Manual del Desarrollador

Release 1.0+0

Equipo de Desarrollo Canaima GNU/Linux

July 27, 2011

Índice general

1. CANAIMA-DESARROLLADOR **3**
1.1. Un conjunto de herramientas para la eliminación de las barreras tecnológicas . . . 3

2. Manual para el Desarrollador **13**
2.1. **Sobre el Ciclo de Desarrollo de Canaima** . . . 13
2.2. **Versionando con GIT** . . . 16
2.3. **Empaquetando con git-buildpackage** . . . 18

3. Guía de Referencia para el Desarrollador **27**
3.1. **Programas que necesitas para el desarrollo** . . . 27
3.2. **Desarrollador oficial de Canaima** . . . 29
3.3. **Primeros Pasos** . . . 30
3.4. **Las cosas necesarias bajo debian** . . . 36
3.5. **Otros ficheros en el directorio debian** . . . 47
3.6. **Construir el paquete** . . . 52
3.7. **Cómo comprobar tu paquete para encontrar fallos** . . . 56
3.8. **Actualizar el paquete** . . . 58

Un conjunto de herramientas para la eliminación de las barreras tecnológicas

Canaima Desarrollador (C-D) es un compendio de herramientas y ayudantes que facilitan el proceso de desarrollo de software para Canaima GNU/Linux. Está diseñado para *facilitar el trabajo* a aquellas personas que participan en dicho proceso con regularidad, como también para *iniciar a los que deseen aprender* de una manera rápida y práctica.

C-D sigue dos líneas de acción principales para lograr éste cometido: *la práctica* y *la formativa*. La práctica permite:

- Agilizar los procesos para la creación de paquetes binarios canaima a partir de paquetes fuentes correctamente estructurados.
- Automatización personalizada de la creación de Paquetes Fuentes acordes a las Políticas de Canaima GNU/Linux.
- Creación de un depósito personal, por usuario, donde se guardan automáticamente y en carpetas separadas los siguientes tipos de archivo: - Proyectos en proceso de empaquetamiento - Paquetes Binarios (*.deb) - Paquetes Fuente (*.tar.gz, *.dsc, *.changes, *.diff) - Registros provenientes de la creación de paquetes binarios (*.build)
- Versionamiento asistido (basado en git) en los proyectos, brindando herramientas para realizar las siguientes operaciones, con un alto nivel de automatización y detección de posibles errores: - git clone - git commit - git push - git pull
- Ejecución de tareas en masa (empaquetar, hacer pull, push, commit, entre otros), para agilizar procesos repetitivos.

En el otro aspecto, el formativo, C-D incluye:

- El Manual del Desarrollador, resumen técnico-práctico de las herramientas cognitivas necesarias para desarrollar paquetes funcionales para Canaima GNU/Linux.
- La Guía de Referencia para el Desarrollador, compendio extenso y detallado que extiende y complementa el contenido del Manual del Desarrollador.
- Éste manual para el uso de Canaima Desarrollador.

Índice de Contenidos

CANAIMA-DESARROLLADOR

1.1 Un conjunto de herramientas para la eliminación de las barreras tecnológicas

Author Luis Alejandro Martínez Faneyth <martinez.faneyth@gmail.com>

Date 2011-01-22

Version 1.0+0

Manual section 1

Manual group Empaquetamiento

1.1.1 MODO DE USO

```
canaima-desarrollador [AYUDANTE] [PARÁMETRO-1] [PARÁMETRO-2] ... [PARÁMETRO-N] [--a
```

1.1.2 DESCRIPCIÓN

Canaima Desarrollador (C-D) es un compendio de herramientas y ayudantes que facilitan el proceso de desarrollo de software para Canaima GNU/Linux. Está diseñado para **facilitar el trabajo** a aquellas personas que participan en dicho proceso con regularidad, como también para **iniciar a los que deseen aprender** de una manera rápida y práctica.

C-D puede ayudarte a:

- Agilizar los procesos para la creación de paquetes binarios canaima a partir de paquetes fuentes correctamente estructurados.
- Automatización personalizada de la creación de Paquetes Fuentes acordes a las Políticas de Canaima GNU/Linux.
- Creación de un depósito personal, por usuario, donde se guardan automáticamente y en carpetas separadas los siguientes tipos de archivo:
  - Proyectos en proceso de empaquetamiento
  - Paquetes Binarios (*.deb)
  - Paquetes Fuente (*.tar.gz, *.dsc, *.changes, *.diff)
  - Registros provenientes de la creación de paquetes binarios (*.build)
- Versionamiento asistido (basado en git) en los proyectos, brindando herramientas para realizar las siguientes operaciones, con un alto nivel de automatización y detección de posibles errores:
  - git clone
  - git commit
  - git push
  - git pull
- Ejecución de tareas en masa (empaquetar, hacer pull, push, commit, entre otros), para agilizar procesos repetitivos.

1.1.3 AYUDANTES DE CANAIMA DESARROLLADOR

- Crear Proyecto / Debianizar
- Crear Fuente
- Empaquetar
- Descargar
- Registrar
- Enviar
- Actualizar
- Descargar Todo
- Registrar Todo
- Enviar Todo
- Actualizar Todo

- Empaquetar Varios
- Empaquetar Todo
- Listar Remotos
- Listar Locales

1.1.4 CREAR PROYECTO / DEBIANIZAR

Crea un proyecto de empaquetamiento desde cero o debianiza uno existente.

USO

```
canaima-desarrollador crear-proyecto|debianizar --nombre="" --version="" --destino=
```

PARÁMETROS

--nombre Un nombre para tu proyecto, que puede contener letras, números, puntos y guiones. Cualquier otro caracter no está permitido.

--version La versión inicial de tu proyecto. Se permiten números, guiones, puntos, letras o dashes (~).

--destino Especifica si es un proyecto de empaquetamiento para Canaima GNU/Linux o si es un proyecto personal. Las opciones disponibles son “canaima” y “personal”.

--licencia Especifica el tipo de licencia bajo el cuál distribuirás tu trabajo. Las licencias soportadas son: apache, artistic, bsd, gpl, gpl2, gpl3, lgpl, lgpl2 y lgpl3.

--ayuda Muestra la documentación para el ayudante.

Si estás debianizando un proyecto existente, lo que ingreses en –nombre=”proyecto” y –version=”X.Y+Z” se utilizará para determinar cuál es el nombre de la carpeta a debianizar dentro del directorio del desarrollador, suponiendo que tiene el nombre proyecto-X.Y+Z. Si no se llama así, habrá un error.

1.1.5 CREAR FUENTE

Crea un paquete fuente a partir de un proyecto de empaquetamiento existente. El resultado es guardado en el depósito de fuentes.

USO

```
canaima-desarrollador crear-fuente --directorio="" [--ayuda]
```

PARÁMETROS

--directorio Nombre del directorio dentro de la carpeta del desarrollador donde se encuentra el proyecto. El directorio debe contener un proyecto debianizado.

--ayuda Muestra la documentación para el ayudante.

1.1.6 EMPAQUETAR

Éste ayudante te permite empaquetar un proyecto de forma automatizada, siguiendo la metodología git-buildpackage, que se centra en el siguiente diagrama:

COMMIT > REFLEJAR CAMBIOS EN EL CHANGELOG > COMMIT > CREAR PAQUETE FUENTE > PUSH > GIT-BUILDPACKAGE

USO

```
canaima-desarrollador empaquetar --directorio="" --mensaje="" --procesadores="" [--
```

PARÁMETROS

--directorio Nombre de la carpeta dentro del directorio del desarrollador donde se encuentra el proyecto a empaquetar.

--mensaje Mensaje representativo de los cambios para el primer commit. El segundo commit es sólo para el changelog. Colocando la palabra "auto" o dejando el campo vacío, se autogenera el mensaje.

--procesadores Número de procesadores con que cuenta tu computadora para optimizar el proceso de empaquetamiento.

--ayuda Muestra la documentación para el ayudante.

1.1.7 DESCARGAR

Éste ayudante te permite copiar a tu disco duro un proyecto que se encuentre en el repositorio remoto para que puedas modificarlo según consideres. Utiliza git clone para realizar tal operación.

Éste ayudante se encarga además de realizar las siguientes operaciones por ti:

- Verifica e informa sobre el éxito de la descarga.

USO

```
canaima-desarrollador descargar --proyecto="" [--ayuda]
```

PARÁMETROS

--proyecto Nombre del proyecto (en caso de que éste se encuentre en el repositorio de Canaima GNU/Linux) o la dirección git pública del proyecto.

--ayuda Muestra la documentación para el ayudante.

1.1.8 REGISTRAR

Éste ayudante te permite registar (o hacer commit de) los cambios hechos en un proyecto mediante el versionamiento basado en git. Utiliza git commit para lograr éste propósito. Éste ayudante se encarga además de realizar las siguientes operaciones por ti:

- Verifica la existencia de la rama git "upstream". En caso de no encontrarla, la crea.
- Verifica la existencia de la rama git "master". En caso de no encontrarla, la crea.
- Verifica la existencia de todos los elementos necesarios para ejecutar la acción git commit (carpetas, variables de entorno, etc..). En caso de encontrar algún error, aborta e informa.
- Autogenera el mensaje de commit, si se le instruye.
- Hace git checkout a la rama master, si nos encontramos en una rama diferente a la hora de hace commit.
- Hace un git merge de la rama master a la upstream, inmediatamente depués del commit.

USO

```
canaima-desarrollador registrar --directorio="" --mensaje=="" [--ayuda]
```

PARÁMETROS

--directorio Nombre de la carpeta dentro del directorio del desarrollador a la que se quiere hacer commit.

--mensaje Mensaje representativo de los cambios para el commit. Colocando la palabra "auto" o dejando el campo vacío, se autogenera el mensaje.

--ayuda Muestra la documentación para el ayudante.

1.1.9 ENVIAR

Éste ayudante te permite enviar los cambios realizados al repositorio remoto especificado en las configuraciones personales, mediante el uso de la acción git push. Éste ayudante se encarga además de realizar las siguientes operaciones por ti:

- Verifica la existencia de la rama git "upstream". En caso de no encontrarla, la crea.
- Verifica la existencia de la rama git "master". En caso de no encontrarla, la crea.
- Verifica la existencia de todos los elementos necesarios para ejecutar la acción git push (carpetas, variables de entorno, etc..). En caso de encontrar algún error, aborta e informa.
- Configura el repositorio remoto para el proyecto, de acuerdo a los parámetros establecidos en ~/.config/canaima-desarrollador/usuario.conf

USO

```
canaima-desarrollador enviar --directorio="" [--ayuda]
```

PARÁMETROS

--directorio Nombre de la carpeta dentro del directorio del desarrollador a la que se quiere hacer push.

--ayuda Muestra la documentación para el ayudante.

1.1.10 ACTUALIZAR

Éste ayudante te permite actualizar el código fuente de un determinado proyecto, mediante la ejecución de "git pull" en la carpeta del proyecto. Éste ayudante se encarga además de realizar las siguientes operaciones por ti:

- Verifica la existencia de la rama git "upstream". En caso de no encontrarla, la crea.
- Verifica la existencia de la rama git "master". En caso de no encontrarla, la crea.
- Verifica la existencia de todos los elementos necesarios para ejecutar la acción git pull (carpetas, variables de entorno, etc..). En caso de encontrar algún error, aborta e informa.

- Configura el repositorio remoto para el proyecto, de acuerdo a los parámetros establecidos en ~/.config/canaima-desarrollador/usuario.conf

USO

```
canaima-desarrollador actualizar --directorio="" [--ayuda]
```

PARÁMETROS

--directorio Nombre de la carpeta dentro del directorio del desarrollador a la que se quiere hacer git pull.

--ayuda Muestra la documentación para el ayudante.

1.1.11 DESCARGAR TODO

Éste ayudante te permite copiar a tu disco duro todos los proyectos de Canaima GNU/Linux que se encuentren en el repositorio remoto oficial. Utiliza git clone para realizar tal operación.

USO

```
canaima-desarrollador descargar-todo [--ayuda]
```

PARÁMETROS

--ayuda Muestra la documentación para el ayudante.

1.1.12 REGISTRAR TODO

Éste ayudante te permite registar (o hacer commit de) todos los cambios hechos en todos los proyectos existentes en la carpeta del desarrollador. Utiliza git commit para lograr éste propósito. Asume un mensaje de commit automático para todos.

USO

```
canaima-desarrollador registrar-todo [--ayuda]
```

PARÁMETROS

--ayuda Muestra la documentación para el ayudante.

1.1.13 ENVIAR TODO

Éste ayudante te permite enviar todos los cambios realizados en todos los proyectos ubicados en la carpeta del desarrollador al repositorio remoto especificado en las configuraciones personales, mediante el uso de la acción git push.

USO

```
canaima-desarrollador enviar-todo [--ayuda]
```

PARÁMETROS

--ayuda Muestra la documentación para el ayudante.

1.1.14 ACTUALIZAR TODO

Éste ayudante te permite actualizar el código fuente de todos los proyectos ubicados en la carpeta del desarrollador, mediante la ejecución de "git pull" en la carpeta del proyecto.

USO

```
canaima-desarrollador actualizar-todo [--ayuda]
```

PARÁMETROS

--ayuda Muestra la documentación para el ayudante.

1.1.15 EMPAQUETAR VARIOS

Éste ayudante te permite empaquetar varios proyectos.

USO

```
canaima-desarrollador empaquetar-varios --para-empaquetar="" --procesadores="" [--a
```

PARÁMETROS

--para-empaquetar Lista de los directorios dentro de la carpeta del desarrollador que contienen los proyectos que se quieren empaquetar, agrupados entre comillas.

--procesadores Número de procesadores con que cuenta tu computadora para optimizar el proceso de empaquetamiento.

--ayuda Muestra la documentación para el ayudante.

1.1.16 EMPAQUETAR TODO

Éste ayudante te permite empaquetar todos los proyectos existentes en la carpeta del desarrollador.

USO

```
canaima-desarrollador empaquetar-todo --procesadores="" [--ayuda]
```

PARÁMETROS

--procesadores Número de procesadores con que cuenta tu computadora para optimizar el proceso de empaquetamiento.

--ayuda Muestra la documentación para el ayudante.

1.1.17 LISTAR REMOTOS

Muestra todos los proyectos contenidos en el repositorio remoto y muestra su dirección git.

USO

```
canaima-desarrollador listar-remotos [--ayuda]
```

PARÁMETROS

--ayuda Muestra la documentación para el ayudante.

1.1.18 LISTAR LOCALES

Muestra todos los proyectos contenidos en la carpeta del desarrollador y los clasifica según su tipo.

USO

```
canaima-desarrollador listar-locales [--ayuda]
```

PARÁMETROS

--ayuda Muestra la documentación para el ayudante.

