Trabajo de Diploma

Universidad de las Ciencias Informáticas

Sistema para la generación de paquetes binarios fragmentados a partir del código fuente.

Trabajo de Diploma

para optar por el título de

Ingeniero en Ciencias Informáticas

Autor:

Daniel Hernandez Bahr

Tutores:

Lic. Dariem Pérez Herrera

Ing. Anielkis Herrera González

Ciudad de la Habana, Cuba

Mayo, 2010

DECLARACIÓN DE AUTORÍA

Declaro ser el autor del presente trabajo de diploma y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo a los ___ días del mes de _____ del año _____.

Daniel Hernandez Bahr

_____

Firma del Autor

Dariem Pérez Herrera

_____

Firma del Tutor

Anielkis Herrera González

_____

Firma del Tutor

Agradecimientos

A Aida Bahr y Jorge Luis Hernández, que desde que tengo memoria me han querido y guiado para convertirme en lo que soy.

A Carmen, por compartir conmigo este año y mantenerme feliz incluso en los momentos en que tuve que volver a empezar todo el trabajo.

A mis tíos Santiago y Esperanza y mis primos Margarita y Jesús, por acogerme en su casa estos cinco años y hacerme sentir que es la mía.

A Ángela Casanova, por ser una suegra tan buena como uno pueda imaginarse.

A Dariem y Anielkis, que más que tutores han sido grandes amigos.

A Abel por toda su ayuda, que fue bastante.

A Jose Jorge, otro amigo que ha sido tutor este año.

A Allan, Miranda, Migue, Micha, Soler, Machín y todo el equipo de Nova, porque por ellos cada día de trabajo se disfruta.

A todos los amigos que en algún momento se han interesado por mi trabajo y más a aquellos que me han sugerido cómo arreglar los problemas que he tenido.

Dedicatoria

Este trabajo es para Aida Bahr, que siempre ha sido un apoyo inmenso para mí y para Jorge Luis Hernández, porque ambos hubiéramos querido que él pudiera leerlo.

Resumen

Se propone un sistema informático que garantiza mayor eficiencia y control sobre el proceso de creación de paquetes binarios de software para repositorios de paquetes de software de distribuciones de GNU/Linux.

Se documenta la investigación realizada con el fin de demostrar que, dada la situación existente, resulta beneficiosa la implementación de un sistema que automatice un proceso que actualmente se desarrolla manualmente.

Se realiza un estudio comparativo entre las distintas tecnologías y tendencias de los procesos de creación de paquetes binarios de software.

Palabras clave

Paquete binario de software, sistema de automatización de compilación, Autotools.

Abstract

Proposal of an informatics system that guarantees greater efficiency and control over the creation process for binary software packages to be included on software packages repositories for GNU/Linux distributions.

Documentation on the investigation performed in order to demonstrate that, given the existent situation, the implementation of a system that automates a process that is currently manually developed results beneficial.

Comparative study between different technologies and tendencies used in the binary software packages creation process.

Keywords

Binary software package, build automation system, Autotools.

Índice

Introducción

Antecedentes

El sistema operativo Nova, de la familia GNU/Linux, comenzó a desarrollarse durante el tercer curso académico de la Universidad de las Ciencias Informáticas con el objetivo de vincular algunas asignaturas del programa docente con actividades prácticas. Desde el principio se eligió Gentoo[1] como distribución base[2] debido a su facilidad de adaptación a distintas y, en el caso específico del país, antiguas configuraciones de hardware. Con el paso de los años el proyecto fue ganando en madurez y pronto surgió una nueva y más ambiciosa meta: apoyar el proceso de migración del país a tecnologías de Software Libre y Código Abierto (FOSS, por sus siglas en inglés). Con tal fin se redirigió el trabajo del equipo de desarrollo hacia lograr un sistema operativo que presupusiera la menor cantidad de conocimiento posible de parte del usuario.

Uno de los principales problemas a los que se enfrentó entonces el proyecto se encontraba en el sistema de gestión de paquetes de software. Hasta entonces este proceso se había heredado de Gentoo sin sufrir modificaciones y consistía en descargar los paquetes de código fuente desde un repositorio y compilarlos en el sistema; lo que permitía al usuario –en tiempo de instalación– elegir, de las posibles, cuáles funcionalidades deseaba que se instalasen con el paquete en cuestión[3]; lográndose esto, en la mayor parte de los casos, al instalar nuevos paquetes de software que proveían dichas funcionalidades al paquete seleccionado por el usuario.

Este método implicaba principalmente dos problemas a saber:

1. El proceso de instalación de nuevos paquetes de software podía resultar extremadamente lento sobre todo para aplicaciones de gran tamaño.
2. Los errores que se pudieran generar durante el proceso resultaban muy complicados de comprender para los usuarios a los que estaba enfocado el nuevo sistema.

En aras de aumentar la aceptación de los usuarios, una de las principales tareas que se emprendieron entonces fue la implementación de un sistema de gestión de paquetes binarios, o precompilados, que redujera el tiempo y la posibilidad de errores durante la instalación de nuevo software en el sistema. Un estudio del estado del arte hecho entonces mostró que *Entropy*, gestor de paquetes oficial de *Sabayon Linux*[4] cumplía con los requerimientos del proyecto y preveía, a corto y

1 Sistema operativo construido sobre un kernel Linux y basado en el sistema de gestión de paquetes Portage.
2 Se entiende por distribución base a aquella que se reutiliza para no incurrir en la duplicación de esfuerzos que conllevaría desarrollar una distribución desde cero.
3 Esto se logra a través de *directivas de compilación* que permiten, al compilar una aplicación o biblioteca, indicar al compilador si es necesario o no incluir secciones del código que incorporan alguna funcionalidad.
4 Distribución de GNU/Linux basada en Gentoo, creada por Fabio Erculiani y el equipo de desarrollo Sabayon.

mediano plazos, la implementación de funcionalidades deseables que permitían integrar a la comunidad al desarrollo y mantenimiento del sistema. La principal deficiencia de este nuevo sistema gestor de paquetes residía en la carencia de una interfaz de escritorio: todo el trabajo debía hacerse a través una interfaz de línea de comandos en la *consola* del sistema. Así comenzó el desarrollo de Summon, una interfaz gráfica para Entropy, que se convertiría en el gestor de paquetes del nuevo sistema.

Sin embargo la asimilación de Entropy generó un nuevo problema. Al existir los paquetes ya compilados en el repositorio se limitaba la libertad del usuario de escoger si deseaba instalar el software con (o sin) alguna funcionalidad en específico; sino que debía instalar las funcionalidades que se le habilitaban al paquete al compilarlo antes de incluirlo al repositorio. Este detalle podía resultar muy molesto cuando se deseaba instalar un paquete relativamente pequeño que había sido compilado incluyendo alguna funcionalidad que dependiera de otro paquete especialmente grande que el usuario pudiera no desear, pues debía instalarse el paquete no deseado antes que el necesitado, lo que conllevaba un gasto adicional en tiempo de descarga de los paquetes en cuestión, con la correspondiente sobrecarga del ancho de banda del que pueda disponer el usuario.

La política al respecto de la dirección del proyecto era habilitar la mayor cantidad de funcionalidades posibles a los paquetes para cubrir un espectro más amplio de funcionalidades.

De lo anteriormente expuesto se puede concluir la siguiente

Situación Problemática

El alto acoplamiento entre los paquetes binarios de software incluidos en el repositorio de Nova y las funcionalidades que proveen aumenta el consumo de ancho de banda y tiempo disponibles por el usuario. Esta situación puede traer como consecuencia una disminución en la aceptación del sistema.

Problema Científico

¿Cómo generar paquetes binarios de forma que se optimice el proceso de instalación de software en Nova?

Objeto de Estudio

El objeto de estudio de este trabajo son los métodos de generación de paquetes binarios de software para distribuciones de GNU/Linux.

Campo de Acción

El campo de acción se restringe a aquellos paquetes que utilizan el *GNU Build System* como sistema de automatización de compilación.

Idea a Defender

La elaboración de una herramienta que automatice el proceso de fragmentación de paquetes binarios de software optimizará el proceso de gestión de paquetes de software en la distribución cubana de GNU/Linux Nova.

Objetivo General

Desarrollar una herramienta que permita seccionar los paquetes binarios de software resultantes para software que utilice el GNU Build System como sistema de automatización de compilación.

Objetivos Específicos

1. Identificar los métodos de fragmentación de paquetes utilizados en diferentes distribuciones de GNU/Linux.
2. Desarrollar una herramienta que determine las distintas secciones en que se puede fragmentar un paquete de software en dependencia de los soportes que se le habiliten a partir de los ficheros de configuración del GNU Build System (también conocido como *Autotools)*.

Tareas de Investigación

1. Análisis de los procesos de fragmentación de paquetes de distintas distribuciones de GNU/Linux e identificación de las herramientas utilizadas.
2. Estudio de las características más relevantes de los ficheros de configuración generados por las Autotools.
3. Elaboración de una herramienta que permita:
    a) Descubrir los posibles paquetes a generar para un software que utilice el *GNU Build System* como sistema gestor de compilación.
    b) Determinar los ficheros que serán contenidos por los los paquetes a generar, así como la interdependencia entre dichos paquetes.

Métodos de Investigación

1. Análisis histórico lógico, que permitió determinar el desarrollo y estado actual del objeto de estudio.
2. Modelación, que permitió estudiar las cualidades y relaciones entre los componentes del objeto de estudio.
3. Observación participante (el observador forma parte del problema a resolver) no controlada (no existe una guía para realizar la observación), que permitió tener un registro visual del comportamiento de algunas herramientas utilizadas dentro del objeto de estudio.

Fundamentación del Tema

El presente capítulo describe las tendencias utilizadas en la actualidad por los principales sistemas de la familia GNU/Linux respecto a la estructura de los paquetes que se incluyen en sus repositorios de software. Se realiza un estudio de las herramientas empleadas por los mantenedores de paquetes de distintas distribuciones que permita apoyar la decisión de adoptar alguna de las técnicas existentes o desarrollar una nueva en caso de no encontrar ninguna factible. Se describe también, muy elementalmente, el uso de los ficheros de configuración generados por el GNU Build System.

