprepro -S main -P low includedsc gutsy $rutadsc";


function svn-tag ()
    svn-buildpackage --svn-only-tag
```

4 Otros aspectos de interés

4.1. Integración de repositorio propio con la jaula *chroot*

En este tutorial no hará falta, pero sí es posible que nos haga falta para casos más complejos. Cuando haya interdependencias entre varios paquetes que deseemos crear, tendremos que añadir nuestro repositorio (que estará alojado en su propio servidor web) a la lista de los repositorios de la jaula *chroot*.

Para ello, nos introduciremos en la jaula e importaremos la firma digital de nuestros paquetes:

```
sudo cowbuilder --login --save-after-login
wget http://direccion.delservidor.org/claveDebian.asc
apt-key add claveDebian.asc
```

Con esto, la jaula ya confiará en nuestros paquetes. Tenemos que añadir entonces la siguiente línea a `/etc/apt/sources.list`, usando el editor *vim*:

```
deb http://direccion.delservidor.org/ubuntu gutsy main
```

Actualizamos la lista de paquetes:

```
aptitude update
```

Hemos terminado, así que sólo queda salir del shell de la jaula:

```
exit
```

4.2. Sincronización del repositorio local con un repositorio en Internet

En muchos casos no dispondremos de los medios necesarios como para dejar nuestro ordenador como servidor web al exterior para que sirva nuestros paquetes. En estos casos, podemos subir nuestros ficheros `.deb` a una forja o algo del estilo y dejar que los usuarios se los bajen.

Pero, ¿y si son muchos paquetes, o si queremos mantener actualizados a los usuarios? Lo mejor sería replicar completo el repositorio en un servidor conectado permantentmente a Internet. La mejor opción es, por tanto, obtener espacio de alojamiento con acceso mediante SFTP o FTP.

Una vez lo hayamos conseguido, podríamos subir el árbol completo de directorios de nuestra máquina al servidor remoto, pero hacer esto una y otra vez tardaría demasiado. La mejor opción es hacer un *mirror inverso* a través de la herramienta *lftp*, que nos pedirá la contraseña antes de conectarse:

```
lftp -d -e "mirror -venR ruta local ruta remota dirección del servidor
```

Si queremos que no nos pregunte una y otra vez la contraseña, siempre podemos añadir líneas como éstas a ~{}/.netrc, cuidando de que sólo sea legible por nosotros:

```
machine mi.servidor.com
login minombredeusuario
password micontraseña
```

4.3. Adaptación de aplicaciones Java

La principal diferencia al crear un paquete Java es, sin duda, el hecho de que, a pesar de tener que compilar, el paquete en sí no tiene ninguna arquitectura definida. Ello debería verse reflejado en `debian/control`.

Además, debería tener en `Build-Depends` algún compilador de Java, y en `Depends` el paquete virtual `java2-runtime`, además de una alternativa concreta, como la de Sun, `sun-java6-jre`, o preferiblemente la basada en OpenJDK, `icedtea-java7-jre`, usando el operador lógico OR (`|`). Los paquetes virtuales no añaden ninguna funcionalidad: se limitan a hacer cosas como ayudarnos a instalar varios paquetes de una sola vez (usándolos en su campo `Depends`), o permitiéndonos usar un paquete cualquiera que nos provea de una cierta funcionalidad.

Así, si curioseamos en el paquete `sun-java6-jre`, veremos que tiene a dicho paquete virtual en el campo `Provides` («Proporciona»):

```
$ aptitude show sun-java6-jre
Paquete: sun-java6-jre
[...]
Proporciona: java-virtual-machine, java1-runtime, java2-runtime
[...]
```

Si usamos el compilador de Sun en Build-Depends (paquete `sun-java6-jdk`), habremos de cambiar nuestro `.pbuilderrc` para que use un interfaz distinto de entrada, que nos deje aceptar los términos de la DLJ de Sun. Añadimos estas líneas:

```
# Para poder aceptar la licencia DLJ
export DEBIAN_FRONTEND="readline"
```

Además de eso, tendremos como de costumbre que adaptar nuestro programa Java para que sea fácilmente compilable de forma automática, y para que se integre bien dentro del sistema. Un problema es que no deberíamos escribir rutas absolutas en el código Java, o si no perderemos toda la transportabilidad que deseábamos en un primer momento. La forma más fácil de implementar lo que deseamos es simplemente utilizar variables de entorno o propiedades del sistema.

A veces no nos servirá cualquier JRE de los disponibles para ejecutar nuestro programa, y tendremos que prescindir de usar `java2-runtime`. Es el caso usual de las aplicaciones basadas en Swing: el JRE por defecto en Gutsy, GCJ, sólo implementa completamente SWT. Tendremos que hacer que el sistema instale y active otro JRE.

Para poder establecerlas de forma separada del programa Java antes de lanzarlo, lo que necesitamos es un guión de shell, que será el que instalaremos en `/usr/bin`. El fichero `.jar`, siguiendo la política de Debian, debe ir en `/usr/share/java`.

El que sea fácilmente compilable de forma automática depende de qué sistema hayamos estado usando. Si nos hemos basado en un IDE, seguramente aquí tengamos problemas. Lo que deberíamos hacer es reescribir la fase de compilación usando un guión para *Apache Ant*. Esto, de paso, nos permitirá aprovechar el perfil de CDBS, simplificando bastante nuestra tarea. *Ant* es efectivamente un sustituto del *GNU Make* escrito en Java, para aplicaciones Java. Utiliza ficheros muy del estilo de los *makefile*, sólo que escritos en XML.

Con todo esto, aún hay un par de complicaciones adicionales. Normalmente, los ficheros fuente Java se hallan escritos en UTF-8. Sin embargo, la jaula *chroot* que antes hicimos en 2.2 en la página 12 no tiene en principio una localización compatible instalada. Vamos a tener que entrar en la jaula e instalar los paquetes necesarios, comprobando la situación que aquí describimos con `locale` (nos mostrará que usamos la localización estándar «C»):

```
sudo cowbuilder --login --save-after-login
aptitude install language-pack-es
exit
```

Ahora, lo que hemos de hacer es asegurarnos que cuando llamemos a *Ant*, lo hagamos estableciendo una localización con UTF-8:

```
LANG=es_ES.UTF-8 ant compile
```

Deberíamos además invocar las pruebas de unidad sobre el código, para asegurarnos de que el código usado en el paquete no presente regresiones. Además de añadir el paquete `junit` a `Build-Depends`, tenemos que tener en cuenta que si en alguna prueba de unidad usamos un componente Swing, aunque no se muestre nada en pantalla, requeriremos un servidor X, o de lo contrario la prueba fallará. Dado que ejecutar un servidor X completo en una jaula *chroot* es un gasto inútil de recursos, vamos a usar un falso servidor X, que añadiremos también a `Build-Depends`: `xvfb`. Para tener disponible el servidor X durante una orden, únicamente ponemos `xvfb-run` antes de ella. Combinándolo con lo de las localizaciones, tendremos que ejecutar una orden como ésta:

```
LANG=es_ES.UTF-8 xvfb-run ant (objetivo-compilación)
```

4.4. Actualización del escritorio

Antes conseguimos crear un paquete con un binario, junto con su página *man*. Sin embargo, para que realmente consideremos al paquete completo, deberíamos integrarlo con el gestor de escritorio del usuario. Veremos cómo añadir el acceso directo al menú, y cómo asociar los ficheros ROM de Super Nintendo al emulador ZSNES.

Esta tarea se ha hecho en los últimos años mucho más sencilla gracias a las especificaciones ofrecidas por el grupo FreeDesktop.org [Fre07], que hace de centro de reunión para diversos proyectos relacionados con entornos gráficos. Así, KDE y GNOME emplean el mismo formato para describir las entradas de su menú, por ejemplo.

4.4.1. Accesos directos e iconos

Normalmente, existen dos sitios del menú de aplicaciones donde podemos instalar un acceso directo a nuestra aplicación: el menú Debian, y el propio menú normal de aplicaciones. Por completitud, trataremos ambos, aunque hoy en día el menú Debian no se usa demasiado. Además, no aparece por omisión en un sistema Ubuntu: hay que instalar el paquete `menu` para poder usarlo.

Para que se añada una entrada al menú Debian, hemos de asegurarnos de que la línea con `dh_installmenu` se halle descomentada en `debian/rules`. Dado que nuestro paquete se llama `zsnes`, crearemos un fichero `debian/zsnes.menu` con este contenido:

```
?package(zsnes):needs="X11" section="Games/Arcade" \
  title="ZSNES" command="/usr/bin/zsnes" \
  icon="/usr/share/pixmaps/zsnes.xpm"
```

El significado de los campos es el siguiente:

`needs="X11"` Indica que el programa tiene una interfaz gráfica, requiriendo un servidor X.

`section="Games/Arcade"` Especifica dónde debería situarse el acceso directo. Las secciones se hallan prefijadas por la política de empaquetado de Debian.

`title="ZSNES"` Éste es el texto que se mostrará en la entrada del menú.

`command=/usr/bin/zsnes"` Ésta es la orden exacta que será ejecutada.

`icon=/usr/share/pixmaps/zsnes.xpm"` Tal y como indica el grupo FreeDesktop.org, hemos instalado el icono en formato XPM y en la ruta `/usr/share/pixmaps`. Para convertir dicha imagen, podemos usar simplemente *Gimp*, o el programa `convert` del paquete `imagemagick`, de esta forma:

```
convert zsnes.png zsnes.xpm
```

Ahora hemos de centrarnos en que aparezca en los menús de KDE y GNOME. Para que sea así, hemos de crear otro fichero más, `debian/zsnes.desktop`, con el siguiente contenido:

```
[Desktop Entry]
Name=ZSNES
Type=Application
Comment=SNES (Super Nintendo) emulator that uses x86 assembly
Comment[es]=Emulador de SNES que emplea código ensamblador de x86
Exec=/usr/bin/zsnes %f
Icon=zsnes.png
MimeType=application/x-snes-rom;
Categories=Game;Emulator;
```

Dicho fichero ha de hallarse codificado tal y como indica el campo `Encoding`: en UTF-8. Además de indicar el nombre y el tipo de la aplicación, podemos especificar una descripción más larga. Adelanto un detalle para realizar la asociación de ficheros: el campo `Exec`, que contiene la orden a ejecutar, permite especificar sustituciones como `%f`, que será sustituido por el primer fichero con el que se active el acceso directo. Si hay varios, se ejecutará el emulador tantas veces como ficheros haya, con un fichero distinto cada vez.

En `Categories` incluimos las categorías a las que pertenece esta aplicación: son más bien anotaciones semánticas que una descripción jerárquica de dónde debería colocarse exactamente. Debe de haber al menos una de las categorías principales según [Fre08] (o si no no aparecerá en el menú), y pueden haber categorías adicionales para dar más información.

Los campos con texto descriptivo como `Name` y `Comment` se pueden personalizar según el idioma añadiendo al nombre de campo, entre corchetes, un identificador de idioma de ISO 639. He incluido un ejemplo para `Comment`. El campo `Icon` no requiere la ruta, ya que el directorio `/usr/share/pixmaps` es uno de los consultados de forma automática.

Otro detalle también importante para crear la asociación de fichero es especificar el tipo MIME de los ficheros que esta aplicación abre. Dado que no existe un tipo estándar MIME para ROM de Super Nintendo, definimos uno nosotros, cuidando de poner el prefijo `x-` en la segunda mitad, indicando que no es estándar.

Validaremos el fichero mediante `desktop-file-validate`, sin más problemas. En caso contrario habría que hacer las correcciones oportunas.

Ahora que tenemos los dos ficheros con la información necesaria, vamos a definir las acciones requeridas para instalarlos debidamente. En primer lugar, la llamada que necesitamos hacer a `dh_installmenu` ya la hace CDBS por nosotros, con lo que ya tendríamos la parte de instalar la entrada en el menú Debian. Aún tenemos que instalar el fichero `.desktop`, el icono, y añadir el icono al caché de GNOME, para mejorar la eficiencia. Añadimos estas líneas a `debian/rules`:

```
install/zsnes::
```

```
dh_install -m 0644 $(CURDIR)/debian/zsnes.desktop  \
                   /usr/share/applications
dh_install -m 0644 $(CURDIR)/debian/zsnes.xpm      \
                   /usr/share/pixmaps
dh_desktop
dh_iconcache
```

Cuando reconstruyamos el paquete, veremos que nos aparecerán dos ficheros nuevos en el directorio `debian`: `zsnes.postrm.debhelper` y `zsnes.postinst.debhelper`. Estos ficheros son guiones Bash generados automáticamente que se ejecutan después de la desinstalación e instalación. También hay guiones equivalentes para antes de la instalación y la desinstalación, llamados `preinst` y `prerm`, respectivamente. El sufijo `.debhelper` indica al mecanismo de construcción de paquetes que debe concatenar su contenido con el que pudieran tener los ficheros sin tal sufijo, como `zsnes.postrm`, que nosotros mismos habríamos escrito manualmente.

En particular, `dh_desktop` ha añadido una llamada a `update-desktop-database`, y `dh_installmenu` otra llamada a `update-menu`. Con esto, nos aseguramos de que el menú sea actualizado debidamente y que el entorno de escritorio sepa que nuestro programa está disponible para abrir cualquier ROM de Super Nintendo.

Posteriormente, tras instalar el paquete, dichos guiones son guardados en `/var/lib/dpkg/info`. Si alguna vez nos equivocamos al escribir alguno de ellos, es posible que no podamos ni terminar de desinstalar ni de instalar el paquete. Podemos forzar a que su ejecución termine con éxito añadiendo al inicio del guión en dicho directorio la línea:

```
exit 0
```

Así ya podremos retirar el paquete e instalar una versión con dicho fallo corregido.

4.4.2. Actualización de los tipos MIME

Ya nuestro sistema sabe que puede abrir ROM de Super Nintendo con ZSNES. Tenemos el acceso directo, y el icono. Sólo falta decirle al sistema cómo identificar una ROM de Super Nintendo.

En general, hay dos formas de identificar el tipo de un fichero: a través de una secuencia binaria específica en la cabecera (como en el caso de las imágenes en formato BMP, que incluyen siempre los caracteres «BM» al inicio del fichero), conocida como *magic cookie*, o a través de la extensión.

Aunque usar el contenido del fichero es mucho más robusto, no conozco realmente si siguen algún formato específico (probablemente no), así que nos limitaremos a la extensión, algo mucho más sencillo. En particular, nos centraremos en las dos más populares: `.sfc` y `.smc`.

Añadimos a nuestro paquete el fichero XML `debian/zsnes.sharedmimeinfo` siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<mime-info xmlns="http://www.freedesktop.org/standards/shared-mime-info">
   <mime-type type="application/x-snes-rom">
     <comment>SNES ROM</comment>
     <comment xml:lang="es">ROM de SNES</comment>
     <glob pattern="*.sfc"/>
     <glob pattern="*.smc"/>
   </mime-type>
</mime-info>
```

Tras la declaración XML, incluimos el elemento raíz `mime-info`, con la declaración del espacio de nombres para documentos de tipos MIME de FreeDesktop.org. Ahora tendríamos una serie de tipos MIME con `mime-type`, dando una línea descriptiva en posiblemente varios idiomas (de nuevo usando códigos de ISO 639), y posteriomente especificando patrones con los que identificar los ficheros que pertenecen a dicho tipo: `glob` usa un simple patrón de ficheros del shell (no se trata de una expresión regular, como podemos ver), y `magic` nos permitiría identificar por contenido.