CAPÍTULO 2

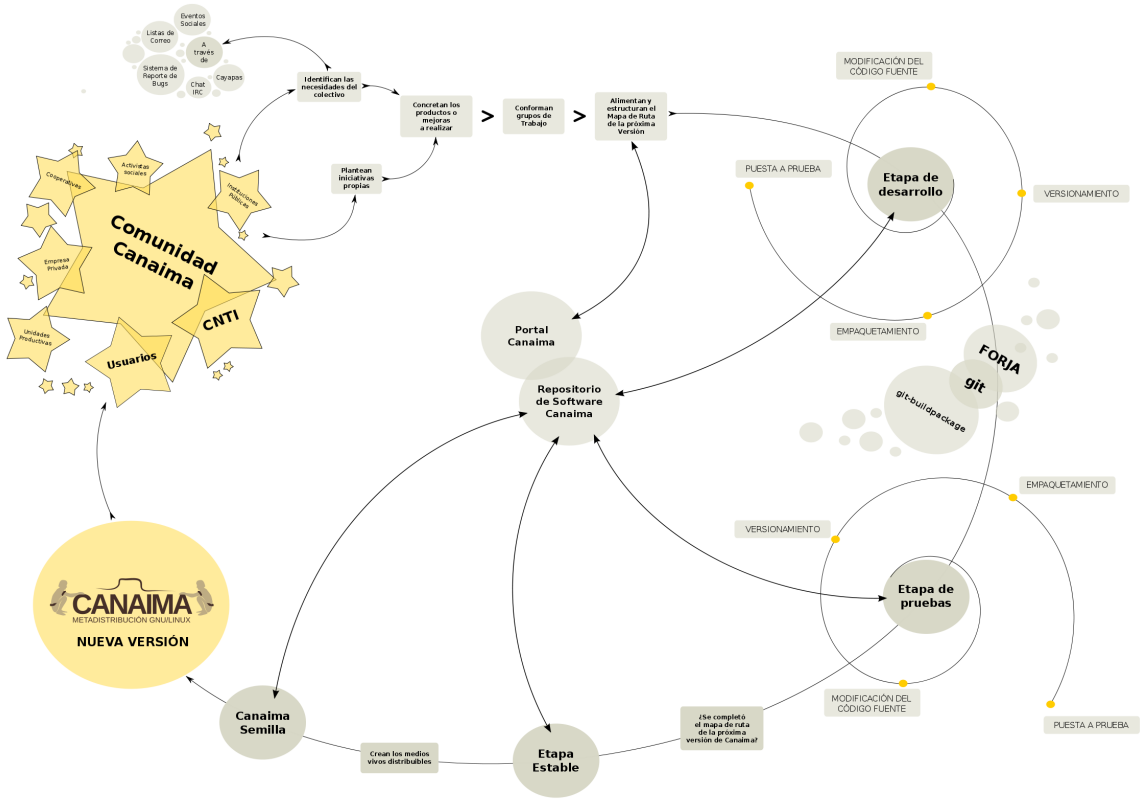
Manual para el Desarrollador

Es bien sabido que Canaima GNU/Linux es una de las distribuciones de Software Libre más importantes de Venezuela. Actualmente se incluye en más de 880 mil portátiles del proyecto Canaima Educativo, ha sido descargado más de 120 mil veces del portal oficial y se planea extender el Proyecto a toda la Administración Pública Nacional.

Sin embargo, la popularidad no lo es todo. Detrás de Canaima GNU/Linux existe un ciclo de desarrollo bastante particular, propio de la dinámica socio-tecnológica en que está envuelto. Éste documento describirá la metodología mediante la cual el colectivo asociado a la Metadistribución, participa en los procesos socio-productivos y de desarrollo que impulsan la generación de Nuevas versiones de Canaima y sustentan su plataforma tecnológica.

2.1 Sobre el Ciclo de Desarrollo de Canaima

Podemos decir que básicamente el ciclo de desarrollo de Canaima sigue un flujo natural de los acontecimientos tal cual se reseñan en el esquema inferior.



2.1.1 ¿Quienes son los actores?

La comunidad canaima está conformada por un gran número de personas que tienen intereses comunes y que comparten un mismo espacio virtual conformado por una serie de herramientas y servicios condensados en una plataforma pública. Éstos actores provienen de distintas organizaciones y colectivos, a saber: el CNTI, Instituciones Públicas en general, Unidades Productivas, Cooperativas, Empresas Privadas, Activistas Sociales y Usuarios en general.

Éste conjunto de personas poseen una dinámica propia, característica de sus realidades particulares; sin embargo, es posible diferenciar un ciclo que se ejecuta cada vez que se publica una nueva versión de Canaima GNU/Linux.

2.1.2 Cocinando una nueva versión de Canaima GNU/Linux

En teoría, una versión de Canaima es publicada cada ~6 meses en sincronía con la Cayapa. Para que ésto se cumpla, ciertos hitos deben ser alcanzados.

La primera meta es realizar un producto. ¿Quien puede realizar un producto para Canaima GNU/Linux? Cualquiera puede hacerlo, sin embargo te recomendamos ciertas cosas que te ayudarán a hacerlo con mayor calidad y eficiencia:

- Aprende a empaquetar con git-buildpackage (más abajo).

- Participa en la lista de correo desarrolladores, discusion y soporte.
- Usa Canaima para todo.
- Infórmate acerca de las conclusiones a las que se llegaron en la última Cayapa.

¿Que producto hago? Existen dos formas de actuar en éste dilema: **determinar y seleccionar una necesidad actual del Proyecto Canaima**, a través de los medios de comunicación destinados a ello (Listas de Correo, Sistema de Reporte de Bugs, Chat IRC, Cayapa y otros eventos sociales) o **seleccionar un proyecto de iniciativa propia**. Una vez concretado el producto que vas a hacer, cómo lo vas a hacer y en que tiempo, anúncialo, comunícalo a la comunidad y ¡Manos a la obra!

Es importante añadir tu proyecto en el Mapa de ruta de la próxima versión de Canaima.

2.1.3 Etapas de Desarrollo

Las etapas de desarrollo iniciales de un producto son locales e involucran casi cualquier elemento que al desarrollador se le ocurra, en fin, es producto de un proceso creativo que se parece mucho a un proceso artístico. Cuando tu proyecto tenga una estructura más o menos definida, es hora de versionarlo con git e ir publicando tu código; en ésta etapa entra en juego forja.softwarelibre.gob.ve.

"La forja" es un espacio público donde puedes alojar proyectos de software libre de una forma práctica y gratuita, bajo la plataforma de Canaima GNU/Linux y usando un repositorio git. Crea un proyecto y ve publicando tu código ahí.

A medida que vas madurando y depurando tu código, es buena idea ir también trabajando en la debianización del paquete fuente para finalmente generar el paquete binario con git-buildpackage. Haz sucesivas pruebas de empaquetado con tu proyecto, y cuando llegues a un nivel en donde sea usable, puedes solicitar a través de la lista de correo desarrolladores permisos suficientes en los servidores de la Plataforma Canaima para subir tus paquetes a la rama de desarrollo. En tu solicitud debes incluir:

1. Nombre del Paquete (Completo).
2. Descripción de su utilidad.
3. Dependencias con otros paquetes.
4. Dirección del código fuente.
5. Direccción del paquete .deb tal cual va a ser incluído en los repositorios.

Una vez otorgados los permisos, puedes subir cuantas versiones consideres, con la frecuencia que necesites. Está demás decir que cualquier intento de violar la privacidad del usuario o de inyectar código malicioso resultará en una severa penalización.

Cuando sientas que haz llegado a una versión estable de tu paquete, es hora de incluirlo en la rama pruebas, para que el colectivo lo use y ofrezca su retroalimentación. El procedimiento es similar al anterior, realiza la petición de inclusión del paquete en la rama pruebas incluyendo la siguiente información:

1. Nombre del Paquete (Completo).
2. Descripción de su utilidad.
3. Dependencias con otros paquetes.
4. Dirección del código fuente.
5. Direccción del paquete .deb tal cual va a ser incluído en los repositorios.

Finalmente, cuando todos los objetivos establecidos en el mapa de ruta se hayan cumplido, todos los paquetes de la rama pruebas serán pasados a estable, generando así una nueva versión de Canaima. Los medios vivos instalables serán generados a través de canaima-semilla.

2.2 Versionando con GIT

GIT es una herramienta de desarrollo muy útil. Con ella cualquier persona podrá manejar de una manera sencilla y práctica el versionamiento de su trabajo.

2.2.1 ¿Por qué necesito versionamiento?

Muchas veces pasa que queremos devolver uno o varios cambios en archivos que ya guardamos y cerramos y nos encontramos con que no tenemos forma de hacerlo.

Con el control de versiones o versionamiento, tenemos la facilidad de gestionar los diferentes cambios que se hacen en el contenido, configuración y propiedades de los archivos de un determinado proyecto. Ésta característica nos permite devolver cambios hacia versiones anteriores, además de facilitar el acceso y distribución de código fuente mediante la utilización de repositorios locales o remotos.

2.2.2 ¿Y por qué GIT?

GIT es una herramienta de versionamiento creada por Linus Torvalds, desarrollador del Kernel Linux. Entre sus beneficios con respecto a otros sistemas de versionamiento tenemos que es un sistema distribuido que permite el trabajo con repositorios locales que luego pueden ser fusionados con el repositorio principal.

2.2.3 ¿Cómo uso GIT?

Lo primero que se debe hacer es instalarlo. Es muy fácil, -como es de costumbre en Linux-, escribimos lo siguiente en una consola con permisos de superusuario:

```
aptitude install git-core
```

Para comenzar a trabajar, accedemos al directorio principal de nuestro proyecto y ejecutamos los siguientes comandos:

```
git init
```

Con ésto inicializamos el versionamiento en el directorio raíz del proyecto:

```
git add .
```

Añadimos todos los archivos del proyecto a ser versionados a nuestro repositorio local:

```
git commit -a -m "Mensaje descriptivo de los cambios"
```

Realizamos la carga de la primera versión de nuestro proyecto.

2.2.4 Uso de un repositorio en línea

Existen varios lugares en internet que brindan servicio gratuito para almacenar proyectos de Software Libre bajo la plataforma GIT. Éstos sitios proveen un repositorio dinámico que permite un versionamiento descentralizado, es decir, que varias personas podrían hacer carga y descarga de datos en nuestro proyecto mediante permisología definida y fusión inteligente, facilitando así el trabajo colaborativo entre diferentes personas.

Uno de éstos sitios es Gitorious (también está Github). Para poder hacer uso de los servicios de Gitorious (o Github), es necesario que tanto el creador del Proyecto como sus colaboradores se registren. Además, cada cuenta creada debe asignarsele la (o las) llave(s) SSH de los equipos autorizados para publicar o descargar contenido. Para conocer nuestra llave SSH utilizamos el comando ssh-keygen; el resultado de ésta consulta debe ser ingresado en el apartado "Manage SSH Keys", de la página de tu perfil en gitorious.org.

2.2.5 Comenzando a trabajar

Inicialmente, debemos agregar el repositorio remoto, que para el caso de gitorious, se indica en la página principal del proyecto. Para ello, creamos un alias o nombre para la dirección del repositorio, de la siguiente forma:

```
git remote add <alias> <dirección>
```

Por ejemplo:

```
git remote add origin git@gitorious.org:miproyecto/mainline.git
```

Seguidamente, el comando para ejecutar la carga de archivos versionados al servidor es el siguiente (recordar hacer git commit -a antes):

```
git push <alias> <rama>
```

En donde rama indica la rama del ciclo de desarrollo al que pertenece esta carga de archivos. La principal es master. Por ejemplo:

```
git push origin master
```

2.2.6 Programación Colaborativa

Otras personas pueden bajar los archivos fuente "clonando" tu repositorio. Para hacerlo creamos una carpeta, digamos "proyecto" y dentro de ella ejecutamos:

```
git clone <dirección>
```

Por ejemplo:

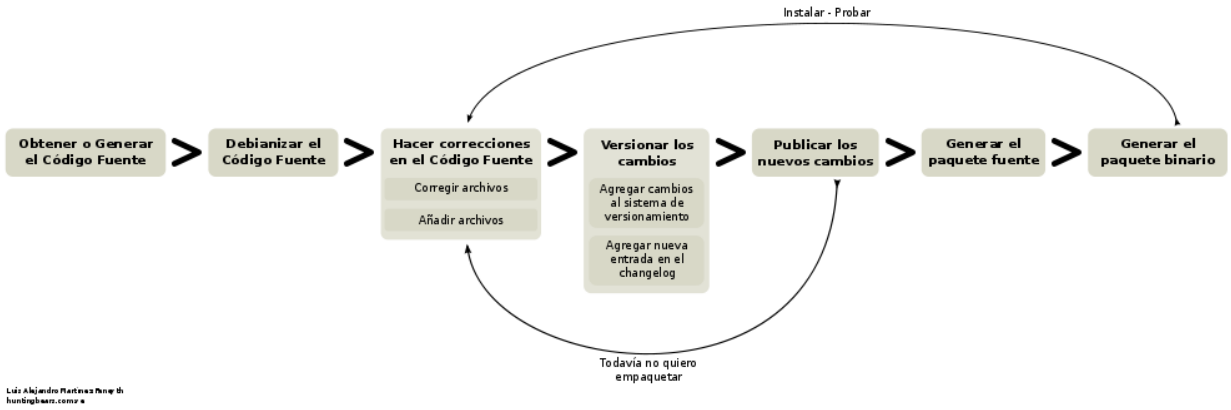
```
git clone http://git.gitorious.org/miproyecto/mainline.git
```

Esto descargará los archivos de la rama principal (master) a la carpeta donde nos encontremos. Una vez en poder de los archivos fuente, podremos realizar modificaciones y subirlos al repositorio en línea con el comando git push, descrito anteriormente (siempre y cuando tengamos la permisología necesaria del creador en gitorious.org). Ésta actividad podemos repetirla cuantas veces sea necesario.

Es recomendable actualizar los archivos fuente cada vez que se vayan a realizar cambios para evitar discordancias.

2.3 Empaquetando con git-buildpackage

Ésta metodología involucra, como eje fundamental, el sistema de versionamiento git fusionado con la metodología de empaquetamiento debian debhelper, permitiéndole al desarrollador mantener el flujo de trabajo estándar en proyectos de Software Libre, usando una sola herramienta. Si necesitas mayor detalle en la descripción de los procesos, puedes consultar la Guía de referencia del Desarrollador.



2.3.1 Términos Fundamentales

Para comenzar, primero debemos revisar algunos conceptos que nos ayudarán a desenvolvernos mejor en el ambiente de desarrollo que necesitaremos. Éstos términos serán descritos de forma que cualquiera lo pueda entender, puesto que es la introducción de éste post y a medida que vayas leyendo, el nivel de dificultad irá aumentando. Sin embargo, sólo se escribirá lo necesario para que empaquetes, es decir, nada faltará ni nada sobrará.

Un **Paquete Fuente** (comúnmente de extensión .tar.gz) es un paquete comprimido que contiene los archivos fuente de un determinado software. Éstos pueden ser por sí mismos los archivos ejecutables (binarios, scripts, entre otros) o, pueden ser los archivos a través de los cuales se generan los ejecutables mediante de un proceso de compilación que depende directamente del lenguaje en que está escrito el software. Para que éstos archivos ejecutables (y demás archivos de contenido y configuración) sean reconocidos por el Sistema de Gestión de Paquetes de Debian, y gocen del beneficio que esto representa (tanto para el desarrollador como para el Sistema operativo), éstos deben ser agrupados y distribuidos a los usuarios en paquetes binarios (.deb).

Por otra parte, los paquetes fuente (adaptados a Debian GNU/Linux) contienen una carpeta llamada "debian" (nótese las minúsculas), en donde se encuentran diferentes archivos que contienen toda la información necesaria para generar el paquete binario a partir del código fuente. Comúnmente la generación de ésta carpeta (proceso al que se le llama "Debianización del Código Fuente") es la parte más difícil del empaquetamiento, ya que se debe editar manualmente y para ello se debe conocer la estructura del Sistema Operativo (donde va cada tipo de cosa) y la estructura del programa que se desea empaquetar (para qué sirve cada cosa).

Los paquetes fuentes son distribuidos por el desarrollador de la aplicación y por el mantenedor del paquete en las diferentes distribuciones en las que esté disponible.

Los **Paquetes Debian** (.deb), también llamados paquetes binarios, son paquetes que contienen software instalable en sistemas operativos Debian y derivados (Ubuntu, Canaima, etc...). Está compuesto por dos partes fundamentales: Archivos de Control y Archivos de Datos.

Los Archivos de Control están agrupados en una carpeta llamada "DEBIAN" (nótese las mayúsculas) y contienen la información necesaria para que el sistema de gestión de paquetes instale (control, md5sum) y configure el paquete (preinst, postinst, prerm, preinst); no debe ser confundido con la carpeta debian de los archivos fuente, la carpeta DEBIAN es generada a partir de la carpeta debian en el proceso de empaquetamiento.

Los Archivos de Datos son los archivos binarios, de texto, configuración y de contenido general propios de la aplicación, dispuestos en la estructura de archivos del sistema tal cual van a ser copiados.

Los paquetes binarios son distribuídos por el mantenedor (o empaquetador) de la aplicación en las diferentes distribuciones en las que esté disponible.