Repositorios de paquetes de software

Los sistemas de la familia GNU/Linux usan, por regla general, sistemas de gestión de paquetes para controlar las aplicaciones y bibliotecas que se instalan y/o desinstalan en el sistema. Estos sistemas de gestión obtienen los paquetes desde alguna ubicación específica que puede encontrarse disponible a través de la red o en algún dispositivo físico como discos duros, compactos o cualquier dispositivo de almacenamiento extraíble. A estas ubicaciones se les conoce como repositorios de paquetes de software. Existen distintas configuraciones estructurales para estos repositorios, en dependencia de qué sistema de gestión se use, pero todas coinciden en que en ellos se pueden encontrar los paquetes de software disponibles para instalar en la distribución que los mantenga. Otro detalle digno de hacer notar reside en que los paquetes incluidos en los repositorios pueden ser de código fuente o precompilados.

En el proceso de instalación, los sistemas de gestión de paquetes se encargan de determinar todos los paquetes que resulte necesario procesar para satisfacer la petición del usuario (paso conocido como Cálculo de Dependencias), proceden entonces a descargar los paquetes del repositorio hacia una ubicación local dentro del sistema, verifican la integridad de los paquetes (esto se refiere a comprobar que no hayan sido adulterados) y solo entonces se instalan y configuran; finalmente se procede a limpiar el sistema de los ficheros temporales generados en tiempo de instalación. Vale destacar que esta descripción del proceso no pretende ser exhaustiva, sino brindar al lector una idea más bien general, para ahondar en los detalles que interesan al tema más adelante.

Como puede suponerse, la diferencia más notable entre los paquetes de código fuente y los precompilados (en lo que al proceso de instalación como tal se refiere) se encuentra, precisamente, en el paso de instalación del paquete después ya de haber sido descargado y validada su integridad. La diferencia está dada en que, por regla general, los paquetes precompilados se instalan con solo descomprimir el paquete y copiar los ficheros en la ubicación final que les corresponde; en cambio los

paquetes de código fuente necesitan ser compilados en primer lugar.

Los sistemas que gestionan paquetes precompilados son mucho más atractivos para la mayoría de los usuarios, especialmente los menos experimentados, puesto que el proceso resulta mucho más rápido. Sin embargo esto no quiere decir que los sistemas que utilizan paquetes de código fuente no tengan su atractivo, mayormente apreciado por usuarios más experimentados y quizá un tanto quisquillosos. Hay dos grandes motivos para esta situación a saber: el primero puede enunciarse como que al compilar los paquetes se pueden asegurar de que queden completamente optimizados para su configuración específica de hardware, cosa que no se puede lograr con paquetes precompilados (ya que estos deben funcionar sobre un espectro más amplio de arquitecturas) y, en segundo lugar, que al compilar los paquetes en su sistema pueden elegir, según lo permita el paquete en cuestión, qué funcionalidades desean compilar del paquete, logrando un grado de personalización mucho mayor.

Los paquetes precompilados, como puede resultar obvio, han pasado ya por el proceso de compilación, durante el cual se genera el paquete lo más genérico posible, de manera que pueda funcionar correctamente sobre la mayor cantidad de configuraciones de hardware y que brinde la mayor cantidad de funcionalidades posible.

Proceso de confección de paquetes de software

El proceso de empaquetamiento del software comienza desde la página web del proyecto. Los mantenedores de paquetes descargan el código fuente y utilizan herramientas de compilación para generar los ficheros binarios con lenguaje entendible por el procesador.

La mayoría de las grandes distribuciones de GNU/Linux proveen a los mantenedores de sus paquetes de un ambiente que favorezca el proceso de compilación y empaquetamiento, pero estas herramientas, aunque gestionan muchas opciones de configuración y automatizan gran parte del trabajo, no son capaces de hacer todo el proceso por sí solas: es responsabilidad del mantenedor elegir qué funcionalidades de las provistas por el software a empaquetar serán incluidas en el paquete específico que se crea. Es asimismo responsabilidad del mantenedor la decisión de si es necesario generar más de un paquete para el software en cuestión. Una práctica frecuente en paquetes de distribuciones como Debian y sus derivados, consiste en crear un paquete aparte para los archivos de cabecera que no resultan necesarios para que la aplicación funcione aunque sí para poder extenderla más adelante, así también se crea un subpaquete para la documentación del paquete.

A continuación se muestran algunas de las herramientas más utilizadas:

Open SuSE brinda el *openSUSE Build Service* (Servicio de Construcción de Open SuSE). Se trata de una plataforma de desarrollo de distribuciones abierta y completa que provee de una infraestructura transparente para el desarrollo de la distribución openSUSE. Es el único servicio que permite a los desarrolladores empaquetar software para todas las principales distribuciones de GNU/Linux. El servicio provee a los desarrolladores de software de una herramienta conveniente y fácil de usar para crear y liberar software de código abierto para openSUSE y otras distribuciones de GNU/Linux en diferentes arquitecturas de hardware y para una amplia audiencia de usuarios. [1]

Posee una interfaz web y un cliente por línea de comandos (CLI por sus siglas en inglés) que permiten a los empaquetadores registrados

- Crear un paquete nuevo
- Modificar un paquete existente
- Elegir para qué distribución se está creando el paquete
- Editar los meta-datos del paquete
- Compilar el paquete (esto se puede realizar localmente o en el servidor de openSUSE)
- Verificar los registros del sistema

Fedora, proyecto comunitario patrocinado por Red Hat, provee un sistema más simplificado utilizando:

- `'rpmdevtools'`: contiene varios guiones (o scripts) para facilitar el desarrollo de paquetes RPM.
- `'rpmbuild'`: usado para construir paquetes de software tanto binarios como de código fuente. [2]

Debian y Ubuntu proponen en sus guías para la mantención de paquetes las siguientes herramientas.

- `'build-essential'`: meta-paquete que provee una serie de herramientas muy utilizadas en la compilación de paquetes en sistemas basados en Debian ('`libc6-dev`', '`gcc`', '`g++`', '`make`', '`dpkg-dev`').
- `'devscripts'`: contiene varios guiones que facilitan el trabajo del mantenedor.
- `'ubuntu-dev-tools'`: también una colección de guiones (como '`devscripts`') pero específicos para Ubuntu.

- `'debhelper'`: guiones que realizan tareas comunes de empaquetamiento.
- `'dh-make'`: puede ser usado para crear plantillas (o templates) para el empaquetamiento.
- `'diff'` y `'patch'`: son usados para crear y aplicar parches, respectivamente.
- `'cdbs'` (*Common Debian Build System):* provee un conjunto de reglas contra las que se pueden compilar los paquetes.
- `'quilt'`: gestiona una serie de parches llevando registro de los cambios que cada uno hace.
- `'gnupg'`: reemplazo libre y completo para *PGP* utilizado para firmar ficheros digitalmente (incluyendo paquetes).
- `'fakeroot'`: simula la ejecución de comandos con privilegios de *root*.
- `'lintian'` y `'linda'`: buscan y reportan bugs y violaciones de políticas en paquetes debianizados.
- `'pbuilder'`: construye un sistema *chroot* y compila paquetes dentro del mismo. [3] [4]

Estos sistemas proveen a los mantenedores de paquetes de un entorno en el que casi todo el trabajo se puede realizar siguiendo una serie de pasos más o menos estándar (que varían de un sistema a otro), por lo que el trabajo humano se limita a ocuparse de las especificidades de cada paquete y las transiciones entre uno y otro paso; pero requieren, de los mantenedores, atención dedicada.

El punto débil de estos sistemas reside en la gran cantidad de capital humano que debe dedicarse a la manutención de paquetes, considerando la cantidad de paquetes incluidos en los repositorios de aplicaciones y la cantidad de paquetes mantenidos por cada mantenedor que, por lo general, realizan estas tareas en sus horas libres, de forma voluntaria y, en la mayor parte de los casos, sin ánimo de lucro, por lo que resulta difícil imponer plazos de entrega y otras restricciones. Por este motivo, distribuciones que cuenten con menos apoyo (cuantitativo) de la comunidad, no pueden darse el lujo de mantener toda la paquetería en su repositorio, y se limitan a reutilizar los paquetes mantenidos por sus distribuciones base.

Claro está que existen plataformas de compilación como la utilizada por Ututo XS que permiten compilar grandes cantidades de paquetes con supervisión mínima, pero esta plataforma utiliza las recetas (ebuilds) de los paquetes del repositorio de Gentoo para la configuración del proceso de compilación, y no se ocupan de separar las distintas funcionalidades de cada paquete pues funcionan con configuraciones preestablecidas para todo el conjunto de paquetes.

Proceso de Compilación de Software

Ahora bien, todos los métodos anteriormente descritos poseen, cuando menos, un punto en común: para crear paquetes binarios de software es necesario compilarlos.

El proceso de compilación del código fuente consiste en traducir el lenguaje de alto nivel utilizado por el desarrollador a un lenguaje de nivel inferior que el procesador pueda reconocer. Este lenguaje de bajo nivel (generalmente “lenguaje máquina”) depende directamente de la arquitectura del hardware del equipo computador. Hoy en día la arquitectura de los procesadores está bastante estandarizada, pero no lo suficiente como para que el código generado por un compilador específico funcione en cualquier equipo sin problemas. Se hace necesario entonces compilar el paquete una vez por cada arquitectura en que se quiera que funcione; trabajo, por demás, bastante engorroso en estos tiempos de estándares, no digamos ya décadas pasadas cuando la variedad de arquitecturas era mucho mayor. Vale mencionar también que compilar y enlazar una aplicación cuyo código fuente está en un solo fichero es bastante sencillo desde una interfaz de línea de comandos, sin embargo cuando la construcción del software requiere separar el código en varios ficheros, que deben ser compilados en un orden específico, el trabajo se complica.

