Infraestructura distribuida para la construcción de paquetes Debian

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

Infraestructura distribuida para la construcción de paquetes Debian

José Luis Sanroma Tato

Septiembre, 2014

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Tecnologías y Sistemas de Información

PROYECTO FIN DE CARRERA

Infraestructura distribuida para la construcción de paquetes Debian

Autor: José Luis Sanroma Tato
Director: Dr. Francisco Moya Fernández

Septiembre, 2014

José Luis Sanroma Tato

Ciudad Real – Spain

E-mail: josel.sanromatato@gmail.com
Web site: `http://www.icebuilder.org`

© 2014 José Luis Sanroma Tato

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Se permite la copia, distribución y/o modificación de este documento bajo los términos de la Licencia de Documentación Libre GNU, versión 1.3 o cualquier versión posterior publicada por la *Free Software Foundation*; sin secciones invariantes. Una copia de esta licencia esta incluida en el apéndice titulado «GNU Free Documentation License».

Muchos de los nombres usados por las compañías para diferenciar sus productos y servicios son reclamados como marcas registradas. Allí donde estos nombres aparezcan en este documento, y cuando el autor haya sido informado de esas marcas registradas, los nombres estarán escritos en mayúsculas o como nombres propios.

TRIBUNAL:

Presidente:

Secretario:

Vocal:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	**SECRETARIO**	**VOCAL**
Fdo.:	Fdo.:	Fdo.:

Resumen

Desde hace algunos años, empresas, universidades, grupos de investigación, o similares, que se dedican al desarrollo de software, desarrollan sus propios paquetes que posteriormente son usados por sus propios trabajadores o clientes. Estos paquetes de software (como todo software), necesitan ser servidos, mantenidos y actualizados de alguna forma. Para ello se usan repositorios. En el caso del Software Libre, una de las opciones es utilizar repositorios no oficiales de Debian que gestionan paquetes Debian. Estos repositorios no oficiales pueden crearse con la ayuda de *reprepro* que permite al usuario crear un repositorio personal con todas las características de uno oficial.

En lugares como el Laboratorio de investigación ARCO de la Escuela Superior de Informática de la Universidad de Castilla-La Mancha, existen computadores de diferentes arquitecturas debido a la renovación que van sufriendo los equipos con el paso del tiempo. Por ello, un problema típico que existe es la construcción de un paquete, por ejemplo, con arquitectura *amd64*. En el momento que se sube al repositorio, sucederá que a este paquete solo podrán acceder los usuarios de *amd64*. Si un usuario de *i386* quiere acceder a ese paquete, tiene que conseguir el código fuente, construirlo y volver a subir el paquete al repositorio. Por lo tanto, es un requisito contar con al menos un computador con esa arquitectura que pueda utilizarse para construir el paquete y que ese usuario, o alguien que sepa hacerlo, se dedique a construirlo usando ese computador u otro con la misma arquitectura. Con lo cual existe una necesidad, y es tener una versión de los paquetes compatible con cada una de las arquitecturas soportadas en el laboratorio para que pueda ser usada por los trabajadores.

Aunque lo ideal sería tener una infraestructura dedicada con al menos un computador de cada una de las arquitecturas soportadas, en pequeños entornos de trabajo como el laboratorio ARCO, llegamos al primer problema, y es que existen recursos limitados y esto no es posible debido a que:

- No se dispone de computadores con el propósito de estar disponibles y dedicados para construir paquetes.
- Los computadores con los que se cuenta podrían no estar siempre disponibles cuando se necesiten para realizar la construcción, bien por estar apagados, bien por estar siendo usados por los trabajadores o cualquier otra circunstancia. Por lo tanto no se puede saber en que instante va a haber un computador disponible.

Algunas veces los usuarios pueden necesitar características que están solamente presentes en la rama *unstable* o incluso *experimental*. Por ejemplo un programador de la biblioteca Orca puede necesitar la versión más actual del paquete ZeroC Ice pero no tiene la necesidad de actualizar todo el sistema a *unstable*. Actualmente los usuarios usan *apt-pinning* para solucionar el problema, pero puede llevar a actualizaciones de paquetes críticos(libc, bibliotecas de tiempo de ejecución de C++) y los binarios pueden acabar enlazados con diferentes versiones de una misma biblioteca. A menudo, los *debian developers* vuelven a construir

el paquete desde el código fuente de *unstable* o *experimental* en un entorno *pbuilder* para *stable* y crean repositorios personalizados.

La *Debian Autbuilder Network* es la infraestructura dedicada de Debian que se encarga tanto de la construcción de paquetes para distintas arquitecturas, como de la distribución de los paquetes a cada uno de los repositorios oficiales de las distintas arquitecturas soportadas por la distribución. Está compuesta por varias máquinas que utilizan *buildd* para recoger los paquetes del archivo de Debian y reconstruirlos.

Esta infraestructura funciona muy bien en Debian porque tienen máquinas dedicadas única y exclusivamente a la construcción de paquetes para servirlos en los repositorios oficiales, siguiendo la política de Debian, y además, tienen unas herramientas (como *wanna-build*, *buildd*, *sbuild* y *quinn-diff*) hechas por y para la infraestructura de Debian. Pero esta infraestructura es difícil de configurar, excede las necesidades de los pequeños entornos (comentadas al principio) y además éstos, requieren de otras necesidades distintas que no se contemplan aquí debido a que no se dispone de los mismos recursos o porque la política de la empresa tiene otras necesidades. Por lo tanto, en este caso *buildd* no cumple con las necesidades y no nos sirve. Se necesita otra infraestructura que sí cumpla con las necesidades de ARCO y es por lo que se propone este proyecto.

Para dar solución a estos problemas, este proyecto se ha ido desarrollando cumpliendo una serie de objetivos que cubren las necesidades listadas anteriormente:

- Diseño de una arquitectura distribuida P2P para donar ciclos de CPU, construcción de paquetes Debian, construcción compartida en un entorno dado y monitorización del proceso, además de ofrecer balanceo de carga y transparencia de localización para saber qué computadores están encendidos, y destinar la construcción de paquetes Debian a aquellos computadores que tengan menos carga de CPU.
- Desarrollo de nodos de construcción, que harán de entornos limpios, basados en máquinas virtuales en lugar de entornos *chroot* para facilitar la disponibilidad de los recursos (la construcción de paquetes se puede parar, reanudar o migrar a otro nodo compatible). Además, un único computador puede ejecutar varios nodos con el fin de construir paquetes para distintas arquitecturas.
- Implementación de construcción de paquetes mediante transacciones, que gracias ellas, el sistema puede recuperarse frente a fallos tanto intencionados (cuando un trabajador termina su jornada laboral y apaga su equipo) como no intencionados (como apagones de luz).
- Repositorios *backports* personalizados impulsados por las necesidades de los usuarios en lugar de la política de Debian.

Abstract

Long time ago, companies, universities, research groups, that dedicates to software development develops their own packages which later are used by their own workers or customers. These software packages as all software needs to be maintained, used and updated somehow, for this pourpose repositories can be used. In the case of Free Software, and in the case that I am going to present, one of the options is to use non-offiial Debian repositorioes which manages Debian packages. These non-official repositories can be created with the help of reprepro, which allows the user to create a personal repository with the official one features.

At some places as ARCO research lab of the computer science faculty of University of Castilla-La Mancha, computers of differents architectures exists due to the undergoing renovation over time. Thereby, a typical problem is when someone built a package, for example with *amd64* architecture, then is uploaded to the repository, and happens that this package is only available for *amd64* users. If an *i386* user wants to accesss to the package, he has to get the source code, built it and upload again the package to the repository. Therefore, one requirement is have at least one computer with that architecture which can be used to compile the package and that user, or someone who knows how to do it, compile it using that computer or another one with the same architecture.

Although ideally would be have an infrastructure dedicated with at least one computer of each architecture supported by the lab, in small workplaces as ARCO research lab, we reach to the first problem, there are limited resources and this is not posible due to:

- No computers are available in order to be available and dedicated to compiling packages.
- Computers which we count with may not always be available when they are nedded for the compilation, because they are turned off or because they can being used by workers or anything else.

Sometimes users may require certain features which are only present in *unstable* or even in *experimental*. For example, a developer of the Orca library may require a bleeding edge ZeroC Ice package but there is no need to upgrade the whole system to unstable. Currently users may use apt pinning but this may lead to unexpected upgrades to some critical packages (libc, C++ compiler runtime libraries) and binaries may end up linked to different versions of the same library. Most often developers rebuild from the unstable or experimental source package in a stable *pbuilder* environment and create custom repositories.

The *Debian Autobuilder Network* is dedicated Debian infrastructure that takes care of compiling packages for different architectures, and also the distribution of packages to each of the official repositories of the various architectures supported by the Debian distribution. It consists of several machines using *buildd* to collect packages from the Debian archive and rebuild them.

This infrastructure works great on Debian because they have machines dedicated solely to the compilation of packages to serve them in the official repositories, following the Debian policy, and also they have some tools made by and for Debian infrastructure. But this infrastructure is difficult to setup, exceeds the needs of small environments (discussed earlier in this section) and besides these, require others different needs that are not covered here because they do not have the same resources or because the company policy has other needs. Therefore, in our case buildd does not fulfill our needs and is not useful.

This project aims at streamlining the process to prepare custom backports which are driven by user requirements rather than imposed policies. It may also be used to help Debian package building infrastructure.

To offer a solution to this problems major milestones of this project are:

- Design of a P2P distributed architecture to donate computing power, issue package build requests, share packages built on a given environment, and monitor the processes.
- Develop build nodes based on virtual machines rather than chroot environments in order to allow easier availability management in a P2P environment (partial builds may be paused, resumed or even migrated to another compatible node). Besides, a single machine may be running several build nodes for different architectures.
- Prepare custom backports which are driven by user requirements rather than imposed policies.
- Packages built using ACID. This allows a transactional system which is also crash recovery.

Índice general

Resumen XI

Abstract XIII

Índice general XV

Índice de cuadros XXI

Índice de figuras XXIII

Índice de listados XXV

Listado de acrónimos XXIX

Agradecimientos XXXI

1. Introducción 1

1.1. Estructura del documento 2
1.2. Nombre y página web 4

2. Antecedentes 5

2.1. El sistema operativo Debian GNU/Linux 5
2.1.1. El funcionamiento interno de Debian GNU/Linux 6
2.1.2. Repositorios en Debian 7
2.1.3. El ciclo de vida de una versión de un paquete Debian 9
2.2. El paquete Debian 11
2.2.1. Elegir el paquete 12
2.2.2. Estructura básica 12
2.2.3. Archivos necesarios en el directorio `debian` 12
2.3. Sistemas distribuidos para construir paquetes Debian 17
2.3.1. Debian Autobuilder Network 18

2.3.2. Pybit 22
2.4. Herramientas de empaquetado en Debian 23
2.4.1. Scripts de empaquetado en Debian 23
2.4.2. El entorno limpio 23
2.4.3. Otras herramientas para la construcción de paquetes 25
2.5. BOINC 25
2.5.1. Objetivos de BOINC 26
2.5.2. Algunos proyectos que usan BOINC 26
2.6. Middlewares orientados a objetos 27
2.6.1. ZeroC ICE 28
2.6.2. CORBA 31
2.6.3. Java RMI 33

3. Objetivos **35**
3.1. Objetivo general 35
3.1.1. Objetivos principales 35
3.1.2. Objetivos secundarios 36

4. Arquitectura del sistema **37**
4.1. Requisitos 37
4.2. Arquitectura 37
4.2.1. Developer 38
4.2.2. Repositorio 38
4.2.3. Cliente 39
4.2.4. Manager 39
4.2.5. Workers 40

5. Metodología y herramientas **41**
5.1. SCRUM 41
5.1.1. Fases de Scrum 42
5.1.2. Prácticas 43
5.2. Herramientas 45
5.2.1. Herramientas para la gestión de proyectos 45
5.2.2. Lenguajes de programación 45
5.2.3. Control de versiones 46
5.2.4. Herramientas de desarrollo 46

5.2.5. Herramientas y tecnologías de bases de datos 47
5.2.6. Herramientas de virtualización 48
5.2.7. Herramientas para realización de pruebas 48
5.2.8. Documentación 48
5.2.9. Sistemas Operativos 49
5.2.10. Hardware empleado 49

6. Desarrollo del proyecto **51**
6.1. Fase Pre-game 51
6.1.1. Requisitos: Product Backlog 52
6.2. Iteración 0: El sistema base 56
6.2.1. Análisis 57
6.2.2. Diseño 60
6.2.3. Implementación 62
6.2.4. Pruebas 64
6.2.5. Entrega 64
6.3. Iteración 1: Entornos limpios para construir paquetes 65
6.3.1. Análisis 68
6.3.2. Diseño 71
6.3.3. Implementación 74
6.3.4. Pruebas 75
6.3.5. Entrega 76
6.4. Iteración 2: Firmado de paquetes: The GNU Privacy Guard 78
6.4.1. Análisis 80
6.4.2. diseño 80
6.4.3. Implementación 81
6.4.4. Pruebas 82
6.4.5. Entrega 82
6.5. Iteración 3: Persistencia de objetos con Freeze 83
6.5.1. User Stories 83
6.5.2. Análisis 84
6.5.3. Diseño 91
6.5.4. Implementación 93
6.5.5. Pruebas 94
6.5.6. Entrega 94
6.6. Iteración 4: El repositorio de paquetes Debian 95

6.6.1. User stories 95
6.6.2. Análisis 96
6.6.3. Diseño 97
6.6.4. Implementación 97
6.6.5. Pruebas 99
6.6.6. Entrega 99
6.7. Iteración 5: El manager para organizar la construcción de paquetes 100
6.7.1. User Stories 100
6.7.2. Análisis 101
6.7.3. Diseño 104
6.7.4. Implementación 105
6.7.5. Pruebas 108
6.7.6. Entrega 108
6.8. Iteración 6: Worker para realizar la construcción de paquetes 109
6.8.1. User Stories 109
6.8.2. Análisis 110
6.8.3. Diseño 111
6.8.4. Implementación 112
6.8.5. Pruebas 113
6.8.6. Entrega 113
6.9. Iteración 7: Despliegue con IceGrid 115
6.9.1. User Stories 115
6.9.2. Análisis 116
6.9.3. Diseño 117
6.9.4. Implementación 117
6.9.5. Pruebas 119
6.9.6. Entrega 120

7. Conclusiones y trabajo futuro **121**
7.1. Conclusiones 121
7.2. Conclusiones personales 122
7.3. Trabajo futuro 122
7.3.1. Infraestructura distribuida y oportunista para construir paquetes Debian confiables 123
7.3.2. Soporte a más arquitecturas 123
7.3.3. Dependencias con repositorios 123

7.4. Premios y honores 124

A. Manual de usuario. Instalación de un Nodo **129**

A.1. Introducción 129

A.1.1. Estructura del manual 129

A.2. ¿Qué es IceBuilder Node? 129

A.2.1. ¿Por qué Icebuilder Node? 130

A.3. Requisitos 131

A.3.1. Requisitos de hardware 131

A.3.2. Paquetes necesarios 131

A.3.3. Requisitos de ejecución 132

A.3.4. El archivo preseed 133

A.4. Instalación 133

A.4.1. Instalación de máquinas virtuales 133

A.5. Configuración 134

A.5.1. Configuración de máquinas virtuales 134

A.5.2. Configuración de sudo 135

A.5.3. Configuración de repositorios 136

A.5.4. Configuración de Grub 137

A.5.5. Prueba y ejecución 137

B. Creación de un repositorio de paquetes Debian **139**

B.1. Instalación 139

B.2. Configuración 140

B.2.1. Configuraración de conf/distributions 140

B.2.2. Configuración de conf/pulls 141

B.2.3. Configuración conf/incoming 141

B.3. Nociones básicas del uso del repositorio 142

B.4. Hacer el repositorio accesible 142

B.4.1. Exportar la clave 142

B.4.2. Importar la clave 142

B.5. Utilizando el repositorio 143

B.5.1. Firmado de paquetes 143

B.5.2. Subida de paquetes al repositorio 143

B.5.3. Añadir más arquitecturas 144

C. Construcción de un paquete Debian **145**
C.1. Primeros pasos 145
C.2. Estructura básica 146
C.2.1. Archivos necesarios en el directorio `debian` 146
C.3. La construcción del paquete 151
C.4. Actualización del paquete 151
C.4.1. Nueva versión del paquete 152
C.4.2. Nueva versión del autor 152
C.4.3. Nueva versión del programa fuente 152

D. Dependencias de un paquete Debian **153**
D.0.4. Repositorio de paquetes Debian 154
D.0.5. Campo architecture 154
D.0.6. Sintaxis en los campos de relación 155
D.1. Dependencias en paquetes Debian 156
D.1.1. Dependencias binarias 157

E. GNU Free Documentation License **159**
E.0. PREAMBLE 159
E.1. APPLICABILITY AND DEFINITIONS 159
E.2. VERBATIM COPYING 160
E.3. COPYING IN QUANTITY 160
E.4. MODIFICATIONS 160
E.5. COLLECTIONS OF DOCUMENTS 161
E.6. AGGREGATION WITH INDEPENDENT WORKS 161
E.7. TRANSLATION 161
E.8. TERMINATION 162
E.9. FUTURE REVISIONS OF THIS LICENSE 162
E.10. RELICENSING 162

Bibliografía **163**

Índice de cuadros

6.1. Iteración 0. 53
6.2. Iteración 1. 54
6.3. Iteración 2. 54
6.4. Iteración 3. 54
6.5. Iteración 4. 55
6.6. Iteración 5. 55
6.7. Iteración 6 55
6.8. Iteración 7. 56

Índice de figuras

2.1. «Debian developers» en el mundo 7
2.2. Camino de un paquete en Debian 11
2.3. Boceto de *Debian Autobuilder Network* [JS12] 19
2.4. Diagrama de flujo de *Wanna-build* 22
2.5. invocación de un método en ICE 27
2.6. Estructura cliente-servidor en ICE 28
2.7. Arquitectura CORBA (David Villa [Vil09]) 32
2.8. Estructura RMI [RMI09] 33

4.1. Developers 38
4.2. Ejemplo de árbol de construcción 39
4.3. Woker 40

5.1. Scrum Process [PAW13] 43
5.2. Esquema de un sprint en Scrum 44

6.1. Sistema base formado por un computador y varias máquinas virtuales . . . 58
6.2. Captura que muestra la interfaz *virbr1* 61
6.3. Arquitectura de un Nodo y del sistema 68
6.4. Firma de claves gpg 78
6.5. Diagrama de clases del patrón composite. 89
6.6. Diagrama de clases del patrón command. 90
6.7. Árbol de dependencias 102
6.8. Diagrama UML composite-command 104
6.9. Paquetes construidos para dos arquitecturas 114
6.10. Máquinas virtuales instaladas 114
6.11. *IceGrid* con los nodos del sistema. 119

7.1. Dependencias con otros repositorios. 124
7.2. Premio al mejor proyecto innovador 125

A.1. Esquema del sistema 130
A.2. Máquinas virtuales. 134

Índice de listados

2.1. debian/control paquete Debian . . . 13
2.2. changelog paquete Debian . . . 15
2.3. copyright paquete Debian . . . 16
2.4. debian/rules paquete Debian . . . 17
2.5. Ejemplo de interfaz Slice . . . 30
6.1. Fichero de configuración de la red net-icebuilder.xml . . . 60
6.2. Fichero de configuración del pool poolt-icebuilder.xml . . . 62
6.3. Hello.ice . . . 62
6.4. Servidor «HelloWorld» distribuido en Python . . . 62
6.5. Cliente «HelloWorld» distribuido en Python . . . 63
6.6. Fichero de configuración preseed . . . 71
6.7. Ficheros souces.list . . . 73
6.8. Fichero `/etc/apt/preferences` . . . 73
6.9. Conexión `SSH` . . . 74
6.10. Orden para actualizar la máquina virtual . . . 74
6.11. Creación snapshot . . . 74
6.12. Obtención del código fuente del paquete . . . 74
6.13. Entrada al directorio para realizar la construcción . . . 75
6.14. Obtener dependencias de construcción . . . 75
6.15. Construcción del paquete . . . 75
6.16. Volver al estado anterior . . . 75
6.17. Apagar máquina virtual . . . 75
6.18. log de construcción de paquetes . . . 76
6.19. DebBuilder.ice . . . 91
6.20. CommandType.ice . . . 92
6.21. interruptCallBack . . . 94
6.22. dupload . . . 97
6.23. Órdenes para construir paquetes . . . 98

6.24. Slice para el manager 104
6.25. Paquete Debian *icebuilder-p1*. 106
6.26. Paquete Debian *icebuilder-p2*. 106
6.27. Paquete Debian *icebuilder-p3*. 106
6.28. Paquete Debian *icebuilder-p4*. 107
6.29. Paquete Debian *icebuilder-p5*. 107
6.30. worker.ice 111
6.31. workerPersistent.ice 111
6.32. operación execute 112
6.33. script que instala una máquina virtual 113
6.34. Configuración del Locator 117
6.35. Configuración del *Registry* 117
6.36. Configuración del *Registry* (continuación) 117
6.37. Configuración del «Worker1» 118
6.38. Configuración del «Worker2» 118
6.39. Makefile para lanzar los nodos de *IceGrid* 118
6.40. Órden para lanzar los nodos junto con *Icegrid*. 119
A.1. Configuración de red virtual. 132
A.2. Configuración de *sudo*. 135
A.3. Configuración de repositorios. 136
A.4. Configuración de repositorios (continuación). 136
A.5. Configuración de *apt-pinning*. 137
B.1. Fichero `conf/distributions` 140
B.2. Fichero `conf/pulls` 141
B.3. Fichero `conf/incoming` 141
B.4. Fichero `.dupload.conf` 143
B.5. Fichero .dput.cfg 144
C.1. debian/control paquete Debian 146
C.2. changelog paquete Debian 149
C.3. copyright paquete Debian 149
C.4. debian/rules paquete Debian 151
D.1. Fichero *debian/control* del paquete *sl*. 153
D.2. Información del paquete fuente *sl*. 154
D.3. Información del paquete binario *sl*. 154
D.4. Relaciones entre las versiones de los paquetes. 155

D.5. relaciones entre arquitecturas. 156
D.6. relaciones entre arquitecturas independientes. 156
D.7. relaciones entre arquitecturas independientes. 156

Listado de acrónimos

ACID	Atomicity, Consistency, Isolation, Durability
Arco	Arquitectura y Redes de Computadores
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
APT	Advanced Package Tool
ASCII	American Standard Code for Information Interchange
BASH	Bourne Again Shell
BOINC	Berkeley Open Infrastructure for Network Computing
BSD	Berkeley Software Distribution
C++	Lenguaje de programación C++
CVS	Concurrent Version System
CORBA	Common Object Request Broker Architecture
CUSL	Concurso Unoversitario de Software Libre
DBTS	Debian Bug Tracking System
DFSG	Debian Free Software Guidelines
DFS	Depth First Search
EMACS	Editor MACroS
FSF	Free Software Foundation
FTP	File Transfer Protocol
GFDL	GNU Free Documentation Document License
GIMPS	Great Internet Mersenne Prime Search
GIOP	General Inter-ORB Protocol
GNOME	GNU Network Object Model Environment
GNU	GNU is Not Unix
GPG	The GNU Privacy Guard
GPL	General Public License

GNOME	GNU Network Object Model Environment
GSoC	Google Summer of Code
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
Ice	Internet Communication Engine
IP	Internet Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
LGPL	Lesser General Public License
MAC	Media Access Control
NASA	National Aeronautics and Space Administration
PFC	Proyecto Final de Carrera
OIS	Objetive Interface System
OMG	Object Management Group
ORB	Object Request Broker
PFC	Proyecto Final de Carrera
RAM	Random Access Memory
QEMU	Quick Emulator
RMI	Java Remote Method Invocation
RPM	RPM Package Manager
Slice	Specification Language for Ice
SSH	Secure Shell
TAO	The ACE ORB
TCP	Transmission Control Protocol
VCS	Version Control System
VBOX	Virtual Box
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

Agradecimientos

«Las organizaciones gastan millones de dólares en firewalls y dispositivos de seguridad, pero tiran el dinero porque ninguna de estas medidas cubre el eslabón más débil de la cadena de seguridad: la gente que usa y administra los ordenadores»
Kevin Mitnick

Este PFC es una realidad gracias a muchas personas. Seguramente me olvide de nombrar a alguna. Si alguien que no ha sido nombrado cree que está en este grupo, le pido perdón junto con una caña. Aunque quizás ahora habrá gente que prefiera estar en el grupo de los «olvidados»...

Lo primero de todo es agradecer a mi familia y a mi padres, ellos me han dado la oportunidad de hacer lo que me gusta y también dedicarme a ello.

Esther, a tí también tengo que darte las gracias, que además de muchos buenos momentos y de haberte leído la memoria entera, me has enseñado el lado oculto de la música con tu violoncello. Una parte de este proyecto también es tuya.

También quisiera dar las gracias a Paco, mi director de proyecto, por haberme permitido trabajar con él en este PFC que tanto me ha gustado, y haber disfrutado de su infinita sabiduría junto con sus valiosos consejos. Este proyecto nunca lo hubiese terminado sin tu ayuda, gracias.

Quiero también agradecer a Juan Carlos el abrirme las puertas del laboratorio ARCO, el cual se ha convertido en mi segunda casa estos dos últimos años.

A mis compañeros de laboratorio me gustaría darles las gracias a todos por igual, en mayor o menor medida todos ellos han contribuido con alguna idea o consejo a que este proyecto salga adelante. Los días se han hecho muy llevaderos con vosotros, con un magnifico ambiente en el laboratorio. Aunque no os nombre todos sabéis quienes sois, además así ninguno me pide estar en el grupo de la caña gratis. No quiero dejar escapar la oportunidad para recordaros que no debe perderse el espíritu, ya sabéis: «No hay otro sistema que GNU y Linux es uno de sus núcleos».

Mis compañeros de carrera también tienen un hueco aquí, Pablo, Eu, Luis Carlos, Leandro, Luis, Luismi etc. Con todos ellos he pasado mucho tiempo realizando prácticas, y trabajos y, por supuesto, ¡muchas fiestas!.

En la universidad y a lo largo de mi vida he tenido muchos profesores, todos ellos, para bien o para mal, también han contribuido a que este proyecto sea una realidad con sus enseñanzas. De este grupo me gustaría destacar a los profesores de física de la ESI, por su infinita paciencia con nosotros.

A los del grupo ibérico que conocí en Noruega... que os voy a decir, no habéis participado directamente en esto, pero si a mejorar mi nivel de inglés y de portugués. Espero que no os

sigáis solidarizando los unos con los otros y paréis ya de romperos los dientes, por favor.

El Concurso Universitario de Software Libre me dio la oportunidad de presentar el proyecto en la Universidad de Sevilla y de darle difusión. Me permitió pasar unos días fantásticos en buena compañía junto con otros compañeros de las distintas universidades españolas. Es por ello que desde aquí le agradezco todo lo que hicieron a todas las personas que forman parte de él.

No puedo terminar esta sección sin nombrar a Richard Stallman y mostrar mi agradecimiento hacia él, por comenzar el proyecto GNU y sin el cual NADA de este proyecto hubiese sido posible.

Tato

A mi familia.
A Esther.
A Luna.

1 Introducción

DESDE hace algunos años, empresas, universidades, grupos de investigación o similares, que se dedican al desarrollo de software desarrollan sus propios paquetes que posteriormente son usados por sus propios trabajadores o clientes. Estos paquetes de software como todo software, necesita ser servido, mantenido y actualizado de alguna forma y para ello se usan repositorios. En el caso del Software Libre, y en el caso que se va a presentar, una de las opciones es utilizar repositorios no oficiales de Debian [Aok12] que gestionan paquetes Debian. Estos repositorios no oficiales pueden crearse con la ayuda de *reprepro* [Lin12], que permite al usuario crear un repositorio personal con todas las características de uno oficial.

En el Laboratorio de investigación ARCO de la Escuela Superior de Informática de la Universidad de Castilla-La Mancha, se tiene como objetivo el diseño de sistemas complejos con componentes tanto hardware como software, haciendo especial énfasis en aspectos como la utilización de recursos, prestaciones y comunicaciones, y su aplicación al desarrollo de servicios avanzados en áreas tales como la computación distribuida, inteligencia ambiental, redes de sensores...