Un **Makefile** es un archivo que forma parte de un paquete fuente y que contiene las instrucciones para probar, compilar, instalar, limpiar y desinstalar el software que se distribuye de forma "estática" (no recibe actualizaciones ni se verifican dependencias mediante el sistema de paquetes de Debian). Es generado por el desarrollador del software, quien conoce exactamente como realizar éstas operaciones.

En algunos casos más complejos, se hace necesario hacer un Makefile para distintos propósitos, por lo que se usa otro elemento que a partir de ciertos procedimientos, genera el Makefile automáticamente; éste elemento es el archivo configure. El archivo configure es producido por un set de herramientas desarrolladas por el proyecto GNU denominadas autotools (aunque también puede ser generado manualmente). Puedes encontrar mayor información de cómo generar el makefile para tus aplicaciones aquí, aquí, también aquí y probablemente aquí y aquí (también aquí, aquí, aquí y aquí).

El archivo rules de la carpeta debian es un archivo Makefile, que contiene las operaciones a realizar para generar la estructura de los Archivos de Datos de un paquete binario. Generalmente son operaciones comunes de movimiento de archivos, y creación de carpetas; sin embargo, pueden incluirse operaciones más complejas dependiendo de las necesidades del mantenedor del paquete. Recientemente, y gracias al conjunto de scripts debhelper, no es necesario realizar éstas operaciones "a mano" ya que existen "ayudantes" que detectan qué debe hacerse con cuales archivos a partir de la presencia de ciertas instrucciones en la carpeta debian durante el proceso de empaquetado. Puedes ampliar la información aquí.

Suficientes términos por ahora, manos a la obra!

2.3.2 Empezando

Para comenzar, necesitaremos varios insumos, uno de ellos es la descripción de nuestro entorno de trabajo. Estamos trabajando en el sistema operativo Canaima, sin embargo, ésta guía también es aplicable a sistemas operativos basados en Debian Lenny (con ligeras diferencias). Usaremos el paquete canaima-semilla para nuestro ejemplo.

Otra cosa que necesitaremos son herramientas de empaquetamiento. A continuación abran una terminal con permisos de Administrador y ejecuten el siguiente comando:

```
aptitude install git-buildpackage build-essential dpkg-dev file libc6-dev patch per
```

Obteniendo el código fuente

Seguidamente obtengamos el código fuente de la aplicación a empaquetar, cosa que podemos hacer de dos formas:

1.- Clonando el repositorio git con el comando gpb-clone:

```
gbp-clone git@gitorious.org:canaima-gnu-linux/canaima-semilla.git
```

2.- O, generando un repositorio git local a partir de un paquete tar.gz:

```
mkdir canaima-semilla
cd canaima-semilla
git init
git-import-orig canaima-semilla-1.5+3.orig.tar.gz
```

Luego de aplicado alguno de los métodos previos, tendremos una carpeta llamada "canaima-semilla", conteniendo nuestro código fuente. Es una buena práctica renombar en ésta etapa la carpeta para que cumpla con el siguiente formato: <Paquete>-<Versión>+<Revisión>, para evitarnos problemas más adelante. En el caso de nuestro ejemplo quedaría: canaima-semilla-1.5+3.

Por supuesto, si se está empezando a escribir el programa desde cero, los métodos anteriores no son válidos, ya que ya tendríamos las fuentes en nuestro computador. En ese caso, simplemente posicionate en la carpeta raíz de tu proyecto y haz tu primera versión con git.

2.3.3 Debianizando el código fuente

Suponiendo que nuestro paquete no contiene la carpeta debian (generalmente se incluye) o que estamos haciendo un desarrollo nuevo (y no ha sido empaquetado antes), necesitaremos realizar éste trabajo por nosotros mismos mediante el comando dh_make (debhelper). También, aunque ya tengamos la carpeta debian en nuestro código fuente, éste comando nos permite generar automáticamente una copia de las fuentes modificadas con el sufijo .orig, el cual es un elemento que será utilizado como insumo en un proceso posterior del empaquetado.

Como precaución, es recomendable declarar las siguientes variables de entorno antes de ejecutar el comando dh_make, para asegurarnos de identificarnos bien:

```
export DEBFULLNAME="<nombre completo del mantenedor>"
export DEBEMAIL="<correo del mantenedor>"
```

Estando dentro de la carpeta del paquete fuente, ejecutaremos el siguiente comando:

```
dh_make --createorig --cdbs --copyright <licencia> --email <correo>
```

En donde:

--createorig Creará una copia de la carpeta donde se encuentra el código fuente, añadiendo el sufijo .orig. Ésto servirá para regenerar el paquete fuente en etapas posteriores del proceso.

--cdbs Le dirá al proceso que vamos a utilizar el Common Debian Build System, por lo que incluirá algunas plantillas útiles en la carpeta debian.

--copyright Especificará bajo cual licencia publicaremos nuestro software.

--email Identificará el código fuente con nuestro correo.

Para nuestro ejemplo haremos:

```
dh_make --createorig --cdbs --copyright gpl3 --email nombre@correo.com
```

Una vez finalizado el proceso, tendremos unas fuentes debianizadas. Sin embargo, ahora hay que adaptarlas a las necesidades del paquete binario que queremos construir. Examinemos lo que ha puesto dh_make en la carpeta debian por nosotros:

- changelog
- compat
- control
- copyright
- cron.d.ex
- docs
- emacsen-install.ex
- emacsen-remove.ex
- emacsen-startup.ex
- init.d.ex
- manpage.1.ex
- manpage.sgml.ex
- manpage.xml.ex
- menu.ex
- postinst.ex
- postrm.ex
- preinst.ex

- prerm.ex
- prueba.cron.d.ex
- prueba.default.ex
- prueba.doc-base.EX
- README.source
- README.Debian
- rules
- source
- watch.ex

Cada uno de éstos archivos son utilizados por algún ayudante de debhelper para construir el paquete. Su configuración es bastante intuitiva, sin embargo proporcionamos algunos ejemplos:

debian/control Este archivo controla el nombre del paquete fuente, el nombre del paquete binario, en qué sección va el paquete, quién es el responsable (aquí podemos definir también co-responsables), si el paquete reemplaza a otro, sugerir y/o recomendar otras cosas y definir dependencias (tanto en fuentes como en binarios). Más información. . .

debian/changelog En este archivo verás el paquete, la versión+revisión Debian, repositorio y la urgencia, algo como canaima-semilla (1.5+3) desarrollo; urgency=low. Donde 1.5 es la versión del programa, +3 es la revisión de Debian, desarrollo es el repositorio al que deberías subirlo y urgency=low establece cuánto tiempo pasará en paquete en «desarrollo» antes de que se intente migrar a «pruebas» («low» significa 10 días), normalmente usarás el valor «low», aunque «medium» y «high» también están disponibles.

debian/copyright En este archivo debes especificar el autor original, el lugar desde el que descargaste el software, y la licencia del programa. Más información. . .

debian/docs Este archivo incluye los documentos que se copiarán a /usr/share/doc/paquete cuando se instale. Deben incluirse uno por línea.

debian/compat Este archivo determina el nivel de compatibilidad con debhelper. Actualmente el nivel recomendado es 7.

2.3.4 Realizar cambios al código fuente

Ésta etapa es bastante flexible y depende en su totalidad de la persona que lo haga. Aquí se harán los cambios que el desarrollador considere de acuerdo con sus objetivos (corregir errores, agregar funcionalidades, entre otros). Usará las herramientas que considere necesarias e incorporará y modificará los archivos que desee sin ningún tipo de restricción, siempre y cuando lo haga dentro

de la carpeta de trabajo e incorpore las nuevas reglas (si las hubiera) en los archivos de construcción e instalación del paquete (Makefile, debian/rules, etc..).

2.3.5 Versionar los cambios

Una vez realizados los cambios, y se considere que son suficientes como para que constituyan una nueva versión de nuestro paquete, es tiempo de versionar el nuevo estado de tu proyecto. Para ello utilizaremos el flujo de trabajo natural de git, que describimos en un post anterior, para luego plasmar los cambios en el archivo debian/changelog mediante el comando git-dch. Éste comando se encargará de recopilar todos los commits nuevos que se han hecho desde la última versión y usará todos sus mensajes para llenar el archivo debian/changelog con una nueva entrada.

Como precaución, es recomendable declarar las siguientes variables de entorno antes de ejecutar el comando git-dch, para asegurarnos de identificarnos bien:

```
export DEBFULLNAME="<nombre completo del mantenedor>"
export DEBEMAIL="<correo del mantenedor>"
```

Ejecutamos en el directorio base, el siguiente comando:

```
git-dch --release --auto --id-length=7 --full
```

En donde:

--release Indica que es una nueva versión y que es definitiva (si en cambio usamos –snapshot, se considerará como una versión temporal)

--auto Indica que se adivinará el número de la versión a partir de la entrada anterior.

--id-length="N" Es el número de caracteres del código del commit que se incluirán.

--full Le indicará que debe incluir todo el mensaje del commit y no un extracto del mismo.

Si por casualidad hemos ejecutado éste comando sin tener commits nuevos, la nueva entrada del debian/changelog será rellenada con la palabra “UNRELEASED”, la cual desaparecerá en el próximo ciclo de versionamiento.

2.3.6 Generar el paquete fuente

Para generar el paquete fuente, necesitamos añadir los cambios a la rama upstream, la cual es usada como rama “fuente”. Si no está disponible debemos crearla con el comando **git branch upstream**. Para añadir los cambios debemos fusionar la rama master con la upstream de la siguiente forma:

```
git checkout upstream
git merge master
git checkout master
```

El siguiente paso es generar la carpeta .orig.tar.gz que va a ser utilizada para generar el paquete fuente, a través de dh-make:

```
dh_make --createorig --cdbs --copyright <licencia> --email <correo>
```

Seguidamente, creamos el paquete fuente en cuestión, excluyendo el directorio git:

```
cd ..
dpkg-source --format="1.0" -i.git/ -I.git -b canaima-semilla-1.5+3
```

2.3.7 Publicar los cambios

En ésta etapa, es hora de hacer saber a los demás que existe una nueva versión del código fuente, y la mejor forma de hacerlo es a través de un repositorio público como github o gitorious.

> git push origin master upstream

2.3.8 Generar el paquete binario

Finalmente podemos generar nuestro paquete binario. Para ello ejecutamos el siguiente comando:

```
git-buildpackage -k<llave> -tc --git-tag -jN
```

En donde:

-k<llave> Especifica la llave pública GPG con que se firmará el paquete.

-tc Limpia el directorio base de los residuos de la construcción del paquete.

--git-tag Crea una etiqueta que agrupa todos los commits de una determinada versión.

-jN Permite utilizar un número N de hilos para ejecutar el proceso. Se recomienda que N sea el número de procesadores más uno.

Si el proceso culmina satisfactoriamente, correrá lintian para indicarnos si hay alguna discrepancia con las normas de empaquetamiento de debian.

Si el proceso se interrumpe, es una buena práctica crear el tag para evitar errores al correr git-dch en el próximo ciclo. Ejecuta `gitbuildpackage --git-tag-only` para asignar el tag sin volver a intentar construir el paquete.

2.3.9 Hoja Resumen (cheat sheet) del flujo de trabajo

```
git add .
git commit --all
git-dch --release --auto --id-length=7 --full
(directorio renombrado)
cd ../nuevo-directorio/
git commit --all
git checkout upstream
git merge master
git checkout master
git push origin master upstream
dh_make --createorig --cdbs --copyright <licencia> --email <correo>
cd ..
(para crear las fuentes formato 1.0)
dpkg-source --format="1.0" -i.git/ -I.git -b nuevo-directorio
cd nuevo-directorio
git push gitorious master upstream
git-buildpackage -k<llave> -tc --git-tag -jN
```

¡Feliz Empaquetado!

CAPÍTULO 3

Guía de Referencia para el Desarrollador

Este documento tratará de describir cómo se construye un paquete Canaima GNU/Linux para el usuario común de Canaima y para futuros desarrolladores en un lenguaje informal, y con multitud de ejemplos. Hay un antiguo dicho romano que dice, Longum iter est per preaecepta, breve et efficax per exempla! (¡Es un largo camino con las reglas, pero corto y eficiente con ejemplos!)

Canaima esta basado en Debian, y una de las cosas que hace a Debian una de las distribuciones más importantes del mercado es su sistema de paquetes. Aunque hay una gran cantidad de programas disponibles en forma de paquetes de Debian, algunas veces necesitarás instalar programas que no están disponible en este formato.

Este documento explicará cada pequeño paso (al principio quizás irrelevantes), le ayudará a crear tu primer paquete, ganar alguna experiencia en construir próximas versiones de él, y quizás otros paquetes después.

Este documento se basa en la "Guía del nuevo desarrollador de Debian". se pueden obtener versiones nuevas de este documento en línea en http://www.debian.org/doc/maint-guide/ y en el paquete «maint-guide-es».

3.1 Programas que necesitas para el desarrollo

Antes de empezar nada, deberías asegurarte de que tienes instalados algunos paquetes adicionales necesarios para el desarrollo. Observa que en la lista no están incluidos paquetes cuyas prioridades son «esencial» o «requerido», que se suponen ya instalados.

Los siguientes paquetes vienen en una instalación estándar de Canaima, así que probablemente ya los tengas (junto con los paquetes de los que dependen). Aún así, deberías comprobarlo con:

```
dpkg -s <paquete>
```

dpkg-dev este paquete contiene las herramientas necesarias para desempaquetar, construir y enviar paquetes fuente de Canaima.

file este útil programa puede determinar de qué tipo es un fichero

gcc el compilador de C de GNU, necesario si el programa, como la gran mayoría, está escrito en el lenguaje de programación C. Este paquete también vendrá con otros paquetes como **binutils** que incluye programas para ensamblar y enlazar ficheros objeto y el preprocesador de C en el paquete cpp.

libc6-dev las bibliotecas y archivos de cabecera de C que gcc necesita para enlazar y crear ficheros objeto.

make habitualmente la creación de un programa consta de varios pasos. En lugar de ejecutar las mismas órdenes una y otra vez, puedes utilizar este programa para automatizar el proceso, creando ficheros **«Makefile»**

patch esta utilidad es muy práctica, ya que permite tomar un fichero que contiene un listado de diferencias (producido por el programa diff) y aplicárselas al fichero original, produciendo una versión "parcheada".

perl Perl es uno de los lenguajes interpretados para hacer guiones (o *«scripts»*) más usados en los sistemas Un*x de hoy en día, comúnmente se refiere a él como la **«navaja suiza de Unix»**.

Probablemente, necesitarás instalar además los siguientes paquetes:

autoconf y *automake*: muchos programas nuevos usan ficheros de configuración y ficheros «Makefile» que se procesan con la ayuda de programas como éstos.

dh-make y *debhelper*: *dh-make* es necesario para crear el esqueleto de nuestro paquete ejemplo, y se usarán algunas de las herramientas de debhelper para crear los paquetes. Aunque no son imprescindibles para la creación de paquetes se recomiendan encarecidamente para nuevos desarrolladores. Hacen el proceso mucho más fácil al principio, y más fácil de controlar también más adelante, mas información en el directorio */doc/debhelper/README*.

devscripts: este paquete contiene algunos guiones útiles para los desarrolladores, pero no son necesarios para crear paquetes, mas información en el directorio */usr/share/doc/devscripts/README.gz*.

fakeroot: esta utilidad te permite emular al usuario administrador (o *root*), lo cual es necesario para ciertas partes del proceso de construcción.

gnupg: herramienta que te permite firmar digitalmente los paquetes. Esto es especialmente importante si quieres distribuir tu paquete a otras personas, y ciertamente, tendrás que hacerlo cuando tu trabajo vaya a incluirse en la distribución de Debian.

g77: el compilador GNU de Fortran 77, necesario si el programa está escrito en Fortran.

gpc: el compilador GNU de Pascal, necesario si el programa está escrito en Pascal. Merece la pena mencionar aquí fp-compiler, un compilador libre de Pascal, que también es bueno en esta tarea.

xutils: algunos programas, normalmente aquellos hechos para X11, también usan programas para generar Makefiles de un conjunto de funciones de macro.

lintian: este es el comprobador de paquetes de Debian, que te indica muchos de los errores comunes después de construir un paquete, y explica los errores encontrados, mas información en */usr/share/doc/lintian/lintian.html/index.html*.

pbuilder: este paquete contiene programas para crear y mantener entornos chroot. Al construir paquetes Debian en estos entornos chroot se verifica que las dependencias son las adecuadas y se evitan fallos al construir desde el código fuente.