Sistemas de automatización de la compilación

Para facilitar el trabajo de los desarrolladores de software surgieron los sistemas de automatización de la compilación (*Build Automation Systems* según el término original en inglés). Estos sistemas, por regla general, consisten en guiones de instrucciones que automatizan una amplia variedad de tareas que los desarrolladores de software realizan en sus actividades diarias entre las que podrían mencionarse:

- compilar código fuente a código binario
- empaquetar código fuente y código binario
- ejecutar pruebas
- despliegues hacia sistemas de producción
- creación de documentación y notas de liberación

Una de las primeras herramientas de este tipo fue *make,* creado en 1977, que permite escribir un guión de compilación que ejecute ordenadamente los pasos necesarios de compilación y enlace para construir una aplicación de software. Este fue el comienzo de la Automatización de la Compilación, su foco primario se centraba en las llamadas a los compiladores y enlazadores. En la medida en que el

proceso de compilación se fue volviendo más complejo los desarrolladores comenzaron a incluir acciones antes y después de las llamadas al compilador, tales como actualizaciones desde el control de versiones hasta la copia de objetos desplegables hacia ubicaciones de prueba. El término “automatización de la compilación” incluye ahora tanto la gestión de actividades pre y post compilación y enlace como las actividades mismas de compilación y enlace.

En años recientes, las soluciones de gestión de la compilación han proveído incluso mayor alivio en lo que al proceso de compilación se refiere. Soluciones tanto comerciales como libres y de código abierto están disponibles para desarrollar una compilación y procesamiento de flujo de trabajo más automatizados. Algunas soluciones se enfocan en automatizar los pasos anteriores y posteriores a las llamadas a los guiones de compilación, mientras otros van más allá del pre y post procesamiento de los guiones de compilación y guían las llamadas de compilación y enlace sin necesidad de mucho trabajo manual. Estas herramientas son particularmente útiles para compilaciones de integración continua (*continuous integration*, según el término original en inglés) donde se requieren llamadas frecuentes al proceso de compilación y se necesita procesamiento incremental de la compilación.

Algunos ejemplos

- Make: mencionado antes, herramienta clásica compilación en UNIX.
    - ClearMake: variante de Make desarrollada por Rational ClearCase.
    - mk: desarrollada originalmente para Plan 9 y UNIX V10.
    - Rake: herramienta desarrollada en Ruby.
    - MPW Make: desarrollada para Mac OS Classic y similar, aunque no compatible, con el make de Unix.
- Anthill Pro: con soporte de tuberías para la automatización y prueba del despliegue. Independiente del lenguaje y la plataforma.
- Apache Ant: popular para desarrollos con Java, usa ficheros con formato XML.
- Apache Buildr: sistema de compilación de código fuente basado en Rake.
- Apache Maven: herramienta en Java para la gestión de proyectos y compilación automatizada de software.
- A-A-P: herramienta de compilación desarrollada en Python.

- Automated Build Studio: sistema para la automatización y gestión de los procesos de compilación, prueba y despliegue de software con programación de compilación y soporte para integración continua.
- buildfactory: *ALM* ágil para integración continua, compilación, prueba y despliegue automatizados.
- BuildIT: herramienta gráfica libre de compilación o tareas para Windows.
- Buildout: sistema de compilación hecho en Python para crear, ensamblar y desplegar aplicaciones de muchas partes.
- CABIE: Entorno de Compilación Automatizada Continua e Integración (*Continuous Automated Build and Integration Environment*, por sus siglas en inglés), código abierto, escrito en Perl.
- Cascade: herramienta de integración continua que compila y prueba componentes de software luego de cada cambio realizado en el repositorio.
- CMake: herramienta que genera ficheros para ambientes de compilación nativos, como los makefiles para Unix o los ficheros de espacios de trabajo para Visual Studio.
- Debian Package Maker: usado para crear paquetes .deb.
- GNU Build Tools: colección de herramientas para compilaciones portables.
- MSBuild: Motor de Compilación de Microsoft.
- Nant: herramienta similar a Ant para el framework .NET.

El presente trabajo centra su atención en el proceso de generación de paquetes binarios de softwares que utilicen el Sistema de Compilación de GNU como sistema de automatización de la compilación por ser uno de los más ampliamente utilizados por los desarrolladores de software para sistemas GNU/Linux.

Sistema de Compilación de GNU

Las *Autotools*, son un conjunto de herramientas producidas por el proyecto GNU y forman parte de *GNU Toolchain*. Estas herramientas están diseñadas para ayudar a crear paquetes de código fuente portable a varios sistemas Unix. Comprende las utilidades Autoconf, Automake y Libtool.

GNU Autoconf

Es una herramienta para producir guiones que configuren automáticamente paquetes de código fuente de software para adaptarse a varios tipos de sistemas de tipo Posix. Los ficheros de configuración producidos por Autoconf son independientes de Autoconf cuando se ejecutan, por lo que los usuarios no necesitan tenerlo en su sistema.

Estos guiones de configuración no requieren intervención manual del usuario cuando se ejecutan; por lo general ni siquiera necesitan que se les especifique el tipo de sistema. En cambio, prueban individualmente la presencia de cada característica que el paquete de software para el que se ejecutan pueda necesitar. Es por esto que trabajan bien con sistemas híbridos o personalizados de variantes de Posix más comunes. [5]

Para cada paquete con el que es utilizado, Autoconf crea un guión de configuración a partir de una archivo plantilla (`'configure.in'` o `'configure.ac'`, siendo este último recomendado sobre el primero [6]) que lista las características del sistema que el paquete necesita o puede usar. También puede procesar otros archivos como los `'Makefile.in'` para producir como salida un archivo `'Makefile'`.

GNU Automake

Es una herramienta para generar automáticamente archivos `'Makefile.in'` a partir de ficheros llamados `'Makefile.am'`. Cada `'Makefile.am'` consiste, básicamente, en una serie de definiciones de variables '`make`', con algunas reglas. Los `'Makefile.in'` generados cumplen con los estándares de GNU Makefile.

El Documento de Estándares de GNU Makefile es largo, complicado y está sujeto a cambio. La meta de Automake es eliminar la carga del mantenimiento de los `'Makefile's` de cada mantenedor individual. El típico archivo de entrada de Automake es simplemente una serie de definiciones de variables. Cada uno de los archivos de entrada es procesado para generar un `'Makefile.in'`. Por lo general debe haber un archivo `'Makefile.am'` por cada directorio del proyecto. [7]

GNU Libtool

Libtool maneja todos los requerimientos de la compilación de *bibliotecas compartidas* (dinámicamente enlazadas) y en estos momentos parece ser la única manera de lograrlo con alguna portabilidad.

También se encarga de muchas otras actividades problemáticas tales como: la interacción con las reglas de Make con los sufijos de variables de bibliotecas compartidas, enlazar seguramente con bibliotecas compartidas antes de ser usadas por el *superusuario* y proveer un sistema consistente de versionado (de forma que diferentes vesiones de una biblioteca puedan ser instaladas o actualizadas sin romper la compatibilidad binaria). Aunque Libtool, como Autoconf, puede ser utilizado independientemente de Automake, es más simplemente utilizado en conjunto con Automake –acá Libtool se utiliza automáticamente cada vez que se necesitan librerías compartidas y no es necesario conocer la sintaxis. [8]

¿Usando Autotools?

Es bueno comenzar diciendo que el usuario de la Autotools es, por regla general, el desarrollador del software. Los usuarios que interactúan con el software de alguna manera, ya sea empaquetándolo o instalándolo (o desinstalándolo) en un sistema, solo utilizan los ficheros que fueron generados por las Autotools y no las Autotools mismas.

El presente trabajo no indaga en el uso de las herramientas del Sistema de Compilación de GNU, sino en el empaquetamiento del software que le utilice, y por ello la presente sección no se enfoca en el uso de Autoconf, Automake o Libtool, sino en el modo de compilar y/o empaquetar el software que haya utilizado estas herramientas para generar los archivos `'configure'`, y `'makefile.am'` y `'makefile.in'`.

El proceso, en su forma más simple, consta de dos o tres pasos a saber.

1. Se ejecuta el archivo guión `'configure'` que preparará el proceso de compilación según las características del sistemas y las opciones que se le pasen como argumentos en la llamada y generará los archivos `'Makefile'` necesarios a partir de las plantillas descritas por el desarrollador.
2. Se invoca a la herramienta `'make'` (sin argumentos) para compilar el software guiada por los archivos `'Makefile'` generados en el paso anterior. Este paso puede ser obviado si, por ejemplo, solo se desea empaquetar el código fuente sin compilar.
3. Se invoca nuevamente la herramienta `'make'`, esta vez se le pasa un argumento que especificará qué se desea hacer exactamente. De esta forma, `'make'` podrá determinar si el usuario desea instalar el software o probar la compilación o empaquetar los archivos resultantes de la compilación.

Invocando a `'configure'`

Son los argumentos pasados a la ejecución del archivo `'configure'` los que determinan cómo será compilado el software. Son muchas las opciones, desde configurar la dirección donde será instalado el paquete hasta habilitar o deshabilitar características o el uso de paquetes externos. Estas últimas mencionadas son el centro de atención de la presente investigación, puesto que influyen directamente en qué archivos binarios resultarán del proceso de compilación y es en ellas que se centra esta sección.

Incluyendo características

Para lograr este efecto `'configure'` provee dos opciones a saber que pueden ser pasadas como argumentos a la invocación del guión:

- `'--enable-CARACTERISTICA[=ARG]'`: incluye la característica identificada como `'CARACTERISTICA'` si `'ARG'` es igual a `'yes'` y la deshabilita si `'ARG'` es igual a `'no'`.
- `'--disable-CARACTERISTICA'`: no incluye la característica identificada como `'CARACTERISTICA'`; es equivalente a usar `'--enable-CARACTERISTICA=no'`.