En el día a día del grupo ARCO se utiliza Debian GNU/Linux como sistema operativo para el desarrolo de software. En el grupo ARCO existen computadores de diferentes arquitecturas, debido a la renovación que van sufriendo los equipos con el paso del tiempo. Por ello, un problema típico que existe es cuando se compila un paquete, por ejemplo con arquitectura «amd64» y se sube al repositorio. Sucederá que a este paquete solo podrán acceder los usuarios de «amd64». Si un usuario de «i386» quiere acceder a ese paquete, tiene que conseguir el código fuente, construirlo y volver a subir el paquete al repositorio. Por lo tanto, es un requisito contar con al menos un computador con esa arquitectura, que pueda utilizarse para compilar el paquete y que ese usuario, o alguien que sepa hacerlo, se dedique a compilarlo usando ese computador u otro con la misma arquitectura. Con lo cual, existe una necesidad, y es tener una versión de los paquetes compatibles con cada una de las arquitecturas soportadas en el laboratorio para que pueda ser usada por lo trabajadores.

Aunque lo ideal sería tener una infraestructura dedicada con al menos un computador de cada una de las arquitecturas soportadas, en pequeños entornos de trabajo como el laboratorio

ARCO, se llega al primer problema, y es que existen recursos limitados y esto no es posible debido a que:

- No se dispone de computadores con el propósito de estar disponibles y dedicados para compilar paquetes.
- Los computadores con los que se cuenta podrían no estar siempre disponibles cuando se necesiten para realizar la compilación, bien por estar apagados, bien por estar siendo usados por los trabajadores o cualquier otra circunstancia. Por lo tanto, no se puede saber en que instante va a haber un computador disponible.

Pero además de estas características, pueden darse unas necesidades como por ejemplo:

- Según la *Debian Policy* [JS12] un paquete se puede subir siempre y cuando existan todas sus dependencias. En este caso puede haber otras necesidades como por ejemplo, que se pueda subir el paquete si se encuentran las dependencias en ese o en otros repositorios.
- Puede haber otros requisitos de calidad totalmente distintos a los de Debian, como por ejemplo, garantizar que se haya utilizado el mismo compilador.
- A la misma vez pueden surgir otras características dependiendo del laboratorio o empresa que no se contemplan en Debian.

El presente proyecto, intenta solventar un problema que se da en el laboratorio ARCO y que es extrapolable a otros entornos o empresas, y es la construcción y distribución de software para múltiples arquitecturas.

1.1 Estructura del documento

El presente documento se encuentra dividido en capítulos donde en cada uno de ellos se aborda el contexto en el que se ha enmarcado este proyecto, además de sus principales aportaciones. El autor del presente documento recomienda al lector hacer una lectura secuencial para comprender mejor todos los aspectos del proyecto.

Si el lector no está familiarizado con la creación de paquetes Debian, se recomienda encarecidamente consultar la Debian Policy y la guía del nuevo mantenenedor de Debian.

Capítulo 2: Antecedentes

En el proyecto «Icebuilder» se pretende construir un sistema con el mismo objetivo final que «buildd» pero aportando cosas distintas, ya que no está arraigado a la política de Debian [JS12].

En este capítulo se hará un resumen de lo que es un paquete Debian, como funciona y para qué sirve, así como un análisis de las herramientas existentes antes de la realización del presente proyecto.

Capítulo 3: Objetivos

El objetivo fundamental del proyecto es construir un sistema distribuido fácil de instalar, de configurar y de mantener que construya paquetes Debian para distintas arquitecturas y los sirva en un repositorio de paquetes.

En este capítulo se desarrollarán objetivos tanto generales como específicos quedando el alcance del proyecto bien definido.

Capítulo 4: Arquitectura del sistema

Para entender un poco mejor en los capítulos posteriores cuáles son los elementos que componen el sistema distribuido, este capítulo pretende ofrecer una serie de esquemas para contribuir a una mejor comprensión del proyecto. Conceptos como *developer*, *manager* o *repositorio* serán explicados en detalle.

Capítulo 5: Metodología y herramientas

Debido al auge que hoy en día tenemos de diferentes metodologías de desarrollo de software, se ha elegido una metodología ágil para el desarrollo del proyecto. De forma que sea posible obtener prototipos funcionales durante el desarrollo del proyecto y que permita realizar cambios.

Capítulo 6: Desarrollo del proyecto

El proyecto ha sido dividido en iteraciones para abordar su desarrollo. Al final de cada una de esas iteraciones se entrega un prototipo añadiendo la funcionalidad requerida. Debido a que los requisitos pueden cambiar durante el desarrollo del mismo, se ha elegido una metodología iterativa e incremental.

En este capítulo se describen los requisitos que debe cumplir el proyecto en cada una de sus iteraciones, así como la evolución obtenida en cada una de ellas. Se ofrece información sobre las decisiones que se han tomado junto con los aspectos de implementación tenidos en cuenta durante el desarrollo.

Capítulo 7: Conclusiones y trabajo futuro

En este capítulo se ofrecen las conclusiones que se han alcanzado tras la finalización del proyecto. Al mismo tiempo, quedan cosas por hacer y se detalla información sobre funcionalidades que pueden resultar interesantes en el futuro. Algunas de esas funcionalidades fueron propuestas como proyecto para el *Google Summer of Code 2014*. Se hace también mención a los premios obtenidos por este Proyecto Fin de Carrera.

Apéndice A: Manual de usuario. Instalación de un Nodo

Guía ilustrativa de la instalación de un nodo del sistema distribuido. Contiene requisitos mínimo y todos los pasos necesarios para poner en funcionamiento un nodo del sistema.

Apéndice B: Creación de un repositorio de paquetes Debian

Creación paso a paso de un repositorio de paquetes Debian con *reprepro* [Lin12].

Incluye archivos de configuración y nociones básicas de uso de un repositorio.

Apéndice C: Construcción de un paquete Debian
Resumen de los pasos que se dan para la creación de un paquete Debian sencillo, así como algunas características que debe tener para seguir la *Debian Policy* [JS12].

Apéndice D: Dependencias de un paquete Debian
Resumen de los aspectos fundamentales relacionados con las dependencias de los paquetes Debian. Las menciones que se hacen aquí son las estrictamente necesarias para comprender algunas partes del desarrollo del proyecto.

Apéndice E: GNU Free Documentation License
Copia de la licencia GFDL que debe acompañar a este documento.

Este documento cumple con la normativa para la elaboración de Proyecto Fin de Carrera indicada por la Escuela Superior de Informática de Ciudad Real, perteneciente a la Universidad de Castilla-La Mancha [Inf13].

1.2 Nombre y página web

Este proyecto participó en el VIII Concurso Universitario de Software libre en el cual cada proyecto requería de una página web y un nombre, el nombre que se utilizó desde entonces fue *Icebuilder*. En algunas partes de este documento, como en los apéndices se nombra como *icebuilder*, porque se tuvieron que realizar paquetes para su instalación y tutoriales para el CUSL.

La página web[1] desarrollada para el concurso es completamente funcional. En ella se puede leer acerca de las últimas novedades del proyecto en sí y desde ella se puede acceder al repositorio[2], realizar seguimiento de bugs,reportar bugs, etc.

[1] http://www.icebuilder.org
[2] `https://bitbucket.org/arco_group/pfc.joseluis.sanroma`

Antecedentes

2

En este capítulo se muestra una visión general del estado del arte en cuanto a la construcción de paquetes Debian se refiere, herramientas actuales que existen y se utilizan para construir paquetes, middlewares de objetos distribuidos y algunos conceptos como *Boinc style*.

2.1 El sistema operativo Debian GNU/Linux

El proyecto Debian GNU/Linux fue fundado en el año 1993 por Ian Murdock. El nombre proviene de una mezcla de varios nombres. Para empezar, la palabra «Debian» está formada por el nombre de la ex-esposa del creador «Deborah» junto con su propio nombre «Ian», de lo cual se obtiene **Debian**. GNU es un acrónimo recursivo de «GNU's Not Unix!» (GNU no es UNIX). El proyecto GNU fue iniciado por Richard Stallman en el año 1984 y es un conjunto de aplicaciones, bibliotecas y herramientas de programación de Software Libre desarrollado o patrocinado por la FSF (*Free Software Foundation*). Linux, es uno de los núcleos de sistema GNU. Todo ello en su conjunto forma la distribución de GNU Debian GNU/Linux que sigue un manifiesto llamado «Directrices de Software Libre de Debian», DFSG [Per04], en el que se define qué software es «suficientemente libre» para incluirse en Debian.

Los objetivos que Ian Murdock buscaba al crear Debian eran dos:

- Calidad, se desarrollaría con el mayor cuidado, para ser digno del núcleo Linux.
- También sería una distribución no comercial, lo suficientemente creíble para competir con las distribuciones comerciales.

A día de hoy, Debian GNU/Linux ha tenido tanto éxito que ha alcanzado un tamaño enorme. Ofrece soporte para 12 arquitecturas y los núcleos Linux, FreeBSD [Pro14] y Hurd [Fou13a]. Además, cuenta con más de 15000 paquetes de software que pueden satisfacer casi cualquier necesidad que uno pueda tener ya sea en casa o en la empresa. Tanto es así, que instituciones del tamaño de la NASA utilizan Debian GNU/Linux. Hay además, alrededor de 100 distribuciones de GNU que se han basado en Debian, entre las cuales destaca **Ubuntu**, por lo tanto, si algo está en Debian estará soportado también para todas sus distribuciones derivadas.

Debian sigue los principios del Software Libre y las nuevas versiones no se publican hasta

que estén listas. Existen cuatro versiones que son *stable*, *testing* y *unstable* y experimental. Para que la versión con etiqueta «estable» sea liberada hacen falta largos meses de pruebas durante los cuales se congelan las versiones de los paquetes y se corrigen bugs críticos. Una vez corregidos todos esos bugs se lanza una nueva «release». Las versiones estables de Debian, siempre reciben nombres de personajes de la película «Toy Story». La última versión estable se denomina «Wheezy», que es el nombre que recibe el pingüino de la citada película.

Legalmente hablando, Debian es un proyecto gestionado por voluntarios sin ánimo de lucro. Cuenta con un millar de *desarrolladores Debian*, (a partir de ahora «Debian developers») pero también con un gran número de colaboradores entre los que se encuentran traductores, informadores de errores, artistas, desarrolladores casuales, mantenedores de paquetes, etc). Para llevar el proyecto a buen puerto se cuenta con una gran infraestructura con muchos servidores que están conectados a través de internet y que suelen ser donados por los patrocinadores.

2.1.1 El funcionamiento interno de Debian GNU/Linux

El proyecto Debian se debe fundamentalmente al trabajo de sus desarrolladores en la infraestructura, trabajo individual o grupal en paquetes y en comentarios y/o sugerencias de usuarios.

Los «Debian developers» tienen varias responsabilidades, y como miembros del proyecto tienen una gran influencia en la dirección que toma Debian con el paso del tiempo. Un «Debian developer» generalmente es responsable de, al menos, un paquete, pero dependiendo de su tiempo libre y de lo que quieran hacer, pueden formar parte de numerosos equipos[1], adquiriendo por lo tanto, más responsabilidad.

Debian tiene una base de datos que contiene a todos los «Debian developers» registrados en el proyecto junto con información relevante (dirección, coordenadas geográficas, tlf, etc). Es pública y cualquiera puede consultarla[2].

Las coordenadas geográficas permiten la creación de un mapa como el de la figura 2.1 donde poder ubicar a todos los «Debian developers».

Hay equipos muy variados, por ejemplo el «Release Team», que es el encargado de cada una de las versiones de Debian. El «QA Team» se encarga de la calidad, pero también hay equipos para la seguridad, releases, etc. De todos ellos los equipos más numerosos están en el empaquetado de software.

El manteniento de paquetes en Debian es una actividad organizada, y muy documentada. Debe adherirse a las reglas establecidas en la «Debian policy» [JS12]. Afortunadamente

[1] Puede encontrar más información sobre los equipos internos de Debian en https://wiki.debian.org/Teams
[2] Lista de los Debian developers en https://www.debian.org/devel/developers.loc

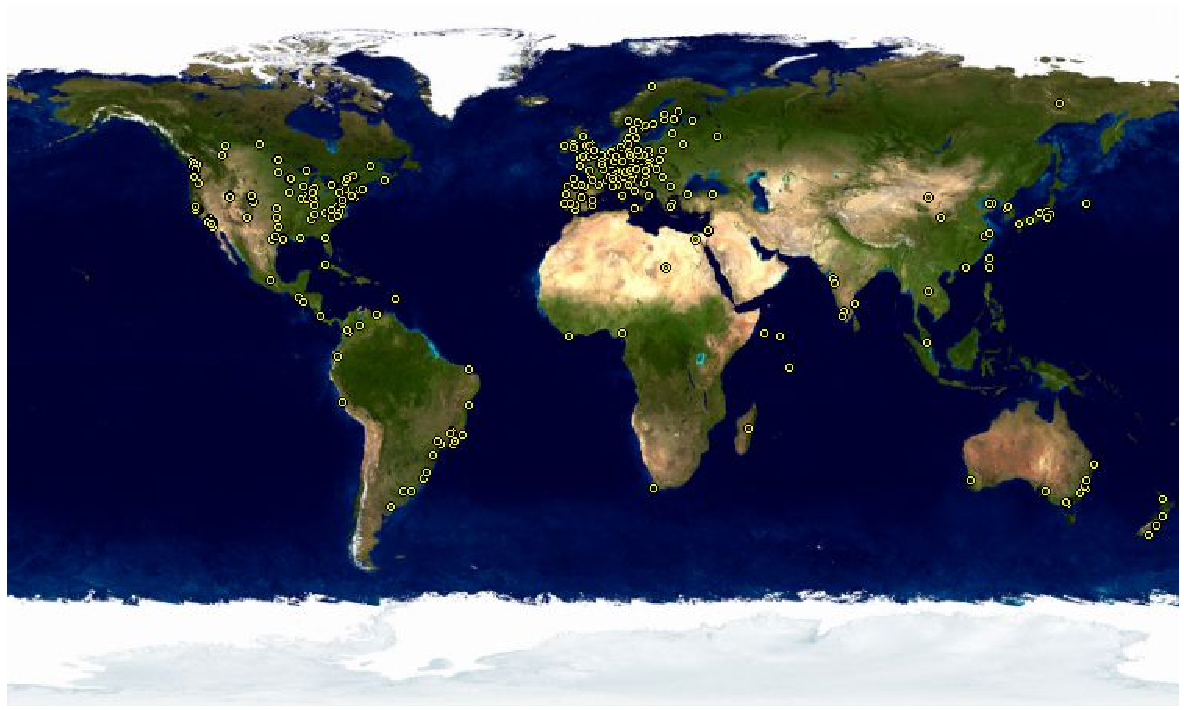

Figura 2.1: «Debian developers» en el mundo

existen multitud de herramientas que sirven de ayuda tanto a los «Debian developers» como a los «Debian maintainers»[3] y que permiten que se concentren en corregir fallos.

La «Debian policy» se va modificando con el paso del tiempo adaptándose e incluyendo modificaciones[4] que son aceptadas por todas las partes interesadas, cubre detalles técnicos en la creación de paquetes.

2.1.2 Repositorios en Debian

A todos los efectos, un repositorio es un archivo ordenado donde se almacenan los paquetes Debian (ya sean paquete binarios o paquetes fuente), con una estructura bien definida y constantemente actualizados.

Dentro del sistema Debian GNU/Linux, los repositorios se configuran en el fichero `/etc/apt/sources.list` en el que cada una de las líneas en él representa un repositorio y la forma en la que los paquetes serán obtenidos.

Los repositorios tienen varias áreas que son: *main*, *contrib* y *non-free*.

[3]Un mantenedor de paquetes Debian o «Debian maintainer», es una persona que necesita de un sponsor para subir los paquetes Debian. La diferencia fundamental con los «Debian Developers», es que estos últimos pueden subir paquetes a Debian sin necesidad de un sponsor.

[4]Las modificaciones que se proponen a la Debian policy puede consultarse en el sistema de seguimiento de errores de Debian en http://bugs.debian.org/cgi-bin/pkgreport.cgi?pkg=debian-policy;dist=unstable.

El área main

Está dentro de la distribución Debian, solamente los paquetes dentro de *main* están considerados parte de la distribución. Ninguno de estos paquetes requieren de otros paquetes que estén fuera del área *main* para funcionar. Estos paquetes deben **cumplir** con la DFSG.

Además de todo ello, los paquetes en *main*:

- No deben tener como requisito un paquete fuera de *main* o recomendar su instalación para compilación o ejecución.
- No deben tener errores que obliguen a dejarlo de soportar.
- Deben cumplir con la política de Debian [JS12].

El área contrib

En este área se proporcionan paquetes suplementarios que pretenden funcionar junto con todos los de la distribución Debian, con la salvedad de que requiere software de fuera de la distribución para compilar o funcionar. Todos los paquetes en el área *contrib* debe cumplir con la DFSG.

Además:

- No deben tener errores que obliguen a dejarlo de soportar.
- Deben cumplir con la política de Debian [JS12].

Algunos ejemplos de paquetes que pueden incluirse en *contrib* son:

- Paquetes libres que requieren de paquetes en *contrib* o *non-free* u otros paquetes que no están en el archivo para compilación o ejecución.
- Paquetes que tienen dentro otros paquetes para paquetes que no son libres.

El área non-free

En el área *non-free* se proporcionan paquetes suplementarios que pretenden funcionar junto con todos los de la distribución Debian, pero que no cumplen con las DFSG o tienen otros problemas que hacen su distribución problemática. Pueden no cumplir con la política de Debian [JS12]

Los paquetes en non-free:

- No deben tener errores que obliguen a dejarlo de soportar.
- Deben cumplir con todos los requisitos de la política [JS12] de Debian que puedan satisfacer.

Repositorio backports

Algunas personas prefieren usar en el día a día la versión «stable» de Debian, esta versión prioriza en todo momento la estabilidad frente a la novedad, por lo que se tienen paquetes de versiones más viejas, pero más estables. Se podría incluso decir que la rama «stable» de Debian está desactualizada con respecto a otras distribuciones de GNU/Linux. Para ayudar a tener paquetes más nuevos surgen los repositorios denominados «backports» que no forman parte de la distribución «stable», pero que cualquier usuario puede añadirlos bajo su propia responsabilidad.

Los paquetes que se alojan en estos repositorios son paquetes que se cogen de la siguiente versión de Debian, llamada «testing», preparados y reconstruidos para su uso en Debian «stable». En ocasiones también se crean los paquetes de la distribución «unstable». Estos paquetes no están tan probados como los de la distribución «stable», pueden incluso tener bugs o incompatibilidades.

2.1.3 El ciclo de vida de una versión de un paquete Debian

El proyecto Debian tiene 3 o 4 versiones diferentes (de ahora en adelante se les llamará «ramas») de cada programa simultáneamente, llamadas «experimental» «unstable»(inestable) «testing»(pruebas) y «stable» (estable). Cada una de ellas corresponde a una fase diferente del desarrollo. Para entender todo ello mejor,se ofrecerá una visión del viaje de un programa cualquiera a través de todas estas versiones.

La rama experimental: Experimental

La primera etapa del paquete suele ser la rama exprimental. Este es un grupo de paquetes Debian que se encuentra actualmente en desarrollo y no tiene por qué estar completado. No todos los paquetes pasan por este paso, algunos desarrolladores utilizan esta rama para recibir comentarios o sugerencias acerca de su programa por usuarios más experimentados. De otro modo, esta rama generalmente tiene modificaciones importantes a paquetes base, cuya integración con «unstable» con errores tendría repercusiones críticas. La rama «experimental» de Debian es una distribución completamente aislada, ya que sus paquetes nunca migran a otra rama salvo intervención directa o expresa de su responsable o los `ftp-master`[5]

La rama unstable

Un «Debian developer» crea un paquete inicial que compila para la rama *Unstable* y se ubica en el servidor **ftp-master.debian.org**. Esta situación hace que el paquete tenga que ser inspecionado y validado por parte de los `ftp-master`. Posteriormente, estará disponible en la rama *unstable*, que es la elegida por los usuarios que prefieren mantenerse con paquetes

[5]El equipo `ftp-master` son responsables de mantener la infraestructura necesaria del archivo de Debian. Esto da lugar a scripts que se utilizan para el procesamiento de paquetes subidos al archivo, y también del flujo de paquetes entre distribuciones.

más actuales a evitar errores serios. Si se encuentran errores en estos paquetes se reportan al mantenedor del paquete. Cuando se corrige ese error el responsable del paquete vuelve a subirlo y es cuando la «Autobuilder network» (que se verá en detalle en la sección 2.3.1) entra en acción.

Migración a Testing

Cuando el paquete ha madurado en la rama *unstable* se vuelve candidato para entrar en la rama *testing* cumpliendo unos determinados niveles de calidad:

1. Falta de fallos críticos o, al menos, menor cantidad de fallos críticos que su versión en *testing*.
2. Al menos ha tenido que estar 10 días en *unstable*, que se considera un tiempo suficiente para encontrar errores o problemas serios.
3. Compilación satisfactoria para todas las arquitecturas oficiales.
4. Sus dependencias deben estar satisfechas en *testing*, o en su defecto, que migren a testing al mismo tiempo.

Este sistema, aunque no es infalible, hace que la rama *testing* tenga para muchos usuarios un buen compromiso entre estabilidad y novedad.

La promoción de testing a stable

El paquete, se encuentra ahora en la rama *testing*. El responsable del paquete debe continuar mejorándolo. El siguiente paso que se produce es la inclusión del paquete en la rama *stable* que no es más que una simple copia de *testing* en un momento elegido por el administrador de la versión. Lo ideal es que esta decisión se tome cuando no exista ningún programa en *testing* que tenga errores críticos conocidos.

Es muy raro que este momento llegue, por lo tanto, en la práctica, se opta por llegar a un compromiso: eliminar los paquetes en los que su encargado no corrigió los errores a tiempo. El gestor de versiones anunciará previamente un periodo de estabilizacion conocido como «freeze» (que en español significa «congelar»), durante el cual cada actualización a *testing* debe ser aprobada. El objetivo que se persigue es evitar cualquier versión nueva, con sus errores, y aprobar solamente correcciones de errores.

Una vez que se ha llegado a este punto se diría que el paquete «ha completado su viaje» dentro de Debian, formando parte por fin de la rama *stable*.

A modo de resumen, se destacan las tres ramas que existen en Debian, stable, testing y unstable, cada una de ellas respectivamente con software más nuevo pero con más errores, así pues, se tiene una versión de Debian para cada uso. Por ejemplo, para una tarea crítica como un servidor, sería recomendable una Debian stable, mientras que para alguien que

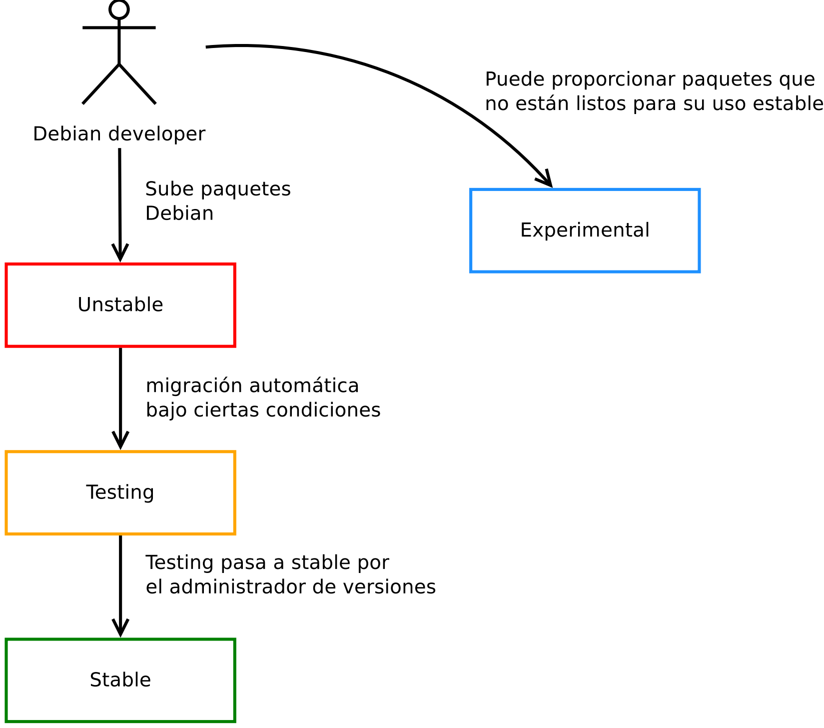

Figura 2.2: Camino de un paquete en Debian

quiere probar nuevas versiones de paquetes sería ideal usar *unstable*. A parte de estas tres ramas está *experimental*, se podría decir que es una distribución a parte, donde se prueba el software más experimental que entra en Debian.

2.2 El paquete Debian

El paquete Debian es una parte muy importante del desarrollo de este proyecto, de hecho, todo gira en torno a él. Aunque para documentar todo lo referente al paquete Debian ya está la *Debian Policy* [JS12] y la *guía del nuevo desarrollador de Debian* [RA13], se mostrarán a modo resumido todos los aspectos considerados más importantes del paquete Debian que pueden resultar de ayuda para entender el presente documento. No obstante, se recomienda al lector que quiera conocimientos más profundos, leer los documentos citados anteriormente.

Se podría hablar de dos tipos de paquetes Debian, el oficial, que es el que se encuentra en los repositorios oficiales de Debian, y el no oficial, que es el que no se encuentra en los repositorios oficiales de Debian y se puede distribuir por los medios que cada uno considere oportunos. Un paquete no oficial puede estar construido de la forma correcta, pero la diferencia es que no ha sido subido a Debian por ningún «Debian Developer».

2.2.1 Elegir el paquete

Si se va a empaquetar algún programa, lo primero que hay que hacer es obtener una copia del programa para probarlo. A partir de aquí, la generación de un paquete Debian requiere nombrar a los ficheros de una determinada manera. Por ejemplo, la copia del programa, comprimido en formato `tar`: `package_version.tar.gz`

Lo siguiente es añadir las modificaciones para «debianizar» el programa:

- `package_version.orig.tar.gz`
- `package_version-revision.debian.tar.gz`
- `package_version.orig.dsc`

Y por último generar el programa: `package_version-revision_arch.deb`

Nótese que la palabra «package» debe ser el nombre del programa en cuestión y «version» el número de versión del programa original. «revision» es la revisión del paquete Debian, y por último «arch» es la arquitectura del paquete según la *Debian Policy* [JS12]

2.2.2 Estructura básica

Normalmente los paquetes Debian se construyen a partir de un código que no está desarrollado por el propio mantenedor del paquete. Es por ello habitual partir de algún archivo comprimido *tar.gz* donde se encuentra todo el código fuente. El programa `dh_make` ayuda a generar las plantillas de ficheros necesarios para construir el paquete. `dh_make` necesita conocer la información del mantenedor, por lo que es necesario proporcionar tanto el nombre completo (DEBFULLNAME) como el e-mail (DEBEMAIL). Es bueno añadir estas variables al `$HOME/.bashrc` para que no se tengan que escribir una y otra vez:

```
DEBFULLNAME="Nombre Apellidos"
DEBEMAIL="tue-mail@ejemplo.com"
export DEBULLFNAME DEBEMAIL
```

Una vez configurado es hora de ejecutar `dh_make` desde dentro del directorio del programa.

```
dh_make -f programa.tar.gz
```

Tras la ejecución de la orden anterior, se crea un directorio llamado `debian` con muchos ficheros, muchos de ellos son plantillas. La «única» tarea que hay que hacer es completar toda la información requerida siguiendo la «Debian Policy».