Las breves descripciones dadas anteriormente sólo sirven para introducirte a lo que hace cada paquete. Antes de continuar, por favor, lee la documentación de cada programa, al menos para su uso normal. Puede parecerte algo duro ahora, pero más adelante estarás muy contento de haberla leído. Nota: debmake es un paquete que incluye otros programas con funciones similares a dh-make, pero su uso específico no está cubierto en este documento porque se trata de una herramienta obsoleta.

3.2 Desarrollador oficial de Canaima

Puede que te quieras convertir en un desarrollador oficial de Canaima una vez hayas construido tu paquete (o incluso mientras lo estás haciendo) para que el paquete se introduzca en la nueva distribución (si el programa es útil, ¿por qué no?).

No puedes convertirte en desarrollador oficial de Canaima de la noche a la mañana porque hace falta más que sólo habilidades técnicas. No te sientas desilusionado por esto. Aún puedes subir tu paquete, si es útil a otras personas, como su mantenedor a través de un patrocinador mientras tu entras en el proceso de nuevos desarrolladores de Debian. En este caso el patrocinador es un desarrollador oficial de Debian que ayuda a la persona que mantiene el paquete a subirlo al archivo de Debian.

Ten en cuenta que no tienes que crear un paquete nuevo para poder convertirte en desarrollador oficial. Un camino posible para ser desarrollador oficial es contribuir al mantenimiento de los paquetes ya existentes en la distribución.

3.2.1 Más información

Puedes construir dos tipos de paquetes: fuentes y binarios. Un paquete fuente contiene el código que puedes compilar en un programa. Un paquete binario contiene sólo el programa terminado. ¡No mezcles los términos como «fuentes de un programa» y el «paquete fuente de un programa»! Por favor, lee los otros manuales si necesitas más detalles sobre terminología.

Canaima usa el término *desarrollador* se utiliza para referirse a la persona que hace paquetes, y el termino *autor original* para la persona que hizo el programa, y *desarrollador original* para la persona que actualmente mantiene el programa fuera de Debian. Generalmente el autor y el desarrollador fuente son la misma persona - y algunas veces incluso el desarrollador es el mismo. Si haces un programa, y quieres incluirlo en Debian, tienes total libertad para solicitar convertirte en desarrollador.

3.3 Primeros Pasos

3.3.1 Elige el programa

Probablemente hayas escogido ya el paquete que deseas construir. Lo primero que debes hacer es comprobar si el paquete está ya en el archivo de la distribución utilizando synaptic.

Si el paquete es nuevo y decides que te gustaría verlo en Canaima GNU/Linux debes seguir los pasos indicados a continuación:

- Comprueba que no hay nadie más trabajando ya en el paquete consultando la lista de paquetes en los que se está trabajando. Si ya hay alguien trabajando en él, contacta con esa persona. Si no, intenta encontrar otro programa interesante que nadie mantenga.
- El programa debe tener una licencia. Preferiblemente la licencia deberá ser libre en el sentido marcado por las Directrices de Debian para el software libre y no puede depender de un paquete que no esté dentro de «main» para compilarse o para poder utilizarse. Si la licencia no sigue alguna de estas reglas aún puede incluirse en las secciones «contrib» o «non-free» de Debian dependiendo de su situación. Si no estás seguro sobre en qué lugar debería ir, envía el texto de la licencia y pide consejo con un correo (en inglés) dirigido a debian-legal@lists.debian.org.
- El programa no debería ejecutarse con «setuid root», o aún mejor: no debería ser «setuid» ni «setgid».
- El programa no debería ser un demonio, o algo que vaya en los directorios */sbin*, o abrir un puerto como usuario administrador.
- El programa debería estar compuesto por binarios ejecutables, no intentes empaquetar aún con bibliotecas.
- El programa debería tener una buena documentación, o al menos un código fuente legible y no ofuscado.
- Deberías contactar con el autor o autores del programa para comprobar si está/n de acuerdo con que se empaquete. Es importante que el autor o autores sigan manteniendo el programa para que puedas en el futuro consultarle/s en caso de que haya problemas específicos. No deberías intentar empaquetar programas que no estén mantenidos.

- Y por último, pero no menos importante, deberías saber cómo funciona, y haberlo utilizado durante algún tiempo.

Por supuesto, esta lista es para tomar medidas de seguridad, y con la intención de salvarte de usuarios enfurecidos si haces algo mal con algún demonio «setuid»... Cuando tengas más experiencia en empaquetar, podrás hacer este tipo de paquetes, incluso los desarrolladores más experimentados preguntan en la lista de correo de *desarrolladores* cuando tienen dudas. La gente allí te ayudará gustosamente.

3.3.2 Obtén el programa y pruébalo

Lo primero que debes hacer es encontrar y descargar el paquete original. A partir de este punto se da por supuesto que ya tienes el código fuente que obtuviste de la página del autor. Las fuentes de los programas libres de GNU/Linux generalmente vienen en formato *tar/gzip*, con extensión *.tar.gz*, y generalmente contienen un subdirectorio llamado *«programa-versión»* con todas las fuentes en él. Si tu programa viene en otro tipo de archivo (por ejemplo, el fichero termina en ".Z" o ".zip"), descomprímelo con las herramientas adecuadas, o pregunta en la lista de correo *desarrolladores* si tienes dudas de cómo se puede desempaquetar correctamente (pista: prueba *«file archivo.extensión»*).

Como ejemplo, usaré el programa conocido como «gentoo», un gestor de ficheros de X11 en GTK+. Observa que el programa ya ha sido empaquetado previamente pero ha cambiado sustancialmente de versión desde que este texto se escribió.

Crea un subdirectorio bajo tu directorio personal llamado «debian» o «deb» o lo que creas apropiado (por ejemplo ~/gentoo/ estaría bien en este caso). Mueve a él el archivo que has descargado, y descomprímelo de la siguiente forma: «tar xzf gentoo-0.9.12.tar.gz». Asegúrate de que no hay errores, incluso errores «irrelevantes», porque es muy probable que haya problemas al desempaquetarlo en sistemas de otras personas, cuyas herramientas de desempaquetado puede que no ignoren estas anomalías.

Ahora tienes otro subdirectorio, llamado «gentoo-0.9.12». Muévete a ese directorio y lee en profundidad la documentación que encuentres. Generalmente se encuentra en ficheros que se llaman *README*, *INSTALL*, *.lsm o *.html. Allí encontrarás instrucciones de cómo compilar e instalar el programa (muy probablemente asumirán que lo quieres instalar en el directorio /usr/local/bin, no harás esto, pero eso lo veremos más adelante en Instalación en un subdirectorio, Sección 3.1).

El proceso varía de un programa a otro, pero gran parte de los programas modernos vienen con un guión *«configure»* que configura las fuentes para tu sistema y se asegura de que el sistema está en condiciones de compilarlo. Después de configurarlo (con *«./configure»*), los programas generalmente se compilan con *«make»*. Algunos de ellos soportan *«make check»* para ejecutarse incluyendo comprobaciones automáticas. Generalmente se instalarán en sus directorios de destino ejecutando *«make install»*.

Ahora intenta compilar, y ejecutar el programa, para asegurarte de que funciona bien y de que no rompe nada mientras está instalándose o ejecutándose.

También, generalmente, puedes ejecutar *«make clean»* (o mejor *«make distclean»*) para limpiar el directorio donde se genera el programa. A veces hay incluso un *«make uninstall»* que se puede utilizar para borrar todos los archivos instalados.

3.3.3 Nombre del paquete y versión

Deberías empezar a construir tu paquete en un directorio de fuentes completamente limpio, o simplemente con las fuentes recién desempaquetadas.

Para construir correctamente el paquete, debes cambiar el nombre original del programa a letras minúsculas (si no lo está ya), y deberías renombrar el directorio de fuentes a *<nombre_de_paquete>-<versión>*.

Si el nombre del programa está formado por varias palabras, contráelas a una palabra o haz una abreviatura. Por ejemplo, el paquete del programa «el editor para X de Javi» se podría llamar javiedx o jle4x, o lo que decidas, siempre y cuando no se exceda de unos límites razonables, como 20 caracteres.

Comprueba también la versión exacta del programa (la que se incluye en la versión del paquete). Si el programa no está numerado con versiones del estilo de X.Y.Z, pero sí con fecha de publicación, eres libre de utilizar la fecha como número de versión, precedida por «0.0» (sólo por si los desarrolladores originales deciden sacar una versión nueva como 1.0). Así, si la fecha de las fuentes es el 19 de diciembre de 1998, puedes utilizar la cadena 0 0.0.19981219 (que utiliza el formato de fecha ISO 8601, N. del T.) como número de versión.

Aún así habrá algunos programas que ni siquiera estén numerados, en cuyo caso deberás contactar con el desarrollador original para ver si tienen algún otro sistema de seguimiento de revisiones.

3.3.4 «Debianización» inicial

Asegúrate que te encuentras en el directorio donde están las fuentes del programa y ejecuta lo siguiente:

```
dh_make -e tu.dirección@de.desarrollador -f ../gentoo-0.9.12.tar.gz
```

Por supuesto, cambia la cadena «tu.dirección@de.desarrollador» por tu dirección de correo electrónico para que se incluya en la entrada del fichero de cambios así como en otros ficheros, y el nombre de fichero de tu archivo fuente original.

Saldrá alguna información. Te preguntará qué tipo de paquete deseas crear. Gentoo es un sólo paquete de binarios - crea sólo un binario, y, por tanto, sólo un fichero .deb - así que seleccionaremos la primera opción, con la tecla «s». Comprueba la información que aparece en la pantalla y confirma pulsando la tecla <intro>.

Tras ejecutar dh_make, se crea una copia del código original con el nombre gentoo_0.9.12.orig.tar.gz en el directorio raíz para facilitar la creación del paquete de fuentes no nativo de Debian con el diff.gz. Observa que hay dos cambios clave en este nombre de fichero:

- El nombre del paquete y la versión están separados por «_».
- Hay un «orig.» antes de «tar.gz».

Como nuevo desarrollador, se desaconseja crear paquetes complicados, por ejemplo:

- múltiples paquetes binarios
- paquetes de bibliotecas
- paquetes en los que el formato del archivo fuente no es en tar.gz. ni en tar.bz2, o
- paquetes cuyas fuentes contienen partes que no se pueden distribuir.

Estos casos no son extremadamente difíciles, pero sí necesita algunos conocimientos más, así que aquí no se describirá el proceso de empaquetado para este tipo de paquetes.

Ten en cuenta que deberías ejecutar dh_make sólo una vez, y que no se comportará correctamente si lo haces otra vez en el mismo directorio ya «debianizado». Esto también significa que usarás un método distinto para crear una nueva revisión o una nueva versión de tu paquete en el futuro.

3.3.5 Modificar las fuentes

Por lo general, los programas se instalan a sí mismos en el subdirectorio /usr/local. Pero los paquetes Debian no pueden utilizar este directorio ya que está reservado para el uso privado del administrador (o de los usuarios). Esto significa que tienes que mirar el sistema de construcción de tu programa, generalmente empezando por el fichero *«Makefile»*. Éste es el guión *make* que se usará para automatizar la creación de este programa.

Observa que si tu programa usa GNU automake y/o autoconf, lo que quiere decir que las fuentes incluyen ficheros Makefile.am y Makefile.in, respectivamente, ya que necesitarás modificar esos ficheros, porque cada invocación de *automake* reescribirá los ficheros *«Makefile.in»* con información generada a partir de los ficheros *«Makefile.am»*, y cada llamada a *./configure* hará lo mismo con los ficheros *«Makefile»*, con información de los ficheros *«Makefile.in»*. Editar los ficheros *«Makefile.am»* requiere algunos conocimientos de *automake*, que puedes obtener leyendo la entrada de *info* para *automake*, mientras que editar los ficheros *«Makefile.in»* es casi lo mismo que editar ficheros *«Makefile»*, simplemente basta con poner atención en las variables, es decir, cualquier cadena que empiece y acabe con el caracter «@», como por ejemplo @CFLAGS@ o @LN_S@, que se sustituyen por otros valores cada vez que se ejecute *./configure*. Por favor, lee */usr/share/doc/autotools-dev/README.Debian.gz* antes de empezar.

Ten en cuenta que no hay espacio aquí para entrar en todos los detalles respecto a los arreglos que deben hacerse en las fuentes originales. Sin embargo, a continuación se detallan algunos de los problemas más frecuentes.

3.3.6 Instalación en un subdirectorio

La mayor parte de los programas tienen alguna manera de instalarse en la estructura de directorios existente en tu sistema, para que los binarios sean incluidos en tu *$PATH*, y para que encuentre la documentación y páginas de manual en los lugares habituales. Sin embargo, si lo instalas de esta forma, el programa se instalará con los demás binarios que ya están en tu sistema. Esto dificultará a las herramientas de paquetes averiguar qué archivos pertenecen a tu paquete y cuales no.

Por lo tanto, necesitas hacer algo más: instalar el programa en un subdirectorio temporal desde el cual las herramientas de desarrollo construirán el paquete *.deb* que se pueda instalar. Todo lo que se incluye en este directorio será instalado en el sistema del usuario cuando instale su paquete, la única diferencia es que *dpkg* instalará los ficheros en el directorio raíz.

Este directorio temporal se creará bajo el directorio *debian/* que está dentro del árbol del código descomprimido, generalmente con el nombre *debian/nombre_de_paquete.*

Ten en cuenta que, aunque necesitas que el programa se instale en *debian/nombre_de_paquete*, también necesitas que se comporte correctamente cuando se instale en el directorio raíz, es decir, cuando se instale desde el paquete *.deb*. Así que no deberías permitir que al construirse lo haga con cadenas como */home/canaima/deb/gentoo-0.9.12/usr/share/gentoo* dentro de los archivos del paquete a distribuir.

Esto será sencillo con los de programas que utilicen la herramienta GNU *autoconf*. La mayoría de estos programas tienen ficheros *«Makefile»* por omisión que permiten configurar la instalación en un subdirectorio cualquiera, aunque recordando que, por ejemplo, */usr* es el prefijo normal. Cuando detecte que tu programa usa *autoconf*, *dh_make* fijará las opciones necesarias para hacer esto automáticamente, así que puedes dejar de leer esta sección. Pero con otros programas puede ser necesario que examines y edites los ficheros *«Makefile»*.

Esta es la parte importante del *Makefile* de gentoo:

```
# ¿Dónde poner el binario cuando se ejecute «make install»?
BIN     = /usr/local/bin

# ¿Dónde poner los iconos cuando se ejecute «make install»?
ICONS   = /usr/local/share/gentoo/
```

Vemos que los ficheros están configurados para instalarse bajo /usr/local. Cambia estas rutas a:

```
# ¿Dónde poner el binario cuando se ejecute «make install»?
BIN     = $(DESTDIR)/usr/bin

# ¿Dónde poner los iconos cuando se ejecute «make install»?
ICONS   = $(DESTDIR)/usr/share/gentoo
```

Pero: ¿por qué en este directorio y no en otro? Porque los paquetes de Debian nunca se instalan bajo */usr/local*, este árbol de directorio, está reservado para el uso del administrador del sistema. Así que estos ficheros deben instalarse en */usr*.

La localización correcta de los binarios, iconos, documentación, etc, está especificada en el *«Estándar de la jerarquía del sistema de ficheros»* (véase */usr/share/doc/debian-policy/fhs*). Te recomiendo que leas las secciones que podrían aplicar a tu paquete.