Usando paquetes

En este caso existen dos opciones similares a las usadas para las características:

- `'--with-PAQUETE[=ARG]'`: usa el paquete descrito por `'PAQUETE'` si `'ARG'` es igual a `'yes'` y deja de usarlo en caso de que `'ARG'` sea igual a `'no'`.
- `'--without-PAQUETE'`: no usa el paquete descrito por `'PAQUETE'`; es equivalente a usar `'--with-PAQUETE=no'`.

Invocando a `'make'`

Una vez que `'configure'` termina su trabajo el resultado es un entorno preparado para compilar, para lo que se utiliza `'make'`, que también posee varias opciones de invocación, pero que no son objetivo del presente trabajo. Sin embargo es bueno proporcionar un entendimiento de cómo se le

orden qué se quiere hacer y esto se logra con objetivos (*target* según el término original en inglés). A continuación se relacionan los objetivos más relevantes para esta investigación:

- `'make install'`: Instala lo que necesite ser instalado.

Como se mencionó antes, invocar a `'make'`, sin indicar un objetivo, compilará todos los programas, bibliotecas y guiones que necesiten se compilados para el paquete. Todos los archivos se contruyen en su lugar dentro de la carpeta del código fuente del software.

De todo lo anteriormente expuesto podemos concluir que, aunque las grandes distribuciones proponen herramientas que facilitan el trabajo de los mantenedores de paquetes, estas herramientas no son capaces de determinar de qué modo debe ser construido el software que se empaquetará, y debido al nivel de atención que requiere cada paquete de su mantenedor, es necesaria la actividad de muchos mantenedores para todo un repositorio, capital humano que podría enfocarse en tareas que requieran de un nivel de creatividad mayor.

Generación de paquetes binarios.

El presente capítulo aborda una propuesta de solución al problema científico identificado, se hace un análisis de las características más relevantes de la sintaxis utilizada por los ficheros de configuración generados por las Autotools así como se describen las herramientas y metodología a utilizar para el desarrollo de dicha propuesta de solución.

Situación actual

En la actualidad el proceso de generación de paquetes binarios en Nova, está dividido en dos momentos principales.

Primeramente se tiene un equipo con el sistema instalado, esto es el sistema para el cual se está creando el repositorio, con una configuración preestablecida que no cambiará durante el tiempo que se le dé soporte. En un primer momento se instala el paquete en el equipo utilizando el sistema de gestión (Portage) y el repositorio de paquetes de Gentoo. El paquete es descargado del repositorio y se compila en el sistema de acuerdo a la configuración que el mismo posee. Una vez instalado, se procede a probar el paquete, paso en que se verifica que no existan incompatibilidades con el sistema, que todas las características previstas funcionen correctamente y que el rendimiento del software sea adecuado.

Una vez se determina que el paquete es apto para ser incluido en el repositorio oficial de la distribución se pasa al segundo momento: es aquí donde entra en juego Entropy. Entropy brinda dos conjuntos de herramientas para la gestión de paquetes, uno para el sistema cliente y otro para el servidor. Son, precisamente, las herramientas de *Entropy Server* las que se utilizan para la creación del repositorio de paquetes de software. La herramienta `'reagent'` busca en la base de datos de Portage los ficheros pertenecientes a los paquetes instalados en el sistema, genera los paquetes (que consisten en *tarballs* que contienen los ficheros con su ruta absoluta desde el directorio raíz del sistema) y crea la base de datos del servidor Entropy. Si se le indica la opción `'update'`, solo se revisan las diferencias, o sea, los paquetes compilados desde la última invocación a la herramienta. Luego se invoca `'activator'` con la opción `'sync'`, que se encarga de sincronizar la base de datos local de Entropy Server con la real del repositorio y sube los paquetes generados por `'reagent'` a la ubicación real del repositorio.

El punto débil consiste en que el paquete incluirá las funcionalidades que se le habiliten en el primer momento del proceso y estas no podrán separarse del mismo de modo que los usuarios puedan

elegir si desean incluirlas en sus sistemas o no. Claro está que algunas de estas funcionalidades no pueden separarse del núcleo, puesto que son incorporadas a este en tiempo de compilación; sin embargo, son varias las que generan ficheros externos al núcleo, esto es que el software puede funcionar sin los mismos, que brindan otro tipo de funcionalidades como, por ejemplo, interfaces para la interacción con otro software.

Autotools

Es una verdad universalmente aceptada que un desarrollador en posesión de un nuevo paquete está buscando un sistema de compilación.

En el mundo de Unix, tal sistema de compilación se alcanza, tradicionalmente, utilizando el comando `'make'`. El desarrollador escribe la receta para construir su paquete en un `'Makefile'`. Este fichero consta de un conjunto de reglas para compilar los ficheros del paquete. Cada vez que se ejecuta `'make'`, lee el `'Makefile'`, chequea la existencia y tiempo de modificación de los ficheros mencionados, decide que ficheros necesitan ser compilados (o recompilados) y ejecuta los comandos asociados.

Cuando un paquete necesita ser compilado en una plataforma distinta de la utilizada para su desarrollo, usualmente su `'Makefile'` necesita ser reajustado. Por ejemplo el compilador puede tener otro nombre o necesitar más opciones. En 1991, David J. MacKenzie se cansó de personalizar su `'Makefile'` para cada una de las 20 plataformas con las que tenía que lidiar. En cambio desarrolló un pequeño guión de consola que llamó `'configure'` para ajustar automáticamente el `'Makefile'`. Compilar su paquete ahora era tan simple como ejecutar `'./configure && make'`.

Hoy en día el proceso se ha estandarizado en el proyecto GNU. El Estándar de Código de GNU (*GNU Coding Standard* en su forma original en inglés) explica cómo cada paquete en el proyecto GNU debe tener un guión `'configure'` y la interfaz mínima que debe tener. El `'Makefile'` también debe seguir algunas convenciones establecidas. ¿Resultado? Un sistema de compilación unificado que hace todos los paquetes casi indistinguibles para aquel que los instale. En su escenario más simple, todo lo que debe hacer el instalador es descompactar el paquete, ejecutar `'./configure && make && make install'`, y repetir para el siguiente paquete para instalarlo.

Este sistema de compilación es conocido como el *Sistema de Compilación de GNU*, ya que fue desarrollado dentro del proyecto GNU. Sin embargo es utilizado por un vasto número de otros paquetes: seguir cualquier convención establecida tiene sus ventajas.

Las Autotools son herramientas que crean un Sistema de Compilación de GNU para un paquete.

Autoconf se enfoca en `'configure'` y Automake en `'Makefile'`. Es posible crear un Sistema de compilación de GNU completo sin la ayuda de estas herramientas, pero resulta muy engorroso y proclive a errores.

Autoconf

El fichero `'configure'` creado por MacKenzie en 1991 funcionó tan bien que decidió adaptarlo (a mano) para crear ficheros `'configure'` similares para varios paquetes de utilidades de GNU. Ese mismo verano descubrió que Richard Stallman y Richard Pixley estaban desarrollando guiones similares para utilizar en las herramientas de compilación de GNU, así que adaptó los suyos para que soportaran las interfaces en evolución de aquellos.

> “En la medida en que fui recibiendo retroalimentación de los usuarios, incorporé muchas mejoras (…). En la medida en que adapté más paquetes de utilidades GNU para que utilizaran guiones `'configure'`, actualizarlos a mano se volvió impracticable. Rich Murphey, mantenedor de las utilidades gráficas de GNU, me envió un correo diciendo que mis guiones `'configure'` eran geniales, y me preguntaba si tenía alguna herramienta para generarlos que le pudiera enviar. No, pensé, ¡pero debería! Así que comencé a trabajar en cómo generarlos. Y el viaje de la esclavitud de guiones `'configure'` escritos a mano a la abundancia y facilidad de Autoconf comenzó.”

Dado que los ficheros `'configure'` de MacKenzie determinaban automáticamente las capacidades del sistema, sin intervención del usuario, decidió llamar Autoconfig al programa que los generaba. Pero al añadirle el número de la versión, ese nombre resultaría demasiado largo para los antiguos sistemas de ficheros de Unix, así que decidió acortarlo a Autoconf.

Autoconf toma un fichero plantilla (comúnmente llamado `'configure.ac' o 'configure.in'`) para generar el fichero `'configure'` expandiendo las macros M4 y el código Bash en ellos.

Es a partir de estas plantillas que la herramienta propuesta “descubre” las funcionalidades que se pueden habilitar al paquete en tiempo de compilación. Para ello se chequean las macros `AC_ARG_WITH` y `AC_ARG_ENABLE` que especifican qué debe hacer el fichero `'configure'` cuando se le indique (o no) si debe utilizar un paquete externo o habilitar una funcionalidad en específico.

Para cada uno de los paquetes de software externos que puedan ser usados, el fichero plantilla debe incluir una llamada a la macro `AC_ARG_WITH` para detectar si el usuario de `'configure'` pidió usarlo. La sintaxis de la macro se relaciona a continuación:

```
AC_ARG_WITH (
      paquete,
      cadena-de-ayuda,
      [acciones-si-se-especifica],
      [acciones-si-no]
)
```

Análogamente, para cada una de las funcionalidades opcionales se incluye una llamada a la macro `AC_ARG_ENABLE:`

```
AC_ARG_ENABLE (
      funcionalidad,
      cadena-de-ayuda,
      [acciones-si-se-especifica],
      [acciones-si-no]
)
```

Los campos *acciones-si-se-especifica* y *acciones-si-no* son opcionales en ambos casos, puesto que se crean variables `with_`*paquete* y `enable_`*funcionalidad,* respectivamente, que almacenan el valor de la opción especificada por el usuario. Así mismo las variables `withval` y `enableval,` respectivamente, contienen el valor de la opción especificado por el usuario de `'configure'` dentro del contexto de las *acciones-si-se-especifica*. Estos campos relativos a las acciones están conformados por sentencias de Bash.