Ahora deberíamos asegurarnos de que la línea que llama a `dh_installmime` se halle descomentada, si usáramos sólo *debhelper*, pero con CDBS no hay que hacer nada más. La llamada a `update-mime-database` será añadida automáticamente a los guiones de postinstalación y postdesinstalación por el guión anterior, y el fichero será instalado en su lugar correcto, `/usr/share/mime/packages`. Podemos comprobar, tras instalar el paquete, que la asociación se ha realizado bien inspeccionando `/usr/share/mime/globs`, que debería contener las líneas:

```
application/x-snes-rom:*.sfc
application/x-snes-rom:*.smc
```

4.5. Generación automática de paquetes

4.5.1. Módulos Perl

El guión `dh-make-perl` nos permite crear rápidamente versiones preliminares de paquetes Debian para módulos Perl del CPAN (Comprehensive Perl Archive Network) [Hie07]. De esta forma, podemos integrar todos los módulos necesarios para nuestra aplicación Perl en paquetes, y que así el usuario pueda instalarla justo como cualquier otra aplicación.

Así, si queremos empaquetar el módulo `XML::Writer` del CPAN, escribiremos:

```
dh-make-perl --cpan XML::Writer
```

Es posible que tengamos que hacer algunos retoques a `debian/control` o a `debian/rules` si el sistema de compilación del paquete no es compatible con el habitual (MakeMaker), o si tiene dependencias que no puedan detectarse automáticamente. Por lo general, no

habrá mucho que hacer: el CPAN impone ciertas cosas que hacen que la tarea sea bastante más sencilla que con otros programas.

Una nota: el servicio CPAN tiene enlaces a otro servicio muy útil que indica exactamente las dependencias de un paquete Perl cualquiera de forma recursiva, y nos indica cuáles módulos se hallan preconfigurados (y no requieren de un paquete por lo tanto) y cuáles no.

También es útil instalar el paquete *apt-file*, que es capaz de encontrar el paquete que contiene un determinado fichero, de tal forma que pueda ser añadido automáticamente a las dependencias del paquete generado. Es importante actualizar el caché cada cierto tiempo de esta forma:

```
sudo apt-file update
```

4.5.2. Adaptación de guiones de instalación

Un caso muy común es cuando nosotros mismos tenemos que compilar alguna aplicación, porque no dispongamos de un paquete apropiado. ¿Deberíamos directamente instalarla, y perder así las ventajas de una desinstalación limpia y segura, y la posibilidad de posteriormente actualizar a una versión posterior si finalmente aparece un paquete?

No hace realmente falta: *checkinstall* [Dur06], cuyo paquete del mismo nombre tendremos que instalar, se ocupa de invocar `make install`, seguir todo el rastro de la instalación, y crear un rudimentario paquete Debian (también hay soporte para paquetes Slackware y RPM) a partir de él. Realiza la instalación automáticamente, y nos deja un paquete en el mismo directorio, que podemos pasar a cualquiera para que también lo instale. Por supuesto, el paquete no podremos subirlo a ningún repositorio serio: no tiene ninguna información de dependencias, por ejemplo, ni se halla firmado.

Si por ejemplo estamos compilando un programa basado en las *autotools* (unión de *autoconf*, *automake* y a veces *libtool*), sólo habría que cambiar el último paso:

```
./configure --prefix=/usr
make
sudo checkinstall
```

4.6. Otros formatos de paquete

4.6.1. Comparativa con otros formatos

Los paquetes Debian son sólo un tipo más de paquete que podemos encontrar. Otras distribuciones han desarrollado sus propios sistemas de empaquetado, con mayor o menor funcionalidad, y con mayor o menor éxito. En esta guía, mencionaremos los otros tres formatos más populares: RPM, Portage y Slackware.