Así pues, deberíamos instalar el binario en */usr/bin* en lugar de */usr/local/bin* y la página de manual en */usr/share/man/man1* en lugar de */usr/local/man/man1*. No hemos mencionado ninguna página de manual en el *Makefile* de gentoo, pero en Debian se requiere que cada programa debe tener una, así que haremos una más tarde y la instalaremos en /usr/share/man/man1.

Algunos programas no usan variables en el *makefile* para definir rutas como éstas. Esto significa que tendrás que editar algunos de los ficheros de código C para arreglarlos y que usen las rutas correctas. Pero, ¿dónde buscar?, y exactamente, ¿el qué? Puedes probar a encontrarlos usando:

```
grep -nr -e 'usr/local/lib' --include='*.[c|h]' .
```

(En cada subdirectorio que contenga ficheros .c y .h, grep nos indicará el nombre del fichero y la línea cuando encuentre una ocurrencia.

Ahora edita esos ficheros y cambia en esas líneas *usr/local/lib* con *usr/share* y ya está. Sólo tienes que reemplazar *usr/local/lib* por tu localización, pero debes ser muy cuidadoso para no modificar el resto del código, especialmente si no sabes mucho sobre cómo programar en C.

Después de esto deberías encontrar el objetivo *«install»* (busca una línea que comience por *«install:»*) y renombra todas las referencias a directorios distintos de los definidos al comienzo del *Makefile*. Anteriormente el objetivo *«install»* decía:

```
install:        gentoo
                install ./gentoo $(BIN)
                install icons $(ICONS)
                install gentoorc-example $(HOME)/.gentoorc
```

Después del cambio dice:

```
install:        gentoo-target
                install -d $(BIN) $(ICONS) $(DESTDIR)/etc
                install ./gentoo $(BIN)
                install -m644 icons/* $(ICONS)
                install -m644 gentoorc-example $(DESTDIR)/etc/gentoorc
```

Seguramente has notado que ahora hay una orden *install -d* antes de las demás órdenes de la regla. El *makefile* original no lo tenía porque normalmente */usr/local/bin* y otros directorios ya existen en el sistema donde se ejecuta *«make install»*. Sin embargo, dado que lo instalaremos en un directorio vacío (o incluso inexistente), tendremos que crear cada uno de estos directorios.

También podemos añadir otras cosas al final de la regla, como la instalación de documentación adicional que los desarrolladores originales a veces omiten:

```
install -d $(DESTDIR)/usr/share/doc/gentoo/html
cp -a docs/* $(DESTDIR)/usr/share/doc/gentoo/html
```

Un lector atento se dará cuenta de que he cambiado *«gentoo»* a *«gentoo-target»* en la línea *«install:»*. A eso se le llama arreglar un fallo en el programa.

Siempre que hagas cambios que no estén específicamente relacionados con el paquete Debian, asegúrate de que los envías al desarrollador original para que éste los pueda incluir en la próxima revisión del programa y así le puedan ser útiles a alguien más. Además, recuerda hacer que tus cambios no sean específicos para Debian o Linux (¡ni siquiera para Unix!) antes de enviarlos, hazlo portable. Esto hará que tus arreglos sean más fáciles de aplicar.

Ten en cuenta que no tienes que enviar ninguno de los ficheros debian/* al desarrollador original.

3.3.7 Bibliotecas diferentes

Hay otro problema común: las bibliotecas son generalmente diferentes de plataforma a plataforma. Por ejemplo, un *Makefile* puede contener una referencia a una biblioteca que no exista en Debian o ni siquiera en Linux. En este caso, se necesita cambiarla a una biblioteca que sí exista en Debian y sirva para el mismo propósito.

Así, si hay una línea en el *Makefile* (o Makefile.in) de tu programa que dice algo como lo siguiente (y tu programa no compila):

```
LIBS = -lcurses -lcosas -lmáscosas
```

Entonces cámbiala a lo siguiente, y funcionará casi con seguridad:

```
LIBS = -lncurses -lcosas -lmáscosas
```

(El autor se ha dado cuenta de que éste no es el mejor ejemplo ya que ahora el paquete libncurses incluye un enlace simbólico a libcurses.so, pero no puedo pensar uno mejor. Cualquier sugerencia sería muy bien recibida :-)

3.4 Las cosas necesarias bajo debian

Ahora hay un nuevo subdirectorio bajo el directorio principal del programa (*«gentoo-0.9.12»*), que se llama *«debian»*. Hay algunos ficheros en este directorio que debemos editar para adaptar el comportamiento del paquete. La parte más importante es modificar los ficheros *«control»*, *«rules»*, *«changelog»*, y *«copyright»* que son necesarios en todos los paquetes.

3.4.1 El fichero «control»

Este fichero contiene varios valores que *dpkg*, *dselect* y otras herramientas de gestión de paquetes usarán para gestionar el paquete.

Aquí está el fichero de control que dh_make crea para nosotros:

```
1  Source: gentoo
2  Section: unknown
3  Priority: optional
4  Maintainer: Josip Rodin <joy-mg@debian.org>
5  Build-Depends: debhelper (>> 3.0.0)
6  Standards-Version: 3.6.2
7
8  Package: gentoo
9  Architecture: any
10 Depends: ${shlibs:Depends}
11 Description: <insertar hasta 60 caracteres de descripción>
12  <inserta una descripción larga, indentada con espacios.>
```

Las líneas 1 a 6 son la información de control para el paquete fuente.

La línea 1 es el nombre del paquete fuente.

La línea 2 es la sección de la distribución dentro de la que estará este paquete.

Como puede que hayas notado, Canaima está dividida en secciones: *«estable»*, *«pruebas»*, *«desarrollo»*, etc. Bajo ellas hay subdivisiones lógicas que describen en una palabra qué paquetes hay dentro. Así que tenemos *«admin»* para programas que sólo usa un administrador, «base» para las herramientas básicas, *«devel»* para las herramientas de programación, *«doc»* para la documentación, *«libs»* para las bibliotecas, *«mail»* para lectores y demonios de correo-e, *«net»* para aplicaciones y demonios de red, *«x11»* para programas específicos de X11, y muchos más.

Vamos a cambiarla para que ponga x11. El prefijo "main/" ya va implícito, así que podemos omitirlo.

La línea 3 describe cómo de importante es para el usuario la instalación de este paquete. Podrás consultar en el manual de normas de Debian («Debian Policy») la guía de los valores que deberían tener estos campos. La prioridad «optional» suele ser lo mejor para los paquetes nuevos.

Como es un paquete de prioridad normal y no tiene conflictos con ningún otro, lo dejaremos con prioridad *«optional»*.

La línea 4 es el nombre y correo electrónico del desarrollador. Para una dirección de correo electrónico, evita usar comas, el signo «&» y paréntesis.

La línea 5 incluye la lista de paquetes requeridos para construir tu paquete. Algunos paquetes como gcc y make están implícitos, consulta el paquete build-essential para más detalles. Si se necesita algún compilador no estándar u otra herramienta para construir tu paquete, deberías añadirla en

la línea *«Build-Depends»*. Las entradas múltiples se separan con comas, lee la explicación de las dependencias binarias para averiguar más sobre la sintaxis de este campo.

También tienes los campos *«Build-Depends-Indep»* y *«Build-Conflicts»* entre otros. Estos datos los usarán los programas de construcción automática de paquetes de Debian para crear paquetes binarios para el resto de arquitecturas. Consulta las normas de Debian para más información sobre las dependencias de construcción y la Referencia del Desarrollador para más información sobre las otras arquitecturas y sobre cómo migrar los programas a ellas.

Aquí tienes un truco que puedes usar para averiguar qué paquetes necesitará tu paquete en su construcción:

```
strace -f -o /tmp/log ./configure
# o make en lugar de ./configure, si el paquete no usa autoconf
for x in `dpkg -S $(grep open /tmp/log|\
                    perl -pe 's!.* open\(\"([^\"]*).*!$1!' |\
                    grep "^/"| sort | uniq|\
                    grep -v "^\(/tmp\|/dev\|/proc\)" ) 2>/dev/null|\
                    cut -f1 -d":"| sort | uniq`; \
      do \
        echo -n "$x (>=" `dpkg -s $x|grep ^Version|cut -f2 -d":"` "), "; \
      done
```

Para encontrar manualmente las dependencias exactas de /usr/bin/foo, ejecuta:

```
objdump -p /usr/bin/foo | grep NEEDED
```

y para cada biblioteca, por ejemplo, libfoo.so.6, ejecuta:

```
dpkg -S libfoo.so.6
```

Debes utilizar la versión *«-dev»* de cada uno de los paquetes dentro de la entrada *«Build-deps»*. Si usas *ldd* para este propósito, también te informará de las dependencias de bibliotecas indirectas, lo que puede llevar a que se introduzcan demasiadas dependencias de construcción.

La aplicación *"gentoo"* también requiere *xlibs-dev*, *libgtk1.2-dev* y *libglib1.2-dev* para su construcción, así que lo añadiremos junto a *debhelper*.

La línea 6 es la versión de los estándares definidos en las normas que sigue este paquete, es decir, la versión del manual de normas que has leído mientras haces tu paquete.

La línea 8 es el nombre del paquete binario. Este suele ser el mismo que el del paquete fuente, pero no tiene que ser necesariamente así siempre.

La línea 9 describe la arquitectura de CPU para la que el paquete binario puede ser compilado. Dejaremos puesto *«any»*, porque dpkg-gencontrol(1) la rellenará con el valor apropiado cuando se compile este paquete en cualquier arquitectura para la cual pueda ser compilado.

Si tu paquete es independiente de la arquitectura (por ejemplo, un documento, un guión escrito en Perl o para el intérprete de órdenes), cambia esto a *«all»*, y consulta más adelante El fichero *«rules»*, Sección 4.4 sobre cómo usar la regla *«binary-indep»* en lugar de *«binary-arch»* para construir el paquete.

La línea 10 muestra una de las más poderosas posibilidades del sistema de paquetes de Debian. Los paquetes se pueden relacionar unos con otros de diversas formas. Aparte de «Depends:» (depende) otros campos de relación son «Recommends:» (recomienda), «Suggests:» (sugiere), «Pre-Depends:» (predepende de), «Conflicts:» (entra en conflicto con), *«Provides:»* (provee), «Replaces:» (reemplaza a).

Las herramientas de gestión de paquetes se comportan habitualmente de la misma forma cuando tratan con esas relaciones entre paquetes; si no es así, se explicará en cada caso.

A continuación se detalla el significado de las dependencias:

- *Depends:*

No se instalará el programa a menos que los paquetes de los que depende estén ya instalados. Usa esto si tu programa no funcionará de ninguna forma (o se romperá fácilmente) a no ser que se haya instalado un paquete determinado.

- *Recommends:*

Programas como dselect o aptitude informarán en la instalación de los paquetes recomendados por tu paquete, dselect incluso insistirá. dpkg y apt-get ignorarán este campo. Usa esto para paquetes que no son estrictamente necesarios pero que se usan habitualmente con tu programa.

- *Suggests:*

Cuando un usuario instale el paquete, todos los programas le informarán de que puede instalar los paquetes sugeridos. Salvo dpkg y apt, que ignorarán estas dependencias. Utiliza esto para paquetes que funcionarán bien con tu programa pero que no son necesarios en absoluto.

- *Pre-Depends:*

Esto es más fuerte que «Depends». El paquete no se instalará a menos que los paquetes de los que pre-dependa esté instalados y correctamente configurados. Utiliza esto muy poco y sólo después de haberlo discutido en la lista de *desarrolladores*. En resumidas cuentas: no lo utilices en absoluto :-)

- *Conflicts:*

El paquete no se instalará hasta que todos los paquetes con los que entra en conflicto hayan sido eliminados. Utiliza esto si tu programa no funcionará en absoluto (o fallará fácilmente) si un paquete en concreto está instalado.

- *Provides:*

Se han definido nombres virtuales para algunos tipos determinados de paquetes que ofrecen múltiples alternativas para la misma función. Puedes obtener la lista completa en el fichero

/usr/share/doc/debian-policy/virtual-package-names-list.text.gz. Usa esto si tu programa ofrece las funciones de un paquete virtual que ya exista.

- *Replaces:*

Usa esto si tu programa reemplaza ficheros de otro paquete o reemplaza totalmente otro paquete (generalmente se usa conjuntamente con «Conflicts:»). Se eliminarán los ficheros de los paquetes indicados antes de instalar el tuyo.

Todos estos campos tienen una sintaxis uniforme se trata de una lista de nombres de paquetes separados por comas. Estos nombres de paquetes también puede ser listas de paquetes alternativos, separados por los símbolos de barra vertical | (símbolos tubería).

Los campos pueden restringir su aplicación a versiones determinadas de cada paquete nombrado. Es <<, <=, =, >= y >> para estrictamente anterior, anterior o igual, exactamente igual, posterior o igual o estrictamente posterior, respectivamente. Por ejemplo:

```
:Depends: foo (>= 1.2), libbar1 (= 1.3.4)
:Conflicts: baz
:Recommends: libbaz4 (>> 4.0.7)
:Suggests: quux
:Replaces: quux (<< 5), quux-foo (<= 7.6)
```

La última funcionalidad que necesitas conocer es $(shlibs:Depends). Después de que tu paquete se compile y se instale en el directorio temporal, dh_shlibdeps(1) lo escaneará en busca de binarios y bibliotecas para determinar las dependencias de bibliotecas compartidas y en qué paquetes están, tales como como libc6 o xlib6g. Luego pasará la lista a dh_gencontrol(1) que rellenará estas dependencias en el lugar adecuado. De esta forma no tendrás que preocuparte por esto.

Después de decir todo esto, podemos dejar la línea de «Depends:» exactamente como está ahora e insertar otra línea tras ésta que diga Suggests: file, porque gentoo utiliza algunas funciones de este paquete/programa.

La línea 11 es una descripción corta. La mayor parte de los monitores de la gente son de 80 columnas de ancho, así que no debería tener más de 60 caracteres. Cambiaré esto a «fully GUI configurable GTK+ file manager» («Gestor de ficheros GTK+ completamente configurable por GUI»).

La línea 12 es donde va la descripción larga del paquete. Debería ser al menos un párrafo que dé más detalles del paquete. La primera columna de cada línea debería estar vacía. No puede haber líneas en blanco, pero puede poner un . (punto) en una columna para simularlo. Tampoco debe haber más de una línea en blanco después de la descripción completa.

Aquí está el fichero de control actualizado:

```
1  Source: gentoo
2  Section: x11
3  Priority: optional
```

```
4  Maintainer: Josip Rodin <joy-mg@debian.org>
5  Build-Depends: debhelper (>> 3.0.0), xlibs-dev, libgtk1.2-dev, libglib1.2-dev
6  Standards-Version: 3.5.2
7
8  Package: gentoo
9  Architecture: any
10 Depends: ${shlibs:Depends}
11 Suggests: file
12 Description: fully GUI configurable X file manager using GTK+
13  gentoo is a file manager for Linux written from scratch in pure C. It
14  uses the GTK+ toolkit for all of its interface needs. gentoo provides
15  100% GUI configurability; no need to edit config files by hand and re-
16  start the program. gentoo supports identifying the type of various
17  files (using extension, regular expressions, or the «file» command),
18  and can display files of different types with different colors and icons.
19  .
20  gentoo borrows some of its look and feel from the classic Amiga file
21  manager "Directory OPUS" (written by Jonathan Potter).
```

3.4.2 El fichero «copyright»

Este fichero contiene la información sobre la licencia y copyright de las fuentes originales del paquete. El formato no está definido en las normas, pero sí en sus contenidos (sección 12.6 «Copyright information»).

dh_make crea por omisión un fichero como este:

```
1  This package was debianized by Josip Rodin <joy-mg@debian.org> on
2  Wed, 11 Nov 1998 21:02:14 +0100.
3
4  It was downloaded from <rellena con el sitio ftp site>
5
6  Upstream Author(s): <pon el nombre del autor y dirección de correo>
7
8  Copyright:
9
10 <Debe incluirse aquí>
```

Las cosas importantes que se deben añadir a este fichero son el lugar de donde obtuviste el paquete junto con la nota de copyright y licencia originales. Debes incluir la licencia completa, a menos que sea una licencia común en el mundo del software libre como GNU GPL o LGPL, BSD o la «Licencia artística», donde basta referirse al fichero apropiado en el directorio /usr/share/common-licenses/ que existe en todo sistema Debian.