Otro aspecto a considerar es el uso de las macros `AC_SUBST` y `AM_CONDITIONAL`, que permiten lograr la persistencia de datos entre los ficheros `'configure'` y los `'Makefile'`. Su modo de uso es sencillo:

```
AC_SUBST (
      variable,
      [valor]
)
```

En caso de no especificarse el *valor* a substituir, se tomará el valor que represente el término *variable* en el momento de la invocación de la macro.

```
AM_CONDITIONAL (
      condicional,
      condición
)
```

condicional debe ser un identificador de variable válido, mientras *condición* debe ser un comando válido para utilizar en una sentencia `if` de Bash.

Para determinar si existe o no, en el sistema, alguna biblioteca requerida (y no proveída) por la aplicación a construir se utilizan ls siguientes macros.

```
AC_CHECK_LIB (
      biblioteca,
      función,
      [acciones-si-se-encuentra],
      [acciones-si-no],
      [otras-bibliotecas]
)
```

Esta macro chequea si existe *biblioteca* intentando utilizar un programa que haga una llamada a *función*; *acciones-si-se-encuentra* es una lista de comandos de consola que se ejecutarán si la prueba resulta satisfactoria, análogamente *acciones-si-no* es una lista de comandos de consola que se ejecutarán en caso de fallar la prueba. Si el intentar vincular *biblioteca* resulta en símbolos perdidos que pudieran ser encontrados vinculando con otras bibliotecas, éstas deben ser proveídas en el campo *otras-bibliotecas*.

```
AC_SEARCH_LIBS (
      función,
      bibliotecas-buscadas,
      [acciones-si-se-encuentra],
      [acciones-si-no],
      [otras-bibliotecas]
)
```

Esta macro funciona parecido a la anterior, solo que en este caso busca en la lista de *bibliotecas-buscadas*, alguna que provea una definición de *función*.

Automake

Para proyectos simples que distribuyen todos sus ficheros fuente en un mismo directorio es suficiente tener un solo `'Makefile.am'` que lo construya todo. En proyectos más grandes es común organizar los ficheros en un árbol de directorios.

La técnica más común es compilar estos directorios recursivamente: cada directorio contiene su

'Makefile' (generado a partir de un 'Makefile.am'), y cuando se ejecuta make desde el directorio raiz, entra en cada uno de los subdirectorios para compilar sus contenidos.

En paquetes con subdirectorios el 'Makefile.am' en el directorio raiz debe indicar a Automake qué subdirectorios deben compilarse. Esto se logra a través de la variable SUBDIRS, que contiene una lista de los subdirectorios en los cuales se puede compilar de distintas maneras.

Es posible definir la variable SUBDIRS de manera condicional si solo se quiere construir un subconjunto del paquete completo. Hay dos macros que permiten lograr este efecto: AC_SUBST y AM_CONDITIONAL.

Compilar programas

Para compilar un programa es necesario indicar a Automake qué fuentes son parte de el mismo, y con qué bibliotecas debería vincularse.

En un directorio contenedor de código fuente a compilar en un programa (en oposición a una biblioteca o guión), se usa la primaria PROGRAMS. Los programas pueden ser instalados en bindir, sbindir, libexecdir, pkglibdir, pkglibexecdir, o no instalarse en lo absoluto. Cada uno de estos lugares está representado por un prefijo que se incluye a la primaria PROGRAMS. La forma de especificar esto es:

bin_PROGRAMS = *nombre*

En este caso se dice que se creará un programa llamado *nombre* y será instalado en bindir. Asociado a este programa existen varias variables auxiliares que se nombran según el programa. Todas estas variables son opcionales y tienen valores por defecto razonables. Para especificar los dicheros que contienen el código fuente del programa nombre se escribe una regla como sigue:

*nombre*_SOURCES = *fuente1.c fuente2.c … fuenteN.c fuenteK.h*

Esto provoca que cada uno de los '.c' y '.h' relacionados se compilen en el correspondiente '.o'. Luego se vinculan para producir a '*nombre*'.

Propuesta de Solución

La presente investigación propone el diseño e implementación de una herramienta que, partiendo de las plantillas utilizadas por las Autotools, determine cuáles funcionalidades pueden ser separadas del núcleo del paquete, proporcionando para las mismas información sobre los ficheros compilados que

brindan dicha funcionalidad, de qué paquetes externos dependerían y/o con cuales entrarían en conflicto.

Dicha herramienta se plantea como un intérprete que analice la sintaxis de los referidos ficheros y reconozca, basado en las macros `AC_ARG_WITH` y `AC_ARG_ENABLE` de las plantillas `'configure.in'` o `'configure.ac'`, las distintas características y paquetes que se le pueden incluir al software a compilar y determine según las plantillas `'Makefile.am'` cuáles de éstas opciones generan ficheros que se puedan excluir del núcleo del paquete.

Herramientas a utilizar

Python

Python es un lenguaje de programación dinámico notablemente potente utilizado en una amplia variedad de dominios de aplicación. A menudo comparado con Tcl, Perl, Ruby, Scheme o Java, algunas de sus principales características distintivas incluyen:

- una sintaxis muy clara y legible
- fuertes capacidades de introspección
- orientación a objetos intuitiva
- total modularidad, soportando paquetes jerárquicos
- gestión de errores basada en excepciones
- tipos dinámicos de datos de muy alto nivel
- librerías estándar extensibles y módulos de terceros para casi cualquier tarea
- extensiones y módulos fácilmente escritos en C, C++ (o Java para Jython, o lenguajes .NET para IronPython)
- puede ser incluido dentro de aplicaciones como interfaz de *scripting*

La rápida velocidad de desarrollo y el alto grado de mantenibilidad del código python, así como el nivel de conocimiento adquirido sobre el mismo, fueron detalles muy importantes que se tuvieron en cuenta para la decisión de usarlo como lenguaje de programación para el desarrollo de la herramienta.

PLY

PLY es una implementación en Python de las herramientas de construcción de compiladores Lex y Yacc. Incluye soporte para análisis LALR(1) así como provee validación de datos de entrada, reporte de errores y diagnósticos.

Consiste de dos módulos separados, `'lex.py'` y `'yacc.py'`, que se encuentran en un paquete de Python llamado `ply`. El módulo 'lex.py' se utiliza para dividir la el texto de entrada en una colección de tokens especificados por un conjunto de expresiones regulares. 'yacc.py' se utiliza para reconocer una sintaxis de lenguaje especificada en la forma de una gramática libre de contexto. 'yacc' utiliza análisis LR, genera sus tablas de análisis a través de algoritmos de generación de tablas LALR (por defecto) o SLR.

Metodología de desarrollo

SXP

A partir de un estudio realizado el Polo Software Libre de la Facultad 10 de la Universidad, al cual pertenece el proyecto Nova, incorporó como metodología de desarrollo SXP, desarrollada por miembros del proyecto Unicornios, que incorpora prácticas de XP y SCRUM.

Entorno de desarrollo

Geany

Licenciado bajo la versión 2 de la GNU General Public License, Geany es un Entorno de Desarrollo Integrado (*IDE*, según sus siglas en inglés) pequeño y ligero. Fue desarrollado para proveer un IDE ligero y rápido que solo tiene unas pocas dependencias con otros paquetes. Otra de las metas fue lograr la independencia de un sistema de escritorio específico (como Gnome o KDE). Geany solo requiere las bibliotecas de ejecución de GTK2.

Descripción del proceso de desarrollo

La metodología SXP categoriza los artefactos que se generan durante la vida del proyecto en 5 secciones a saber:

1. **Planificación**: cuyo propósito es establecer la visión, fijar las expectativas y asegurar el financiamiento.

2. **Desarrollo**: que se centra en implementar un sistema listo para entrega mediante una serie de iteraciones .
3. **Entrega**: que consiste en la puesta en operación del sistema desarrollado.
4. **Mantenimiento**: mientras la primera versión se encuentra en producción, el proyecto debe mantener el sistema en funcionamiento al tiempo que desarrolla nuevas iteraciones.
5. **Legal**: que recoge una serie de documentos que dan formalidad al proyecto.

El presente epígrafe se enfoca en los artefactos de *Planificación*, mientras que el siguiente capítulo tratará los generados para el *Desarrollo*. Los artefactos de *Entrega*, *Mantenimiento* y *Legal* no se relacionan en este documento, puesto que el interés principal es centrarse en la herramienta como tal y no hacer una caso de estudio de la metodología en cuestión.

Planificación

La Planificación ocurre, en mayor medida, durante la fase de definición. Se centra en el qué: se establece qué información necesita ser procesada, qué funcionalidades se desea, qué tiempo se necesita para definir un sistema correcto. Durante esta fase se genera un conjunto de artefactos entre los que se pueden mencionar:

- **Plantilla de Concepción del Sistema**: refleja la visión general del producto a implementar, recoge los diferentes roles que intervendrán en el desarrollo así como las tecnologías utilizadas.
- **Plantilla de Lista de Reserva del Producto**: lista priorizada que define el trabajo que se va a realizar en el proyecto.
- **Plantilla de Historias de Usuario**: técnica utilizada en XP para especificar los requisitos de software.
- **Plantilla de Lista de riesgos**: define los posibles riesgos que actuarán sobre el proceso de desarrollo de software, así como la estrategia trazada para mitigarlos.
- **Plantilla de Modelo de Diseño**: define un esbozo inicial del diseño del sistema, sin entrar en especificaciones ni detalles, solo lo que el diseñador necesita para un primer entregable. [9] (Ver Anexo 1. Modelo de diseño)

Las historias de usuario a implementar son:

- Descubrir paquetes candidatos

- Obtener datos por paquete

Historia de Usuario	
Número: 1	**Nombre Historia de Usuario:** Descubrir paquetes candidatos
Modificación de Historia de Usuario Número: 1	
Usuario: Daniel Hernandez Bahr	**Iteración Asignada:** 1
Prioridad en Negocio: Alta	**Puntos Estimados:** 3
Riesgo en Desarrollo: Alto	**Puntos Reales:**
Descripción: El sistema debe identificar los paquetes externos con los que se puede compilar y funcionalidades que se le pueden habilitar al software, así como las dependencias auxiliares y/o conflictos relativos a cada uno.	
Observaciones:	
Prototipo de interfaz:	

Historia de Usuario	
Número: *2*	**Nombre Historia de Usuario:** Obtener datos por paquete
Modificación de Historia de Usuario Número: 1	
Usuario: Daniel Hernandez Bahr	**Iteración Asignada:** 2
Prioridad en Negocio: Alta	**Puntos Estimados:** 3
Riesgo en Desarrollo: Alto	**Puntos Reales:**
Descripción: El sistema debe asignar a cada uno de los paquetes candidatos los ficheros que incluirían.	
Observaciones: Si algún paquete candidato queda sin ficheros asignados es descartado.	
Prototipo de interfaz:	

A modo de cierre, podemos comentar que, después de haber realizado un análisis de las características claves de los ficheros plantilla utilizados por las Autotools ha sido posible planificar un proceso de desarrollo que arroje como resultado una herramienta capaz de proponer qué paquetes han de ser generados a partir del código fuente de una aplicación.