2.2.3 Archivos necesarios en el directorio `debian`

A continuación se describirán los archivos importantes para el paquete. Cuando se refiera a algún archivo se hará mediante la ruta relativa al directorio que se acaba de crear con `dh_-`

make, es decir, si se refiere al archivo debian/control quiere decir que el archivo estará dentro del directorio debian y se llamará control. El debian/copyright es el archivo copyright que está dentro del directorio debian y así sucesivamente.

debian/control

Es uno de los ficheros más importantes del paquete *Debian*, es una especie de manifiesto donde se incluye información relevante para la el sistema de gestión de paquetes. Véase un archivo debian/control cualquiera, por ejemplo el del paquete *gentoo*.

```
Source: gentoo
Section: x11
Priority: optional
Maintainer: Innocent De Marchi <tangram.peces@gmail.com>
Build-Depends: debhelper (>= 9), dh-autoreconf, libgtk2.0-dev, libglib2.0-dev
Standards-Version: 3.9.3
Homepage: http://www.obsession.se/gentoo/

Package: gentoo
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Suggests: file
Description: fully GUI-configurable, two-pane X file manager
gentoo is a two-pane file manager for the X Window System. gentoo lets the
user do (almost) all of the configuration and customizing from within the
program itself. If you still prefer to hand-edit configuration files,
they're fairly easy to work with since they are written in an XML format.
.
gentoo features a fairly complex and powerful file identification system,
coupled to an object-oriented style system, which together give you a lot
of control over how files of different types are displayed and acted upon.
Additionally, over a hundred pixmap images are available for use in file
type descriptions.
.
gentoo was written from scratch in ANSI C, and it utilizes the GTK+ toolkit
for its interface.
```

Listado 2.1: debian/control paquete Debian

En este fichero hay dos secciones diferenciadas, el paquete fuente y la información necesaria para su construcción se pueden ver en la sección Source. El paquete o los paquetes binarios están separados por una línea en blanco y están en la sección Package.

A continuación, se repasarán los campos del paquete fuente donde si no se especifica lo contrario, el campo a rellenar **no** es obligatorio:

Source Es el nombre del paquete fuente y es obligatorio. Tanto en el debian/control como en el archivo .dsc, este campo debe tener solamente el nombre del paquete fuente.

Maintainer Nombre del mantenedor del paquete y su dirección de correo electrónico. Este campo es obligatorio. El nombre aparece primero y luego el correo eléctrónico entre <>.

Uploaders Nombre o nombres con sus respectivas direcciones de correo electrónico de los

mantenedores del paquete. El formato es el mismo que en el campo anterior.

Section Este campo no es obligatorio pero si recomendable. Especifica un área en la cual clasificar el paquete [6].

Priority Este campo es recomendable y representa el nivel de importancia que tiene para el usuario que este paquete esté instalado. Estos niveles pueden ser: `required`, `important`, `standard`, `optional` ó `extra`. Para ver una descripción detallada de niveles de prioridad, consultar el capítulo 2.5 de la «Debian Policy» [JS12].

Build-depends Es una lista con los nombres de los paquetes y en algunos casos sus versiones que son necesarios para construir el paquete. Estos paquetes deben estar instalados en el sistema a la hora de construir el paquete fuente para poder construir los paquetes binarios.

Standars-Version Recomendado, la versión más reciente de las normas con el que el paquete cumple.

Homepage La dirección del sitio web para el paquete. Preferiblemente la web desde donde puede obtenerse el paquete fuente y cualquier documentación que pueda servir de ayuda. El contenido desde este campo es **solamente** la URL, sin nada más.

Vcs-Browser Los paquetes están desarrollados con controles de versiones. El propósito es indicar el repositorio público donde se está desarrollando el paquete fuente.

Campos del paquete binario:

Package Campo obligatorio en el que se pone el nombre del paquete binario, el que dará lugar al `.deb`. Este nombre deberá cumplir una serie de reglas dependiendo del lenguaje en el que esté escrito, su finalidad, etc.

Architecture Dependiendo del contexto, este campo obligatorio, puede incluir una de las siguiente opciones:

- Una única palabra identificando una de las arquitecturas descritas. Estos nombres pueden obtenerse con la orden `dpkg-architecture -L`.
- Una arquitectura comodín como puede ser `any` y que sirve para todas las arquitecturas listadas con la orden `dpkg-architecture -L`. Es la más usada.
- `all` lo que indica que el paquete es independiente de la arquitectura. Para que el lector se haga una idea, `all` se refiere a programas que funcionan en todas las arquitecturas tales como ficheros multimedia, manuales o programas escritos en lenguajes interpretados.
- `source` lo cual indica un paquete fuente.

Section Este campo aunque no es obligatorio, si es recomendable. Tal y como en el paquete fuente, indica un área en donde ubicar le paquete.

[6]Lista completa de las secciones en «SID» http://packages.debian.org/unstable/

Priority Recomendado. Indica como de importante para el usuario es tener el paquete instalado. La prioridad `optional` se utiliza para paquetes nuevos que no entran en conflicto con otros de prioridad `required`, `important` o `standar`

Depends y otras Son las dependencias binarias del paquete que describen sus relaciones con el resto de paquete. Básicamente son los paquete que se requiere que estén instalados en el sistema para que el paquete funcione. Una relación de las dependencias binarias puede verse en el capítulo 7.2 de la *Debian policy* [JS12]

Description Este campo es obligatorio. Contiene la descripción del paquete binario. Consiste en dos parte, una descripción corta y la descripción larga basándose en el siguiente formato y sin los símbolos <>.

Homepage La dirección del sitio web para el paquete. Preferiblemente la web desde donde puede obtenerse el paquete fuente y cualquier documentación que pueda servir de ayuda. El contenido desde este campo es **solamente** la URL, sin nada más.

Package-Type Campo simple donde se indica el tipo de paquete, `deb` para paquetes binarios y `udeb` para paquetes micro binarios.

debian/changelog

Este fichero contiene una descripción muy breve de los cambios que el mantenedor del paquete hace a los ficheros específicos del paquete. No se describen los cambios que sufre el programa, eso corresponde a su autor y eso suele estar en un fichero `.changes`

```
 miproyecto (1.0-1) unstable; urgency=low

 * Initial release (Closes: #nnnn)  <nnnn is the bug number of your ITP>

-- John Doe <johnDoe@icebuilder.com>  Wed, 25 Dec 2013 14:02:25 +0100
```

Listado 2.2: changelog paquete Debian

Cada vez que el mantenedor realice un cambio en el paquete debe crear una nueva entrada en el `debian/changelog`. Si el mantenedor hace cambio en la versión del paquete, se incrementará el número de versión del paquete. Cuando el mantenedor del paquete empaquete una nueva versión, el número de versión del paquete vuelve a empezar desde 1.

En el listado 2.2 «miproyecto» es el nombre del paquete, y entre parentesis (1.0-1), del cual «1.0» es la versión del código fuente, mientras que «1» es la versión del paquete Debian.

debian/copyright

Este fichero contiene información sobre el autor del programa y las licencias que se utilizan en el programa y en cada una de sus partes. Debería indicar también quien es el autor y la licencia de los ficheros del paquete Debian.

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: miproyecto
Source: <url://example.com>

Files: *
Copyright: <years> <put author's name and email here>
<years> <likewise for another author>
License: GPL-3.0+

Files: debian/*
Copyright: 2013 John Doe <johnDoe@icebuilder.com>
License: GPL-3.0+

License: GPL-3.0+
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
.
This package is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
.
On Debian systems, the complete text of the GNU General
Public License version 3 can be found in "/usr/share/common-licenses/GPL-3".

# Please also look if there are files or directories which have a
# different copyright/license attached and list them here.
# Please avoid to pick license terms that are more restrictive than the
# packaged work, as it may make Debian's contributions unacceptable upstream.
```

Listado 2.3: copyright paquete Debian

debian/rules

Es el `Makefile` que debe inlcuir una serie de objetivos como : `clean`, `binary`, `binary-arch`, `binary-indep` y `build`.

Estas reglas objetivo serán ejecutadas por `dpkg-buildpackage` o `debuild` cuando se construya el paquete. Cabe destacar que todas ellas son de carácter obligatorio, salvo `install` y `get-orig-source` que son opcionales.

- `clean` (obligatorio): elimina todos los archivos generados, compilados o innecesarios del árbol de directorios de las fuentes.
- `build` (obligatorio): para la construcción de archivos compilados a partir de los archivos fuenteo o la construcción de documentos formateados.
- objetivo `build-arch` (obligatorio): para la compilación de las fuentes en programas compilados (dependientes de la arquitectura) en el árbol de directorios de la compilación.
- objetivo `build-indep` (obligatorio): para la compilación de las fuentes en documentos formateados (independientes de la aqruitectura) en el árbol de directorios de la

compilación

- `install` (opcional): para la instalación en la estructura de directorios temporales para el directorio `debian` de los archivos para cada uno de los paquetes binarios. Si existe el objetivo `binary` dependerá de este.

- `binary` (obligatorio): para la construcción de cada uno de los paquetes binarios (combinando con los objetivos `binary-arch` y `binary-indep`).

- `binary-arch` (obligatorio): para la construcción de paquetes dependientes de la aquitectura (`Architecture: any`).

- `binary-indep` (obligatorio): para la construcción de paquetes independientes de la aquitectura (`Architecture: all`).

- `get-orig-source` (opcional): para obtener la versión más reciente de las fuentes originales desde el lugar de almacenaje del autor.

Cuando se utiliza `dh_make` se genera un `debian/rules` como el siguiente:

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

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

%:
	dh $@
```

Listado 2.4: debian/rules paquete Debian

debhelper hace casi todo el trabajo aplicando las reglas por defecto por lo que la única tarea que habría que realizar sería la de sobreescribir dichos objetivos.

2.3 Sistemas distribuidos para construir paquetes Debian

La necesidad de un sistema distribuido para construir paquetes Debian parte de querer agilizar el proceso de construcción en el que el mayor cuello de botella se encuentra en la construcción de las dependencias de construcción de un paquete. Más aún si cabe si ese paquete debe ser construido para más de una arquitectura.

Para automatizar este proceso, la propia distribución de Debian GNU/Linux tiene una red de autocompiladores para todas las arquitecturas que están soportadas en Debian, por lo que será un caso de estudio. Es la que se usa actualmente para que todos los paquetes de Debian estén listos para ser distribuidos a través de sus repositorios oficiales.

2.3.1 Debian Autobuilder Network

La «Debian Autobuilder Network» es el sistema distribuido desarrollado por Debian que gestiona la construcción de paquetes para todas las arquitecturas soportadas[7]. Esta red se constituye por varias máquinas que usan el paquete *buildd* que es el encargado de coger los paquetes del repositorio y reconstruirlos para la arquitectura requerida.

Necesidad de la Debian autobuilder network en Debian

Debian soporta actualmente unas pocas arquitecturas, pero los mantenedores de los paquetes suelen construir los paquetes binarios para una sola arquitectura, que suele ser *i386* o *amd64* por ser las más comunes. La construcción para el resto de arquitecturas se producen automáticamente asegurando que cada paquete solamente se construya una vez. Todos los fallos ocurridos se ven reflejados en una base de datos.

En sus inicios, los desarrolladores tenían que vigilar cuando había nuevas versiones de los paquetes y reconstruirlos si querían estar actualizados con respecto a la distribución intel (entonces las arquitecturas disponibles eran para *intel* y *m68k*). Todo el proceso se hacía manualmente. En las listas de correo, los desarrolladores miraban los nuevos paquetes y cogían algunos de ellos para construirlos. La coordinación para que ningún paquete sea construido más de una vez por personas distintas se hacía anunciándolo en una lista de correo. Queda patente que este procedimiento es propenso a errores y puede consumir mucho tiempo. Sin embargo, esta fue la forma en la que se mantuvo Debian durante mucho tiempo.

El demonio (explicar que es un demonio) de construcción del sistema automatiza la mayor parte de este proceso. Consiste en un conjunto de scripts (escritos en *Perl* [Int11] y *Python* [Fou13b] para ayudar a los que hacen las adaptaciones. Finalmente todo ello ha evolucionado en un sistema que puede mantener las distribuciones de Debian casi automáticamente. Las actualizaciones de seguridad se construyen en el mismo conjunto de máquinas para asegurar su disponibilidad a tiempo.

Funcionamiento de buildd

Buildd es el nombre que recibe el software que utiliza la «Debian Autobuilder Network», y que en realidad está compuesto por varias partes:

wanna-build: Es la herramienta que ayuda a coordinar la reconstrucción de paquetes a través de una base de datos que mantiene una lista de paquetes y su estado. Hay una base de datos central por arquitectura que almacena los estados, versiones y alguna otra información relevante acerca de los paquetes. Se alimenta de los ficheros fuentes y de paquetes obtenidos de los distintos archivos de paquetes que tiene Debian como `ftp-master` y `security-master`.

[7]Arquitecturas oficiales: amd64, armel, armhf, i386, ia64, kfreebsd-amd64, kfreebsd-i386, mips, mipsel, powerpc, s390, s390x, sparc. Además de otras no oficiales http://www.debian.org/ports/

buildd: Es el demonio que periódicamente comprueba la base de datos mantenida por `wanna-build` y llama a `sbuild` para construir los paquetes.

sbuild: Es responsable de la construcción real de los paquetes en entornos enjaulados y aislados. Se asegura que las dependencias estén instaladas en el `chroot` antes de realizar la construcción y después llama a las herramientas estándar de Debian para iniciar el proceso de construcción. Los registros se envían a una base de datos para su posterior análisis.

En la figura 2.3 se muestra un boceto de las partes de *buildd* de las que se ha hablado en esta sección.

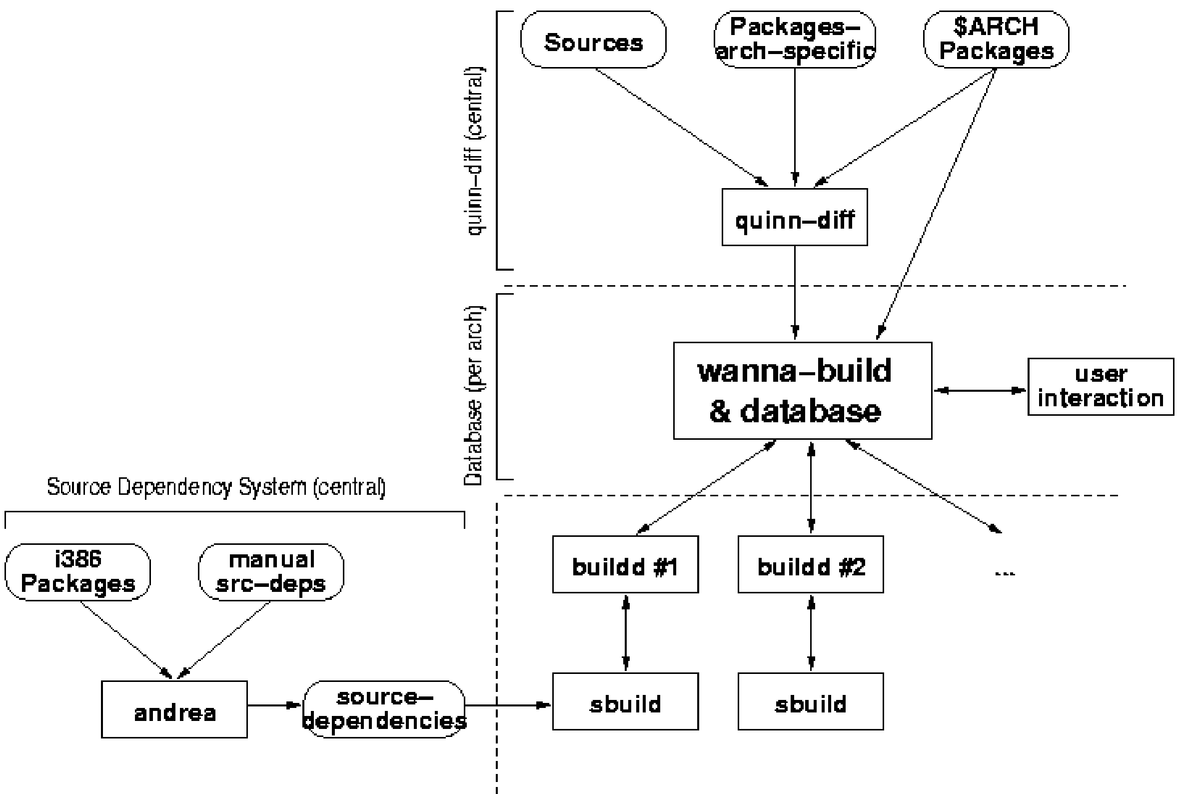

Figura 2.3: Boceto de *Debian Autobuilder Network* [JS12]

Tareas de un Debian Developer

En principio un desarrollador de Debian no necesita usar explícitamente *buildd*. Cuando envíe un paquete al repositorio, se añadirá a la base de datos para todas las arquitecturas en `needs-build` (se verá más adelante). Las máquinas de construcción pedirán a la base de datos de construcciones, una lista con paquetes en este estado, y tomarán dichos paquetes de esta lista para realizar la construcción. Esta lista está priorizada por el estado de construcción que puede ser «out-of-date» o «uncompiled» , prioridad, sección y finalmente nombre del paquete. Además, con el fin de evitar que un paquete se queden sin construir se ajustan dinámicamente las prioridades de forma dinámica con un tiempo incremental de espera.

Si la construcción de un paquete alcanza un estado satisfactorio para todas las arquitecturas, el responsable del paquete no tendrá que hacer nada, y estos paquetes serán enviados al archivo correspondiente. Si por el contrario, la construcción no es satisfactoria, el paquete entrará en unos estados especiales: `build-attempted` en el caso de fallos de construcción que no han sido revisados, `failed` para su revisión y fallos reportados en los paquetes o `dep-Wait`, si las dependencias de construcción específicas para construir el paquete no están disponibles. Los administradores revisarán los paquetes que han fallado y avisarán al responsable. Este proceso normalmente se hace abriendo un error en el sistema de seguimiento de errores del que dispone Debian `DBTS`.

En ocasiones, algún paquete tarda mucho tiempo en ser construido y eso lastra su inclusión en la distribución `testing`. Si se da este caso, se ajusta bajo la petición del equipo responsable de la publicación de Debian.

Para comprobar el estado actual de los diferentes intentos de construcción que ha hecho *buildd* con los paquetes se puede comprobar el resgistro[8].

Estados de «Wanna-build»

Como se ha descrito en la sección 2.3.1, *Wanna-build* [Deb12] es la herramienta que ayuda a coordinar la construcción de los paquetes a través de una base de datos que mantiene una lista de paquetes y su estado. A alguno de esos estados, como «needs-build» ya se les ha hecho referencia en la sección 2.3.1.

Para cada una de las arquitecturas a las que Debian da soporte, hay una base de datos de *wanna-build* instalada en buildd.debian.org con los paquetes y sus respectivos estados de construcción. Actualmente hay 7 estados que un paquetes pueden tener: *needs-build*, *building*, *uploaded*, *dep-wait*, *failed*, *not-for-us* y *installed*.

El significado de cada uno de esos estados se describe a continuación:

needs-build Cuando un paquete se marca con este estado, quiere decir que su responsable ha enviado una nueva versión del paquete para una arquitectura distinta de la arquitectura para la que es esta base de datos de *wanna-build*, por lo tanto es necesaria su reconstrucción para el resto de arquitecturas. Si el estado es *needs-build* quiere decir que el paquete está en cola para la reconstrucción, es decir, aún no ha sido escogido por ningún constructor de paquetes, pero se hará. La cola `needs-buils` se basa en los siguiente criterios de prioridad:

1. Paquetes construidos previamente. A los paquetes que se han construido previamente se les da prioridad sobre otros que no lo han sido.
2. Los paquetes con prioridad `required` se construyen antes que los de `extra`.

[8]La comprobación de los intentos de construcción de un paquete https://buildd.debian.org/status/

3. La sección del paquete. Hay paquete que son más importantes que otros, por ejemplo, los paquetes en `games` se construyen después que los de `base`.
4. Orden «ASCIIbético» en el nombre del paquete.

building Un paquete se marca como `building` en el momento en el que un constructor de paquetes lo coja del principio de la cola hasta el momento en el que se genera el registro del paquete. Como los paquetes no se cogen de uno a uno, puede estar marcado como `building` antes de que empiece su construcción.

uploaded Este estado queda marcado en un paquete cuando su construcción es satisfactoria. Un constructor de paquetes no volverá a tocar un paquete marcado con este estado hasta el siguiente envío.

dep-wait Es el estado que se le marca a un paquete cuando la construcción del mismo falla porque no se encuentran sus dependencias de construcción. Posteriormente, se marcará automáticamente como `needs-build` hasta que las dependencias estén disponibles.

failed Un paquete se marca con estado `failed` cuando falló su construcción. El paquete tendrá este estado hasta que un responsable decida que debe hacerse o hasta que una nueva versión esté disponible. Sin embargo, cuando una nueva versión de una paquete que se marcó como `failed` exista, el autocompilador preguntará a su administrador si se debe o no reintentar construirlo. Esto es así para que los paquetes que obviamente fallarán de nuevo no hagan perder el tiempo a *buildd*. Aunque descartar un paquete antes de intentar compilarlo no es la manera correcta de hacerlo, la opción está disponible para el administrador del autocompilador. Un paquete **nunca** se marcará como failed sin intervención humana.

not-for-us Algunos paquetes son específicos de algunas arquitecturas y no deberían recompilarse para otras. `Wanna-build` solamente mira el nombre del paquete y su sección por lo tanto no sabe si un paquete pertenece solamente a una arquitectura. Los constructores de paquetes no deberían perder el tiempo construyendo estos paquetes, por eso, una solución inicial fue marcarlos como `not-for-us`. Sin embargo, como es difícil de mantener, `not-for-us` está desaprobado actualmente; los responsables deberían usar en su lugar `packages-arch-specific`

installed Un paquete que se marca como `installed` está construido para la arquitectura para la que es la base de datos de `wanna-build`. Un paquete pasa de `uploaded` a `installed` cuando se acepta en el repositorio.

Para ilustrar todo los descrito anteriormente, en la figura 2.4 se muestra el diagrama de flujo del procedimiento que se sigue con `wanna-build`

En el diagrama de flujo puede observarse claramente que la intervención humana en el proceso de construcción de un paquete se hace en ocasiones muy contadas y especiales.

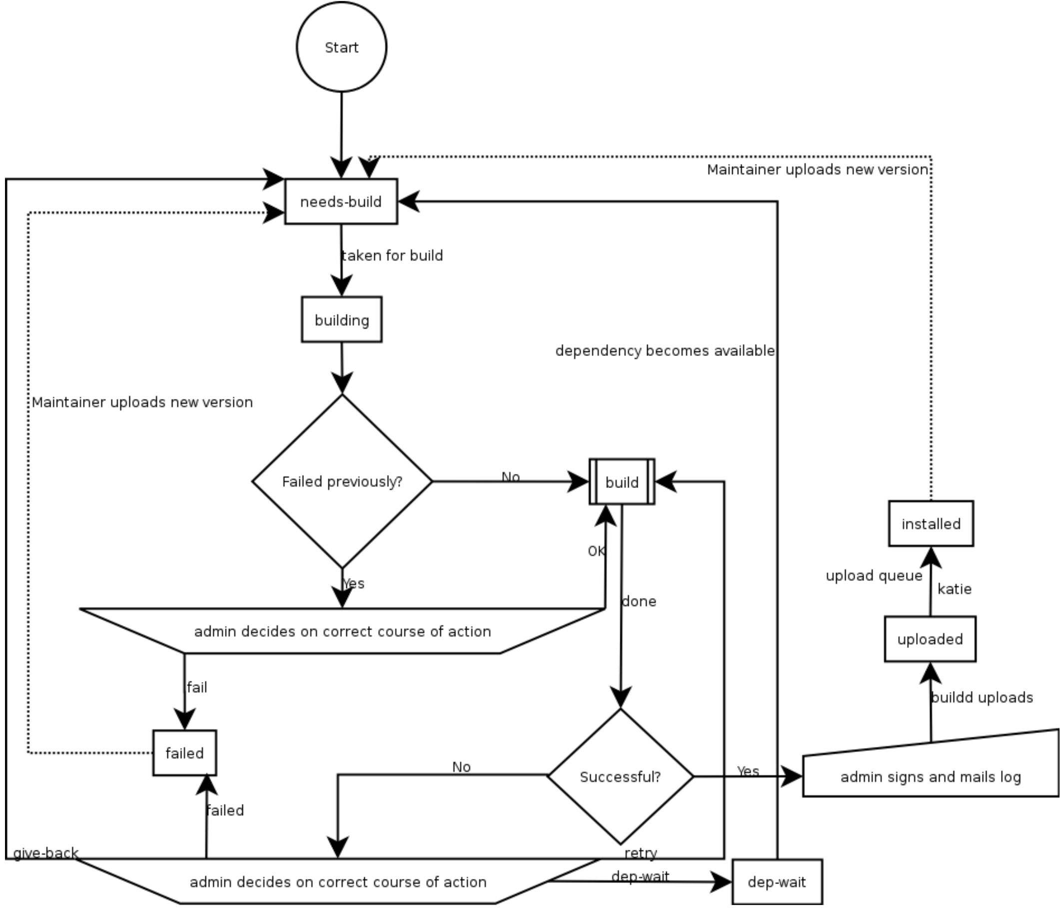

Figura 2.4: Diagrama de flujo de *Wanna-build*

2.3.2 Pybit

Dado que la «Debian Autobuilder Network» está fuertemente arraigada a la política de Debian, han surgido otros intentos de imitar esta red de autocompiladores que suplen distintas necesidades. Una de estas herramientas es *Pybit*.

El objetivo de esta herramienta es crear un sistema distribuido basado en AMQP para construir paquetes usando una colección de buildd directamente desde varios clientes de VCS. Permite cancelar las construcciones y usar varios buildd por arquitectura, plataforma y conjunto de herramientas.

Utiliza el lenguaje de programación *Python* [Fou13b] y *RaqbbitMQ*. La cola de mensajes se trabaja usando *python-amqplib* con los mensajes encolados usando *python-jsonpickie*.

El sistema consiste en dos partes:

- Un servidor
- Uno o muchos clientes.

pybit-web es la parte del servidor, y *pybit-client* el cliente. Necesita una base de datos *PostgreSQL* [Gro13] con la que interactuar utilizando *python-psycopg2*.

El front-end web consulta con el back-end utilizando *JQuery Javascript* [Pro13]. Esta parte del sistema no interactúa directamente con la base de datos ni con el sistema.

El cliente solamente interactúa con el controlado utilizando la cola y la API HTTP [HTT99], nunca con la base de datos. Se pretende construir cualquier combinación de paquetes (por ejemplo DEB O RPM) para cualquier arquitectura y en cualquier sistema, incluido MS Windows.

2.4 Herramientas de empaquetado en Debian

Para crear un paquete Debian, la propia Debian proporciona algunas herramientas para ayudar en el proceso de construcción de paquetes. Algunas de estas herramientas son colecciones de `scripts` que ayudan en la generación del paquete Debian, mientras que otras son programas completos que facilitan al desarrollador o al mantenedor la interacción diaria con los paquetes. Hoy en día la creación de paquetes para Debian sigue siendo un proceso muy artesanal, no así el proceso de reconstrucción.

2.4.1 Scripts de empaquetado en Debian

Algunos de los *scripts* más conocidos son los que se recomiendan en la documentación oficial de Debian como la *Guía del nuevo desarrollador de Debian* [RA13], la *política de Debian* [JS12] o la *Developer's reference* [JS13]:

debhelper [Hes10], un framework para la generación de paquetes Debian que está formado por un conjunto de scripts para ayudar al mantenedor del paquete Debian.

svn-buildpackage [Blo09], permite mantener un paquete alojado en un repositorio subversion, proporciona métodos de construcción sencillos.

git-buildpackage [Gue13], al igual que el anterior, permite mantener un paquete alojado en un repositorio, esta vez git, y proporciona métodos de construcción sencillos.

dhmake [ClS09], se utiliza para crear el directorio Debian inicial. Genera plantillas de los archivos que son obligatorios u opcionales en el paquete para su construcción.

Tanto svn-buildpackage [Blo09] como git-buildpackage [Gue13] están pensadas para ayudar a mantener el paquete con el paso del tiempo facilitando las tareas al mantenedor. Por el contrario, las otras dos están más orientadas a ayudar a empaquetar desde cero. Por ejemplo, `dh-make` [ClS09] ayuda a generar los ficheros necesarios cuando se crea por primera vez el paquete.