La aplicación *gentoo* está publicado bajo la Licencia Pública General GNU, así que cambiaremos el fichero a esto:

```
1  This package was debianized by Josip Rodin <joy-mg@debian.org> on
2  Wed, 11 Nov 1998 21:02:14 +0100.
3
4  It was downloaded from: ftp://ftp.obsession.se/gentoo/
5
6  Upstream author: Emil Brink <emil@obsession.se>
7
8  This software is copyright (c) 1998-99 by Emil Brink, Obsession
9  Development.
10
11 You are free to distribute this software under the terms of
12 the GNU General Public License  either version 2 of the License,
13 or (at your option) any later version.
14 On Debian systems, the complete text of the GNU General Public
15 License can be found in the file `/usr/share/common-licenses/GPL-2'.
```

3.4.3 El fichero «changelog»

Este es un fichero requerido, que tiene un formato especial descrito en las normas, sección 4.4 “debian/changelog”. Este es el formato que usan dpkg y otros programas para obtener el número de versión, revisión, distribución y urgencia de tu paquete.

Para ti es también importante, ya que es bueno tener documentados todos los cambios que hayas hecho. Esto ayudará a las personas que se descarguen tu paquete para ver si hay temas pendientes en el paquete que deberían conocer de forma inmediata. Se guardará como «/usr/share/doc/gentoo/changelog.Debian.gz» en el paquete binario.

dh_make crea uno por omisión, el cual es como sigue:

```
1  gentoo (0.9.12-1) unstable; urgency=low
2
3   * Initial Release.
4
5  -- Carlos Guerrero <cguerrero@cnti.gob.ve> Wed, 11 Nov 2009 21:02:14 +0100
6
```

La línea 1 es el nombre del paquete, versión, distribución y urgencia. El nombre debe coincidir con el nombre del paquete fuente, la distribución debería ser, por ahora, «pruebas» (o incluso «desarrollo») y la urgencia no debería cambiarse a algo mayor que «low». :-)

Las línea 3-5 son una entrada de registro, donde se documentan los cambios hechos en esta revisión del paquete (no los cambios en las fuentes originales - hay un fichero especial para este propósito, creado por los autores originales y que instalarás luego como */usr/share/doc/gentoo/changelog.gz*). Las nuevas líneas deben insertarse justo antes de la línea que hay más arriba que comienza por un asterisco («*»). Puede hacerlo con dch(1), o manualmente con cualquier editor de texto.

Terminarás con algo así:

```
1  gentoo (0.9.12-1) unstable; urgency=low
2
3   * Version inicial del paquete.
4   * Este es mi primer paquete.
5   * Sin modificaciones adicionales del archivo fuente.
6
7  -- Carlos Guerrero <cguerrero@cnti.gob.ve> Wed, 11 Nov 2009 21:02:14 +0100
8
```

Puedes leer más sobre cómo actualizar el fichero changelog más adelante en Actualizar el paquete, Capítulo 9.

3.4.4 El fichero «rules»

Ahora necesitamos mirar las reglas exactas que dpkg-buildpackage utilizará para crear el paquete. Este fichero es en realidad otro Makefile, pero diferente al que viene en las fuentes originales. A diferencia de otros ficheros en debian/, éste necesita ser un fichero ejecutable.

Cada fichero *«rules»*, como muchos otros *Makefiles*, se compone de varias reglas que especifican cómo tratar las fuentes. Cada regla se compone de objetivos, ficheros o nombres de acciones que se deben llevar a cabo (por ejemplo, *«build:»* o *«install:»*). Las reglas que quieras ejecutar deberían llamarse como argumentos de la línea de órdenes (por ejemplo, *«./debian/rules build»* o *«make -f rules install»*). Después del nombre del objetivo, puedes nombrar las dependencias, programas o ficheros de los que la regla dependa. Después de esto, hay un número cualquiera de instrucciones (¡indentado con <tab>!), hasta que se llega a una línea en blanco. Ahí empieza otra regla. Las líneas múltiples en blanco, y las líneas que empiezan por almohadillas («#») se tratan como comentarios y se ignoran.

Probablemente ya te hayas perdido, pero todo quedará más claro después de ver el fichero «rules» que *dh_make* pone por omisión. Deberías leer también la entrada de *«make»* en *info* para más información.

La parte importante que debes conocer sobre el fichero de reglas creado por *dh_make*, es que sólo es una sugerencia. Funcionará para paquetes simples pero para más complicados, no te asustes y añade o quita cosas de éste para ajustarlo a tus necesidades. Una cosa que no debes cambiar son los nombres de las reglas, porque todas las herramientas utilizan estos nombres, como se describe en las normas.

Éste es, más o menos, el contenido del fichero debian/rules que *dh_make* genera por omisión:

```
1  #!/usr/bin/make -f
2  # -*- makefile -*-
3  # Sample debian/rules that uses debhelper.
4  # This file was originally written by Joey Hess and Craig Small.
```

```
 5  # As a special exception, when this file is copied by dh-make into a
 6  # dh-make output file, you may use that output file without restriction.
 7  # This special exception was added by Craig Small in version 0.37 of dh-make.
 8  # Uncomment this to turn on verbose mode.
 9  #export DH_VERBOSE=1
10  configure: configure-stamp
11  configure-stamp:
12          dh_testdir
13          # Add here commands to configure the package.
14          touch configure-stamp
15  build: build-stamp
16  build-stamp: configure-stamp
17          dh_testdir
18          # Add here commands to compile the package.
19          $(MAKE)
20          #docbook-to-man debian/testpack.sgml > testpack.1
21          touch $@
22  clean:
23          dh_testdir
24          dh_testroot
25          rm -f build-stamp configure-stamp
26          # Add here commands to clean up after the build process.
27          $(MAKE) clean
28          dh_clean
29  install: build
30          dh_testdir
31          dh_testroot
32          dh_clean -k
33          dh_installdirs
34          # Add here commands to install the package into debian/testpack.
35          $(MAKE) DESTDIR=$(CURDIR)/debian/testpack install
36  # Build architecture-independent files here.
37  binary-indep: build install
38  # We have nothing to do by default.
39  # Build architecture-dependent files here.
40  binary-arch: build install
41          dh_testdir
42          dh_testroot
43          dh_installchangelogs
44          dh_installdocs
45          dh_installexamples
46  #       dh_install
47  #       dh_installmenu
48  #       dh_installdebconf
49  #       dh_installlogrotate
50  #       dh_installemacsen
51  #       dh_installpam
```

```
52  #       dh_installmime
53  #       dh_python
54  #       dh_installinit
55  #       dh_installcron
56  #       dh_installinfo
57          dh_installman
58          dh_link
59          dh_strip
60          dh_compress
61          dh_fixperms
62  #       dh_perl
63  #       dh_makeshlibs
64          dh_installdeb
65          dh_shlibdeps
66          dh_gencontrol
67          dh_md5sums
68          dh_builddeb
69  binary: binary-indep binary-arch
70  .PHONY: build clean binary-indep binary-arch binary install configure
```

Probablemente estés familiarizado con líneas como la primera de guiones escritos en shell o Perl. Esta línea indica que el fichero debe ejecutarse con */usr/bin/make*.

El significado de las variables DH_* que se mencionan en las líneas 8 y 9 debería ser evidente de la descripción corta. Para más información sobre DH_COMPAT consulte la sección «Debhelper compatibility levels» del manual de debhelper(1).

Las líneas de la 11 a la 16 son el esqueleto de apoyo para los parámetros de DEB_BUILD_OPTIONS. Basicamente, estas cosas controlan si los binarios se construyen con los símbolos del depurador y si deberían eliminarse tras la instalación. De nuevo, es sólo un esqueleto, una pista de lo que deberías hacer. Deberías comprobar cómo el sistema de construcción de las fuentes maneja la inclusión de los símbolos del depurador y su eliminación en la instalación e implementarlo por ti mismo.

Habitualmente puedes decirle a gcc que compile con "-g" usando la variable CFLAGS. Si este es el caso de tu paquete, pon la variable añadiendo CFLAGS="$(CFLAGS)" a la invocación de $(MAKE) en la regla de construcción (ver más abajo). Alternativamente, si tu paquete usa un guión de configuración de autoconf puedes definir la cadena arriba mostrada anteponiéndola a la llamada a ./configure en la regla de construcción.

Los programas a los que se le quitan los símbolos del depurador con *strip* se configuran normalmente para instalarse sin pasar por *strip*, y a menudo sin una opción para cambiar esto. Afortunadamente, tienes *dh_strip* que detectará cuando la bandera DEB_BUILD_OPTIONS=nostrip está activada y finalizará silenciosamente.

Las líneas 18 a la 26 describen la regla *build* (y su hija *«build-stamp»*), que ejecuta *make* con el propio *Makefile* de la aplicación para compilar el programa. Si el programa utiliza las utilidades de configuración de GNU para construir los binarios, por favor, asegúrate de

leer */usr/share/doc/autotools-dev/README.Debian.gz*. Hablaremos sobre el ejemplo comentado docbook-to-man más adelante en *manpage.1.ex*, *manpage.sgml.es*, Sección 5.8.

La regla «clean», como se especifica en las líneas 28 a la 36, limpia cualquier binario innecesario o cosas generadas automáticamente, dejadas después de la construcción del paquete. Esta regla debe funcionar en todo momento (¡incluso cuando el árbol de fuentes esté limpio!), así que, por favor, usa las opciones que fuercen a hacer cosas (por ejemplo para rm, sería «-f»), o ignora los valores devueltos (con un «-» al principio de la orden).

El proceso de instalación, la regla «install», comienza en la línea 38. Básicamente ejecuta la regla «install» del Makefile del programa, pero lo instala en el directorio $(CURDIR)/debian/gentoo. Esta es la razón por la que especificamos $(DESTDIR) como el directorio raíz de instalación del Makefile de gentoo.

Como sugiere el comentario, la regla *«binary-indep»*, en la línea 48, se usa para construir paquetes independientes de arquitectura. Como no tenemos ninguno, aquí no se hará nada.

Lo siguiente es la regla *«binary-arch»*, en las líneas 52 a 79, en la que ejecutamos varias pequeñas utilidades del paquete debhelper que nos permiten hacer diversas operaciones en nuestro paquete para que cumpla las normas de Debian.

Si tu paquete es del tipo *«Architecture: all»* necesitarás incluir todas las órdenes para crear el paquete bajo esta regla, y dejar la siguiente regla (*«binary-arch»*) vacía en su lugar.

Los nombres comienzan con *dh_* y el resto del nombre es la descripción de lo que la utilidad en particular realmente hace. Es todo más o menos auto-explicativo, pero a continuación tienes algunos añadidos a las explicaciones:

- *dh_testdir* comprueba que estás en el directorio correcto (esto es, el directorio raíz de la distribución de las fuentes),
- *dh_testroot* comprueba que tienes permisos de superusuario que son necesarios para las reglas *«binary-arch»*, *«binary-indep»* and *«clean»*,
- *dh_installman* copiará todas las páginas de manual que encuentre en el paquete fuente en el paquete, sólo has de indicarle donde están de forma relativa, desde el nivel más alto del directorio de codigo.
- *dh_strip* elimina las cabeceras de depuración de los ficheros ejecutables para hacerlos más pequeños,
- *dh_compress* comprime las páginas de manual y los ficheros de documentación que sean más grandes de 4 kB con gzip(1),
- *dh_installdeb* copia los ficheros relativos al paquete (es decir, los guiones del desarrollador que mantiene el paquete) bajo el directorio *debian/gentoo/DEBIAN*,
- *dh_shlibdeps* calcula las dependencias de los ejecutables y bibliotecas con las bibliotecas compartidas,
- *dh_gencontrol* genera e instala el fichero de control en *debian/gentoo/DEBIAN*,

- *dh_md5sums* genera las sumas de comprobación MD5 para todos los ficheros del paquete.

:Para información más completa de lo que hacen cada uno de estos guiones *dh_** , y qué otras opciones tienen, por favor lee sus páginas de manual respectivas. Hay otros guiones con la misma nomenclatura (*dh_**) que no se han mencionado aquí, pero pueden serte útiles. Si los necesitas, lee la documentación de *debhelper*.

La sección binary-arch es en una de las que deberías comentar o eliminar las líneas que llamen a funciones que no necesites. Para gentoo, comentaré de ejemplos, cron, init, man e info, simplemente porque gentoo no las necesita. Tan sólo, en la línea 68, reemplazaré «ChangeLog» con «FIXES», porque este es el nombre del fichero de cambios de las fuentes.

Las últimas dos líneas (junto con otras que no se explican) son cosas más o menos necesarias, sobre las que puedes leer en el manual de *make*, y las normas. Por ahora no es importante que sepas nada de ellas.

3.5 Otros ficheros en el directorio debian

Verás que existen otros ficheros en el subdirectorio debian/, muchas de los cuales tendrán el sufijo «.ex», que indica que son ejemplos. Echale un vistazo a todos. Si lo deseas o necesitas usar alguna de estas características:

- revisa todo la documentación relacionada,
- si es necesario, modifica los ficheros para ajustarlos a tus necesidades,
- renómbralos para eliminar el sufijo «ex.», si lo tiene,
- renómbralos para eliminar el prefijo «.ex», si lo tiene,
- modifica el fichero «rules» si fuera necesario.

Algunos de los ficheros que se usan habitualmente se detallan en las secciones que siguen.

3.5.1 README.debian (LÉEME.debian)

Cualquier detalle extra o discrepancias entre el programa original y su versión debianizada debería documentarse aquí.

dh_make crea una por omisión, y éste es su aspecto:

> gentoo for Debian
>
> <possible notes regarding this package - if none, delete this file>
>
> > – Carlos Guerrero <cguerrero@cnti.gob.ve>, Wed, 11 May 2009 21:02:14 +0100

Dado que no tenemos que poner nada aquí - está permitido borrarlo.

3.5.2 conffiles

Una de las cosas más molestas de los programas es cuando pasas mucho tiempo y esfuerzo adaptando un programa y una actualización destroza todos tus cambios. Debian resuelve este problema marcando los ficheros de configuración de forma que cuando actualizas un paquete se te pregunta si deseas mantener la nueva configuración o no.

Eso se consigue poniendo la ruta completa a cada fichero de configuración (se encuentran generalmente en */etc*), una por línea, en un fichero llamado *«conffiles»* (abreviatura de ficheros de configuración). Gentoo tiene un fichero de configuración, */etc/gentoorc*, y meteremos éste en el fichero *conffiles*.

En el caso de que tu programa utilice ficheros de configuración pero también los reescriba él mismo es mejor no marcarlos como «conffiles». Si lo haces, dpkg informará a los usuarios que verifiquen los cambios de estos ficheros cada vez que lo actualicen.

También deberías considerar no marcar el fichero como un conffile si el programa que estás empaquetando requiere que cada usuario modifique su fichero de configuración para poder trabajar.

Puedes tomar ejemplos de ficheros de configuración de los guiones ya existentes de desarrolladores, para más información consulta postinst.ex, preinst.ex, postrm.ex y prerm.ex.

Puedes eliminar el fichero conffiles del directorio debian/ si tu programa no tiene «conffiles».

3.5.3 cron.d.ex

Si tu paquete requiere tareas periódicas para funcionar adecuadamente, puedes usar este fichero como patrón.

Ten en cuenta que ésto no incluye la rotación de archivos de registro, para hacer eso consulta *dh_installlogrotate* y *logrotate*.

Elimina el fichero si el paquete no utiliza dichas tareas.

3.5.4 dirs

Este fichero especifica los directorios que se necesitan pero que por alguna razón no se crean en un proceso de instalación normal (*«make install»*).