Desarrollo del Sistema

El presente capítulo describe los tres momentos a saber de la fase de desarrollo: Planificación, Implementación y Prueba. Para ello se relacionan diferentes artefactos generados durante la fase.

Esta fase se centra en el cómo, se definen las estructuras de datos, cómo se implementa la funcionalidad dentro de la arquitectura de software, los detalles de los procedimientos, cómo se traduce el diseño del software al lenguaje de programación.

Planificación

Durante la planificación de la fase de desarrollo se generan otros artefactos:

- **Plantilla de Glosario de Términos**: recoge los términos que se relacionan con el sistema y la metodología utilizada que puedan causar dudas al cliente para un mejor entendimiento de todo el proceso de desarrollo de software.
- **Plantilla de Tareas de Ingeniería**: recoge las tareas por Historia de Usuario a realizar.
- **Plantilla de Cronograma de Producción**: recoge las actividades realizadas en el equipo de desarrollo durante la iteración.
- **Plantilla de Plan de Releases**: recoge las iteraciones a realizar con sus características, además del orden de las Historias de Usuario con su planificación estimada para ser implementadas. [10]

Durante la primera iteración se desarrollan las tareas de ingeniería involucradas con la Historia de Usuario #1.

		Tarea de Ingeniería
Número Tarea: 1	**Número Historia de Usuario:** HU-1	
Nombre Tarea: Nombrar paquetes candidatos		
Tipo de Tarea : Desarrollo		**Puntos Estimados:** 1
Fecha Inicio: 5/4/2010		**Fecha Fin:** 11/4/2010
Programador Responsable: Daniel Hernandez Bahr		
Descripción: Se descubre qué características pueden habilitarse al software, así como los paquetes de software externos con los que puede compilarse. Cada característica y paquete se identificará como “Paquete Candidato”.		

Tarea de Ingeniería	
Número Tarea: 2	**Número Historia de Usuario:** HU-1
Nombre Tarea: Trascendencia	
Tipo de Tarea : Desarrollo	**Puntos Estimados:** 1
Fecha Inicio: 12/4/2010	**Fecha Fin:** 18/4/2010
Programador Responsable: Daniel Hernandez Bahr	
Descripción: Se identifican las variables que trascenderán del fichero 'configure' a los 'Makefile', y se determina cuáles están ligadas a qué “Paquetes Candidatos”	

Tarea de Ingeniería	
Número Tarea: 3	**Número Historia de Usuario:** HU-1
Nombre Tarea: Chequear dependencias	
Tipo de Tarea : Desarrollo	**Puntos Estimados:** 1
Fecha Inicio: 19/4/2010	**Fecha Fin:**
Programador Responsable: Daniel Hernandez Bahr	
Descripción: Se determina qué dependencias necesitan los “Paquetes Candidatos”, así como los paquetes con los que entran en conflicto.	

Quedando para la segunda iteración la tarea de ingeniería correspondiente a la segunda Historia de Usuario:

Tarea de Ingeniería	
Número Tarea: 4	**Número Historia de Usuario:** HU-2
Nombre Tarea: Obtener datos por paquete	
Tipo de Tarea : Desarrollo	**Puntos Estimados:** 3
Fecha Inicio:	**Fecha Fin:**
Programador Responsable: Daniel Hernandez Bahr	
Descripción: Se determinan qué ficheros binarios serán generados y se calcula a qué “Paquete Candidato” pertenecen. Aquellos paquetes candidatos a los que no pertenezca ningún fichero, desaparecerán; los que tengan ficheros comunes se unirán.	

Implementación

Los artefactos generados durante la implementación son

- **Plantilla de Código Fuente**: Recoge el código fuente del sistema desarrollado.
- **Plantilla de Estándar de Código**: Recoge el estándar utilizado y su explicación. [11]

Es en esta fase que cae el mayor peso de las iteraciones, dado que aquí es donde se genera el producto como tal. El proceso de codificación es antecedido por un momento de diseño durante el cual se generan distintos diagramas que apoyen la actividad (Ver Anexo 2. Diagramas de Clases).

Como se puede apreciar en los diagramas el sistema implementa una arquitectura en capas, donde cada componente solo puede interactuar con componentes de su misma capa y componentes de la capa inferior.

Prueba

Durante el momento de prueba se generan los artefactos:

- **Plan de Prueba**: recoge la organización de cada una de las Historias de Usuario que serán probadas.
- **Plantilla Caso de Prueba de Aceptación**: en esta plantilla el desarrollador escribe las pruebas realizadas según la Historia de Usuario seleccionada para realizar la comprobación y validación de las funcionalidades del sistema y, de esta forma, saber si está apto para ser liberado. [12]

Las pruebas de aceptación que se ejecutaron sobre el sistema se relacionan a continuación.

Durante la primera iteración se hicieron pruebas de aceptación para las funcionalidades brindadas en la Historia de Usuario #1.

Caso de Prueba de Aceptación	
Código Caso de Prueba: S-1-1	**Nombre Historia de Usuario:** Descubrir paquetes candidatos
Nombre de la persona que realiza la prueba: Daniel Hernandez Bahr	
Descripción de la Prueba: Se verificará que el sistema seleccione el fichero plantilla correctamente.	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución: Se invoca el método `'get_template'` de la clase `'PackageDiscoverer'`.	
Resultado Esperado: Que se obtenga el texto del fichero plantilla, ya sea `'configure.ac'` o `'configure.in'`.	
Evaluación de la Prueba: Satisfactoria	

Caso de Prueba de Aceptación	
Código Caso de Prueba: S-1-2	**Nombre Historia de Usuario:** Descubrir paquetes candidatos
Nombre de la persona que realiza la prueba: Daniel Hernandez Bahr	
Descripción de la Prueba: Se intentará obtener los nombres de los paquetes candidatos.	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución: Se invoca el método `'name_candidates'` de la clase `'PackageDiscoverer'`.	
Resultado Esperado: Que por cada macro `AC_ARG_WITH` y `AC_ARG_ENABLE` en la plantilla del fichero `'configure'` se cree una instancia de la clase `'PkgInfo'` se añada a la lista de paquetes candidatos.	
Evaluación de la Prueba: Satisfactoria	

Caso de Prueba de Aceptación	
Código Caso de Prueba: S-1-3	**Nombre Historia de Usuario:** Descubrir paquetes candidatos
Nombre de la persona que realiza la prueba: Daniel Hernandez Bahr	
Descripción de la Prueba: Se intentará obtener las variables que trascienden a los 'Makefile'.	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución: Se invoca el método `'trascendance'` de la clase `'PackageDiscoverer'`.	
Resultado Esperado: Que por cada macro `AC_SUBST` y `AM_CONDITIONAL` en la plantilla del fichero `'configure'` se verifique qué paquete candidato influye sobre la misma y se añada a la lista `'subst_vars'` de la instancia pertinente en la clase `'PkgInfo'`.	
Evaluación de la Prueba: Satisfactoria	

Caso de Prueba de Aceptación	
Código Caso de Prueba: S-1-4	**Nombre Historia de Usuario:** Descubrir paquetes candidatos
Nombre de la persona que realiza la prueba: Daniel Hernandez Bahr	
Descripción de la Prueba: Se intentará determinar qué dependencias puedan surgir para cada opción que se pueda habilitar.	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución: Se invoca el método `'check_dependencies'` de la clase `'PackageDiscoverer'`.	
Resultado Esperado: Que por cada macro `AC_CHECK_LIB` o `AC_SEARCH_LIBS` en la plantilla del fichero `'configure'` se verifique si algún paquete candidato influye sobre la misma y se añada a la lista `'dependencies'` de la instancia pertinente de la clase `'PkgInfo'`, así como que verfique cuáles entran en conflicto y las añada la lista 'conflicts'.	
Evaluación de la Prueba: Satisfactoria	

Caso de Prueba de Aceptación	
Código Caso de Prueba: S-1-5	**Nombre Historia de Usuario:** Descubrir paquetes candidatos
Nombre de la persona que realiza la prueba: Daniel Hernandez Bahr	
Descripción de la Prueba: Se intentará determinar qué todas las funcionalidades de la clase `'PackageDiscoverer'` se integren correctamente.	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución: Se invoca el método `'discover_packages'` de la clase `'Main'`.	
Resultado Esperado: Que se llene la lista `'candidates'` de la clase `'PackageDiscoverer'` con una instancia de la clase `'PkgInfo'` por cada macro `AC_ARG_WITH` o `AC_ARG_ENABLE` presente en la plantilla del fichero `'configure'`, y que para cada elemento en la lista se tengan las variables de trascendencia relacionadas en la lista `'subst_vars'` así como las dependencias y conflictos relativas en las listas `'dependencies'` y `'conflicts'` respectivamente.	
Evaluación de la Prueba: Satisfactoria	

En la segunda iteración se realizaron las pruebas de aceptación a la Historia de Usuario #2.