2.4.2 El entorno limpio

A la hora de generar un paquete Debian, se necesita cumplir una serie de requisitos con sus dependencias. En la sección 2.2 se explica la estructura del paquete Debian con los campos que tiene cada fichero que se genera y lo que ha de incluirse en las descripciones. Uno de

esos campos hace referencia a las dependencias de construcción.

Para construir un paquete Debian **todas** sus dependencias de construcción deben estar instaladas en el sistema donde se va a realizar la construcción. Esto puede ocasionar un problema, y es que si un mantenedor de paquete mantiene más de un paquete, práctica bastante habitual, tendrá instalados en el sistema paquetes que no le hacen falta para su uso diario, y que solamente necesita cuando va a realizar la construcción del paquete que mantiene. Esto da lugar a que poco a poco el sistema se va *ensuciando* con dependencias que no se usan y que pueden dar lugar a problemas y errores. Es por ello que para reconstruir los paquetes se necesita un entorno limpio, lo más aislado posible, donde se realizará la construcción del paquete y las pruebas necesarias para su correcto funcionamiento y con ello poder avisar de algún posible error.

Con el paso del tiempo, se han ido creando poco a poco herramientas automáticas para la reconstrucción de paquetes en entornos limpios, de hecho, es recomendable construir los paquetes Debian utilizando estas herramientas.

Herramientas para la construicción de paquetes en entornos limpios

En esta subsección se hará un repaso de las herramientas más utilizadas para la construcción de paquetes Debian en entornos limpios y que se encuentran en los repositorios oficiales de Debian:

sbuild Es la herramienta que se utiliza en la «Debian Autobuilder Network», ya comentada en la seccion 2.3.1.

pbuilder [Uek07] Permite al usuario configurar un «chroot» [Deb] para construir paquetes de forma automática. Realiza una instalación limpia donde se construyen los paquetes indicando la rama (*stable*, *testing* o *unstable*). Además, esta instalación es mantenible y se puede actualizar para tener en todo momento un entorno actualizado con las versiones más recientes de los paquetes. Una de las ventajas de esta herramienta es que también permite construir paquetes para «Ubuntu», la distribución de GNU basada en Debian más conocida.

debomatic [Fal12] Esta herramienta está basada en *pbuilder* [Uek07], por lo que su funcionamiento es análogo. Sin embargo, añade más funcionalidad con el uso de «plugins», permitiendo automatizar entre otras cosas el firmado de paquetes y la subida de los paquetes a un repositorio Debian.

qemubuilder [Uek08] QEMU permite emular procesadores de otras arquitecturas. «Qemubuilder» usa «pbuilder» en otros procesadores emulados consiguiendo que se realice la construcción de un paquete dado en más de una arquitectura. La principal ventaja es que no se necesita de la máquina nativa para realizar la construcción ya que se realiza en máquinas virtuales.

2.4.3 Otras herramientas para la construcción de paquetes

Al margen de las herramientas comentadas hasta aquí, existen muchas otras herramientas que se han ido desarrollando con el paso del tiempo para ayudar a la construcción de paquetes Debian. Estas herramientas disfrutan de menos popularidad, seguramente porque son más jóvenes e inestables y no disfrutan de la madurez de las otras.

- **pdebuild-cross** crea un entorno de compilación cruzada compatible con pbuilder.
- **debian-builder** reconstruye un paquete dado ejecutando una serie de órdenes de forma automática.
- **cowbuilder** herramienta que envuelve el uso de pbuilder.
- **cowdancer** Es un wrapper para pbuilder.
- **edos-distcheck** Comprueba la «instalabilidad» de los paquetes Debian. Lee el conjunto de las descripciones de un pauqete Debian en formato deb-control del «Packagefile», y paquetes fuentes del «SourcePackagefile»
- **edos-builddebcheck** Comprueba si un paquete es «construible» en un entorno Degbian dado.
- **edos-debcheck** Comprueba si se satisfacen las dependencias de un paquete Debian. Lee de la entrada estandar un conjunto de descripciones de un paquete debian, en el formato *debian-control*.
- **dose-builddebcheck** Actualización de los paquetes «edos» mencionados anteriormente, el autor cambió de nombre añadiendo nueva funcionalidad a estos en detrimento de los «edos». Comprueba si un paquete es «construible» en un entorno dado.
- **dose-distcheck** Al igual que el anterior, muestra información acerca de si un paquete es «instalable» o no.
- **dose-debcheck** Otra actualización de los «edos», facilita información acerca de si se satisfacen las dependencias de un paquete Debian.

2.5 BOINC

La capacidad de cómputo y el espacio en disco no está concentrado en un solo sitio, ya no está solamente en habitaciones con super computadores realización miles de operaciones. En su lugar, está distribuido a lo largo de cientos de millones de computadores personales.

La utilización de los recursos públicos en computación, comenzó a mediados de los años 90 con dos proyectos GIMPS, un proyecto colaborativo en el cual voluntarios buscan números primos de Mersenne[9] y *Distributed.net*[10], un proyecto en el cual se donan ciclos de CPU para investigación y proyectos de interés público.

[9] http://www.mersenne.org/
[10] http://www.distributed.net

El número de computadores personales conectados a internet está creciendo rápidamente, se estima que puede alcanzar los 1.000 millones en 2015. Todos esos computadores pueden ofrecer una capacidad de varios *PetaFLOPS*. Si 100 millones de computadores dan 10 Gigabytes de almacenamiento, el total sería 1 ExaByte o 10^{18} bytes y excedería la capacidad de cualquier centro de almacenamiento.

BOINC [And12] (Berkeley Open Infrastructure for Network Computing) es una plataforma para computación distribuida que ha sido desarrollada en U.C. Berkeley Spaces Sciences Laboratory. Es *open source* y está disponible en su página web[11].

2.5.1 Objetivos de BOINC

El objetivo general de BOINC es alentar a la gente para que participe en proyectos donando su capacidad de cómputo. Algunos objetivos específicos:

Reducir las barreras para participar en proyectos públicos. Permitir a la investigación científica que cuenta con un moderado número de computadores crear y hacer funcionar una gran red de computadores. Un servidor para un proyecto basado en BOINC consiste en una máquina configurada con *Apache [Fou14], PHP [MA14], MySQL y Python [Fou13b].*

Compartir recursos entre proyectos autónomos. Los proyectos no están registrados de forma centralizada. Cada proyecto opera con sus propios servidores y a parte. Sin embargo, los usuarios de los computadores pueden participar en varios proyectos al mismo tiempo y pueden asignar a cada proyecto una parte de sus recursos a donar. Por ejemplo, en un proyecto pueden donar ciclos de CPU mientras que en otro pueden donar ancho de banda.

Soporte para distintas aplicaciones. BOINC tiene un amplio abanico de aplicaciones y aporta mecanismos flexibles y escalables para distribuir datos. Aplicaciones en distintos lenguajes de programación como C, C++ o FROTRAN pueden funcionar como BOINC con unas pequeñas modificaciones. Además las nuevas versiones pueden desplegarse automáticamente.

Recompensar a los participantes. Los proyectos de interés público deben proporcionar incentivos para atraer al mayor número posible de participantes. El primer incentivo que se da es el crédito, por ejemplo, midiendo la cantidad de recursos que ha aportado al proyecto (CPU, espacio en disco, ancho de banda, etc).

2.5.2 Algunos proyectos que usan BOINC

Los listados que se presentan a continuación son algunos de los proyectos que usan BOINC, y son los requisitos de los mismos los que han ido dando forma a BOINC:

[11] `http://boinc.berkeley.edu`

SETI@Home Realiza procesamiento de señales digitales de los datos del radio telescopio. Una versión basada en BOINC de este proyecto tiene alrededor de 500.000 participantes.

Folding@Home Está en la Universidad de Stanford. Estudia diversos aspectos de las proteínas y enfermedades relacionadas.

Climateprediction.net El objetivo de este proyecto es cuantificar y reducir incertidumbres en predicciones meteorológicas que se basan en simulación por computador.

Existen muchos más proyectos que utilizan BOINC, estos son solamente algunos de ellos.

2.6 Middlewares orientados a objetos

Las nuevas necesidades de los usuarios, la mejora en las comunicaciones y otros muchos factores han contribuido a que los sistemas de información que se usan actualmente ofrezcan servicios en red y funcionen en un entorno distribuido.

Los *middlewares* son plataformas que proporcionan soporte para desarrollar aplicaciones distribuidas abstrayendose en mayor o menor medida de la tecnología y de los protocolos de red. El objetivo de cualquier middleware es ofrecer al programador una visión abstracta sobre la tecnología de red que separa la comunicación real de las aplicaciones.

En la figura 2.5, se representa una invocación entre objetos de un mismo programa. Ambos objetos se encuentran en la memoria del *Nodo A* en el cual están siendo ejecutados. Se comunican entre ellos mediante mensajes que son invocaciones a métodos.

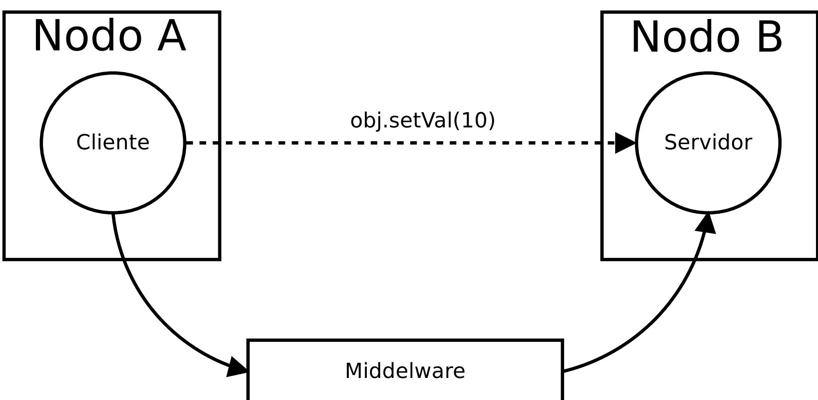

Figura 2.5: invocación de un método en ICE

La filosofía de la orientación a objetos se basa en que los problemas del mundo pueden modelarse como objetos que tienen relaciones con otros objetos y se comunican entre ellos. Los middlewares orientados a objetos, mantienen esta filosofía extendiéndola a la red. Las plataformas distribuidas permiten pensar en términos de clientes y servidores de modo que el «cliente» es el que demanda un servicio, mientras que el «servidor» es la entidad que proporciona esos servicios.

Un mismo cliente puede actuar también como servidor y viceversa. Por ejemplo, si el servidor recibe una invocación y este a su vez realiza otra invocación sobre otro objeto, se podría decir que está actuando como servidor y como cliente.

ZeroC ICE tiene una gran importancia en el desarrollo de este proyecto y se estudiará más a fondo, pero también existen otros middlewares en el mercado. En las siguientes secciones se presentarán algunos de los middlewares más utilizados actualmente. Los conceptos que aquí se presentan son fácilmente trasladables a otros middlewares.

2.6.1 ZeroC ICE

Internet Communications Engine (Internet Communication Engine (ICE)) [HS13] es un middleware orientado a objetos construido por la empresa ZeroC y con licencia GPL.

Soporta varios lenguajes de programación, lo cual permite una mayor adaptabilidad según las necesitadades del momento.

Clientes y servidores

Los conceptos cliente y servidor varian un poco en lo fundamental. El cliente requiere la funcionalidad que el servidor proporciona. Los servidores ICE necesitan un adaptador de objetos donde poder añadir los sirvientes necesarios para ofrecer el servicio. En cuanto a los clientes, necesitan un proxy al servidor para poder solicitar la funcionalidad requerida por la aplicación.

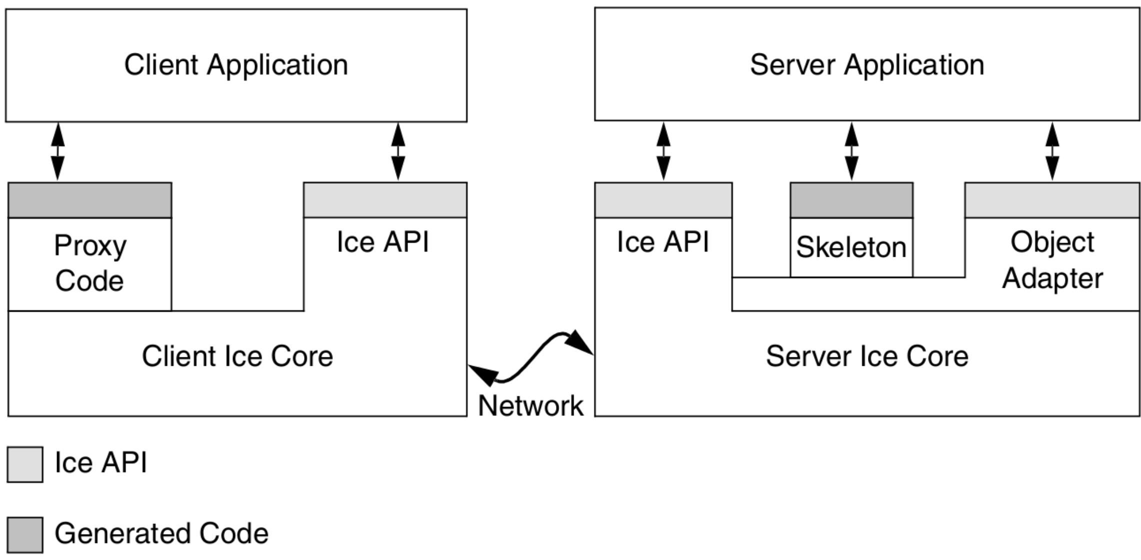

Figura 2.6: Estructura cliente-servidor en ICE

Adaptador de objetos

El adaptador de objetos actúa como un contenedor de objetos del servidor que son accedidos mediante invocaciones remotas. En cada adaptador de objetos hay unas direcciones asignadas que se conocen como **endpoints**. En ICE un ejemplo de **endpoint** sería el siguiente:

```
tcp -h 127.0.0.1 -p 9999
```

Este ejemplo es un endpoint `tcp`, o explicado de otra forma, utilizará el protocolo TCP y escuchará en la interfaz con dirección IP 127.0.0.1 y en el puerto 9999.

Objetos y sirvientes

Los **objetos** en ICE son el mismo concepto que los de cualquier lenguaje de programación. Pero con algunas características especiales:

- Es una entidad en el espacio de direcciones, remoto o local, capaz de responder a las peticiones de los clientes.
- Un objeto puede instanciarse en un servidor o en varios servidores.
- Cada objeto tiene una o más interfaces, la cual tiene varias operaciones soportadas por un objeto. Los clientes son los que invocan estas operaciones.
- Una operación tiene cero o más parámetros y un valor de retorno. Tanto los parámetros como el valor de retorno son de un tipo específico.
- Cada objeto tiene una identidad única, con esta identidad única, se distingue a un objeto de otros.

Las peticiones de los clientes deben terminar ejecutando el código en el servidor. El componente en el lado del servidor que proporciona el comportamiento que está asociado a la invocación, se denomina **sirviente**. Cuando un sirviente se añade a un adaptador de objetos, es necesario asignarle una **identidad de objeto**, para identificarlo globalmente en el sistema distribuido.

Proxy

Para identificar un sirviente bastará con conocer su identidad y el `endpoint` del adaptador de objetos donde se encuentra:

```
Objeto1 -t:tcp -h 127.0.0.1 -p 9999
```

El sirviente con la identidad `Objeto1` es accesible en el endpoint `tcp -h 127.0.0.1 -p 9999`. El valor `-t` quiere decir que el modo de acceso al objeto es **twoway**, o lo que es lo mismo, que el sirviente puede recibir mensajes y se esperarán sus respuestas.

Slice

¿Cómo sabe el cliente el tipo de objeto que hay al otro lado? El servidor y el cliente deben compartir información sobre los objetos involucrados en la comunicación, así como sus atributos, métodos, etc.

Specification Language for Ice (Slice) [sli14] es el lenguaje para especificar el contrato entre el cliente y el sevidor. Se define la interfaz y las operaciones que serán accesibles mediante un *proxy* a un determinado sirviente. ICE proporciona herramientas como `slice2cpp` o `slice2java` para traducir la interfaz SLICE a C++ u Java respectivamente. ICE también da soporte a: Python, .NET, PHP, Pbjetive-C, Ruby y ActionScript.

```
module Prueba {
  interface Cajero {
    void sacar_dinero(int dinero);
    void ingresar_dinero(int dinero);
  };
};
```

Listado 2.5: Ejemplo de interfaz Slice

El listado 2.5 muestra un ejemplo muy sencillo de SLICE. Este archivo debe ser compartido entre el cliente y el servidor. En el ejemplo se puede ver una interfaz `Prueba.Cajero` que tiene dos métodos `ingresar()` y `sacar()`.

Un objeto que implemente la interfaz `Prueba.Cajero` podrá ser añadido a un adaptador de objetos de un servidor. A su vez, el cliente, puede acceder al sirviente asegurándose así que cumple con la interfaz que se ha definido previamente en el archivo SLICE. Tanto el cliente como el servidor pueden estar implementados en diferentes lenguajes de programación, esto, sin duda alguna, es una ventaja porque se pueden implementar clientes en diferentes lenguajes sin necesidad de hacer cambios en el servidor.

Servicios de ICE

ICE ofrece una serie de servicios. Estos servicios suministran la funcionalidad que la mayoría de aplicaciones distribuidas requieren. Estos servicios proporcionan un excelente rendimiento y pueden ser replicados para lograr tolerancia a fallos y escalabilidad.

- **IceGrid**: *IceGrid* es uno de los servicios más importantes de ICE, proporciona una gran cantidad de funcionalidades para un *grid* de ordenadores como puede ser activación automática de objetos, balanceo de carga y transparencia de localización. Permite que un cliente se comunique con un objeto remoto sin saber en qué nodo se encuentra ni en qué puerto está escuchando. *IceGrid* incluye dos herramientas de administración para controlar sus características, `icegridadmin` la cual tiene una interfaz por linea de comandos y `icegrid-admin` con una interfaz gráfica. Es importante conocer algunos

de los conceptos que maneja *IceGrid*:

Nodo Identifica a un «nodo lógico», puede haber más de un nodo en un computador.

Servidor Identifica mediante un nombre único a un programa que se ejecutará en un nodo.

Adaptador de objetos Tiene datos específicos de un adaptador de objetos utilizado en un servidor ICE como endpoints.

Aplicación Conjunto de servicios, objetos y configuraciones que forman la aplicación distribuida. Se puede almacenar en un fichero XML.

- **IceStorm**: *IceStorm* es el servicio de ICE que proporciona comunicación entre clientes y servidores a través de canales de eventos. En este ámbito es más común utilizar *publicador* y *suscriptor* en lugar de cliente y servidor. El publicador o los publicadores envían datos a un canal mediante invocaciones remotas, que serán enviadas a los suscriptores de dicho canal. Por ejemplo, imagínese que hay nodos que quieren recibir audio de un nodo servidor. El publicador puede publicar el audio en un canal de eventos de tal forma que los clientes que quieran recibir el audio solamente tendrán que suscribirse a ese canal. Cada canal se identifica por su nombre y pueden tener múltiples publicadores y suscriptores.
- **Freeze y FreezeScript**: *Freeze* es un servicio de persistencia que permite guardar el estado de los objetos en una base de datos *Berkeley*. *Freeze* puede recuperar automáticamente los objetos de una base de datos bajo demanda, y automáticamente actualizar la base de datos cuando cambia el estado del objeto. *FreezeScript* es un lenguaje de comandos que permite consultar el contenido de una base de datos *Freeze*. Resulta de mucha utilidad para la depuración de aplicaciones. *FreezeScript* ofrece funciones de transformación que permiten migrar de forma sencilla de una base de datos existente para adaptarse a un nuevo esquema.
- **Glacier2**: Es un servicio de firewall que permite la comunicación segura entre clientes y servidores.
- **IcePatch2**: *IcePatch* permite la distribución de actualizaciones de software a los nodos que componen el *grid*. *IcePatch* comprueba automáticamente la versión que tiene cada nodo y envía las actualizaciones disponibles.

2.6.2 CORBA

Uno de los middlewares más conocidos es al que se le conoce como *Common Object Request Broker Architecture* (CORBA). Es un estándar el *Object Management Group (*OMG*)* en el cual se describe el modelo de comunicación de un sistema distribuido. En este caso el OMG [OMG09] es el encargado de definir la API, funcionalidad, servicios y arquitectura que una implementación en CORBA debe proporcionar.

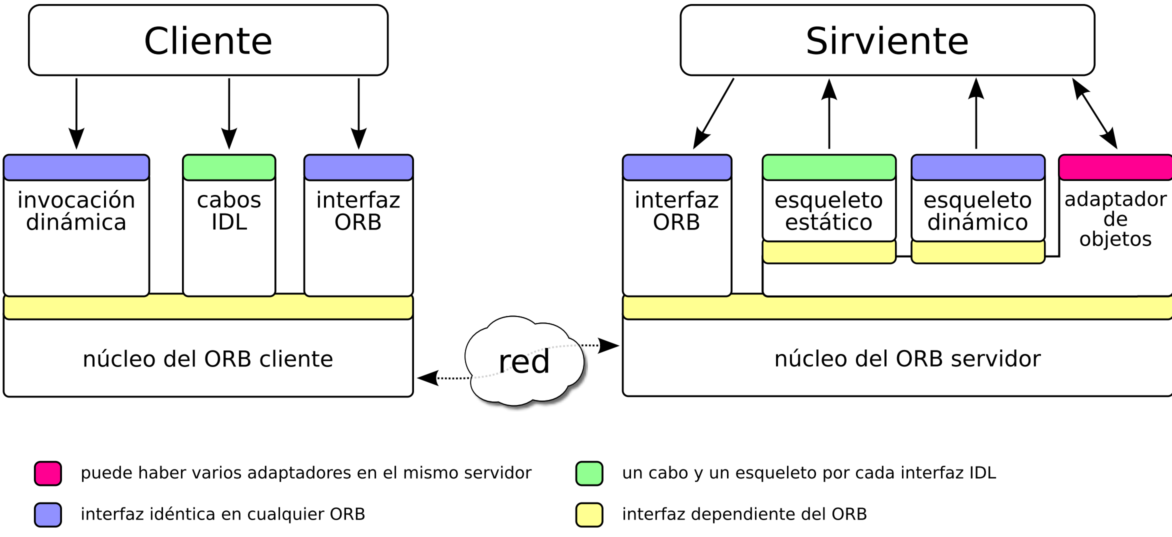

Figura 2.7: Arquitectura CORBA (David Villa [Vil09])

La arquitectura cliente-servidor en CORBA es muy similar a la de ICE. El lenguaje de especificación de interfaces (IDL), al igual que SLICE en ICE, permite definir el contrato entre el cliente y el servidor. Las aplicaciones clientes y servidores pueden estar escritas en distintos lenguajes de programación ya que existen herramientas que traducen IDL a un lenguaje concreto para que pueda ser utilizado.

Muchas de las estructuras como adaptadores de objetos, Object Request Broker (ORB), *skeleton*, etc, son muy similares a las de ICE en cuanto a concepto. El proceso de creación de ciertas identidades puede diferir entre los middlewares, pero la filosofía es la misma.

En el caso de CORBA el protocolo de red se llama *General Inter-ORB Protocol* (GIOP). La implementación sobre TCP de este protocolo se conoce como *Internet Inter-ORB Protocol* (IIOP). En este protocolo se definen los formatos y tipos de los mensajes, las normas de comunicación y la representación de las estructuras de los mensajes.

Existen variedad de implementaciones de CORBA algunas de ellas con licencia pública como *The ACE ORB* (TAO) [TAO09]

Y otras con licencias propietarias:

- OpenFusion [OPF13]: Se trata de un conjunto de implementaciones de CORBA para distintos niveles de computación que ha desarrollado la empresa *PrismTech*. Se ofrecen versiones de CORBA desde circuito integrado hasta otras de alto nivel para servidores y computación distribuida.
- ORBexpress [ORB09]: desarrollada por la empresa Objetive Interface System (OIS), es una colección de implementaciones de CORBA de las que destaca ORBexpressRT,

que implementa la especificación en tiempo real del estándar.

2.6.3 Java RMI

Java Remote Method Invocation RMI (ref manual) es un middleware que fue creado por la empresa *Sun Microsystems* para aplicaciones Java. En la figura (ref figure)) se muestra la arquitectura cliente-servidor de RMI.

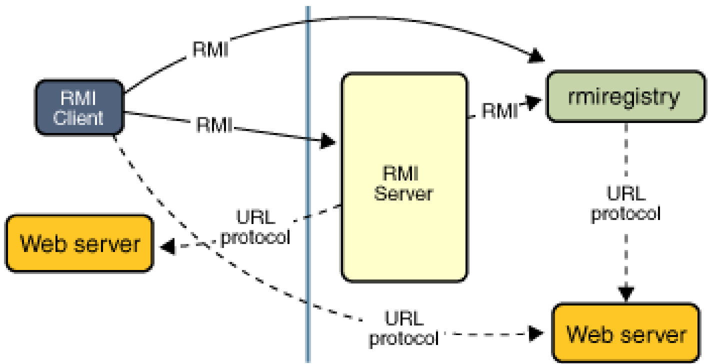

Figura 2.8: Estructura RMI [RMI09]

El esquema general de funcionamiento es el siguiente:

- Cada objeto registrado en el `rmiregistry` tiene que implementar, al menos, la interfaz `Remote` para que pueda ser accesible. Esta interfaz se encuentra en la biblioteca estándar de RMI
- El servidor utiliza el `rmiregistry` para registrar los diferentes objetos que serán accesibles para los clientes. RMI utiliza un sistema de nombres gracias al cual el servidor identifica a cada uno de los objetos dentro del `rmiregistry`
- El cliente obtiene la referencia a los objetos que el servidor publicó a través del `rmiregistry`
- Por último, el cliente puede invocar métodos sobre los objetos remotos, utilizando la referencia obtenida previamente de forma muy parecida a si el objetio estuviese accesible localmente.

En el caso de RMI el contrato entre cliente y servidor se espcifica utilizando interfaces Java. Por ello, el lenguaje de la implementación tanto de los clientes como de los servidores que utilicen RMI solamente puede ser Java.

3 Objetivos

En este capítulo se detallan los objetivos que se fijaron en el inicio de este proyecto, tanto generales como específicos. De esta manera, se determina el alcance del proyecto y los resultados que se esperan tras su implementación.

3.1 Objetivo general

Crear una infraestructura distribuida para la construcción de paquetes en un repositorio Debian. Una infraestructura que, en definitiva, automatice la construcción de paquetes Debian para ayudar a generar los paquetes para el resto de arquitecturas soportadas por el entorno de trabajo donde se instale y que sirva esos paquetes a través de un repositorio de paquetes Debian.

3.1.1 Objetivos principales

En general se pueden definir los siguientes objetivos fundamentales que se abordarán durante el desarrollo:

- Diseño de una arquitectura distribuida al estilo BOINC para donar ciclos de CPU con el fin de construir paquetes Debian, construcción compartida en un entorno dado y monitorización del proceso, además de ofrecer balanceo de carga y transparencia de localización para saber que computadores están encendidos, y destinar el trabajo de la construcción de paquetes a aquellos computadores que tengan una menor carga de CPU.
- Desarrollo de nodos de construcción basados en máquinas virtuales en lugar de entornos «chroot» para facilitar la disponibilidad de los recursos (la construcción de paquetes se puede parar, reanudar o migrar a otro nodo compatible). Además, un único computador puede ejecutar varias máquinas virtuales con el fin de construir paquetes para distintas arquitecturas.
- Implementación de construcción de paquetes mediante transacciones, que gracias ellas, el sistema pueda recuperarse frente a fallos tanto intencionados (cuando un trabajador termina su jornada laboral y apaga su equipo) como no intencionados (apagones de luz).

- Repositorios «backports» personalizados impulsados por las necesidades de los usuarios en lugar de la política de Debian.

3.1.2 Objetivos secundarios

Además de los objetivos fundamentales, se persiguen otros objetivos un poco menos importantes y que se describen a continuación:

Log del sistema distribuido Se debe tener en todo momento información de lo que pasa con los paquetes. Si la construcción del paquete ha finalizado con éxito, y si no lo ha hecho, cual ha sido el motivo, o en su lugar las pistas necesarias para poder deducirlo.