RPM

El formato RPM (RedHat Package Manager) es el empleado actualmente en las distribuciones SUSE y Fedora, entre otras, y es originario de la ahora desaparecida RedHat Linux. Tiene funcionalidad en el mismo nivel de los paquetes Debian, con guiones de pre/postinstalación y desinstalación, y metadatos, como por ejemplo las dependencias. Sin embargo, esta información de dependencias es menos rica que la de los paquetes Debian: sólo existe un tipo de dependencia, mientras que en los paquetes Debian tenemos recomendaciones (paquetes que casi siempre necesitaremos junto con el actual) y sugerencias (paquetes que mejorarían la funcionalidad del actual). Además, en los paquetes Debian podemos especificar alternativas en las dependencias, y usar paquetes virtuales, que indiquen la disponibilidad de una cierta funcionalidad, como «navegador web» o «entorno de ejecución Java», más que un software determinado.

Por otro lado, a diferencia de los paquetes Debian, incorpora guiones de verificación de la correcta instalación de un paquete, y disparadores al cambiar el estado de algún otro paquete. También incluye la capacidad de definir dependencias sobre ficheros, aunque a muchos no les parece realmente útil.

El principal problema de este formato era la falta de resolución automática de dependencias en la herramienta de gestión de paquetes `rpm` estándar: ésta simplemente informaba de las dependencias directas que no habían sido cumplidas, obligándonos a reintentar el proceso bajando un paquete tras otro hasta finalmente conseguir instalar el paquete deseado. Sin embargo, ya existen herramientas que añaden esta resolución de forma transparente, como `yum` de Fedora, *YaST* de SUSE, o `urpmi` de Mandriva.

Gracias a esas herramientas, hoy en día el problema es distinto, y se basa más en la forma y contenido de los paquetes en sí: no existe una política estricta y estándar de empaquetado como la que tienen los paquetes Debian [JS98], ni herramientas de validación. Por ello, un RPM no suele funcionar bien fuera de la versión específica de la distribución en que se desarrolló. Además, no hay tanto software empaquetado en formato RPM como lo hay en formato Debian, ni de forma tan accesible.

Portage

El sistema Portage de Gentoo es una versión para Linux del sistema *ports* de FreeBSD: realmente, un paquete de Gentoo se halla formado por un fichero `ebuild`, que contiene las instrucciones de cómo obtener, compilar, instalar y configurar el software.

Normalmente, tras compilar el paquete, obtenemos también un paquete parecido a los que estamos acostumbrados a ver. Así, si tenemos varias máquinas con la misma versión de Gentoo, no tendremos que compilar en cada máquina. De la misma forma, existen también paquetes binarios precompilados si los necesitamos. De esa forma evitaremos tener que recompilar muchas aplicaciones grandes, como OpenOffice o KDE, por ejemplo.

Los ficheros `ebuild` incluyen información de dependencias, y la herramienta usada para la instalación, `emerge`, las sigue de forma transitiva. También disponemos de paquetes virtuales, con la misma semántica que en Debian. El único problema se halla en las dependencias inversas: a la hora de retirar un paquete, Portage no comprueba si

estamos rompiendo algún otro paquete. Existen herramientas como `revdep-rebuild` que hacen esta comprobación por nosotros, pero no se hallan integradas con el proceso de desinstalación.

Otro problemas incluyen el tener que realizar mantenimiento constante sobre los ficheros de configuración de los paquetes que actualicemos, o el ciclo de desarrollo más corto de los paquetes Gentoo frente a los de Debian. Aunque normalmente tendremos software mucho más reciente, no estará tan probado como lo podría estar un paquete Debian, y podemos llevarnos más de un disgusto.