Caso de Prueba de Aceptación	
Código Caso de Prueba: S-2-1	**Nombre Historia de Usuario:** Obtener datos por paquete
Nombre de la persona que realiza la prueba: Daniel Hernandez Bahr	
Descripción de la Prueba: Se intentará determinar qué subdirectorios será necesario visitar según las variables de trascendencia que se hayan obtenido para los distintos paquetes candidatos.	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución: Se invoca el método `'get_templates_order'` de la clase `'PackageValidator'`.	
Resultado Esperado: Que según las variables de trascendencia de cada paquete se seleccionen los subdirectorios que serán construidos.	
Evaluación de la Prueba: Satisfactoria	

Caso de Prueba de Aceptación	
Código Caso de Prueba: S-2-2	**Nombre Historia de Usuario:** Obtener datos por paquete
Nombre de la persona que realiza la prueba: Daniel Hernandez Bahr	
Descripción de la Prueba: Se intentará determinar los archivos binarios que pertenecerán a cada paquete candidato según las variables de trascendencia que se hayan obtenido para los mismos.	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución: Se invoca el método `'search_template'` de la clase `'PackageValidator'` para cada uno de las plantillas a examinar.	
Resultado Esperado: Que para cada paquete candidato que genere archivos binarios independientes del núcleo del sistema se llene la lista `'files'` de la instancia correspondiente de la clase `'PkgInfo'`.	
Evaluación de la Prueba: Satisfactoria	

Caso de Prueba de Aceptación	
Código Caso de Prueba: S-2-3	**Nombre Historia de Usuario:** Obtener datos por paquete
Nombre de la persona que realiza la prueba: Daniel Hernandez Bahr	
Descripción de la Prueba: Se intentará mezclar dos paquetes candidatos que generen ficheros binarios comunes entre sí, pero independientes del núcleo del sistema	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución: Se invoca el método `'merge_packages'` de la clase `'PackageValidator'` pasando como argumentos los paquetes candidatos a mezclar.	
Resultado Esperado: Que se obtenga una instancia de la clase `'PkgInfo'` cuyos atributos sean resultado de la unión de los atributos de los paquetes pasados como argumentos a la llamada al método `'merge_packages'` de la calse `'PackageValidator'`.	
Evaluación de la Prueba: Satisfactoria	

Caso de Prueba de Aceptación	
Código Caso de Prueba: S-2-4	**Nombre Historia de Usuario:** Obtener datos por paquete
Nombre de la persona que realiza la prueba: Daniel Hernandez Bahr	
Descripción de la Prueba: Se intentará eliminar los paquetes candidatos que no generen ningún archivo binario independiente del núcleo del sistema.	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución: Se invoca el método `'purge_empty_packages'` de la clase `'PackageValidator'`.	
Resultado Esperado: Que se eliminen las instancias de la clase `'PkgInfo'` cuyo atributo `'files'` contenga una lista vacía.	
Evaluación de la Prueba: Satisfactoria	

Caso de Prueba de Aceptación	
Código Caso de Prueba: S-2-5	**Nombre Historia de Usuario:** Obtener datos por paquete
Nombre de la persona que realiza la prueba: Daniel Hernandez Bahr	
Descripción de la Prueba: Se intentará determinar qué todas las funcionalidades de la clase `'PackageValidator'` se integren correctamente.	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución: Se invoca el método `'get_packages_data'` de la clase `'Main'`.	
Resultado Esperado: Que la lista `'candidates'` contenga solo instancias de la clase `'PkgInfo'` cuyos atributos `'files'` contengan listas de archivos binarios independientes del nucleo del sistema.	
Evaluación de la Prueba: Satisfactoria	

Hemos podido apreciar cómo se diseñó el sistema y que sus partes funcionan de manera correcta e integrada, por lo que se puede concluir que el sistema funciona adecuadamente para el conjunto de paquetes que se eligió para ejecutar las pruebas de aceptación.

Conclusiones

Como resultado del estudio anterior ha sido posible la construcción de un sistema que da respuesta al Problema Científico planteado en el capítulo introductorio del mismo:

> ¿Cómo generar paquetes binarios de forma que se optimice el proceso de instalación de software en Nova?

La herramienta permite:

1. Determinar distintos paquetes como resultado del proceso de compilación.
2. Determinar las dependencias y conflictos entre los mismos y otros paquetes externos.
3. Determinar qué archivos binarios serán contenidos por los distintos paquetes resultantes.

Lo cual logra un menor acoplamiento entre el núcleo de los paquetes y las funcionalidades que proveen, permitiendo a los usuarios del sistema ahorrar ancho de banda en tiempo de descarga de los paquetes de sus repositorios de paquetes de software.

Recomendaciones

1. Extender el sistema de modo que pueda funcionar con sistemas de automatización de compilación distintos del Sistema de Construcción de GNU.
2. Aplicar el uso del sistema a los procesos de compilación de paquetes de software para la distribución cubana de GNU/Linux Nova.

Referencias bibliográficas

[1] "Welcome to openSUSE Build Service"; https://build.opensuse.org/. Consultada en enero 28, 2010.

[2] "Building Packages in Fedora"; http://fedoraproject.org/wiki/Docs/Drafts/BuildingPackagesGuide. Consultada en enero 28, 2010.

[3] "Packaging Guide/Complete"; https://wiki.ubuntu.com/PackagingGuide/Complete. Consultada en enero 28, 2010.

[4] "Guía del nuevo desarrollador de Debian"; http://www.debian.org/doc/maint-guide/ch-start.es.html. Consultada en enero 28, 2010.

[5] McKenzie D, Elliston B y Demaille A. *Autoconf. Creating Automatic Configuration Scripts*. Free Software Foundation, Inc. 2009

[6] "Writing Autoconf Input – Autoconf"; http://www.gnu.org/software/autoconf/manual/html_node/Writing-Autoconf-Input.html#Writing-Autoconf-Input. Consultada en enero 28, 2010.

[7] McKenzie D, Tromey T y Duret-Lutz A. *GNU Automake*. Free Software Foundation, Inc. 2009

[8] Ibídem 5.

[9] Colectivo de Autores. SXP, metodología ágil para proyectos de software libre. Universidad de las Ciencias Informáticas. 2009.

[10] Ibídem 9.

[11] Ibídem 9.

[12] Ibídem 9.

Bibliografía

- “Application lifecycle management”; http://en.wikipedia.org/wiki/Application_lifecycle_management
- McKenzie D, Elliston B y Demaille A. *Autoconf. Creating Automatic Configuration Scripts*. Free Software Foundation, Inc. 2009.
- “Alioth: build-common”; http://alioth.debian.org/projects/build-common.
- “Building Packages in Fedora”; http://fedoraproject.org/wiki/Docs/Drafts/BuildingPackagesGuide.
- “Chroot”; http://en.wikipedia.org/wiki/Chroot.
- “Computer software”; http://en.wikipedia.org/wiki/Software.
- Núñez Camallea, N y Coutin Abalo R. *Diccionario de Informática*. Edit. Científico-Técnica. 2005.
- “Ebuild”; http://es.wikipedia.org/wiki/Ebuild.
- “Entropy”; http://wiki.sabayon.org/index.php?title=Entropy
- “Free and open source software”; http://en.wikipedia.org/wiki/Free_and_open_source_software.
- “Gentoo Linux”; http://en.wikipedia.org/wiki/Gentoo_Linux.
- McKenzie D, Tromey T y Duret-Lutz A. *GNU Automake*. Free Software Foundation, Inc. 2009.
- “GNU Build System”; http://en.wikipedia.org/wiki/Autotools.
- “GNU Toolchain”; http://en.wikipedia.org/wiki/GNU_toolchain
- “Guía del nuevo desarrollador de Debian”; http://www.debian.org/doc/maint-guide/ch-start.es.html.
- “Hardware”; http://en.wikipedia.org/wiki/Hardware.
- Hernández Sampier R, *Metodología de la Investigación*. Tomo 1. Edit. Félix Varela. 2004.
- Hernández Sampier R, *Metodología de la Investigación*. Tomo 2. Edit. Félix Varela. 2004.
- “Package management system”; http://en.wikipedia.org/wiki/Package_management_system.
- “Packaging Guide/Complete”; https://wiki.ubuntu.com/PackagingGuide/Complete.
- “Pretty Good Privacy”; http://en.wikipedia.org/wiki/Pretty_Good_Privacy.
- “Sabayon Linux”; http://en.wikipedia.org/wiki/Sabayon_Linux.

- “Scripting language”; http://en.wikipedia.org/wiki/Scripting.
- “Software framework”; http://en.wikipedia.org/wiki/Software_framework.
- “Superuser”; http://en.wikipedia.org/wiki/Superuser.
- Colectivo de Autores. SXP, metodología ágil para proyectos de software libre. Universidad de las Ciencias Informáticas. 2009.
- “Tar”; http://es.wikipedia.org/wiki/Tarball.
- “Welcome to openSUSE Build Service”; https://build.opensuse.org/.
- “Writing Autoconf Input – Autoconf”; http://www.gnu.org/software/autoconf/manual/html_node/Writing-Autoconf-Input.html#Writing-Autoconf-Input.