Construcción y distribución de software Estudiar y conocer el proceso de construcción y distribución de software. La familiarización y el entendimiento en esta materia, permitirá una mayor comprensión de los entresijos que sigue el proceso de desarrollar software, empaquetarlo y distribuirlo para distintas arquitecturas. Para ello se cuenta con la documentación proporcionada por Debian entre la cual está la *Debian Policy* [JS12], *guía del nuevo mantenedor de Debian* [RA13] y *guía de referencia del desarrollador de Debian* [JS13]

4 Arquitectura del sistema

En el trascurso este capítulo se mostrará una visión general de la arquitectura del sistema que ayudará a comprender lo que se mostrará más en detalle en el capítulo destinado al desarrollo del proyecto.

Se pretende dar una visión general de cada uno de los componentes que integran el sistema distribuido y de las funciones que desempeñan. Primero se hablará acerca de los requisitos, que bien pueden tomarse como una explicación mucho más profunda y detallada de los objetivos del proyecto.

Tras revisar los requisitos, se presentará cada uno de los componentes indicando que función realizan, para qué han sido construidos con qué propósito y por qué.

4.1 Requisitos

Una de las cosas más importantes de las cuales se hace referencia a la hora de explicar por qué la infraestructura que tiene Debian no servía, es porque es difícil de configurar y difícil de mantener. Bien, ese es una de los principales requisitos del sistema, que sea fácil de instalar, configurar y de mantener. Tanto, que incluso alguien que no conozca lo que hay detrás de la construcción de los paquetes Debian pueda utilizar esta infraestructura sin problemas.

4.2 Arquitectura

Se pretende dar a conocer de una manera introductoria los componentes más importantes del sistema distribuido que son fundamentalmente 4:

- Developers: Los que hacen los paquetes.
- Repositorio de paquetes: Aloja paquetes Debian y los sirve.
- Cliente: Hace peticiones de construcción.
- Manager: Genera el árbol de dependencias.
- Workers: Realizan la construcción y/o reconstrucción de los paquetes Debian.

4.2.1 Developer

El «developer» es aquella persona que sabe como construir paquetes Debian y conoce el proceso. Usualmente es el encargado de construir paquetes Debian desde el código fuente de cada uno de los programas que necesitan ser empaquetados.

Otra de las acciones que realiza es la subida de paquetes al repositorio para su construcción para el resto de arquitecturas. Solamente los «developers» tienen permisos para subir los paquetes.

Si por algún caso en concreto se detecta un *bug*[1], el «developer» también será el encargado de subir la nueva versión al repositorio con el bug corregido. El bug puede haber sido corregido por otro usuario o por él mismo.

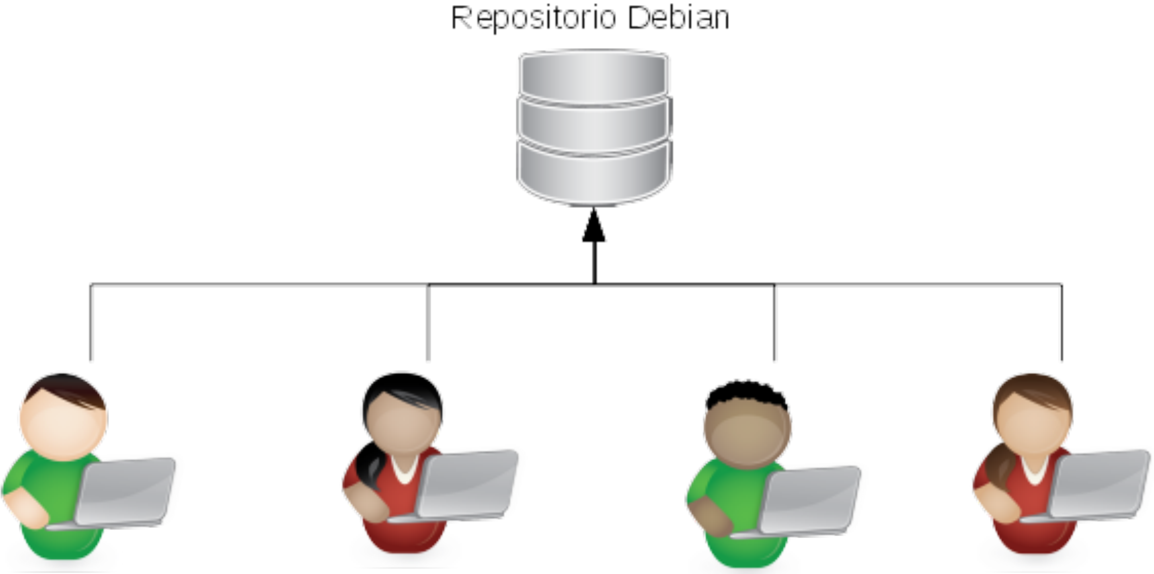

Figura 4.1: Developers

4.2.2 Repositorio

El repositorio es el lugar donde se almacenan todos los paquetes Debian. En el caso concreto de este proyecto se utiliza la herramienta reprepro [Lin12], la cual permitirá registro de claves GPG [Koc13] para asegurar que solamente los «developer» conocidos suben los paquetes. También permite subir paquetes firmados por los «Workers4.2.5» usando GPG, donde esas claves GPG estarán dentro del anillo de claves conocidas rechazando así cualquier paquete del cual no se conozca su firma GPG, y por lo tanto sea no confiable.

El repositorio puede estar en cualquier máquina dentro de la red, siempre y cuando todos los computadores, tanto virtuales como reales que forman el sistema distribuido, tengan acceso a él.

[1]Error en el software

4.2.3 Cliente

La labor del «Cliente» puede estar desempeñada por un humano, por una máquina o por ambos al mismo tiempo. Supóngase un caso en el cual una persona quiere construir el paquete `Zeroc-ice35`[2] para que funcione en la rama *stable* de Debian. En este caso, el cliente es humano. El paquete ya existía previamente, pero en este caso una persona quiere tener un paquete disponible para una determinada versión que por defecto no se construye para *stable*.

Supóngase ahora que un «developer» ha corregido un bug en el mismo paquete y el sistema recibe una alerta y reconstruye el paquete para el resto de arquitecturas. En este caso el «Cliente» sería una máquina, o el sistema distribuido en sí mismo. Por último, supóngase el caso en el que mientras el sistema automáticamente está construyendo paquetes, una persona le manda una petición. En este caso, el «Cliente» serían ambos.

Las peticiones que realiza el «Cliente» son enviadas al «Manager»

4.2.4 Manager

El «Manager» recibe las peticiones de construcción de los paquetes que provienen del «Cliente». Una vez recibidas, son procesadas generando las dependencias de construcción necesarias para construir el paquete solicitado previamente. Estas dependencias son generadas utilizando una estructura en forma de árbol mediante un algoritmo DFS[3] similar a la de la figura para construir el paquete «A».

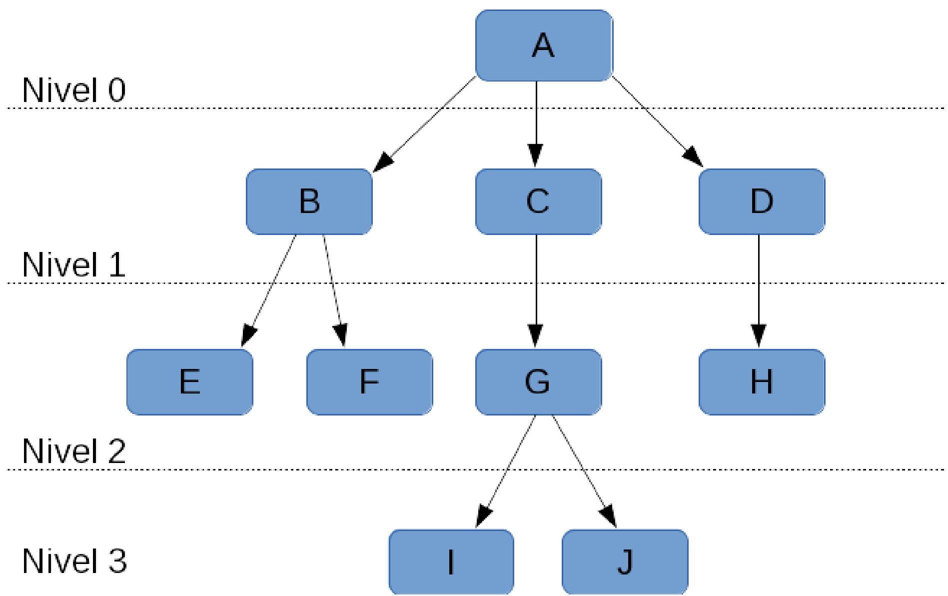

Figura 4.2: Ejemplo de árbol de construcción

[2]Paquete ZeroC-ice en su versión 3.5.1 que en el momento de la realización del presente documento, puede encontrarse en los repositorios oficiales de Debian en las ramas *unstable* y *sid*

[3]Depth First Search. En español es más conocido como búsqueda en profundidad

Una vez obtenido el árbol de dependencias, estas se procesan por niveles en orden inverso al que se han generado, empezando por los niveles más altos hasta terminar por los niveles más bajos del árbol.

Las dependencias de construcción se envían a los «Worker», para realizar la construcción de los paquetes.

4.2.5 Workers

Los «Workers» reciben los paquetes que tienen que construir del «Manager». Un «Worker» es un computador que se encuentra en la misma red en la que se ejecuta `icebuilder` y, cada uno de ellos puede tener una o varias máquinas virtuales instaladas (dependiendo del número de `cores` de ese computador) de distintas arquitecturas, donde se realizará la construcción de los paquetes.

Las máquinas virtuales tienen una distribución Debian GNU/Linux instalada, con los paquetes necesarios para realizar todo el proceso de construcción de los paquetes. Además de construir los paquetes, pueden firmarlos y subirlos al repositorio para lo cual, tienen una clave GPG [Koc13] que está dentro del anillo de `icebuilder`.

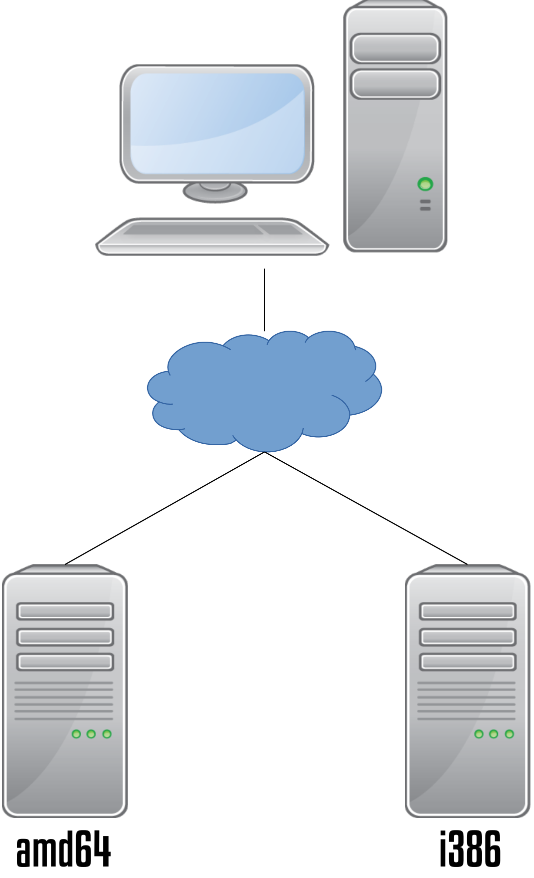

Figura 4.3: Woker

5 Metodología y herramientas

En el presente capítulo se explica la metodología de desarrollo aplicada para lograr los objetivos descritos con anterioridad en el capítulo 3.

En la segunda parte del capítulo se ofrecerá un visión de todas las herramientas que se han utilizado para el desarrollo de este PFC, en las cuales se engloban lenguajes de programación, editores de código fuente, etc.

La metodología de desarrolo escogida es SCRUM [Kni07], una metodología ágil que puede ser aplicada a casi cualquier proyecto, pero sin embargo se suele usar en desarrollo de software. SCRUM se adapta muy bien en proyectos en los cuales aparecen nuevos requisitos o estos cambian a medida que se avanza en el tiempo. Se realiza por medio de iteraciones planificadas al principio de cada una de ellas, y en las que se concluye con una revisión del trabajo realizado comprobando si cumplen con las necesidades.

El caso de este proyecto es un claro ejemplo en el cual aparecen nuevos requisitos o estos cambian por la aparición de nuevas tecnologías que permiten hacer las cosas de forma diferente. Se requiere pues de una metodología que sea adaptable a cambios y es por ello que se ha elegido SCRUM.

5.1 SCRUM

El enfoque de Scrum [PAW13] ha sido desarrollado para manejar el desarrollo de procesos. Se trata de un enfoque empírico aplicando las ideas de la teoría de control de proceso industrial al desarrollo de sistemas informáticos que resulta en un enfoque que retoma las ideas de flexibilidad, adaptabilidad y productividad.

La idea principal de Scrum es que el desarrollo del sistema envuelve muchas variables (requisitos, tiempo, recursos y tecnología) que pueden cambiar durante el proceso. Esto hace el proceso de desarrollo impredecible y complejo, requiriendo flexibilidad para ser capaz de responder a los cambios que irán surgiendo.

El proceso Scrum ayuda a mejorar la existencia de prácticas ingenieriles en la organización [NBM09], ya que implica frecuentes actividades de gestión destinadas a identificar las deficiencias o impedimentos en el desarrollo del proceso, así como las prácticas que se

utilizan.

5.1.1 Fases de Scrum

Scrum se desarrolla fundamentalmente en tres fases:

- **Pre-game:** Esta fase se puede dividir en dos sub-fases. La primera de ellas es la planificación, incluyendo la definición del sistema que se va a desarrollar. Se crea un «Product Backlog list», que es una lista con todos los requisitos del proyecto, también llamados «historias de usuario» realizada por el cliente y en el lenguaje del cliente. Los requisitos tienen prioridades para saber cual debe desarrollarse antes y una estimación de esfuerzo para medir la dificultad. Esta lista es actualizada continuamente con nuevos requisitos, herramientas, recursos, etc. La segunda sub-fase consiste en un diseño en alto nivel del sistema con las decisiones de diseño a tomar para aplicar *a posteriori*.
- **Development:** También conocida como fase de desarrollo. Mediante reuniones se decide que parte de las historias de usuario van a formar parte de cada uno de las iteraciones o «Sprints». Las iteraciones son intervalos de tiempo entre 2 y 4 semanas. Incluye fases más pequeñas para el desarrollo del sistema como *análisis*, *diseño*, *implementación*, *pruebas* y *entrega*. Cada una de las iteraciones realizadas concluyen con una reunión donde se trata si los objetivos han sido alcanzados y si el resultado es satisfactorio.
- **Postgame phase:** A esta fase se llega cuando se han cumplido los requisitos de entrega y se van a añadir las nuevas funcionalidades a una nueva versión. Ya no pueden añadirse nuevos requisitos o historias de usuario. Se llevan a cabo pruebas de integración y se realiza la documentación.

En al figura 5.1 se muestra un esquema de cada una de las actividades que se realizan en cada fase, con sus entradas y salidas. La fase de desarrollo es iterativa e incremental y de donde se obtienen las versiones del producto que posteriormente se incorporan poco a poco al producto final.

Roles y responsabilidades

Se pueden identificar seis roles en SCRUM que tienen distintas tareas y propósitos, estos son: Scrum Master, Product Owner, Scrum Team, Customer, User y Management.

Scrum Master: Es una función de administrador dentro de Scrum. Es el responsable de asegurarse que el proyecto se haga de acuerdo a las prácticas establecidas y de que el proyecto avanza según lo previsto. Se encarga también de liderar al equipo de desarrollo cuando se encuentra algún obstáculo, y de mantener el equipo de modo que se ocupe de las tareas de la forma más productiva posible.

Product Owner: Es el responsable del proyecto, y de su gestión ycontrol. Lo elige el Scrum

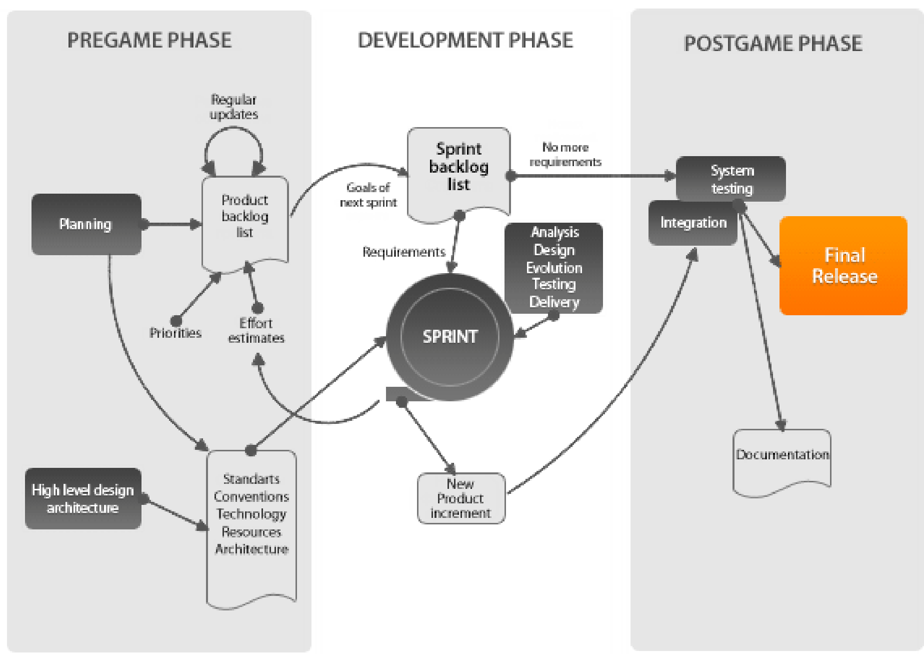

Figura 5.1: Scrum Process [PAW13]

Master. Participa sobre todo en la estimación de esfuerzo de los elementos.

Scrum Team: Es el equipo o uno de los equipos de desarrollo del proyecto, tienen la autoridad para decidir sobre las acciones y la organización necesaria para alcanzar los objetivos. También está implicado en toma de decisiones en cuanto a estimación del esfuerzo.

Customer o cliente: Participa en las tareas relacionadas con el «Product backlog» para el sistema que está siendo desarrollado o mejorado.

Management: Es responsable de las decisiones finales y de las normas que se tienen que seguir en el proyecto. También participa en la redacción de requisitos y objetivos a cumplir.

5.1.2 Prácticas

En SCRUM se llevan a cabo algunas prácticas que permiten llevar una mejor organización del proyecto, de lo que queda por hacer, de lo que se ha hecho y, que además, permite ser flexible frente a cambios que pueden surgir durante el desarrollo del proyecto.

Product Backlog Define todo lo que es necesario en el producto final basado en el conocimiento actual que se tiene. El «Product Backlog» define el trabajo que se tiene que hacer en el proyecto. Se compone de una lista de requisitos con prioridades y que se actualizan constantemente. Pueden incluir características, funciones, correcciones de errores, mejoras solicitadas, etc. Varias personas pueden participar en la generación del «Product Backlog», como clientes, equipos de desarrollo,... Este componente re-

presenta la visión del cliente respecto a los objetivos del proyecto. Está en constante evolución y se modifica con el paso de las iteraciones.

Estimación de esfuerzo Es un proceso interactivo, en el cual las estimaciones son más precisas cuanta más información se tiene. El «Product Owner» junto con el equipo de desarrollo son los responsables de realizar la estimación. Es común recurrir a diferentes métodos de estimación como el juicio de expertos[1]

Sprint Es el proceso por el cual se van adaptando los cambios al proyecto. El equipo de desarrollo se organiza él mismo para producir nuevos ejecutables en cada iteración que dura entre 2 y 4 semanas. Reuniones de planificación de sprints, reuniones diarias y Sprint Backlog son algunas de las actividades que se realizan en los Sprint.

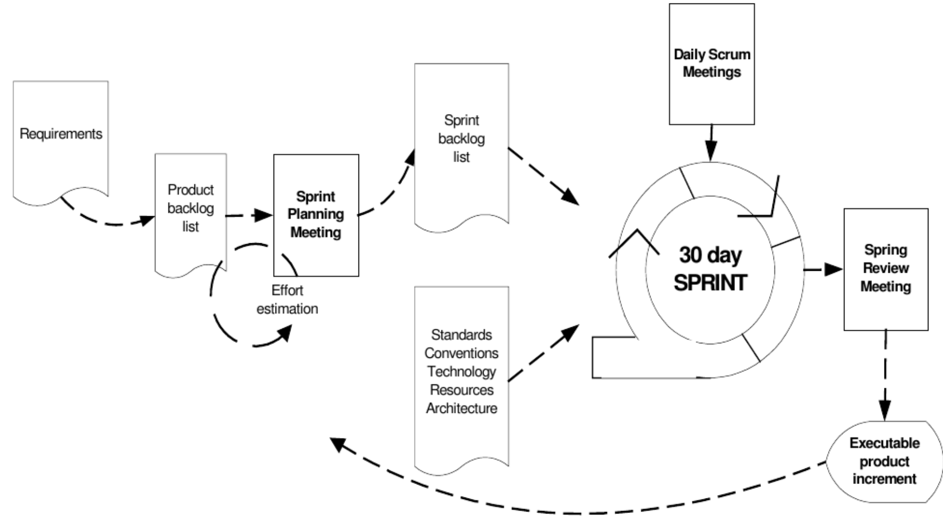

Figura 5.2: Esquema de un sprint en Scrum

En la figura anterior se muestra un esquema que ilustra los elementos con sus entradas y salidas que toman parte en una iteración completa de la metodología Scrum.

Sprint Planning meeting Es la reunión organizada por el Scrum Master y tiene dos fases. En la primera de ellas participan el cliente, usuarios, administradores y scrum team. En esta fase se definen los objetivos y la funcionalidad para el siguiente Sprint. La segunda fase se realiza entre el Scrum Master y el Scrum Team, centrándose en como se implementarán las cosas durante el Sprint.

Sprint Backlog Es el punto de partida de cada Sprint, es una lista de elementos del «Product Backlog» seleccionados para ser implementados en el siguiente Sprint o iteración. Estos elementos son seleccionados por el equipo de desarrollo y el «Product Owner» en la «Sprint Planning meeting».

[1]Es una técnica que se aplica para realizar estimaciones, en la cual se reúnen expertos que han trabajado en varios proyectos para estimar el presente proyecto.

Daily Scrum meeting Son reuniones diarias para mantener un seguimiento. Responde a preguntas del tipo: Qué se ha hecho desde la anterior reunión y que se va a hacer para la siguiente. También se discuten problemas que han surgido. Duran en torno a 15 minutos y es el «Scrum Master» quien las dirige.

Sprint Review meeting El último día del Sprint, se reúnen el Scrum Team y el Scrum Master y se presentan los resultados a los administradores, clientes, usuarios etc. Los participantes el prototipo entregado. Esta reunión debe dar un nuevo Backlog con nuevos elementos y si se debe o no cambiar de dirección en el desarrollo.

5.2 Herramientas

La realización de un proyecto de tal embergadura lleva consigo la utilización de muchas herramientas, que en algún momento del desarrollo han ayudado a resolver algún problema aportando alguna funcionalidad. En esta sección se nombran todas esas herramientas con una pequeña descripción. El por qué se usan esas herramientas y no otras está justificado durante la etapa del desarrollo del proyecto en el capítulo 6.

5.2.1 Herramientas para la gestión de proyectos

A la hora de gestionar el proyecto se ha elegido *redmine* [Lóp09]. Una herramienta para la gestión de proyectos que permiten abrir tickets para asignar tareas a los usuarios con prioridades. Dispone de diferentes plugin que añaden funcionalidad a la herramienta

5.2.2 Lenguajes de programación

Se han utilizado diferentes lenguajes de programación según las necesidades a cubrir en cada una de las etapas.

Python Lenguaje de alto nivel, interpretado y orientado a objetos. Se pueden construir prototipos en poco tiempo sin mucho esfuerzo. Ha sido utilizado para implementar algunas pruebas y para la primera iteración del proyecto.

C++ Creado por Bjarne Stroustrup [Str13], C++ proporciona mecanismos de orientación a objetos y compatibilidad con C. Es un lenguaje compilado que ofrece distintos niveles de abstracción al programador. Se utiliza como lenguaje de programación principal utilizando el excelente soporte que ofrece ICE para este lenguaje, tanto la implementación de persistencia como en otros servicios.

Bash BASH [Koc10] utilizado para realizar diferentes scripts para automatizar tareas relacionadas con la construcción de paquetes como puede ser el firmado, comprobación de claves o subida al repositorio. La gestión de las máquinas virtuales también se realiza mediante scripts permitiendo el encendido, apagado o toma de «snapshots» entre otras.

5.2.3 Control de versiones

Para el desarrollo del proyecto, se ha optado por usar controles de versiones para llevar el día a día del proyecto. Permitiendo de este modo la creación de ramas de desarrollo, teniendo así, siempre una versión estable y una versión inestable donde se desarrollan las distintas iteraciones.

Mercurial [Mac05] Ofrece un sistema de control de versiones distribuido y la posibilidad de llevar ese control fuera de línea, frente a los sistemas de control de versiones centralizados como CVS o Subversion. Se utiliza como control de versiones principal en el que se desarrolla la mayor parte del proyecto.

Git Sistema de control de versiones utilizado para mantener algunos de los paquetes Debian de prueba que se realizan durante el desarrollo del proyecto. Debian ofrece la herramienta *git-buildpackage*~citegit-buildpackage que ayuda en esta tarea.

5.2.4 Herramientas de desarrollo

Para la programación del proyecto se han utilizado multitud de herramientas tanto para programar, como para automatizar ciertas tareas como compilación y realización de pruebas.

Emacs [Sta13] Entorno de desarrollo de GNU que se usa como editor principal durante todo el proyecto, tanto de la programación del sistema distribuido como de la documentación.

Make [SM88] Permite la construcción y compilación automática de las aplicaciones. El proceso de compilación del proyecto está desarrollado con esa herramienta, incluido también el presente documento. También se utiliza en la creación de paquetes Debian.

equivs Herramienta para crear paquetes Debian vacíos. Estos paquetes suelen utilizarse para hacer un meta paquete como por ejemplo *gnome-desktopenvironment*, el cual instala el escritorio GNOME al completo. En concreto se ha utilizado para crear los paquetes de pruebas y para hacer un meta paquete que instale las dependencias necesarias para poner en funcionamiento este PFC.

clusterssh [Fer10] Herramienta utilizada para la configuración de varias máquinas en red. Permite ejecutar la misma orden en un número determinado de máquinas diferentes.

preseed [Tea10] Herramienta que permite responder a las preguntas que se realizan durante el proceso de instalación de Debian GNU/linux. Esto permite realizar instalaciones automáticas e incluso ofrece distintas posibilidades que la instalación normal, como por ejemplo, crear usuarios con sus contraseñas, elegir algunos paquetes, etc. *preseed* se utiliza en la instalación de las máquinas virtuales para automatizar en todo lo posible ese proceso.

GDB [SP14] Depurador para el compilador de GNU, elegido para realizar la depuración del código fuente programado en C++ de forma local o remota cuando sea necesario.

GCC-4.8 [Pla13] Compilador de GNU para C y C++.

reprepro [Lin12] Herramienta que permite crear un repositorio de paquetes Debian para varias arquitecturas con características similares a las de un repositorio de Debian oficial.

GPG [Koc13] Implementación del estándar OpenPGP definido por el RFC4880 [RFC07]. Permite cifrar y firmar datos entre muchas otras cosas. Se utiliza para crear las claves que se utilizarán para el firmado de paquetes que estén dentro del anillo de confianza del sistema distribuido.

devscripts-el Scripts que ayudan en las tareas de empaquetado en Debian GNU/Linux y que se usa con el editor de textos EMACS.

debhelper Colección de herramientas que se utilizan en las labores de empaquetado, por ejemplo generando el directorio debian inicial de un paquete.

Jed [Dav99] editor de textos similar a EMACS en cual se realizan algunas tareas de edición y programación de documentos pequeños tales como ficheros de configuración.