Por omisión, tiene este aspecto:

```
1 usr/bin
2 usr/sbin
```

Observa que la barra precedente no está incluida. Normalmente lo cambiaríamos a algo así:

```
1 usr/bin
2 usr/man/man1
```

pero estos directorios ya se crean en el Makefile, así que no necesitaremos este fichero y lo podremos borrar.

3.5.5 docs

Este fichero especifica los nombres de los ficheros de documentación que *dh_installdocs* instalará en el directorio temporal.

Por omisión, se incluirán todos los ficheros existentes en los directorios de más alto nivel del código que se llamen «BUGS», «README*», «TODO» etc.

También incluiré algunos otros para gentoo:

```
BUGS
CONFIG-CHANGES
CREDITS
ONEWS
README
README.gtkrc
TODO
```

También podemos eliminar este fichero y en su lugar listar estos ficheros en la línea de órdenes de *dh_installdocs* en el fichero rules, de esta forma:

```
dh_installdocs BUGS CONFIG-CHANGES CREDITS ONEWS README \
               README.gtkrc TODO
```

Es posible que no tengas ninguno de estos ficheros en las fuentes de tu paquete. Puedes eliminar este fichero si este es tú caso. Pero no elimines la llamada a *dh_installdocs* desde el fichero rules porque también se usa para instalar el fichero copyright entre otras cosas.

3.5.6 emacsen-*.ex

Si tu paquete proporciona ficheros Emacs que pueden ser compilados a bytes en el momento de la instalación, puede usar estos ficheros.

dh_installemacsen los instala en el directorio temporal, así que no olvides descomentar esta línea en el fichero rules si los usas.

Elimínalos si no los necesitas.

3.5.7 init.d.ex

Si tu paquete es un demonio que necesita ejecutarse en el arranque del sistema, obviamente has desatendido mi recomendación inicial, ¿o no? :-)

Este fichero es prácticamente un esqueleto genérico para un fichero de guiones en */etc/init.d/*, así que probablemente tendrás que editarlo y mucho. *dh_installinit* lo instalará en el directorio temporal.

Elimina el fichero si no lo necesitas.

3.5.8 manpage.1.ex, manpage.sgml.es

El programa debería tener una página de manual. Cada uno de estos ficheros es una plantilla que puedes rellenar en el caso de que no tengas una.

Las páginas de manual se escriben normalmente con *nroff*. El ejemplo *manpage.1.ex* está también escrito con *nroff*. Consulta la página de manual *man* para una breve descripción de como editar el fichero.

Por otro lado, puede que prefieras escribir usando *SGML* en lugar de *nroff*. En este caso, puedes usar la plantilla *manpage.sgml.ex*. Si haces esto, tendrás que:

- instalar el paquete docbook-to-man
- añadir docbook-to-man a la línea de Build-Depends en el fichero de control
- eliminar el comentario de la llamada a docbook-to-man en la regla «build» de tu fichero rules

¡Y recuerda renombrar el fichero a algo como gentoo.sgml!

La página final del nombre debería incluir el nombre del programa que está documentando, asi que lo renombraremos de “manpage” a “gentoo”. El nombre del fichero incluye también “.1” como primer sufijo, lo que significa que es una página de manual para una programa de usuario. Asegurate de verificar que esa sección es la correcta. Aquí tienes una pequeña lista de las secciones de las páginas de manual:

```
Sección |     Descripción        |     Notas
   1     Ordenes de Usuario       Programas o guiones ejecutables.
   2     Llamadas al Sistema      Funciones que ofrece el núcleo.
   3     Llamadas a Bibliotecas   Funciones dadas por las bibliotecas del sistema.
   4     Ficheros Especiales      Generalmente se encuentran en /dev.
   5     Formatos de Fichero      Por ejemplo, el formato del /etc/passwd.
   6     Juegos                   U otros programas frívolos.
   7     Paquetes de Macros       Como las macros de man.
   8     Administración del Sist. Programas que sólo suele ejecutar el superusuari
   9     Rutinas del Núcleo       Llamadas al sistema no estándar.
```

Así que la página de manual de gentoo debería llamarse gentoo.1. No había una página de manual gentoo.1 en el paquete fuente asi que la escribí usando la información del ejemplo y de los documentos del programador original.

3.5.9 menu.ex

Los usuarios de X Windows suelen tener un gestor de ventanas con menús que pueden adaptarse para lanzar programas. Si tienen instalado el paquete menu de Canaima, se creará un conjunto de menús para cada programa del sistema para ellos.

Éste es el fichero *menu.ex* que *dh_make* crea por omisión:

```
?package(gentoo):needs="X11|text|vc|wm" section="Apps/lea-manual-menu"\
  title="gentoo" command="/usr/bin/gentoo"
```

El primer campo tras la coma (*«needs»*) son las necesidades, y especifica qué tipo de interfaz necesita el programa. Cambia ésta a una de las alternativas que se listan, como por ejemplo *«text»* o *«X11»*.

Lo siguiente (*«section»*) es la sección donde deberían aparecer la entrada del menú y del submenú. La lista actual de secciones está en: */usr/share/doc/debian-policy/menu-policy.html/ch2.html#s2.1*

El campo *«title»* es el nombre del programa. Puedes comenzar este en mayúsculas si lo quieres, pero hazlo lo más corto que puedas.

Finalmente, el campo *«command»* es la orden que ejecuta el programa.

Ahora cambiaremos la entrada del menú por ésta:

```
?package(gentoo): needs="X11" section="Apps/Tools" title="Gentoo" command="gentoo"
```

También puedes añadir otros campos como son *«longtitle»* (título largo), *«icon»* (icono), *«hints»* (pistas), etc. Para más información consulta *menufile*, *update-menus* y */usr/share/doc/debian-policy/menu-policy.html/*.

3.5.10 watch.ex

Este fichero se usa para configurar los programas *uscan* y *uupdate* (en el paquete devscripts), que se usan para vigilar el servidor de donde obtuviste las fuentes originales.

Esto es lo que he puesto yo:

```
# watch control file for uscan
# Site             Directory       Pattern                  Version Script
ftp.obsession.se   /gentoo         gentoo-(.*)\.tar\.gz     debian  uupdate
```

Pista: conéctate a Internet, e intenta ejecutar el programa *«uscan»* en el directorio donde has creado el fichero. Consulta la página de manual para más detalles.

3.5.11 ex.package.doc-base

Si tu paquete tiene documentación aparte de las páginas de manual y documentos *«info»*, deberías usar el fichero *«doc-base»* para registrarla, así el usuario puede encontrarlos con *dhelp*, *dwww* o *doccentral*.

Esto incluye generalmente ficheros *HTML*, *PS* y *PDF* que se instalen en */usr/share/doc/nombre_de_paquete/*.

Así es como el fichero *doc-base* de gentoo gentoo.doc-base debe ser:

```
Document: gentoo
Title: Gentoo Manual
Author: Emil Brink
Abstract: This manual describes what Gentoo is, and how it can be used.
Section: Apps/Tools

Format: HTML
Index: /usr/share/doc/gentoo/html/index.html
Files: /usr/share/doc/gentoo/html/*.html
```

Para información sobre el formato del fichero revisa install-docs(8) y el manual de doc-base en /usr/share/doc/doc-base/doc-base.html/index.html.

3.5.12 postinst.ex, preinst.ex, postrm.ex y prerm.ex

Estos ficheros se llaman *guiones del desarrollador* o *«maintainer scripts»*, y son guiones que se colocan en el área de control del paquete y que *dpkg* ejecuta cuando tu paquete se instala, se actualiza o se elimina.

Por ahora, deberías intentar evitar editar manualmente estos guiones si puedes porque suelen hacerse muy complejos. Es recomendable echar un vistazo a los ejemplos dados por *dh_make*.

3.6 Construir el paquete

Deberíamos estar preparados para construir el paquete.

3.6.1 Reconstrucción completa

Entra en el directorio principal del programa y ejecuta la siguiente orden:

```
dpkg-buildpackage -rfakeroot
```

Esto lo hará todo por tí:

- limpia el árbol del código (debian/rules clean), usando fakeroot
- construye el paquete de código (dpkg-source -b)
- construye el programa (debian/rules build)
- construye el paquete binario (debian/rules binary), usando fakeroot
- firma el fichero fuente .dsc, usando gnupg
- crea y firma el fichero de subida .changes, usando dpkg-genchanges y gnupg

Lo único que se te pedirá es que escribas tu contraseña secreta de la clave GPG, dos veces.

Después de hacer todo esto, verás las siguientes líneas en el directorio encima del que está (~/gentoo/):

gentoo_0.9.12.orig.tar.gz

:Este es el código fuente original comprimido, simplemente se ha renombrado para seguir los estándares de Debian. Nótese que ha sido creado usando la opción «-f» de *dh_make* cuando lo ejecutamos en el inicio.

gentoo_0.9.12-1.dsc

:Este es un sumario de los contenidos del código fuente. Este fichero se genera a partir del fichero de *«control»* y se usa cuando se descomprimen las fuentes con *dpkg-source*. Este fichero está firmado con GPG de forma que cualquiera pueda estar seguro de que es realmente suyo.

gentoo_0.9.12-1.diff.gz

Este fichero comprimido contiene todos y cada uno de los cambios que hizo al código fuente original,
si no renombras el archivo comprimido original nombre_de_paquete_versión.orig.tar.gz ¡*dpkg-source* fallará al generar el fichero .diff.gz!

Si alguien quiere volver a crear tu paquete desde cero, puede hacerlo fácilmente usando los tres fiche
sólo se debe copiar los tres ficheros en algún lado y ejecutar dpkg-source -x gentoo_0.9.12-1.dsc.

gentoo_0.9.12-1_i386.deb

:Este es el paquete binario completo. Puedes usar *dpkg* para instalar o eliminar tanto este paquete como cualquier otro.

gentoo_0.9.12-1_i386.changes

:Este fichero describe todos los cambios hechos en la revisión actual del paquete, y se usa por los programas de gestión del archivo FTP para instalar los paquetes binarios y fuentes en él. Se genera

parcialmente a partir del fichero *«changelog»* y el fichero *«.dsc»*. Este fichero está firmado con GPG, de forma que cualquiera puede estar aún más seguro de que es realmente tuyo.

:Mientras sigues trabajando en el paquete, éste cambiará su comportamiento y se le añadirán nuevas funciones. Las personas que descarguen tu paquete pueden leer este fichero y ver qué ha cambiado. Los programas de mantenimiento del archivo de Canaima, también enviarán el contenido de este fichero a la lista de correo.

:Las largas listas de números en los ficheros .dsc y .changes son las sumas MD5 para los ficheros. Las personas que descarguen estos ficheros pueden comprobarlos con *md5sum* y si los números no coinciden, sabrán que el fichero está corrupto o ha sido modificado.

3.6.2 Reconstrucción rápida

Con un paquete grande, puede que no quieras recompilar desde cero cada vez que tocas un detalle en el fichero debian/rules. Para propósitos de prueba, puedes hacer un fichero .deb sin necesidad de recompilar las fuentes originales de esta forma:

```
fakeroot debian/rules binary
```

Una vez que has terminado la puesta a punto, recuerda reconstruir el paquete siguiendo el procedimiento adecuado que está arriba. Puede que no seas capaz de enviar correctamente el paquete si intentas enviar los archivos .deb construidos de esta forma.

3.6.3 La orden debuild

Puedes automatizar aún más el proceso de construcción de paquetes con la orden *debuild*.

La personalización de la orden *debuild* puede hacerse a través de */etc/devscripts.conf* o *~/.devscripts*. Te sugiero al menos los siguientes valores:

```
DEBSIGN_KEYID="Tu_ID_clave_GPG"
DEBUILD_DPKG_BUILDPACKAGE_OPTS="-i -ICVS -I.svn"
```

Con estos valores, puedes construir paquetes siempre con tu clave GPG y evitar incluir componentes no deseados. (Esto también es bueno para patrocinar). Por ejemplo, limpiar el código y reconstruir el paquete desde una cuenta de usuario es tan simple como:

```
debuild clean
debuild
```

3.6.4 Los sistemas dpatch y quilt

El uso de las órdenes *dh_make* y *dpkg-buildpackage* creará un gran fichero *diff.gz* que contendrá los archivos de mantenimiento del paquete en *debian/* así como los parches de los ficheros fuente. Este tipo de paquetes es un poco engorroso de inspeccionar y entender para cada una de las modificaciones de código posteriores.

Se han propuesto y se utilizan distintos métodos para la gestión de conjuntos de parches en Debian. Los sistemas *dpatch* y *quilt* son los dos más simples de todos los propuestos. Otros son *dbs*, *cdbs*, etc.

Un paquete que haya sido empaquetado correctamente con el sistema *dpatch* o *quilt* tiene las modificaciones al código fuente claramente documentadas como un conjunto de ficheros parche de tipo *«-p1»* en *debian/patches/* y el árbol de código permanece más allá del directorio *debian/*. Si estás pidiendo a un patrocinador que suba tu paquete, este clara separación y documentación de los cambios son muy importantes para acelerar la revisión del paquete por parte del patrocinador. El modo de empleo de *dpatch* y *quilt* se explica en *dpatch*, *dpatch-edit-patch* y *quilt*. Ambos programas ofrecen ficheros que se pueden incluir en *debian/rules*: */usr/share/dpatch/dpatch.make* y */usr/share/quilt/quilt.make*.

Cuando alguien (incluyéndote a ti) proporciona un parche para las fuentes, modificar el paquete con es muy sencillo:

- Edita el parche para crear un parche -p1 sobre el árbol el código fuente.
- En el caso de dpatch, añade una cabecera empleando la orden dpatch patch-template.
- Pon ese fichero en debian/patches
- Añade el nombre de fichero de este parche a debian/patches/00list (en dpatch) o debian/patches/series (en quilt).

Además, dpatch puede crear parches dependientes de la arquitectura usando macros CPP.

3.6.5 Incluir orig.tar.gz para subir

Cuando subes por primera vez un paquete al archivo, necesitas incluir las fuentes originales orig.tar.gz. Si la versión del paquete no es una revisión de Debian -0 o -1, debes proporcionarle la opción *«-sa»* a la orden *dpkg-buildpackage*. Por otro lado, la opción *«-sd»* forzará la exclusión del código original *orig.tar.gz*.

3.7 Cómo comprobar tu paquete para encontrar fallos

3.7.1 Los paquetes lintian

Ejecuta *lintian* sobre tu fichero de cambios *.changes*. Estos programas comprobarán muchos errores comunes al empaquetar. La orden es:

```
lintian -i gentoo_0.9.12-1_i386.changes
```

Por supuesto, cambia el nombre de fichero con el nombre del fichero de cambios generado por tu paquete. Si parece que hay algunos errores (líneas que comienzan por E:), lee la explicación (líneas N:), corrige errores, y reconstruye como se describe en Reconstrucción completa, Sección 6.1. Las líneas que comienzan con W: son sólo avisos (Warnings), así que afina el paquete o verifica que los avisos son falsos (y haz que *lintian* los acepte, consulta la documentación para más detalles).

Observa que puedes construir el paquete con *dpkg-buildpackage* y ejecutar *lintian* todo con sólo una orden si utilizas *debuild*.

3.7.2 La orden mc

Puedes descomprimir el contenido del paquete **.deb* con la orden *dpkg-deb*. Puedes listar el contenido de un paquete Debian con *debc*.

Este proceso puede ser muy intuitivo si empleamos un gestor de ficheros como *mc*, que permite visionar tanto el contenido del paquete **.deb*, como el de los ficheros **.diff.gz* y **.tar.gz*.

Vigila que no haya ficheros innecesarios extra o de tamaño cero, tanto en el binario como en el paquete fuente. A veces, hay cosas que no se limpiaron adecuadamente, debes ajustar tu fichero *«rules»* para arreglar esto.

Pista: *«zgrep ^+++ ../gentoo_0.9.12-1.diff.gz»* te dará una lista de tus cambios o contribuciones a las fuentes, y *«dpkg-deb -c gentoo_0.9.12-1_i386.deb»* o *«debc gentoo_0.9.12-1_i386.changes»* listará los ficheros en el paquete binario.