Anexos

Modelo de diseño

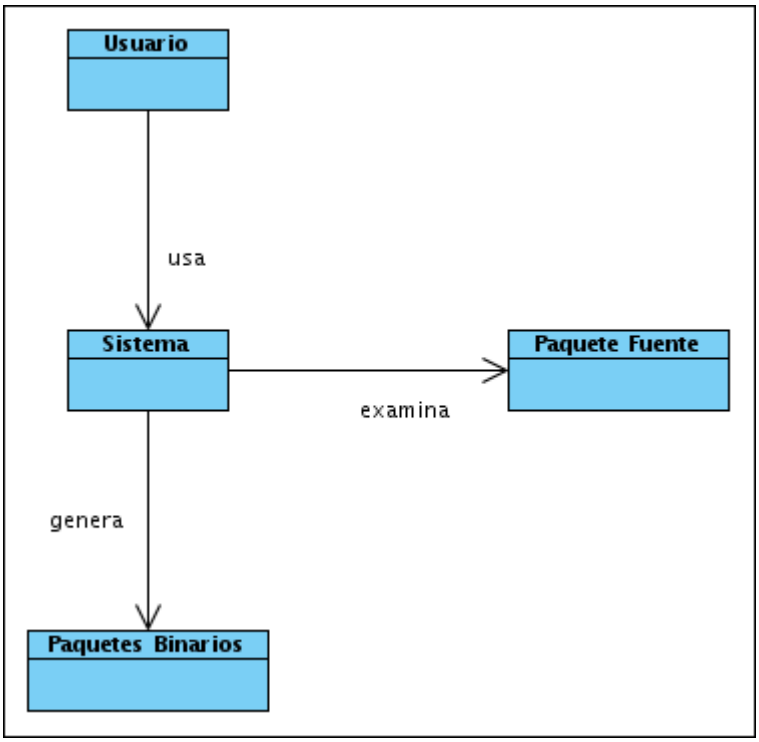

Figura 1: Modelo de diseño

Diagramas de Clase

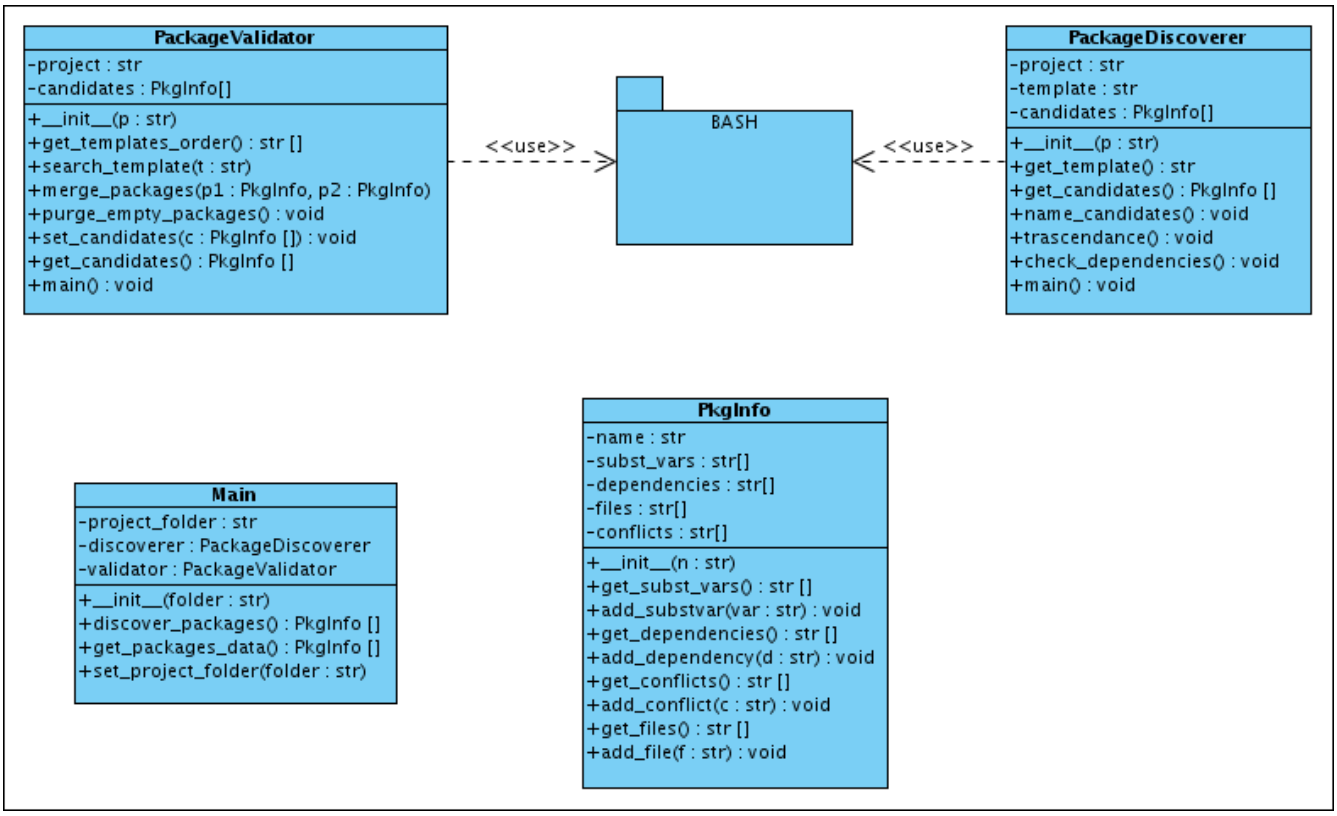

Figura 2: Diagrama de clases del Sistema

Fig.2 Diagrama de clases del paquete BASH

Glosario de Términos

ALM (Application lifecycle management): es la unión de la gestión de negocios y la ingeniería de software, hecha posible por herramientas que facilitan e integran la gestión de requisitos, arquitectura, codificación, prueba, traceo y gestión de liberaciones.

Autotools: ver *GNU Build System*.

Chroot: es una operación que cambia el aparente directorio raíz para el proceso actualmente en ejecución y sus hijos.

Código fuente: Instrucciones y expresiones de un programa, escritas por el programador en un lenguaje determinado. No es ejecutable directamente por la computadora. Puede ser escrito en un editor de texto y guardado en un archivo que luego hay que convertir a código máquina para que la computadora "lo entienda". Cuando el código fuente de un programa es de libre acceso se dice que es código abierto.

Common Debian Build System: sistema de compilación para paquetes de Debian basados en Makefiles. Es muy flexible; puede ser utilizado para crear un sistema de compilación personalizado o simplemente utilizar reglas estándar y obtener un sistema de compilación en cuatro líneas de código.

Compilación: Proceso durante el cual se traduce un lenguaje de alto nivel a un programa de lenguaje máquina.

Consola: Intérprete de comandos y lenguaje de programación que permite interactuar con el sistema operativo y ejecutar programas.

Distro: apócope de Distribución; se refiere a las distribuciones de GNU/Linux (p.e: Debian, Fedora, Gentoo, Nova, etc)

Ebuild: fichero de procesamiento por lotes especializado, creado por el proyecto Gentoo Linux para usarlo con el sistema de mantenimiento de software: Portage. Es una forma automática de compilar e instalar software.

Entropy: gestor de paquetes binarios de Sabayon Linux. Es una infraestructura completa, compuesta por un cliente de línea de comandos (Equo), un cliente gráfico (Sulfur) y las aplicaciones servidoras Reagent y Activator.

FOSS (Free and Open Source Software): software libremente licenciado para permitir el derecho de los usuarios a usarlo, estudiarlo, modificarlo y mejorar su diseño mediante la disponibilidad del código fuente.

Framework: abstracción en la cual código común que provee funcionalidades genéricas puede ser re-escrito o especializado selectivamente por el usuario de forma que provea funcionalidades específicas.

Gentoo: es un sistema operativo construido sobre un kernel Linux y basado en el sistema de gestión de paquetes Portage. Es distribuido como software libre y de código abierto, pero incluye algunos paquetes privativos. A diferencia de distribuciones de software convencionales, el usuario compila su software localmente, de acuerdo a una configuración escogida.

GNU Build system: conjunto de herramientas de programación desarrolladas por el proyecto GNU. Están diseñadas para asistir el trabajo de portabilizar varios paquetes de código fuente a muchos sistemas tipo Unix. Es parte de GNU Toolchain y es ampliamente utilizado en muchos paquetes de software libre y código abierto. Las herramientas comprendidas por el GNU Build System son software libre licenciadas bajo la Licencia Pública General de GNU (GNU General Public License, o GPL) con excepciones especiales de licencia que permiten el uso del GNU Build System con software privativo.

GNU Toolchain: término que agrupa a una serie de proyectos que contienen las herramientas de programación producidas por el proyecto GNU. Estos proyectos forman un sistema integrado que es usado para programar tanto aplicaciones como sistemas operativos. Componente vital en el desarrollo del núcleo Linux, el desarrollo del BSD y software para sistemas empotrados. Partes del toolchain de GNU también son usadas en Solaris Operating Environment y la programación de Microsoft Windows con Cygwin y MinGW/MSYS.

Hardware: término general para los artefactos físicos de una tecnología. Puede aplicarse a los componentes físicos de un sistema de cómputo.

PGP (Pretty Good Privacy): programa informático que provee privacidad criptográfica y autenticación. A menudo se utiliza para firmar, encriptar y desencriptar correos electrónicos para aumentar la seguridad de las comunicaciones por esta vía. Fue creado por Philip Zimmermann en 1991.

Scripting: lenguaje de programación que permite controlar una o más aplicaciones de software. Los “scripts” o ”guiones” son distintos del núcleo del código de la aplicación, puesto que usualmente están escritos en un lenguaje diferente y a menudo son creados, o al menos modificados, por el usuario final.

Sistema de Gestión de Paquetes: colección de herramientas que automatizan el proceso de instalación, actualización, configuración y eliminación de paquetes de software en una computadora. Distribuciones de Linux y otros sistemas tipo Unix típicamente consisten en cientos

o incluso miles de diferentes paquetes de software; en el primer caso un sistema de gestión de paquetes es algo agradable, en el último es esencial.

Software: término general fundamentalmente utilizado para datos almacenados digitalmente, tales como programas de computadoras y otros tipos de información leída y escrita por computadoras. El término fue acuñado de manera que contrastara con el término *hardware* más antiguo. En contraste con el hardware el software es intangible, significando que “no puede ser tocado”. En ocasiones el término se utiliza más estrechamente para referirse únicamente a aplicaciones de software.

Superusuario: cuenta de usuario especial utilizada en la administración del sistema.

Tarball: formato de archivos ampliamente utilizado en entornos UNIX.