GNU Nano [Mal99] Editor de textos que viene instalado por defecto con la instalación de Debian GNU/Linux. Las configuraciones dentro de las máquinas virtuales se han realizado con este editor de textos.

Vi [Moo99] editor de textos que también viene instalado por defecto en Debian GNU/Linux y con el que se editan los ficheros de configuración de máquinas virtuales.

dose-builddebcheck [PA13] Permite conocer si un paquete es construible en un determinado entorno.

apt-rdepends Herramienta que se utiliza en algunas partes del sistema para conocer las dependencias de construcción de un paquete dado.

dupload Permite la subida automática de paquetes Debian al archivo, hacer un registro de la subida y enviarlo por e-mail. Como métodos de subida tiene FTP, *scp* y *rsync*.

dput Herramienta para subir paquetes Debian al archivo. Se puede especificar a que servidor se sube por medio de un argumento. El paquete se sube indicando uno o más `package.changes` y se subirán de uno en uno. Este programa puede subir los paquetes utilizando uno de lo métodos de subida habilitados, que actualmente son: FTP, scp, rsync, HTTP, HTTPS y local.

5.2.5 Herramientas y tecnologías de bases de datos

Una parte del proyecto necesita persistencia, y la persistencia se consigue mediante el uso de una base de datos donde se almacenen los datos necesarios para garantizarla. Ade-

más, para poder ser tolerable a fallos se necesita realizar ciertas operaciones dentro de una transacción. ICE ofrece su propio servicio de persistencia en el cual todo ello queda resuelto e integrado en el middleware. Para ello ICE utiliza una base de datos *Berkeley DB* donde almacena los estados de los objetos.

5.2.6 Herramientas de virtualización

La virtualización, que es una de las partes más importantes de este proyecto, se lleva a cabo con las siguientes herramientas:

libvirt [RH11] API de virtualización que permite administrar diferentes soluciones de virtualización como KVM [CA12] y Xen a través de una interfaz común. Se utiliza para crear y organizar las máquinas virtuales que se instalan en cada uno de los nodos del sistema distribuido.

QEMU [Bel13] Emulador de máquinas virtuales. Junto con *libvirt* se utiliza en la instalación y utilización de máquinas virtuales.

5.2.7 Herramientas para realización de pruebas

Para la realización de las pruebas se utiliza *prego*[2], una biblioteca para Python realizada dentro del grupo ARCO que permite la realización de pruebas soportando órdenes por consola, envío de señales a procesos, etc.

5.2.8 Documentación

La realización de documentación se ha llevado a cabo utilizando tanto herramientas de maquetación de textos como herramientas que permitan hacer diagramas para ilustrar y facilitar la comprensión del proyecto.

GIMP Editor de imágenes y retoque fotográfico. Utilizado en la maquetación de fotografías e ilustraciones.

Inkscape Editor de imágenes vectorial.

Libreoffice Draw Editor de imágenes vectorial utilizado para la realización de algunos diagramas tanto para documentación como para presentaciones.

Libreoffice Impress Programa para la realización de presentaciones, utilizado en algunas de las presentaciones que se han realizado de este proyecto.

Dia Editor de diagramas. Utilizado para construir los diagramas que se han utilizado para el desarrollo del proyecto.

LaTeX Lenguaje de maquetación de textos, utilizado para la elaboración del presente documento y algunas presentaciones.

[2] `https://bitbucket.org/arco_group/prego`

noweb Herramienta de programación literal utilizada durante algunas de las iteraciones.

arco-pfc Plantilla de LaTeXutilizada en la documentación de este proyecto.

5.2.9 Sistemas Operativos

De los sistemas operativos que existen a día de hoy se han utilizado dos, aunque uno de ellos, *raspbian*, únicamente ha sido para realizar algunas mediciones.

Debian GNU/Linux En su version *testing/unstable* compilada para arquitectura *amd64*. Es una distribución del sistema operativo GNU con núcleo Linux.

Raspbian Sistema operativo basado en Debian [Aok12] en el cual se realizan algunas pruebas de rendimiento.

5.2.10 Hardware empleado

Se han utilizado varios computadores tanto personales como proporcionados por el laboratorio ARCO. Estos computadores se han utilizado para construir un entorno de pruebas, realización de pruebas de rendimiento, etc:

Raspberry Pi [Com12] Computador de bajo coste, con arquitectura ARM. Ha sido utilizado para realizar pruebas de rendimiento cuyos resultados han sido comparados luego con las máquinas virtuales. Las características del *Raspberry Pi* son:

- ARM1176JZF-S a 700 MHz.
- 512 MB RAM
- GPU broadcom
- Alimentación de 3.5 W
- Sistema operativo Raspbian, que está basado en Debian [Aok12].

Ordenador Personal Utilizado durante la realización del proyecto cuando no se podía acceder al laboratorio ARCO. Características:

- Intel Core i5 3750k [Int13a].
- 8GB de Memoria RAM.
- Disco duro Wester Digital Black 1TB.

Dell XPS 420 Ordenador proporcionado por el laboratorio ARCO cuyas características son:

- Intel Core 2 Duo.
- 4 GB RAM
- Disco duro 500 GB.

Dell Precision T1650 Ordenador proporcionado por el laboratorio ARCO cuyas características son:

- Intel Core i7 3770 [Int13b].

- 8GB de Memoria RAM.
- Disco duro Wester Digital Black 1TB.

6 Desarrollo del proyecto

En el presente capítulo se describe la planificación junto con las distintas fases que se han dado durante el desarrollo del proyecto, divididas en iteraciones. Cada iteración corresponde con un incremento que aporta una nueva funcionalidad al conjunto del proyecto. Así mismo, en cada una de las iteraciones se ofrece información sobre las decisiones de diseño que se han tomado poco a poco junto con sus justificaciones e implementación.

Aunque la metodología Scrum, reserva una fase para documentar el desarrollo al final de cada iteración, este no ha sido el caso y la documentación se ha ido haciendo al mismo tiempo que que se desarrollaba el proyecto. Ello ha permitido plasmar con mayor detalle todo el proceso llevado a cabo junto con la toma de decisiones en cada momento, y así ofrecer una documentación mucho maś rica en detalles para facilitar su comprensión.

6.1 Fase Pre-game

En esta primera fase del proyecto se realiza la planificación. Se harán iteraciones entre 2 y 4 semanas, dependiendo de la dificultad y la duración de las tareas en cada iteración. Cabe destacar que el desarrollo se ha compatibilizado con las clases por lo que en ocasiones no se dedican más de 2 o 3 horas al día.

Dados los objetivos planteados en el capítulo 3, se trata de construir un sistema distribuido para construir software que sea fácil de instalar, actualizar y de mantener. Además de ello tiene que ser tolerable a fallos para poder conocer en todo momento que es lo que queda por hacer y que es lo que ya se ha hecho. Para realizar un proyecto de esta índole, es preciso contar con un entorno limpio donde realizar la construcción de paquetes. Por lo tanto se podría subdividir el proyecto en dos proyectos:

1. Construcción de un entorno limpio.
2. Construcción de la infraestructura distribuida para la construcción de software.

El entorno limpio deberá ser capaz de construir paquetes Debian, firmarlos[1] y subirlos al repositorio de forma automática. Para realizar esto se necesitaría un repositorio de paquetes debian, y que solamente permita la subida de paquetes a los usuarios que tienen el rol de

[1] Las firmas se realizarán mediante clave GPG

«Developers» dentro de la organización.

La infraestructura distribuida estará compuesta por nodos que serán los computadores de la organización y que será donde se instale el entorno limpio para realizar la construcción de paquetes. La infraestructura debe disponer de una buena escalabilidad.

6.1.1 Requisitos: Product Backlog

Como se explicó en el capítulo 5, en la sección 5.1, el «Product Backlog» es la visión que tiene el cliente de los objetivos del proyecto y es un documento en constante evolución. Los requisitos están compuestos de:

- ID: Identificador del requisito.
- Descripción breve: Lo más breve posible.
- Coste: El coste que representa en el proyecto.
- Condiciones de satisfacción: Condiciones que harán válido ese requisito.
- Valor aportado: Un valor numérico con el valor aportado al proyecto.

Dado que el «Product Backlog» es un documento en constante evolución, para facilitar la lectura de la memoria, el campo *ID* se hace coincidir con las iteraciones desarrolladas para que sea más fácil saber qué se ha añadido en cada momento y poder así ver una evolución.

A continuación se listan los requisitos desde el punto de vista del cliente, son requisitos descritos generalmente sin lenguaje excesivamente técnico.

A1
- Descripción: Construir un entorno de pruebas simulando uno real.
  - Coste: 100.
  - Condiciones de satisfacción: Tener varios computadores de diferentes arquitecturas (*i386*, *amd64*)
  - Valor: 200.

A2
- Descripción: Instalar el middleware ZeroC Ice en el entorno de pruebas.
  - Coste: 50.
  - Condiciones de satisfacción: Instalar ZeroC Ice en los computadores para realizar las pruebas.
  - Valor: 100.

A3
- Descripción: Construir paquetes en máquinas aisladas.
  - Coste: 150.
  - Condiciones de satisfacción: Entorno para construir paquetes Debian.
  - Valor: 350.

A4
- Descripción: Recuperación de información frente a fallos.

- Coste: 200.
- Condiciones de satisfacción: Si surge un fallo inesperado la información esté guardada.
- Valor: 250.

A5
- Descripción: Utilizar los ordenadores disponibles para realizar los trabajos.
- Coste: 100.
- Condiciones de satisfacción: Saber qué ordenadores están conectados a la red y cuales no
- Valor: 150.

A6
- Descripción: Repositorio para servir los paquetes.
- Coste: 100.
- Condiciones de satisfacción: Crear un repositorio en un servidor que sirva los paquetes construidos a los usuarios.
- Valor: 100.

A7
- Descripción: Utilizar paquetes seguros.
- Coste: 110.
- Condiciones de satisfacción: El sistema garantice que los paquetes que se utilizan no han sido alterados.
- Valor: 150.

En las siguientes tablas se muestran las iteraciones, una a una con las tareas que se han ido realizando en cada una de ellas.

En la primera iteración, se trata de construir un entorno de pruebas donde poder probar los futuros prototipos.

Iteración	Id	User Story
It0: El sistema base	0.0	Elección de alternativas de virtualización
	0.1	Creación de una estructura básica
	0.2	Diseño y creación de una red virtual
	0.3	Crear una máquina virtual con *amd64*
	0.4	Crear una máquina virtual con *i386*
	0.5	Instalación de ICE en todas las máquinas
	0.6	Diseño e implementación «Hello world» en ICE

Cuadro 6.1: Iteración 0.

Los requisitos de la siguiente iteración van orientados a construir un entorno limpio, uno de los grandes objetivos de este proyecto. Se hace un estudio sobre las aplicaciones que existen y se construye:

Iteración	Id	User Story
It1: Entorno limpio para construir paquetes	1.0	Elección de alternativas de virtualización
	1.1	Diseño y creación de una red virtual
	1.2	Crear un grupo de máquinas virtuales
	1.3	Automatizar instalación de máquinas virtuales
	1.4	Dotar de permisos adecuados a las máquinas virtuales
	1.5	Construir un fichero preseed
	1.6	Diseño e implementación de scripts para construir paquetes
	1.7	Automatizar la construcción de paquetes con una orden
	1.8	Habilitar salvado de «snapshots» de máquinas virtuales
	1.9	Scripts para actualizar máquinas virtuales

Cuadro 6.2: Iteración 1.

Tras tener un prototipo con una mínima funcionalidad como es la de construir paquetes en entornos limpios, se decide hacer el sistema un poco más seguro habilitando la firma de paquetes con GPG.

Iteración	Id	User Story
It2: Firmado de paquetes con GPG	2.0	Firmar paquetes usando GPG
	2.1	Generar firmas GPG para las Máquinas virtuales
	2.2	Exportar las claves a su correspondiente computador
	2.3	Firmar las claves de los computadores

Cuadro 6.3: Iteración 2.

Una de las cosas que hacen a este proyecto innovador y diferente es la persistencia, que permitirá que se pueda recuperar la información frente a un fallo, estos son los requisitos de la siguiente iteración:

Iteración	Id	User Story
It3: Persistencia de objetos con Freeze	3.0	Jerarquía de objetos para crear un árbol
	3.0	Jerarquía de objetos
	3.1	Elección de lenguaje de programación compatible con Freeze
	3.2	Implementar persistencia usando Freeze
	3.3	Almacenar los datos en la base de datos de ICE
	3.4	Implementar persistencia con un «evictor»
	3.5	Añadir información a los paquetes Debian

Cuadro 6.4: Iteración 3.

Los paquetes que se construyan, deben servirse en un repositorio para que todos los usuarios tengan acceso a ellos. En esta iteración la tarea fundamental será la de crear un repositorio y firmarlo para servir los paquetes.

Iteración	Id	User Story
It4: Repositorio de paquetes Debian	4.0	Creación de un repositorio de paquetes Debian
	4.1	Añadir soporte para arquitectura *i386* y *amd64*
	4.2	Habilitar subida de paquetes solamente a los «developers»
	4.3	Hacer el repositorio accesible
	4.4	Firmar las claves de las máquinas virtuales
	4.5	Firmar el repositorio

Cuadro 6.5: Iteración 4.

La próxima iteración se centra en la construcción de un componente del sistema que será el encargado de organizar todas las construcciones de paquetes, el *manager*:

Iteración	Id	User Story
It5: Mánager para organizar la construcción de paquetes	5.0	Generar el árbol de dependencias de construcción de un paquete
	5.1	Conocer las dependencias de construcción de un paquete dado
	5.2	Saber si un paquete se encuentra disponible en el repositorio
	5.3	Determinar si un paquete se puede construir
	5.4	Implementar persistencia
	5.5	Implementar algoritmo DFS para recorrer el árbol
	5.6	Saber qué computadores están conectados a la red

Cuadro 6.6: Iteración 5.

El manager, enviará los paquetes a que se construyan a los *Workers*, la realización de está iteración se centra en ello:

Iteración	Id	User Story
It6: Worker para construir los paquetes	6.0	Tener «Workers» en los nodos del sistema distribuido
	6.1	Detener la construcción de un paquete
	6.2	Implementar persistencia
	6.3	Tener varias máquinas virtuales en un «Worker»
	6.4	Implementar persistencia
	6.5	Enviar las órdenes de construcción por transacciones

Cuadro 6.7: Iteración 6

Por último, es necesario desplegar el sistema, se usará *IceGrid*, que es uno de los servicios más importantes que tiene ICE.

Una vez vistos los requisitos se procede con el desarrollo de cada una de las iteraciones donde se detallarán las decisiones tomadas durante el desarrollo del proyecto.

Iteración	Id	User Story
It7: Despliegue con IceGrid	6.0	Balanceo de carga
	6.1	Transparencia de localización
	6.2	Dar de alta nodos («Workers») en el sistema
	6.3	Tener información de los paquetes Debian

Cuadro 6.8: Iteración 7.

6.2 Iteración 0: El sistema base

En esta iteración se define el sistema base para todo el proyecto. Al tratarse de un sistema distribuido, y dado que no se dispone de infinitos computadores, se opta por la utilización de máquinas virtuales como medio para la realización de las pruebas y montar el sistema, por lo tanto, se requiere de un estudio de las alternativas que existen para la virtualización.

User Stories

- **Id:** 0.0
  - **Como:** Programador.
  - **Quiero:** Hacer un estudio de alternativas de virtualización
  - **De modo que:** Pueda obtener una visión clara de lo que hay y elegir la opción que mejor se adapte a las necesidades.
  - **Notas:** Estudio de diferentes tecnologías de virtualización existentes que sean compatibles con GPL y que se adapten a las necesidades.
  - **Prioridad:** Alta
- **Id:** 0.1
  - **Como:** Programador.
  - **Quiero:** Crear una estructura básica.
  - **De modo que:** Pueda realizar las pruebas.
  - **Notas:** Plataforma con una máquina virtual de cada arquitectura para probar el sistema en el futuro.
  - **Prioridad:** Alta
- **Id:** 0.2
  - **Como:** Programador.
  - **Quiero:** Crear una red virtual.
  - **De modo que:** Pueda recrear un entorno real con muchos computadores.
  - **Notas:** Se podrán añadir tantos computadores como se quiera para recrear un entorno real de trabajo.
  - **Prioridad:** Alta

- **Id:** 0.3
  - **Como:** Programador
  - **Quiero:** Crear una máquina virtual con arquitectura *amd64*
  - **De modo que:** Pueda simular una máquina de esa misma arquitectura.
  - **Notas:** Permitirá construir paquetes para arquitectura amd64, simulando una máquina real con Debina GNU/Linux. Además se podrá clonar para añadir n máquinas virtuales iguales.
  - **Prioridad:** Normal.
- **Id:** 0.4
  - **Como:** Programador
  - **Quiero:** Crear una máquina virtual con arquitectura *i386*
  - **De modo que:** Pueda simular una máquina de esa misma arquitectura.
  - **Notas:** Permitirá construir paquetes para arquitectura amd64, simulando una máquina real con Debina GNU/Linux. Además se podrá clonar para añadir n máquinas virtuales iguales.
  - **Prioridad:** Normal.
- **Id:** 0.5
  - **Como:** Programador
  - **Quiero:** instalar el middleware ZeroC Ice[2]
  - **De modo que:** Se pueda desarrollar una aplicación distribuida.
  - **Notas:** Esta tarea permitirá disponer de los medios que permitan programar un «hello world!» para probar que todo funciona.
  - **Prioridad:** Normal.
- **Id:** 0.6
  - **Como:** Programador
  - **Quiero:** Diseñar, implementar y probar un «Hello World» en ZeroC Ice
  - **De modo que:** Se pueda probar que todo funciona correctamente.
  - **Notas:** Esta tarea permitirá ejecutar un «hello world!» para probar que todo funciona.
  - **Prioridad:** Normal.

6.2.1 Análisis

El fin que se pretende alcanzar es construir un sistema base con máquinas virtuales de arquitecturas *i386*, *amd64* y *armel* tal y como se muestra en la figura 6.1

[2] `http://www.zeroc.com/`

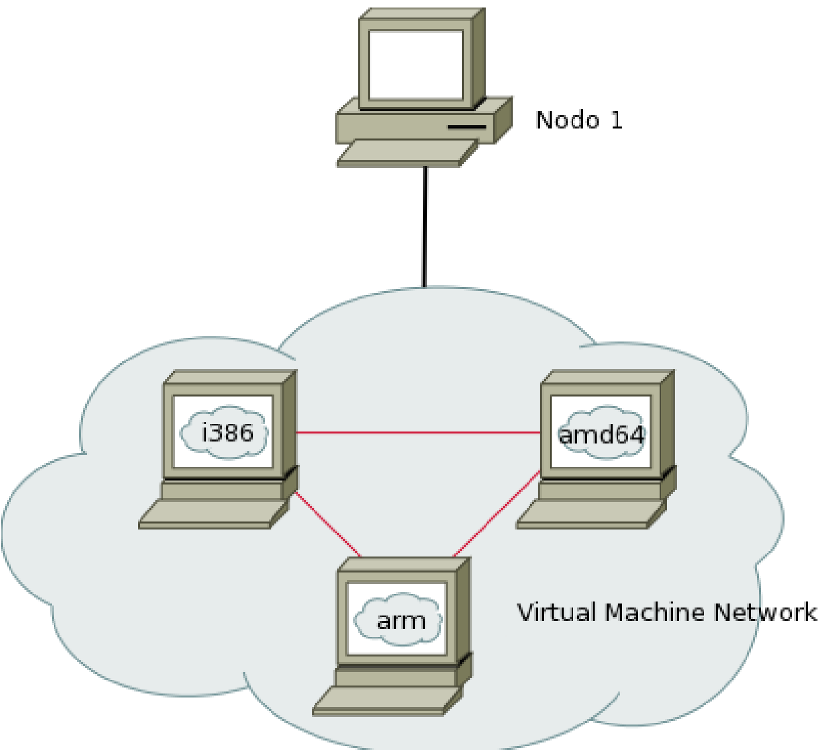

Figura 6.1: Sistema base formado por un computador y varias máquinas virtuales

Una máquina virtual es un sistema de virtualización por software, es decir, simula un sistema físico, en este caso un computador, con unas características de hardware determinadas. Cuando se ejecuta, proporciona un ambiente de ejecución similar a todos los efectos a un computador físico, con su CPU (puede ser más de una), memoria RAM, tarjeta gráfica, tarjeta de red, sistema de sonido, dispositivos de almacenamiento secundario.

Con esto se consigue ejecutar uno o más computadores dentro de un mismo computador de manera simultanea, y se resuelve así el problema de no tener muchos computadores disponibles donde poder probar el sistema.

Alternativas de virtualización

Se procede al estudio de las distintas alternativas que existen para virtualizar computadores de diferentes arquitecturas. Para ello se consultan varias herramientas basando la búsqueda en los los siguientes requisitos:

- Se requiere de la virtualización de distintas arquitecturas, como mínimo *i386* y *amd64*, aunque también sería deseable poder tener *armel* y otras como las que soporta oficialmente Debian GNU/Linux.
- Tener versión para Debian GNU/Linux.
- Deben poderse realizar las siguientes acciones con las máquinas virtuales:
    - **Iniciar**: La herramienda debe permitir el inicio de una o varias máquinas virtuales

cuando se requiera.

- **Parar**: La herramienta debe permitir parar una o varias maquinas virtuales cuando se requiera manteniendo otras encendidas.
- **Snapshot**: es necesario que se permita la captura de «snapshots» para congelar el estado de las máquinas virtuales.
- **Clonar**: Se debe permitir el clonado de una o varias máquinas virtuales con el fin de poder añadir más máquinas virtuales de forma rápida.
- **Reiniciar**: Se debe permitir que una o varias máquinas virtuales se reinicien cuando se requiera.
- Que su licencia sea una de las licencias de software libre que se define en la FSF como compatibles con la GPL.

- Además de todo ello sería deseable que las máquinas virtuales se puedan gestionar mediante comandos con el fin de construir algún script para automatizar las operaciones.

En base a estos criterios se procede al estudio de las siguientes tecnologías de virtualización: VMware [VMw13], VirtualBox [Cor13], QEMU [Bel13], Gnome Boxes, Libvirt [RH11]

VMWare: Software de virtualización disponible para computadores x86. Este programa no dispone de una licencia de software libre, por lo tanto no cumple con uno de los requisitos y se descarta su uso por esta razón.

VirtualBox: VBOX es un software de virtualización para arquitecturas *x86* y *amd64*, es multiplataforma, lo cual quiere decir que se pueden tener instancias en varios sistemas operativos como *Windows Mac* o *GNU/Linux*. Dispone de gestión mediante comandos a través del comando `VBoxManager`, así como de una interfaz gráfica donde poder instalar el sistema operativo deseado. Su licencia es *GPL*, sin embargo, un punto en contra es la no virtualización de la arquitectura *arm*. Se pueden realizar todas las operaciones descritas en los requisitos, por lo tanto se convierte en candidato.

QEMU: En inglés Quick EMUlator, es capaz de virtualizar diferentes tipos de CPU, y de este modo se pueden virtualizar diferentes arquitecturas. Dispone de interfaz gráfica y de administración mediante comandos, gracias a esto y con la ayuda del manual, es muy sencillo emular un procesador con arquitectura *arm* y por consiguiente realizar la instalación de una versión de Debian para esa arquitectura. Se pueden realizar todas las acciones descritas en los requisitos, por lo que se convierte en candidato.

gnome-boxes: Gnome boxes: Es una aplicación del escritorio GNOME para acceder a sistemas virtuales ya sea a través del propio computador o remotamente. Pretende tener funcionando rápidamente y de forma fácil máquinas virtuales. Gracias a su sencilla interfaz resulta extremadamente fácil de usar. Está diseñado para alguien que sólo requiera encender o apagar lás máquinas virtuales. Utiliza *qemu* como motor para vir-

tualizar pero no se pueden realizar todas las operaciones que sí se pueden realizar con *qemu* o *VirtualBox*.

Libvirt: Es una API para virtualización que puede realizar todas las operaciones descritas en los requisitos además de otras como monitorizar y modificar las máquinas virtuales entre otras. Dispone de una interfaz gráfica fácil de usar donde se puede ver la carga de los sistemas virtualizados. Permite ejecutar órdenes por comando a través del comando `virsh` o `virt`. Tiene licencia LGPL, y al igual que *gnome-boxes* usa *qemu*, por lo tanto se convierte en candidato.

Tras este análisis y tras haber probado cada una de las alternativas, la opción final ha sido utilizar *Libvirt*. Dispone de un abanico de operaciones más amplio para interactuar con los sitemas virtualizados, además al utilizar *qemu*, permite emular procesadores *arm* y con ello también se puede dar soporte a esta arquitectura, cosa que con *VirtualBox* no se puede hacer. Cumple también con el último requisito, que es que se pueda administrar por comandos para poder así construir scripts que automaticen todo lo posible el proceso. Teniendo además la posisibilidad de usar *Python* o *C* entre otros para programar las operaciones.

6.2.2 Diseño

Para poder ejecutar las máquinas virtuales, *libvirt* ofrece unas configuraciones por defecto. Ofrece una interfaz de red llamada *default* donde cada máquina virtual que se cree obtendrá una dirección *IP*. Por otra parte ofrece un *pool*[3] por defecto.

Estas opciones, aunque vienen bien para probar, se antojan insuficientes para los objetivos de este proyecto. Se crearán una red virtual y un *pool* expeciales para el proyecto donde se configurarán las máquinas virutales necesarias para su correcto funcionamiento. Esta decisión se toma con vistas a aislar el sistema lo más posible, ya que los usuarios de un computador pueden usar *libvirt* para otro uso totalmente diferente a este. Por lo tanto se dejan las configuraciones por defecto intactas y sin usar y se crean unas nuevas.

Creación de la red

La creación de la red puede comprobarse en el listado 6.1. Las configuraciones en *libvirt* se hacen mediante lenguaje XML.

```
<network>
<name>icebuilder</name>
<forward mode='nat'/>
<bridge name='virbr1' stp='on' delay='0'/>
<mac address='52:54:00:6F:5E:D8'/>
<ip address='192.168.122.1' netmask='255.255.255.0'>
  <dhcp>
    <range start='192.168.122.2' end='192.168.122.254'/>
    <host mac="00:AA:00:00:00:11" name="Bully" ip='192.168.122.11'/>
```

[3]El pool se denomina al directorio o directorios donde se guardarán las máquinas virtuales.

```
      <host mac="00:AA:00:00:00:12" name="Rigodon" ip='192.168.122.12'
         />
      % <host mac="00:AA:00:00:00:13" name="Phobos" ip='192.168.122.13
         '/>
      % <host mac="00:AA:00:00:00:14" name="Deimos" ip='192.168.122.14
         '/>
    </dhcp>
  </ip>
</network>
```

Listado 6.1: Fichero de configuración de la red net-icebuilder.xml

La IP de la red será la 192,168,122,1, pudiendo añadir 254 máquinas virtuales en el caso en el que fuese necesario. Esta red, tendrá una dirección MAC de *52:54:00:6F:5E:D8*, y el nombre por el que se conocerá dentro del sistema GNU/Linux será de *virbr1* tal y como se muestra en la figura 6.2.

```
tato@Superman: ~
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
tato@Superman:~$ /sbin/ifconfig
eth0      Link encap:Ethernet  HWaddr bc:5f:f4:53:96:7b
          inet addr:192.168.1.5  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::be5f:f4ff:fe53:967b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4405919 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3847616 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5089982076 (4.7 GiB)  TX bytes:948332628 (904.4 MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:899 errors:0 dropped:0 overruns:0 frame:0
          TX packets:899 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:143607 (140.2 KiB)  TX bytes:143607 (140.2 KiB)

virbr1    Link encap:Ethernet  HWaddr 52:54:00:6f:5e:d8
          inet addr:192.168.122.1  Bcast:192.168.122.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

tato@Superman:~$
```

Figura 6.2: Captura que muestra la interfaz *virbr1*

Si en un futuro se necesitase o se quisiese incluir más máquinas virtuales bastaría con añadirlas el fichero XML con una IP y una MAC. Aunque esto *a priori* no es necesario, ya que cuando una nueva máquina virtual se instala cogería la dirección IP a través del DNS, se

hace así con el fin de conocer de antemano las *IP* para poder conectarse luego a las máquinas virtuales mediante SSH para realizar tareas de administración o mantenimiento.