Por lo demás, se trata de un sistema muy completo, con la capacidad de actualizar todo nuestro sistema comprobando nuestras dependencias de forma transitiva automáticamente, entre otras cosas. Obtenemos las actualizaciones sincronizando de forma periódica nuestro árbol de ficheros `ebuild` con un directorio remoto mediante `rsync`.

Slackware

El formato `tgz`, de Slackware: es únicamente un `tar.gz` que será descomprimido directamente a `/`, para después ejecutar un guión de postinstalación. Se corresponde con la filosofía de dicha distribución: el usuario es el que sabe qué hay que hacer.

No incorpora metadatos de ningún tipo, ni control de dependencias. Es posiblemente el formato más sencillo de empaquetado que podríamos imaginarnos.

4.6.2. Conversión desde paquetes Debian

Puede que queramos que usuarios de SUSE, Slackware o Fedora, por ejemplo, empleen nuestro programa. Quizás no tengamos tiempo como para mantener un paquete para todos y cada uno de los otros muchos formatos disponibles.

La herramienta *alien* (como siempre, antes deberemos instalar su paquete) nos deja convertir entre paquetes Debian, Slackware, Solaris, RPM, LSB y Stampede. Es bastante experimental, y los paquetes generados no tendrán la misma calidad que uno hecho a mano, pero puede ser útil en muchos casos.

Así, para convertir cualquier paquete a formato Debian, usaremos:

```
alien (ruta al paquete)
```

Por otro lado, si queremos crear un paquete RPM a partir de un paquete Debian, podríamos usar, como en el caso de ZSNES:

```
alien --to-rpm ~/packages/build-area/zsnes_1.510.SVN5113-0ubuntu1_i386.deb
```

Sólo hemos de tener cuidado de que la conversión se haya hecho lo bastante bien: por ejemplo, los guiones de preinstalación, postinstalación y demás no se convierten bien al formato RPM. Posiblemente tendremos que editar los ficheros generados para asegurarnos de que funcionen bien.

Bibliografía

[CSFP07] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Control de versiones con Subversion. http://svnbook.red-bean.com/, 2007.

[DP07] Marc Dequènes and Arnaud Patard. CDBS Documentation. https://perso.duckcorp.org/duck/cdbs-doc/cdbs-doc.xhtml, 2007.

[Duc98] Marius Ducea. Ubuntu package version naming explanation. http://www.ducea.com/2006/06/17/ubuntu-package-version-naming-explanation/, 1998.

[Dur06] Felipe Eduardo Sánchez Díaz Durán. CheckInstall. http://asic-linux.com.mx/~izto/checkinstall/index.php, 2006.

[Fre07] FreeDesktop.org. FreeDesktop.org Specifications. http://www.freedesktop.org/wiki/Specifications, 2007.

[Fre08] FreeDesktop.org. FreeDesktop.org Desktop Menu Specification. http://standards.freedesktop.org/menu-spec/latest/, 2008.

[Hie07] Jarkko Hietaniemi. Comprehensive Perl Archive Network. http://www.cpan.org, 2007.

[JS98] Ian Jackson and Christian Schwarz. Debian Policy Manual. http://www.debian.org/doc/debian-policy/, 1998.

[RA07] Josip Rodin and Osamu Aoki. Debian New Maintainer's Guide. http://www.debian.org/doc/maint-guide/index.html, 2007.

[Tea07a] Ubuntu MOTU Team. MOTU School: Packaging Basics. https://wiki.ubuntu.com/MOTU/School/PackagingBasics, 2007.

[Tea07b] Ubuntu MOTU Team. MOTU School: Patching Sources. https://wiki.ubuntu.com/MOTU/School/PatchingSources, 2007.

[VETT06] Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. Autoconf, Automake and Libtool. http://sourceware.org/autobook/, 2006.

[Wik08] Ubuntu Wiki. PackagingGuide/Complete. https://wiki.ubuntu.com/PackagingGuide/Complete, 2008.