3.7.3 La orden debdiff

Puedes comparar la lista de ficheros de dos paquetes binarios de Debian con la orden *debdiff*. Este programa es útil para verificar que no hay ficheros que se hayan cambiado de sitio o eliminado por error, y que no se ha realizado ningún otro cambio no deseado al actualizar el paquete. Puedes comprobar un grupo de ficheros **.deb* simplemente con *«debdiff paquete-viejo.change paquete-nuevo.change»*.

3.7.4 La orden interdiff

Puedes comparar dos ficheros *diff.gz* con la orden *interdiff*. Esto es muy útil para verificar que no se han realizado cambios inadvertidos por el mantenedor al actualizar el paquete. Ejecuta *«interdiff -z paquete-viejo.diff.gz paquete-nuevo.diff.gz»*.

3.7.5 La orden debi

Instala el paquete para probarlo tú mismo, por ejemplo, usando la orden *debi* como superusuario. Intenta instalarlo y ejecutarlo en otras máquinas distintas de la tuya, y presta atención para detectar errores o avisos tanto en la instalación como en la ejecución del programa.

3.7.6 El paquete pbuilder

El paquete *pbuilder* es muy útil para conseguir un entorno limpio (chroot) donde verificar las dependencias. Esto asegura una construcción limpia desde el código en los programas que realizan la compilación automática de paquetes para diferentes arquitecturas y evita fallos serios del tipo FTBFS (Fallo al construir desde la fuente o «Fail to Build From Source»), que son siempre del tipo RC (fallos criticos o «release critical»). Para más informacon del paquete debian auto-builder véase *http://buildd.debian.org/*.

El uso más básico del paquete pbuilder es la ejecución directa de la orden *pbuilder* como administrador. Por ejemplo, puedes construir un paquete si escribes las siguientes órdenes en el directorio donde se encuentran los ficheros *.orig.tar.gz*, *.diff.gz* y *.dsc*:

```
pbuilder create # si se ejecuta por segunda vez, pbuilder update
pbuilder build foo.dsc
```

Los paquetes recién construidos se pueden encontrar en /var/cache/pbuilder/result/ y el propietario será el usuario administrador.

La orden pdebuild te ayuda a usar las funciones del paquete pbuilder como usuario sin permisos de administración. Desde el directorio raíz del código fuente, con el archivo orig.tar.gz en el directorio padre, escribe las siguientes órdenes:

```
$ sudo pbuilder create # si se ejecuta por segunda vez, sudo pbuilder update
$ pdebuild
```

Los paquetes construidos se pueden encontrar en /var/cache/pbuilder/result/ y el propietario no será el administrador. [2]

Si deseas añadir fuentes de apt para que las utilice el paquete pbuilder, configura *OTHERMIRROR* en *~/.pbuilderrc* o */etc/pbuilderrc* y ejecuta (para sarge):

```
$ sudo pbuilder update --distribution sarge --override-config
```

Es necesario el uso de *–override-config* para actualizar las fuentes de apt dentro del entorno chroot.

3.8 Actualizar el paquete

3.8.1 Nueva revisión Debian del paquete

Supongamos que se ha creado un informe de fallo en tu paquete con el número #54321, y que describe un problema que puedes solucionar. Para crear una nueva revisión del paquete, necesitas:

- Corregir, por supuesto, el problema en las fuentes del paquete.
- Añadir una nueva revisión en el fichero de cambios (changelog), con *«dch -i»*, o explícitamente con *«dch -v <versión>-<revisión>»* y entonces insertar los comentarios con tu editor favorito.

  Sugerencia ¿Como obtener la fecha fácilmente en el formato requerido? Usa «822-date», o «date -R».
- Incluir una breve descripción del error y su solución en la entrada del fichero de cambios, seguido por: *«Closes: #54321»*. De esta forma, el informe del error será automágicamente cerrado por los programas de gestión del archivo en el momento en que tu paquete se acepte en el archivo de Canaima.
- Repite lo que hiciste en *Reconstrucción completa*, *Cómo comprobar tu paquete para encontrar fallos* y *Enviar el paquete*. La diferencia es que esta vez, las fuentes originales del archivo no se incluirán, dado que no han cambiado y ya existen en el archivo de Canaima.

3.8.2 Nueva versión del programa fuente (básico)

Ahora consideremos una situación diferente y algo más complicada: ha salido una versión nueva de las fuentes originales, y, por supuesto, deseas empaquetarla. Debes hacer lo siguiente:

- Descarga las nuevas fuentes y pon el archivo tar (pongamos que se llama gentoo-0.9.13.tar.gz) un directorio por encima del antiguo árbol de fuentes (por ejemplo ~/gentoo/).
- Entra en el antiguo directorio de las fuentes y ejecuta:

  ```
  uupdate -u gentoo-0.9.13.tar.gz
  ```

:Por supuesto, reemplaza este nombre de fichero con el nombre de las fuentes de tu programa. uupdate(1) renombrará apropiadamente este fichero tar, intentará aplicar los cambios de tu fichero .diff.gz previo y actualizará el nuevo fichero debian/changelog.

- Cambia al directorio «../gentoo-0.9.13», el nuevo directorio fuente del paquete, y repite la operación que hiciste en *Reconstrucción completa*, *Cómo comprobar tu paquete para encontrar fallos*, y *Enviar el paquete*.

:Observa que si has puesto el fichero «debian/watch» como se describe en watch.ex, puedes ejecutar *uscan* para buscar automáticamente fuentes revisadas, descargarlas, y ejecutar uupdate

3.8.3 Nueva versión de las fuentes (realista)

Cuando prepares paquetes para el archivo de Debian, debes comprobar los paquetes resultantes en detalle. A continuación, tienes un ejemplo más realista de este procedimiento.

- Verificar los cambios en las fuentes.
- De las fuentes, lee los ficheros changelog, NEWS, y cualquier otra documentación que se haya publicado con la nueva versión.
- Ejecuta «diff -urN» entre las fuentes viejas y las nuevas para obtener una visión del alcance de los cambios, donde se ha trabajado más activamente (y por tanto donde podrían aparecer nuevas erratas), y también busca cualquier cosa que pudiera parecer sospechosa.
- Porta el paquete Debian viejo a la nueva versión.
- Descomprime el código fuente original y renombra la raíz del árbol de las fuentes como <nombrepaquete>-<versión_original>/ y haz «cd» en este directorio.
- Copia el código fuente en el directorio padre y renombrarlo como <nombrepaquete>_<versión_original>.orig.tar.gz .
- Aplica el mismo tipo de modificación a el nuevo código que al viejo. Algunos posibles métodos son:
- orden «zcat /path/to/<nombrepaquete>_<versión-vieja>.diff.gz | patch -p1»,
- orden «uupdate»,
- orden «svn merge» si gestionas el código con un repositorio Subversion o,
- simplemente copia el directorio debian/ del árbol de código viejo si se empaquetó con dpatch o quilt.
- Conserva las entradas viejas del fichero *«changelog»* (puede parecer obvio, pero se han dado casos...)
- La nueva versión del paquete es la versión original añadiéndole el número de revisión de Canaima, por ejemplo, “0.9.13-canaima1”.
- Añade una entrada en el fichero *«changelog»* para esta nueva versión al comienzo *debian/changelog* que ponga *«New upstream release»* (nueva versión original). Por ejemplo, *«dch -v 0.9.13-1»*.

- Describe de forma resumida los cambios en la nueva versión del código fuente que arreglan fallos de los que ya se ha informado y cierra esos fallos en el fichero *«changelog»*.
- Describe de forma resumida los cambios hechos a la nueva versión del código por el mantenedor para arreglar fallos de los que se ha informado y cierra esos fallos en el fichero «changelog».
- Si el parche/fusión no se aplicó limpiamente, inspecciona la situación para determinar qué ha fallado (la clave está en los ficheros .rej). A menudo el problema es que un parche que has aplicado a las fuentes se ha integrado en el código fuente original, y, por lo tanto, el parche ya no es necesario.
- Las actualizacioens de versión deberían ser silenciosas y no intrusivas (los usuarios sólo deberían advertir la actualización al descubrir que se han arreglado viejos fallos y porque se han introducido algunas nuevas características).
- Si necesitas añadir plantillas eliminadas por alguna razón, puedes ejecutar dh_make otra vez en el mismo directorio ya «debianizado», con la opción -o. Una vez hecho esto edítalo como sea necesario.
- Deberías reconsiderar todos los cambios introducidos para Canaima: elimina aquello que el autor original haya incorporado (de una forma u otra) y recuerda mantener aquellos que no hayan sido incorporados, a menos que haya una razón convincente para no incluirlos.
- Si se ha realizado algún cambio en el sistema de construcción (esperemos que lo supieras desde el primer paso), actualiza los ficheros *debian/rules* y las dependencias de construcción en *debian/control* si es necesario.
- Construye el nuevo paquete como se describe en La orden *debuild*, o El paquete *pbuilder*. Es conveniente el uso de pbuilder.
- Comprueba que los paquetes nuevos se han construido correctamente.
- Ejecuta *Cómo comprobar tu paquete para encontrar fallos*.
- Ejecuta *Verificar actualizaciones del paquete*.
- Si realizaste algún cambio en el empaquetado durante el proceso, vuelve al segundo paso hasta que todo esté correcto.
- Si tu envío necesita que se patrocine, asegúrese de comentar cualquier opción especial que se requiera en la construcción del paquete (como *«dpkg-buildpackage -sa -v ...»*) y de informar a tu patrocinador, así podrá construirlo correctamente.
- Si lo envías tú, ejecuta *Enviar el paquete*.

3.8.4 El archivo orig.tar.gz

Si intentas construir los paquetes sólo desde el nuevo código fuente con el directorio debian/, sin que exista el fichero orig.tar.gz en el directorio padre, acabarás creando un paquete de fuentes

nativo sin querer. Estos paquetes se distribuyen sin el fichero diff.gz. Este tipo de empaquetamiento sólo debe hacerse para aquellos paquetes que son específicos de Debian, es decir, aquellos que no serían útiles en otra distribución. [5]

Para obtener un paquete no nativo de fuentes que consista tanto en un archivo orig.tar.gz como en un archivo diff.gz, debes copiar manualmente el archivo tar del código fuente original al directorio padre con el nombre cambiado a <nombrepaquete>_<versión>.orig.tar.gz. Igual que como lo hizo la orden dh_make en «Debianización» inicial, Sección 2.4.

3.8.5 La orden cvs-buildpackage y similares

Deberías considerar el utilizar algún sistema de administración de código para gestión del proceso de empaquetado. Hay varios guiones adaptados para que puedan utilizarse en algunos de los sistemas de control de versiones más populares.

- **CVS**
  - cvs-buildpackage
- **Subversion**
  - svn-buildpackage
- **GIT**
  - git-buildpackage

Estas órdenes también automatizan el empaquetado de nuevas versiones del código fuente.

3.8.6 Verificar actualizaciones del paquete

Cuando construyas una nueva versión del paquete, deberías hacer lo siguiente para verificar que el paquete puede actualizarse de forma segura:

- actualiza el paquete a partir de la versión previa,
- vuelve a la versión anterior y elimínala,
- instala el paquete nuevo,
- elimínalo y reinstálalo de nuevo,
- púrgalo.

Si el paquete hace uso de unos guiones pre/post/inst/rm complicados, asegúrate de probar éstos con las distintas rutas posibles en la actualización del paquete.

Ten en cuenta que si tu paquete ha estado previamente en Canaima, lo más frecuente es que gente actualice el paquete desde la versión que estaba en la última versión de Canaima. Recuerda que debes probar también las actualizaciones desde esa versión.

3.8.7 Dónde pedir ayuda

Antes de que te decidas a preguntar en lugares públicos, por favor, simplemente RTFM («Lee el dichoso manual»), que incluye la documentación en */usr/share/doc/dpkg*, */usr/share/doc/debian*, */usr/share/doc/autotools-dev/README.Debian.gz*, */usr/share/doc/package/** y las páginas de *man/info* para todos los programas mencionados en este documento.

Si tienes dudas sobre empaquetado a las que no has podido encontrar respuesta en la documentación, puedes preguntar en la lista de correo de desarrolladores: desarrolladores@canaima.softwarelibre.gob.ve

Aunque todo funcione bien, es el momento de empezar a rezar. ¿Por qué? Por que en sólo unas horas (o días) usuarios de todo el mundo empezarán a usar tu paquete, y si cometiste algún error crítico centenares de usuarios furiosos de Canaima te bombardearán con correos... sólo bromeaba :-)

3.8.8 Ejemplos

En este ejemplo empaquetaremos el código fuente original gentoo-1.0.2.tar.gz y subiremos todos los paquetes al nm_objetivo.

- A.1 Ejemplo de empaquetado sencillo:

```
$ mkdir -p /ruta/a # nuevo directorio vacío
$ cd /ruta/a
$ tar -xvzf /ruta/desde/gentoo-1.0.2.tar.gz # obtén la fuente
$ cd gentoo-1.0.2
$ dh_make -e nombre@dominio.com -f /ruta/desde/gentoo-1.0.2.tar.gz
... Responde a las preguntas
... Arregla el árbol de las fuentes
... Si es un paquete que contiene programas guiones, indica en debian/control
... No borres ../gentoo_1.0.2.orig.tar.gz
$ debuild
... Asegúrate de que no hay ningún aviso
$ cd ..
$ dupload -t nm_objetivo gentoo_1.0.2-1_i386.changes
```

- A.2 Ejemplo de empaquetado con dpatch y pbuilder:

```
$ mkdir -p /ruta/a # nuevo directorio vacío
$ cd /ruta/a
$ tar -xvzf /ruta/desde/gentoo-1.0.2.tar.gz
$ cp -a  gentoo-1.0.2 gentoo-1.0.2-orig
$ cd gentoo-1.0.2
$ dh_make -e nombre@dominio.com -f /ruta/de/gentoo-1.0.2.tar.gz
... Responde a las preguntas
```

En un principio, debian/rules es así:

```
configure: configure-stamp
configure-stamp:
        dh_testdir
        # Add here commands to configure the package.
        touch configure-stamp
build: build-stamp
build-stamp: configure-stamp
        dh_testdir
        # Add here commands to compile the package.
        $(MAKE)
        #docbook-to-man debian/gentoo.sgml > gentoo.1
        touch $@
clean:
        dh_testdir
        dh_testroot
        rm -f build-stamp configure-stamp
        # Add here commands to clean up after the build process.
        -$(MAKE) clean
        dh_clean
```

Cambia lo siguiente con un editor en debian/rules para usar dpatch y añade dpatch a la línea Build-Depends: en el fichero debian/control:

```
configure: configure-stamp
configure-stamp: patch
        dh_testdir
        # Add here commands to configure the package.
        touch configure-stamp
build: build-stamp
build-stamp: configure-stamp
        dh_testdir
        # Add here commands to compile the package.
        $(MAKE)
        #docbook-to-man debian/gentoo.sgml > gentoo.1
        touch $@
clean: clean-patched unpatch
        dh_testdir
        dh_testroot
        rm -f build-stamp configure-stamp
        # Add here commands to clean up after the build process.
        -$(MAKE) clean
        dh_clean
patch: patch-stamp
patch-stamp:
     dpatch apply-all
```

```
     dpatch call-all -a=pkg-info >patch-stamp
unpatch:
     dpatch deapply-all
     rm -rf patch-stamp debian/patched
```

Ahora está todo preparado para reempaquetar el árbol de código con el sistema dpatch y con la ayuda de dpatch-edit-patch:

```
$ dpatch-edit-patch patch 10_firstpatch
... Arregla el arbol de fuentes con el editor
$ exit 0
... Intenta construir el paquete con «debuild -us -uc»
... Limpia las fuentes con «debuild clean»
... Repite con dpatch-edit-patch hasta que las fuentes compilen.
$ sudo pbuilder update
$ pdebuild
$ cd /var/cache/pbuilder/result/
$ dupload -t nm_objetivo gentoo_1.0.2-1_i386.changes
```

:)