Creación del pool

Para la creación del pool simplemente se necesita un directorio donde se instalarán las máquinas virtuales. El fichero de configuración es el siguiente:

```
<pool type="dir">
<name>icebuilder</name>
<target>
  <path>/home/user/.icebuilder</path>
</target>
</pool>
```

Listado 6.2: Fichero de configuración del pool poolt-icebuilder.xml

Como nombre se tiene *Icebuilder* y como ruta de instalación un directorio oculto llamado `.icebuilder` dentro del *home* del usuario.

Diseño del Hello World

Con el fin de construir un programa que sirva para probar el entorno que se quiere generar, se implementará un *Hello World*. El diseño del fichero SLICE es el que se presenta a continuación:

```
module Prueba {
  interface Hello {
      void write(string str);
  };
};
```

Listado 6.3: Hello.ice

Se creará un módulo llamado `Prueba` que tendrá la interfaz `Hello` y que implementará un método llamado `write` cuyo argumento es un `string` en el cual irá el *Hello World*.

6.2.3 Implementación

Tras instalar ambas máquinas virtuales con la API libvirt se procede a la implementación del *Hello World* [DV13]. Para ello se ha elegido el lenguaje de programación *Python*q que permite crear prototipos de forma rápida. El cliente y el servidor del *HelloWorld* se muestran a continuación.

```
import sys
import Ice
Ice.loadSlice('Printer.ice')
import Example
```

```
class PrinterI(Example.Printer):
  def write(self, message, current=None):
      print message
      sys.stdout.flush()

class Server(Ice.Application):
  def run(self, argv):
      broker = self.communicator()
      servant = PrinterI()

      adapter = broker.createObjectAdapter("PrinterAdapter")
      proxy = adapter.add(servant, broker.stringToIdentity("printer1
         "))

      print proxy

      adapter.activate()
      self.shutdownOnInterrupt()
      broker.waitForShutdown()

      return 0

server = Server()
sys.exit(server.main(sys.argv))
```

Listado 6.4: Servidor «HelloWorld» distribuido en Python

ZeroC Ice permite que el cliente o servidor puedan ser desarrollados en lenguajes de programación diferentes, por requisitos o por cualquier otra razón. Para la implementación del cliente se ha utilizado también *Python* como lenguaje:

```
import sys
import Ice
Ice.loadSlice('Printer.ice')
import Example

class client (Ice.Application):
  def run(self, argv):
      proxy = self.communicator().stringToProxy(argv[1])
      printer = Example.PrinterPrx.checkedCast(proxy)

      if not printer:
          raise RuntimeError('Invalid proxy')

      printer.write('Hello World!')

      return 0

sys.exit(client().main(sys.argv))
```

Listado 6.5: Cliente «HelloWorld» distribuido en Python

6.2.4 Pruebas

Las pruebas realizadas para comprobar el funcionamiento de la presente iteración se centran en:

- Que el servidor, muestra el nombre del sirviente, en este caso `Printer1`.
- Comprobar que el servidor está funcionando.
- Comprobar que se obtiene el mensaje «Hello World!».

Los resultados de las pruebas arrojan resultados satisfactorios, por lo que se procede a la fase de entrega para validar la iteración.

6.2.5 Entrega

Una vez realizadas todas las tareas, se observa que se obtienen todos los resultados esperados satisfactoriamente. A modo de resumen se enumeran las teras que se han llevado a cabo durante esta iteración.

- Se ha creado un entorno de pruebas que servirá para probar en futuras iteraciones el avance del proyecto.
- Hay una máquina virtual con *amd64* y otra con *i386*, por lo que se tienen distintas arquitecturas para poder construir los paquetes en un futuro.
- Las máquinas virtuales están aisladas del resto de usuarios con una configuración específica.
- Se ha instalado ZeroC Ice con éxito.
- Se ha implementado un «Hello World» en Python.
- Las pruebas han arrojado resultados satisfactorios

Se valida el prototipo y se procede con el siguiente incremento o iteración.

6.3 Iteración 1: Entornos limpios para construir paquetes

La construcción de los paquetes debe realizarse en entornos aislados. Un entorno aislado o limpio es aquel que tiene instalados los programas necesarios para realizar la construcción de los paquetes. Cuando un paquete se instala, también se instalan sus dependencias. Esas dependencias tienen a su vez otras dependencias, o incluso puede que para construir un paquete no hagan falta tener instalado un determinado programa. En este caso se dice que el entorno está sucio.

Un computador el cual usa alguien siempre tendrá mucho software instalado, ese software ensucia el sistema porque pueden tener *bugs* que afecten a su funcionamiento o volverse inestable con alguna actualización.

Por estas razones, es necesario disponer de un entorno limpio donde realizar la construcción de los paquetes.

En esta iteración se aborda este problema y la solución que se ha dado para este proyecto. Es por ello que se seleccionan las siguientes historias de usuario para ser tratadas en esta iteración.

User Stories

Se seleccionan las siguientes historias de usuario para ser desarrolladas en esta iteración:

- **Id:** 1.0
  - **Como:** Programador.
  - **Quiero:** hacer un estudio de los entornos limpios que existen
  - **De modo que:** Pueda obtener una visión clara de lo que hay y elegir la opción que mejor se adapte a las necesidades.
  - **Notas:** Estudiar que ofrece Debian al respecto y decidir si se utiliza algo existente o crear algo nuevo.
  - **Prioridad:** Alta
- **Id:** 1.1
  - **Como:** Programador.
  - **Quiero:** Crear una red virtual
  - **De modo que:** Pueda aislarse las máquinas virtuales que formarán parte del sistema.
  - **Notas:** No se deben mezclar las máquinas virtuales con algunas posibles máquinas que puedan tener los usuarios del computador instaladas para sus propias necesidades, es por ello la necesidad de ponerlas en una red distinta.
  - **Prioridad:** Alta

- **Id:** 1.2
  - **Como:** Programador.
  - **Quiero:** Crear un grupo de máquinas virtuales
  - **De modo que:** Se conecten a la red virtual creada previamente y estén aisladas del resto del equipo.
  - **Notas:** No se deben mezclar las máquinas virtuales con máquinas virtuales que puedan tener los usuarios del computador instaladas para sus propias necesidades. Por ello tendrán sus propios directorios.
  - **Prioridad:** Alta
- **Id:** 1.3
  - **Como:** Programador.
  - **Quiero:** Realizar la instalación de máquinas virtuales automáticamente
  - **De modo que:** El proceso sea desatendido.
  - **Notas:** Ninguna.
  - **Prioridad:** Alta
- **Id:** 1.4
  - **Como:** Programador.
  - **Quiero:** Instalar las máquinas virtuales con un sistema mínimo.
  - **De modo que:** Tengan todas las herramientas necesarias para realizar la construcción de paquetes Debian.
  - **Notas:** Ninguna.
  - **Prioridad:** Alta
- **Id:** 1.5
  - **Como:** Programador.
  - **Quiero:** Dotar a las máquinas virtuales con los permisos y usuarios necesarios.
  - **De modo que:** Se puedan construir paquetes sin intervención del usuario.
  - **Notas:** Ninguna.
  - **Prioridad:** Alta
- **Id:** 1.6
  - **Como:** Programador.
  - **Quiero:** Diseñar un fichero preseed
  - **De modo que:** Responda automáticamente a las preguntas del instalador de Debian.
  - **Notas:** *Preseed* permite responder automáticamente a las preguntas del instalador de Debian, pudiendo así configurar los valores que se le pasarán al instalador.

- **Prioridad:** Alta

- **Id:** 1.7
  - **Como:** Programador.
  - **Quiero:** Construir scripts para automatizar la construicción de paquetes.
  - **De modo que:** Se cumplan todos los pasos en la etapa de construcción de un paquete Debian
  - **Notas:** Los scripts usarán algunas herramientas que proporciona Debian para facilitar la tarea.
  - **Prioridad:** Alta

- **Id:** 1.8
  - **Como:** Usuario.
  - **Quiero:** Tener una orden para construir un paquete del tipo **orden nombe_del_-paquete arquitectura**.
  - **De modo que:** Todo el proceso de construcción se realice con una única orden.
  - **Notas:** Esta orden será ejecutada desde el sistema distribuido, por eso se envuelve todo el proceso en una única orden
  - **Prioridad:** Alta

- **Id:** 1.9
  - **Como:** Usuario.
  - **Quiero:** Poder salvar el estado de las máquinas virtuales
  - **De modo que:** Pueda volver a un estado anterior.
  - **Notas:** Esta historia puede servir para recuperación frente a fallos, ya que se podría volver al estado inicial de la máquina virtual en cualquier momento.
  - **Prioridad:** Alta

- **Id:** 1.10
  - **Como:** Usuario.
  - **Quiero:** Poder actualizar las máquinas virtuales
  - **De modo que:** Pueda tener siempre un entorno actualizado.
  - **Notas:** Esta historia permitirá que antes de construir un paquete, se actualice la máquina virtual. Cuando se termine el proceso se podrá volver al estado anterior, con una máquina virtual mínima y lista para construir el siguiente paquete.
  - **Prioridad:** Alta

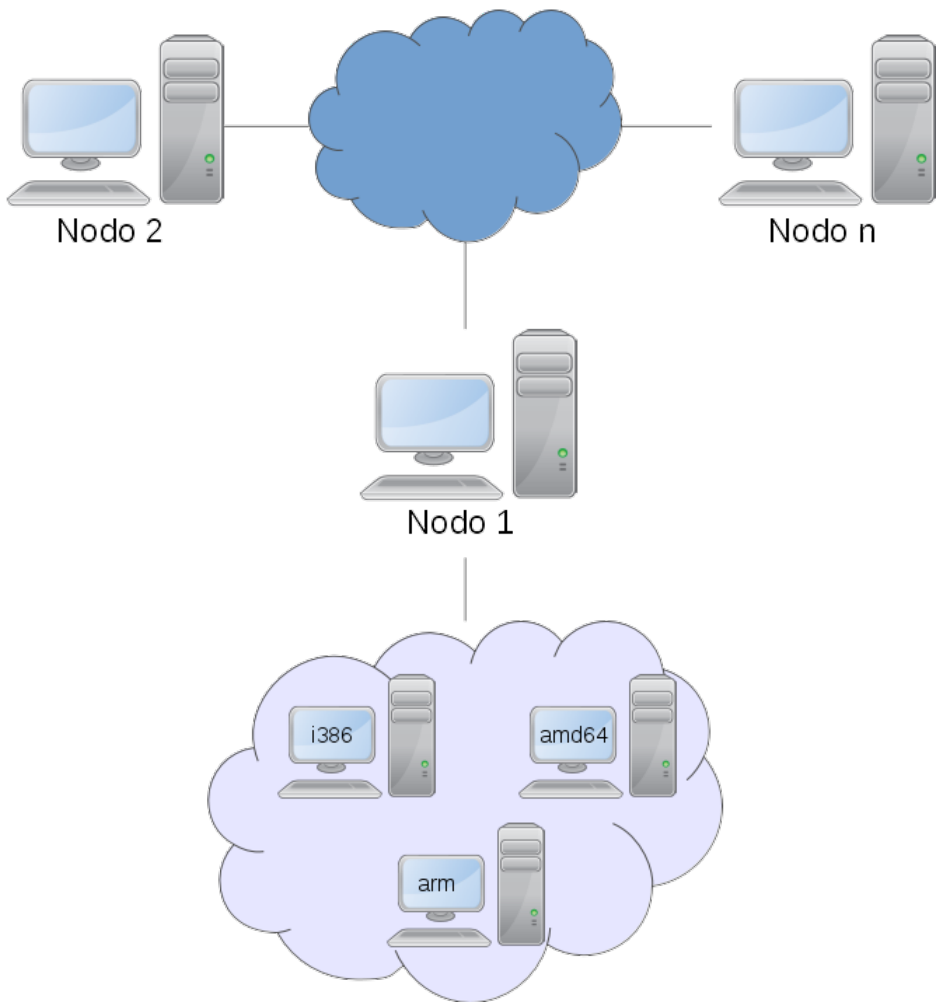

Figura 6.3: Arquitectura de un Nodo y del sistema

6.3.1 Análisis

Tras tener el sistema mínimo instalado que consta de una máquina virtual de cada arquitectura (*i386* y *amd64*), se parte entonces, de una infraestructura como la de la figura 6.3

A partir de aquí, el proyecto puede tomar dos vertientes, que están basadas en el uso o no de *pbuilder* [Uek07]. Un sistema de construcción automática de paquetes Debian donde el objetivo, es poder generar paquetes Debian sin ensuciar el sistema con las dependencias de los paquetes que se están construyendo. El dilema es, ¿usar pbuilder para construir los paquetes o aprovechar las máquinas virtuales que ya son un entorno limpio?

Entorno limpio a utilizar

Uno de los requisitos del sistema es, que aprovechando los computadores disponibles en el laboratorio ARCO, se construirían los paquetes, pero surge un problema al que aún no se le ha dado solución, ¿qué pasaría si un computador se apagaba en medio de la construcción de un paquete?. Este caso sirve de ejemplo ilustrativo para poder plantear la resolución del

camino a tomar a partir de ahora, porque, ¿qué pasaría si se necesita construir un paquete muy grande y en mitad de la construcción se apaga el computador? Una de las opciones sería empezar al día siguiente con un computador de la misma arquitectura elegido por el sistema, pero, ¿qué sucedería si la construcción durara más tiempo del que los computadores estén encendidos? Se tendría que resolver eso de algún modo. Una posible solución sería, no apagar ese computador hasta que termine la construcción del paquete, pero entonces se estaría alterando la forma en la que funciona el laboratorio. El edificio donde se encuentre el entorno de trabajo puede apagar la electricidad por multitud de razones y esto ya supone un problema de por sí, y el cual hay que tener en cuenta.

Con el uso de máquinas virtuales se abren nuevas opciones y nuevas formas de resolver el problema, incluso dando soluciones a otros nuevos problemas. Las *snapshots* son una operación que se puede hacer sobre las máquinas virtuales y que puede resolver este problema. Una *snapshot* o, traducido al español, una imagen, es la operación por la cual se toma un estado en un instante t cualquiera de la máquina virtual y se congela, pudiendo en cualquier momento volver a ese estado de *congelación* y, empezar o continuar desde ese estado la tarea que se estuviese desarrollando la próxima vez que se encienda el ordenador o la máquina virtual. Esta operación también tiene la ventaja de que se puede revertir y volver a un estado anterior cuando se requiera.

La idea de esto, es que ya que se necesita un entorno limpio para construir los paquetes, aprovechar que al generar el sistema base en la iteración anterior, se tienen máquinas virtuales mínimas muy similares a los entornos que ofrece *pbuilder* para construir los paquetes. Tal y como se describe parece que usar las máquinas virtuales, a priori, no tiene ninguna desventaja, pero si que tiene. Por ejemplo, se necesita almacenamiento en disco extra cada vez que se genera una *snapshot* de la máquina virtual en cuestión, si ocupa 3GB se necesitarían 6GB (3 GB de la *snapshot* y 3GB de la original). Ahora la pregunta es, ¿es un problema el almacenamiento? La respuesta es que no, ya que hoy en día la relación Euro-Giga es de más o menos 0.039 en discos duros no SSD, y en el futuro bajará cada vez más. De hecho, el computador usado para realizar este proyecto (Dell XPS 420) salió a la venta en el año 2007 y ya tenía un disco duro de 300GB, de los cuales a día de hoy para trabajar no se utilizan ni la mitad.

Otra de las ventajas que tiene utilizar este sistema es que se pueden generar paquetes por medio de máquinas virtuales utilizando la emulación de algunos procesadores y que permitirían construir paquetes para más arquitecturas de las disponibles. A priori no sería necesario, pero de esta forma se generaliza la solución al problema dando soporte a más arquitecturas. Esto también puede resultar ventajoso porque la máquina emulada, puede resultar en ocasiones más rápida que la propia máquina real. Se puede utilizar de ejemplo el famoso *Raspberry Pi*, en el cual la construcción de un paquete tardaría más que en una máquina virtual emulando ese procesador en un PC de escritorio.

Pbuilder es un sistema ya probado, y que funciona, pero tiene la desventaja de que no se pueden construir paquetes para otras arquitecturas diferentes de las del computador que tengamos en posesión, por ejemplo, dado un computador con *amd64*, solamente podría generar paquetes para esa arquitectura.

Inicialmente se construyó un entorno de pruebas mínimo con una máquina virtual de cada arquitectura, y ese entorno podría ser el sistema para seguir adelante ya que se ha observado una ventaja importante: el hecho de poder congelar la construcción de paquetes.

Ventajas y desventajas de pbuilder y máquinas virtuales

Un resumen de las ventajas e inconvenientes de cada opción son:

- Pbuilder
  - Ventajas:
    - Ya está hecho, y funciona.
    - No hace falta crear nada adicional salvo configurarlo.
    - Menor consumo que las máquinas virtuales en cuanto a memoria principal y secundaria.
  - Inconvenientes:
    - Está acotado, pues solamente se pueden construir paquetes para la arquitectura del computador que lo ejecuta.
    - Si un computador se apaga se tiene que empezar la construcción del paquete la próxima vez que se encienda el computador por lo que se presenta el riesgo de que no se pueda construir el paquete.
- Máquinas virtuales
  - Ventajas:
    - Con una instalación mínima se tiene un entorno muy similar al que se obtiene con *pbuilder*
    - Se puede parar la construcción del paquete y continuarla cuando se desee gracias a las *snapshots*.
    - Da soporte a más arquitecturas gracias a la emulación a través de *qemu* de diferentes arquitecturas de procesadores. Un mismo computador puede construir varios paquetes de distintas arquitecturas.
    - Posibilidad de construir con una misma máquina paquetes Debian para las otras ramas de Debian.
  - Inconvenientes:
    - Mayor consumo de memoria principal y de memoria secundaria.
    - Tarda más tiempo en la construcción de paquetes.

Por todo ello, y viendo que el tiempo que tarde en construir los paquetes no es un problema, y que tampoco lo es el espacio en disco duro, se eligen las máquinas virtuales y se cambia el entorno de pruebas por el entorno final donde se desarrollará el proyecto.

6.3.2 Diseño

Una vez elegido el entorno limpio que se va a utilizar, es hora de describir que características va a aportar al sistema. Las máquinas virtuales se instalarán con una versión Debian *testing* y tendrán los repositorios de las tres ramas de Debian.

Instalación automática de máquinas virtuales

Al utilizar máquinas virtuales para construir los paquetes, sería deseable disponer de una instalación automática de las mismas, especificando qué paquetes de Debian son necesarios durante la instalación con el fin de automatizar en la medida de lo posible el proceso. De tal forma que se pueda rápidamente tener al menos una máquina virtual de cada una de las arquitecturas. Esto daría la opción de clonar tantas máquinas virtuales como se desee en el caso de cada usuario.

Afortunadamente, Debian proporciona mecanismos de instalación automática desde uno de los CD oficiales o desde una réplica de red oficial. Estas instalaciones automáticas se realizan gracias a la instalación con «preseed» [Tea10].

Preseed es una archivo de texto en el que básicamente se responde a las preguntas que se realizan desde el instalador de debian de forma automática. Estas preguntas suelen cambiar con cada «release» de Debian `stable`[4].

A parte de todo esto, los ficheros *preseed* ofrecen la opción de hacer cosas distintas que, instalando a mano, no serían posible, como por ejemplo la instalación de ciertos paquetes o crear ciertos usuarios.

Lo que se requiere en este caso es introducir los repositorios necesarios en las máquinas virtuales, introducir contraseñas específicas y generar el usuarios sudo para permitir al usuario de las máquinas virtuales ejecutar órdenes con permisos de administrador.

```
#LOCALOZATION
d-i debian-installer/locale string es_ES

#KEYBOARD SELECTION
d-i console-keymaps-at/keymap select es
d-i keyboard-configuration/xkb-keymap select es

#NETWORK CONFIGURATION
d-i netcfg/choose_interface select auto
d-i netcfg/disable_dhcp boolean true
d-i netcfg/dhcp_failed note
d-i netcfg/dhcp_options select Configure network manually
```

[4]Debian tiene tres ramas *unstable*, *testing* y *stable*, siendo esta última, como su propio nombre, indica la versión más estable de todas.

```
d-i netcfg/get_nameservers string 192.168.122.1
d-i netcfg/get_ipaddress string 192.168.122.42
d-i netcfg/get_netmask string 255.255.255.0
d-i netcfg/get_gateway string 192.168.122.1
d-i netcfg/confirm_static boolean true
d-i hw-detect/load_firmware boolean false

#MIRROR SETTINGS
d-i mirror/protocol string http
d-i mirror/country string manual
d-i mirror/http/hostname string http.us.debian.org
d-i mirror/http/directory string /debian
d-i mirror/http/proxy string
d-i mirror/suite select testing
d-i mirror/udeb/suite string testing

#CLOCK AND TIME ZONES SETUP
d-i clock-setup/utc boolean true
d-i time/zone string Europe/Eastern
d-i clock-setup/ntp boolean true

#PARTITIONING
d-i partman-auto/choose_recipe select atomic
d-i partman/confirm_write_new_label boolean true
d-i partman/choose_partition select finish
d-i partman/confirm boolean true
d-i partman/confirm_nooverwrite boolean true

#ACCOUNT SETUP
#SUDO
d-i passwd/root-login boolean false
#NORMAL-USER
d-i passwd/user-fullname string ARCO
d-i passwd/username string arco
d-i passwd/user-password-crypted password $1$PsPemPVl$5PM3C4xlGAMb9KQ8Ump$zU0

#APT SETUP
#d-i apt-setup/local0/repository string \
#       http://babel.esi.uclm.es/arco sid main
#d-i apt-setup/local0/comment string local server
#d-i apt-setup/local0/source boolean true
#d-i apt-setup/local0/key string http://babel.esi.uclm.es/debian/dists/sid/Release.gpg

#PACKAGE SELECTION
tasksel tasksel/first multiselect standard
d-i pkgsel/include string openssh-server make build-essential dpkg-dev devscripts


#BOOT LOADER INSTALLATION
d-i grub-installer/only_debian boolean true
d-i grub-installer/with_other_os boolean true

#FINISHING UP THE INSTALLATION
d-i finish-install/reboot_in_progress note
```

Listado 6.6: Fichero de configuración preseed

Todas las preguntas y respuestas del listado 6.6, además de otras opciones pueden consultarse en la documentación online[5] de Debian y de *preseed*. Lo más destacable de esta configuración es que se instalan paquetes necesarios para construir paquetes Debian como **openssh-server make build-essential dpkg-dev devscripts**. Al estar las máquinas virtuales dentro de una red virtual, se accederá a ellas por `SSH`, la razón por la cual se instala el paquete `openssh-server`.

[5] `http://d-i.alioth.debian.org/tmp/en.amd64/apbs04.html`URL del contenido de los ficheros de preconfiguración.

Las tres ramas en las máquinas virtuales

Al utilizar las máquinas virtuales, se tiene instalada una distribución de Debian *testing* mínima. Si a esa distribución se le añaden los repositorios del resto de ramas de Debian, se tendría una máquina virtual con acceso a las tres ramas de Debian (*stable*, *testing* y *sid*) y con la posibilidad de construir paquetes para las tres ramas. Como se tiene que implementar la operación de capturar una imagen, la máquina virtual siempre volvería a su estado original como si nunca hubiese pasado nada.

Expuesto lo anterior se adopta la opción de poner los repositorios de todas las ramas de Debian en el archivo `/etc/apt/sources.list`6.7 de las máquinas virtuales:

```
#UNSTABLE
deb http://ftp.debian.org/debian/ unstable main
deb-src http://ftp.debian.org/debian/ unstable main

#TESTING
deb http://ftp.debian.org/debian/ testing main
deb-src http://ftp.debian.org/debian/ testing main

#STABLE
deb http://ftp.debian.org/debian/ stable main
deb-src http://ftp.debian.org/debian/ stable main

#ARCO
deb http://babel.esi.uclm.es/debian sid main
deb-src http://babel.esi.uclm.es/debian sid main
```

Listado 6.7: Ficheros souces.list

Las líneas debajo del comentario `#ARCO` son del repositorio privado y no oficial de ARCO. Esas dos líneas deben ser cambiadas con el fin de poner ahí el repositorio que más convenga al lector.

Es importante configurar las máquinas virtuales para que pese a tener los repositorios de las tres ramas de Debian, estén siempre con versiones de *testing*. Esto permite tener un equilibrio entre estabilidad y actualización de las versiones de paquetes. Para ayudar a que esto sea así se configuran las opciones de la herramienta APT [Deb14] para que tengan preferencia los paquetes de *testing* editando el fichero `/etc/apt/preferences`:

```
Package: *
Pin:release a=testing
Pin-Priority: 900

Package: *
Pin:release a=unstable
Pin-Priority: 800

Package: *
Pin:release a=stable
Pin-Priority: 700
```

Listado 6.8: Fichero `/etc/apt/preferences`

Con las líneas del listado 6.8 se consigue darle prioridad a los paquetes de Debian `testing`

frente a los de `unstable` y los de `stable`, siendo esta última rama la que tiene menos prioridad.

6.3.3 Implementación

Como se verá más adelante se requerirá en algunas órdenes el uso de privilegios de administrador, por lo que las máquinas virtuales se han configurado para usar `sudo` sin que pida la contraseña. Para conectar cada una de las máquinas virtuales se hace a través del protocolo *SSH* y *sshpass* para pasar el usuario y la contraseña como argumentos a través de la siguiente orden:

```
sshpass -p 'passwd' ssh -o StrictHostKeyChecking=no user@host <nombre\_de\_paquete>
```

Listado 6.9: Conexión SSH

La orden `sshpass -p 'passwd'` indica que la contraseña será 'passwd'. El argumento `-o` de *SSH* sirve para indicarle una opción, que en este caso es `StrictHostKeyChecking=no`, y que sirve para controlar los *logins* en las máquinas virtuales donde no se conocen las claves o estas han cambiado.

Se procede a definir las órdenes necesarias para construir un paquete y que se ejecutarán cada vez que se realice una petición de construcción de un paquete Debian.

Para la actualización del sistema se requiere de las típicas órdenes.

```
sudo apt-get update && sudo apt-get upgrade
```

Listado 6.10: Orden para actualizar la máquina virtual

Ahora que el sistema está actualizado es el momento de crear la “snapshot” con

```
sudo virsh snapshot-delete <nombre_dominio> <nombre_snapshot>
```

Listado 6.11: Creación snapshot

Cuando sistema está actualizado y con la “snapshot” hecha, una de las primeras cosas que hay que hacer para construir un paquete Debian que ya está en el repositorio es ejecutar el comando

```
apt-get source <nombre_de_paquete>
```

Listado 6.12: Obtención del código fuente del paquete

para conseguir las fuentes. Esto creará los archivos y directorios necesarios para la construcción del paquete, por lo que hay que estar dentro del directorio con nombre `nombre_del_paquete-version`

```
cd ls -F | grep /
```

Listado 6.13: Entrada al directorio para realizar la construcción

Un paquete Debian necesita cumplir unas dependencias de construcción, estas dependencias de construcción están indicadas en el archivo `debian/rules` en la línea *Build-Depends*. Los paquetes que figuran en esa línea necesitan estar instalados en el sistema para construir el paquete, ya que se depende de ellos para su construcción. En el caso del paquete *gentoo* las dependencias son: `debhelper (>= 9), dh-autoreconf, libgtk2.0-dev, libglib2.0-dev`. Para solventar este problema *apt-get* tiene una opcion “build-dep” que instala las dependencias necesarias para la construcción del paquete. Este comando hay que ejecutarlo como usuario `root`, por lo tanto se usa la orden `sudo` que permite ejecutar comandos como administrador.

```
sudo apt-get build-dep <nombre_del_paquete>
```

Listado 6.14: Obtener dependencias de construcción

Como último paso dentro de la máquina virtual hay que ejecutar `debuild` para construir el paquete

```
debuild -us -uc
```

Listado 6.15: Construcción del paquete

Para terminar el proceso hay que revertir la «snapshot» para dejar la máquina virtual actualizada y preparada para realizar la siguiente construcción.

```
sudo virsh snapshot-revert <nombre_dominio> <nombre_snapshot>
```

Listado 6.16: Volver al estado anterior

Lo único que falta es apagar la máquina virtual:

```
sudo virsh shutdown <nombre_dominio>
```

Listado 6.17: Apagar máquina virtual

Para automatizar todo este proceso se han realizado unos *scripts* que se ejecutarán en cada uno de los nodos cuando se requiera para construir los paquetes. Estos *scripts* se encuentran en `src/scripts`.

6.3.4 Pruebas

Las pruebas en esta iteración se resuelven de forma muy simple, ya que son los paquetes oficiales de Debian, los que ofrecen un *log* que será en sí mismo la prueba de si un paquete

se construye o no y por qué. En el listado 6.18, se muestra parte del *log* de la construcción de un paquete.

```
make[3]: No se hace nada para ‘install-data-am’.
make[3]: se sale del directorio ‘/home/arco/gentoo-0.19.13’
make[2]: se sale del directorio ‘/home/arco/gentoo-0.19.13’
make[1]: se sale del directorio ‘/home/arco/gentoo-0.19.13’
dh_install
dh_installdocs
dh_installchangelogs
dh_installman
dh_installmenu
dh_perl
dh_link
dh_compress
dh_fixperms
dh_strip
dh_makeshlibs
dh_shlibdeps
dh_installdeb
dh_gencontrol
dh_md5sums
dh_builddeb
dpkg-deb: construyendo el paquete ‘gentoo’ en ‘../gentoo_0.19.13-2_i386.deb’.
dpkg-genchanges  >../gentoo_0.19.13-2_i386.changes
dpkg-source --after-build gentoo-0.19.13
dpkg-buildpackage: subida de binarios y diferencias (NO se incluye la fuente original
    )
Now running lintian...
W: gentoo source: field-name-typo-in-dep5-copyright file -> files (line 122)
W: gentoo source: field-name-typo-in-dep5-copyright file -> files (line 171)
W: gentoo source: ancient-standards-version 3.9.3 (current is 3.9.5)
Finished running lintian.
Reverting snapshot...
snapshots reverted...[ OK ]
deleting snapshot...
snapshots deleted...[ OK ]
Domain bully-i386 is being shutdown
```

Listado 6.18: log de construcción de paquetes

En este listado se ve como el paquete `gentoo_0.19.13-2_i386.deb` se ha creado, y en las líneas sucesivas se pueden ver los avisos de «warning» de la herramienta *lintian* [NT10]. Todo ello da lugar al fichero *log* de la construcción de cada uno de los paquetes.

6.3.5 Entrega

El incremento que se le ha hecho al sistema es la funcionalidad de poder construir paquetes en un entorno limpio. Ese entorno limpio está compuesto por diferentes máquinas virtuales con arquitecturas diferentes, lo que permite construir varios paquetes para distintas arquitecturas en un nodo del sistema.

La instalación del entorno limpio se realiza automáticamente por lo que ayuda a cumplir uno de los objetivos que no es otro de tener una infraestructura «fácil de instalar y de mantener».

Se ofrece el propio *log* de la creación de paquetes proporcionado por la aplicaciones oficiales de Debian.

Este incremento, no solamente puede usarse con este proyecto, si no que puede usarse

como una herramienta externa al proyecto, del mismo modo que se usa *pbuilder*. Por lo tanto, no solamente se ha dado una solución al problema planteado al inicio de la iteración. La versatilidad del entorno limpio creado le permite ser utilizado como una herramienta externa o como parte de la infraestructura distribuida que se desarrolla con este proyecto.

Por todo ello, se valida el prototipo y se procede con el siguiente incremento.

6.4 Iteración 2: Firmado de paquetes: The GNU Privacy Guard

Cuando se construye software es importante utilizar algún mecanismo que diferencie que paquetes han sido construidos con una firma confiable y cuales no. Esto es lo que permitirá al sistema ofrecer paquetes de software seguros o que han sido construidos por el sistema. En Debian GNU/Linux, se utiliza las claves GPG para estas tareas.

GnuPG es el la implementación libre del estándar OpenPGP definido en el **RFC4880**. *GnuPG* permite cifrar y firmar tus datos y comunicaciones. Cuenta con un sistema de gestión de claves, así como módulos de acceso para todo tipo de directorios de claves. Gnupg es conocido por *GPG*, y es una herramienta para la línea de comandos para integrarla fácilmente con el resto de aplicaciones.

El funcionamiento consiste en que cada usuario tiene una clave privada y una clave pública. La clave privada es la que se utiliza para desencriptar aquello que te envían cifrado con la clave pública. La clave privada solamente la debe conocer el propietario, si alguien más la conociese, podría desencriptar lo que te envían y sería un problema de seguridad.

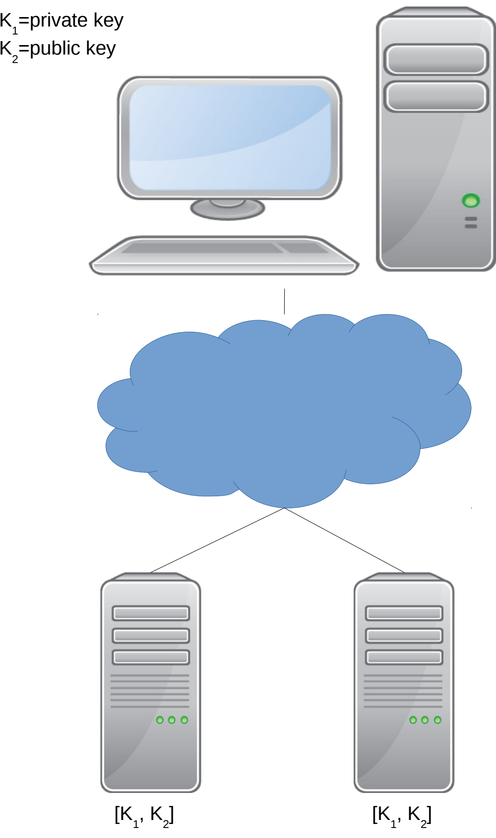

Figura 6.4: Firma de claves gpg

Lo que se pretende conseguir al firmar los paquetes es que se tenga la seguridad de que esos paquetes no han sido alterados por máquinas o usuarios ajenos al sistema cuando pasan

por las numerosas fases de construcción. De este modo, si un paquete no tuviese firma o tuviese una de la que no se tiene su clave, se considera como no confiable.

APT utiliza *GPG* para validar los paquetes .deb descargados y asegurarse de que no han sido alterados.

User Stories

Las historias de usuario que tras las reuniones oportunas se ha decidido incluir en esta iteración, son las siguientes.

- **Id:** 2.0
  - **Como:** Usuario.
  - **Quiero:** Firmar los paquetes
  - **De modo que:** Los paquetes construidos sean confiables y estén firmados.
  - **Notas:** Se recomienda utilizar firmas GPG como se utiliza en Debian GNU/Linux.
  - **Prioridad:** Alta
- **Id:** 2.1
  - **Como:** Usuario.
  - **Quiero:** Generar una firma GPG para cada una de las máquinas virtuales
  - **De modo que:** Puedan firmar los paquetes que se construyen poco a poco en el tiempo.
  - **Notas:** Generar tantas firmas como máquinas virtuales tenga el sistema. Las firmas no se pueden generar directamente en las máquinas virtuales porque no se genera entropía dentro de ellas.
  - **Prioridad:** Alta
- **Id:** 2.2
  - **Como:** Usuario.
  - **Quiero:** Exportar las firmas generadas a las máquinas virtuales
  - **De modo que:** Cada máquina virtual tenga la suya.
  - **Notas:** Exportar las claves utilizando las opciones `export` del comando gpg.
  - **Prioridad:** Alta
- **Id:** 2.3
  - **Como:** Usuario.
  - **Quiero:** Las firmas de cada una de las máquinas virtuales estén en el anillo del sistema.

- **De modo que:** El sistema distribuido pueda «conocer» esas máquinas como confiables.
- **Notas:** Las claves de las máquinas virtuales deben estar en el anillo del sistema, para que este las conozca y puedan confiar en esas claves.
- **Prioridad:** Alta

6.4.1 Análisis

Para realizar el firmado de claves y tal y como está diseñado este proyecto, se presentan dos alternativas. Recuérdese que cada uno de los nodos tienen un número determinado de máquinas virtuales. Las opciones que se barajan son:

1. Que cada uno de los nodos tenga su propia firma y que una vez construido el paquete se copie al PC «host» de alguna forma.
2. Que cada uno de las máquinas virtuales que están en cada uno de los nodos tenga su propia firma.

Los argumentos a favor y en contra de cada una de las aproximaciones están bien definidos.

Por un lado, si el firmado se realiza en cada uno de los nodos, se debe copiar el paquete para que se firme, el copiado de paquete, aunque casi instantáneo requiere de un esfuerzo innecesario.

Si el firmado de claves se realizase directamente en cada una de las máquinas virtuales, esa firma se podría meter dentro de la transacción que se hace para construir los paquetes, por lo que «el firmado» de paquete en sí, pasaría a formar parte de los *commands* que tiene cada uno de los *CompositeCommand*.

El segundo argumento ofrece una mayor simplicidad a la hora de implementar la solución y además se puede meter todo dentro de la misma transacción por lo que si algo va mal, el propio *evictor* se encargaría de rehacer la transacción evitando así muchos problemas.

Por lo tanto y expuestos los argumentos a favor y en contra de cada una de las soluciones, se opta por incluir el firmado de paquetes dentro de las máquinas virtuales para que formen parte de las transacciones.

6.4.2 diseño

Se necesita crear un anillo de claves entre el *Manager* del sistema y los nodos. Si el *Manager* tiene en su anillo de claves a los nodos, estos nodos serán conocidos por el sistema. Si por el contrario hay algún nodo que no se encuentra en el anillo del *Manager*, el sistema no lo reconocerá como firma confiable y no aceptará los paquetes que construya.

6.4.3 Implementación

Generación de un nuevo par de claves

Es el turno de la fase de creación de claves en la parte del computador que tiene las máquinas virtuales.

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

Y se elige la primera opción `RSA and RSA (default)`. Cuando pregunte por el tamaño, éste se deja por defecto a 2048, en cuanto a la duración también se deja por defecto a 0. Se siguen los pasos poniendo lo que pida el programa (Nombre, e-mail y comentario).

Una vez se generan las claves, es bueno generar el certificado de revocación, que en el caso de perder la clave o de haber comprometido su seguridad, el certificado de revocación se haría público para notificar al resto de usuarios que la clave pública no debe ser usada nunca más.

```
gpg --output DD123456.asc --gen-revoke 0xDD123456
```

Ahora que se tienen las claves es hora de exportarlas para que se puedan firmar y copiar.

```
gpg --output ARCO.gpg --export e-mail
```

Tras esto es necesario copiar la clave a las máquinas virtuales, y una vez que las tengamos allí se importará a cada una de ellas:

```
gpg --import ARCO.gpg
```

Cuando se construye un paquete y se quiere firmar, basta con utilizar la orden `debuild`, pero esto solamente funciona si el nombre y los datos coinciden con los del mantenedor del paquete. Como en este caso el mantenedor no va a construir el paquete, si no que va a ser una máquina que no tiene nada que ver con el mantenedor, se debe construir el paquete utilizando el argumento `-k` seguido de la clave. Por lo tanto el comando quedaría:

```
debuild -kE65B0539
```

Firmado automático de claves

Se puede automatizar el uso de GPG para firmar los paquetes automáticamente:

1. Si se quiere hacer firmado automático lo primero es crear una `subkey` para la `key`.

```
gpg --edit-key keyID
```

Introducir `addkey` y seleccionar el tipo DSA key.

2. Asegurarse de introducir la frase cuando se pida,

```
gpg --export-secret-subkeys --no-comment foo >secring.auto
```

6.4.4 Pruebas

Al igual que la iteración anterior, las herramientas que ofrece Debian disponen de un *log* el cual será la prueba de si se ha firmado un paquete o no se ha firmado. Al ser un añadido de la iteración anterior, esta información se verá reflejada cuando se construya un paquete.

6.4.5 Entrega

La funcionalidad extra que se ha añadido tras implementar esta iteración es la de ofrecer seguridad al sistema por medio de claves GPG, para que los paquetes que se construyan estén firmados y sean conocidos dentro del anillo del sistema. De otra forma, como ya se ha explicado anteriormente, si un paquete no tiene un firma conocida, no se distribuirá, y no se podrá construir.

6.5 Iteración 3: Persistencia de objetos con Freeze

Una vez que la iteración anterior cumple con los requisitos y las pruebas, es hora de añadir funcionalidad cambiando su comportamiento para cumplir poco a poco los requisitos del proyecto.

Se pretende implementar una de las cosas que hacen a este proyecto diferente e interesante, y es la persistencia. Con la persistencia se abre un abanico de posibilidades bastante amplio porque permite empezar la construcción de un paquete y continuarla en otro momento cuando el computador donde se está realizando esa tarea esté de nuevo disponible.

Esto permitirá que no se necesite tener una infraestructura dedicada como la de *Debian* y permitir con ello un ahorro en los costes, ya que los computadores que se usarán para construir un paquete serán los mismos que se usen para el trabajo diario de las personas que esten trabajando dentro de la red.

En la iteración 1 de la sección 6.3, concrétamente en la subsección 6.3.1, se hablaba del problema de usar o no *pbuilder*, y de cómo se decidió utilizar las máquinas virtuales que se generaron en aquel momento como un entorno limpio donde construir los paquetes debian. Una de las ventajas tenidas en cuenta a la hora de cambiar el rumbo y usar las máquinas virtuales, era que con ellas se permitiría el uso de "snapshots", lo cual a su vez, permitía congelar la construcción de paquetes cuando se requiera o se necesite apagar un computador, y continuar de este modo cuando ese computador se encienda la próxima vez.

6.5.1 User Stories

Para representar unos requisitos las *user stories* serán las siguientes:

- **Id:** 3.0
  - **Como:** Usuario.
  - **Quiero:** Jerarquía de objetos.
  - **De modo que:** Se representen los paquetes y sus dependencias mediante una estructura arbórea.
  - **Notas:** Permitirá poder construir las dependencias de un paquete que no estén construidas. Pueden usarse patrones de software como el «Composite».
  - **Prioridad:** Alta.
- **Id:** 3.1
  - **Como:** Programador.
  - **Quiero:** Cambiar el lenguaje de programación.
  - **De modo que:** Prueda utilizar todas las características que ofrece ICE.
  - **Notas:** Hasta ahora se había utilizado Python, y se propone el cambio a C++..
  - **Prioridad:** Alta.

- **Id:** 3.2
  - **Como:** Usuario.
  - **Quiero:** Usar persistencia.
  - **De modo que:** Pueda deterner la construicción de los paquetes.
  - **Notas:** Esto permitirá poder recuperar el sistema de un fallo, o simplemente pararlo porque sea necesario.
  - **Prioridad:** Alta.
- **Id:** 3.3
  - **Como:** Usuario.
  - **Quiero:** Almacenar la inforamción en una base de datos.
  - **De modo que:** Permita obtener los atributos de los objetos en cualquier momento
  - **Notas:** Va ligada con la implementación de persistencia, se almacenarán en. una base de datos los atributos de los objetos para que estos puedan ser recuperados.
  - **Prioridad:** Normal.
- **Id:** 3.4
  - **Como:** Programador.
  - **Quiero:** Impementar persistencia con un «evictor» de Ice.
  - **De modo que:** Pueda manejar el desalojo de objetos automáticamente.
  - **Notas:** Aumentará la simpleza de la implementación.
  - **Prioridad:** Alta.
- **Id:** 3.5
  - **Como:** Usuario.
  - **Quiero:** Tener más información de los paquetes.
  - **De modo que:** Pueda conocer su estado, versión, arquitectura, etc.
  - **Notas:** Se añaden más campos a la inforamción del paquete con el fin de tener más información de los mismos.
  - **Prioridad:** Alta.

6.5.2 Análisis

Lo primero que hay que hacer en el análisis es estudiar las posibilidades que ofrece ICE para el uso de persistencia. El servicio de ICE que ofrece persistencia recibe el nombre de *Freeze*.

Freeze

Es es el conjunto de servicios de persistencia que ofrece ICE, los cuales se dividen en los *evictor* y *map*:

- *Freeze evictor*, una implementación de un *servant locator* de ICE que provee persistencia automática y desalojo de los sirvientes.
- *Freeze Map*, un contenedor asociativo genérico. Las aplicaciones interactúan con un *Freeze Map* como si de cualquier otro contenedor se tratase, excepto que las claves y los valores de un Freeze Map son persistentes.

Freeze Evictor

Un *Freeze evictor* considera que el estado del sirviente se ha modificado cuando una operación de lectura-escritura se completa en ese mismo sirviente.

Para indicar las operaciones se añaden meta datos a los archivos SLICE. Aunque se habla de objetos persistentes, son en realidad los atributos de estos los que son persistentes.

Para implementar un evictor se requiere de los siguientes pasos.

- Una extensión de SLICE para especificar estado y que métodos alterarán el estado.
- Una factoría de sirvientes que permite que el núcleo de comunicaciones instancie los sirvientes al leer su estado en la base de datos.
- Un inicializador de sirvientes, que completa la inicialización una vez que se ha leído el estado de la base de datos.
- El propio Evictor, que se comporta como un localizador de sirvientes.

Estos pasos pasarán a formar parte de las *user stories* si finalmente se elige la implementación basada en el evictor.

ICE ofrece dos tipos de evictor cada cual cumpliendo una serie de características:

BackgroundSave evictor

Guarda todos los sirvientes en un mapa y escribe el estado de los sirvientes de nueva creación, modificados y borrados del disco duro de forma asíncrona en otro hilo de ejecución. Se puede configurar el intervalo en el que se guardan los sirvientes, lo cual viene bien si se necesitan hacer salvados de muchos sirvientes al mismo tiempo. Esta característica es muy interesante ya que ofrece una gran versatilidad para la mayoría de aplicaciones, pero tiene una desventaja, y es la recuperación frente a fallos. Al guardar de forma asíncrona no hay forma de forzar un guardado frente a una actualización.

Para ilustrar ese caso, se usará el clásico ejemplo del cajero que todo el mundo usa en los bancos. Si se quisiesen transferir fondos entre dos cuentas bancarias y en el momento de realizar la transferencia se produce un fallo en el sistema, es posible que al recuperar el sistema se pueda comprobar que el saldo de una de las dos cuentas ha sido actualizado, mientras que el saldo de la otra cuenta permanece como estaba antes de realizar la transferencia. Esto da

lugar a un problema, y es que no se puede asegurar donde se ha quedado ese dinero ni a quien pertenece.

Transactional evictor

Mantiene un mapa de sirvientes, pero de solo lectura. El estado de los sirvientes corresponde con el último dato almacenado en disco. Cada operación de crear, actualizar o borrar de un sirviente se hace a través de un transacción a la base de datos. Esta transacción se lleva a cabo o se deshace inmediatamente, típicamente al finalizar la operación actual.

El *transactional* evictor utiliza los mismos objetos para realizar transacciones que los *Freeze Maps*, lo cual permite actualizar un Freeze Map dentro de una transacción manejada por el "transactional.[ev]ictor. Usando los freeze Maps se realizan las transacciones utilizando ACID (atomicity, concurrency, isolation, durability).

Este tipo de evictor puede resultar más lento debido a que realiza muchas más lecturas y escrituras en almacenamiento secundario. El almacenamiento secundario es una de las memorias más lentas que puede haber en un computador, por lo tanto será menos eficiente que el *BackgroundSave* evictor, pero por el contrario permite recuperarse frente a fallos gracias al uso de las transacciones.

Freeze Maps

FreezeMap no es más que un contenedor asociativo persistente. ICE aprovecha la infraestructura de serialización de mensajes para implementar la persistencia.

Cada *Freeze Map* necesita realizar una conexión con la base de datos que le permita crear transacciones o borrar los índices que ya no se usen. Además, esta conexión puede ser compartida por varios *Freeze Map* cuando participan en operaciones conjuntas. El uso de transacciones es opcional en *Freeze* porque si no se usan los *Freeze Map* envolverán cada operación de actualización en una única transacción y los iteradores crearán una transacción en el constructor y la cerrarán en el destructor.

Conclusiones del estudio y elección

Tomando cada una de las opciones con cuidado objetivamente. La utilización del *evictor* es una opción muy interesante, porque se encargan de prácticamente todo automáticamente, no siendo así en los *Freeze Map*, por lo tanto se toma una primera decisión que es la de desechar el uso de los *Freeze Map* y implementar un evictor. La razón de la elección no es otra que aprovechar la facilidad que brinda el hecho de que se encargue el evictor de realizar todas las tareas de forma automática.

En cuanto a los evictor hay que elegir cual de los dos tipos utilizar. A priori se destaca al *BackgroundSaveEvictor* como una alternativa todo terreno de utilizar la persistencia con ICE, y de hecho, no sería descabellado su uso ya que ofrece una mayor eficiencia que el

TransactionalEvictor y se puede configurar el intervalo de guardado. Veamos pues, como debería comportarse el sistema y que se espera de él para elegir uno u otro evictor.

El sistema no debe quedarse en un estado en el que no sepa que es lo que tiene que hacer o en qué momento de todas las tareas que tiene que ejecutar está. Se requiere que el sistema ofrezca una seguridad frente a fallos, es decir, que conozca en todo momento que ha hecho y que es lo que queda por hacer. Es por ello que es muy importante que se vayan realizando las operaciones una a una y si alguna no se ha realizado se pueda empezar por ahí.

Imaginemos que el sistema tiene que construir un paquete p_1. Uno de los nodos del sistema ha decidido hacerse cargo de la construcción del paquete p_1 y procede a ello. Lo primero que debería hacer es encender la máquina virtual para la arquitectura deseada. La enciende, y tras ello se pone a hacer las operaciones requeridas. De repente, justo en el momento en el que se han terminado de instalar las dependencias de construcción del paquete, se va la luz o el ocupante del puesto acaba de terminar su jornada laboral, apaga el ordenador y decide irse a casa a comer. ¿Se sabe si el sistema ha llegado a guardar la operación en la base de datos? ¿qué pasaría si la próxima vez que el ocupante del puesto vuelva a trabajar y enciende su puesto? El sistema intentaría seguir con la construcción del paquete y se abren dos escenarios. Si el sistema continua con la construcción, la siguiente operación que debería hacer sería la de construir el paquete p_1. Si el sistema continuase con la construcción de p_1 podría tener las dependencias instaladas o no. Si las tuviese no pasaría nada, se procedería su construcción y las siguientes operaciones se realizarían de forma normal. Pero si no se hubiesen instalado las dependencias de construcción, el sistema intentaría construir el paquete p_1 sin tener sus dependencias, por lo que daría un error en la construcción de p_1, y se tendría que volver a poner a la cola para ser construido con la consiguiente pérdida de tiempo ya que estas dependencias podrían ser de tamaños de varios MB, y haber estado varias horas instalándose.

De todo lo anterior se puede concluir que el sistema debe poder recuperarse de fallos, tener el control y saber en todo momento donde y en que fase de la construcción de un paquete se ha quedado y cual es el siguiente paso a realizar. Esta característica y este problema planteado, hace ver que utilizar un *BackgroundSaveEvictor* puede ser contraproducente ya que aunque se configure el tiempo de guardado en la configuración nunca se sabe cuando se va a producir un fallo, y normalmente los fallos ocurren cuando menos se esperan.

Por todo ello, la alternativa a utilizar entre los dos tipos de evictor para implementar persistencia será un *Trasactional Evictor*. Aunque la aplicación sea más lenta en términos de rendimiento, es tolerable a fallos y ofrecerá una mayor seguridad al saber en todo momento por donde se ha quedado la construcción de paquetes.

Al utilizar el *Transactional Evictor* se consigue que los datos persistentes se modifiquen única y exclusivamente cuando se ha realizado cada una de las operaciones, sin intervenir

unas con otras. La unidad atómica que se elegirá para tratar las transacciones se verá con más detalle en las siguientes subsecciones.

C++

En las anteriores iteraciones la implementación se realizó utilizando el lenguaje de programación *Python*, el cual está bien para hacer prototipado rápido de los programas. ICE y sus implementaciones de persistencia solamente se encuentran disponibles para *Java* y *C++*, por lo que surge el problema de que no se puede utilizar *Python* si se quiere utilizar los servicios de persistencia que ofrece ICE. Por ello se plantea el dilema de si elegir algún servicio de persistencia disponible con *Python* o cambiar a *Java* o *C++* y utilizar los servicios de ICE.

Java es un lenguaje de programación orientado a objetos, y es uno de los que más he usado a los largo de la carrera, por lo que resulta familiar a la hora de utilizarlo para el desarrollo del proyecto. Pero la parte negativa de *Java* es que requiere de la máquina virtual cd *Java* para funcionar.

C++ es el otro lenguaje soportado para implementar la persistencia con ICE, además, también es un lenguaje de programación orientado a objetos. La parte negativa es que no lo he utilizado durante las asignaturas en la universidad y no conozco nada de él, pero he utilizado *C* de sintaxis parecida. Habría que invertir tiempo en aprenderlo o al menos aprender su sintaxis, pero por otro lado el sistema no tendría el inconveniente de utilizar una máquina virtual de *Java* para ejecutarse.

Es por todo ello y una vez vistas las diferentes alternativas, por lo que se seguirá implementando el proyecto utilizando *C++* en lugar de *Java* o *Python* con el que no está soportada la persistencia en ICE.

Patrones de Diseño

Los patrones de diseño son formas bien **conocidas** y **probadas** de resolver problemas de diseño que son recurrentes en el tiempo. Los patrones son utilizados en multitud de disciplinas tanto creativas como técnicas.

Los patrones sintetizan la experiencia de diseñadores de software que han evaluado y demostrado que la solución proporcionada es una **buena solución**

La implementación que se pretende realizar se puede hacer de muchas maneras, pero una de ellas es utilizando patrones de software, concrétamente los patrones *Composite* y *Command*.

El patrón Composite

El patrón *Composite* es un patrón de diseño estructural con la intención de representar objetos como una estructura jerárquica, con lo cual se permite tratar a los objetos de forma individual o de forma compuesta (un objeto compuesto de varios objetos que a su vez pueden contener varios objetos).

Este patrón de software se puede utilizar cuando se quiere representar algo del tipo uno a muchos, tiene un o es un X y tiene varios objetos X.

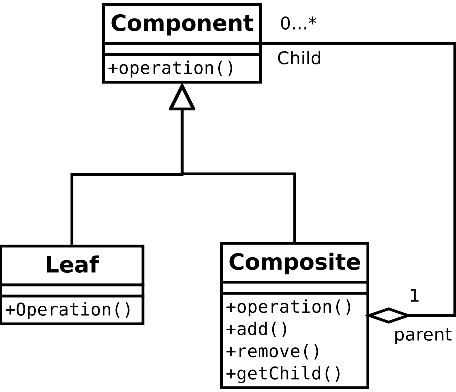

Figura 6.5: Diagrama de clases del patrón composite.

En este caso, para construir un paquete se necesita hacer un comando de comandos, es decir, un commando «construye este paquete» y que dentro de él tenga todos los commandos necesarios, por lo que la utilización de este patrón de software encaja perfectamente con los requisitos del sistema que se pretende desarrollar.

Además permitirá crear una estructura arbórea para representar las dependencias de cada uno de los paquetes que se construyan en el futuro. En la sección anterior se dejaba abierta la elección de la unidad atómica para tratar las excepciones. Bien, esa unidad atómica serán los «CompositeCommand», que serán tratados como un paquete con sus comandos de construcción. Esto quiere decir que cada vez que se añada o elimine un «CompositeCommand» se hará dentro de una transacción, así como construir un paquete.

El patrón Command

El patrón Command permite encapsular peticiones como objetos y con ello permitir gestionar colas de varios mensajes. Además soporta restaurar a partir de un momento dado, por lo que permite volver atrás en el caso en que una operación no se termine de realizar o repetirla con el método `redo()`.

En la figura 6.6 se puede ver el diagrama de clases del patrón *command*.

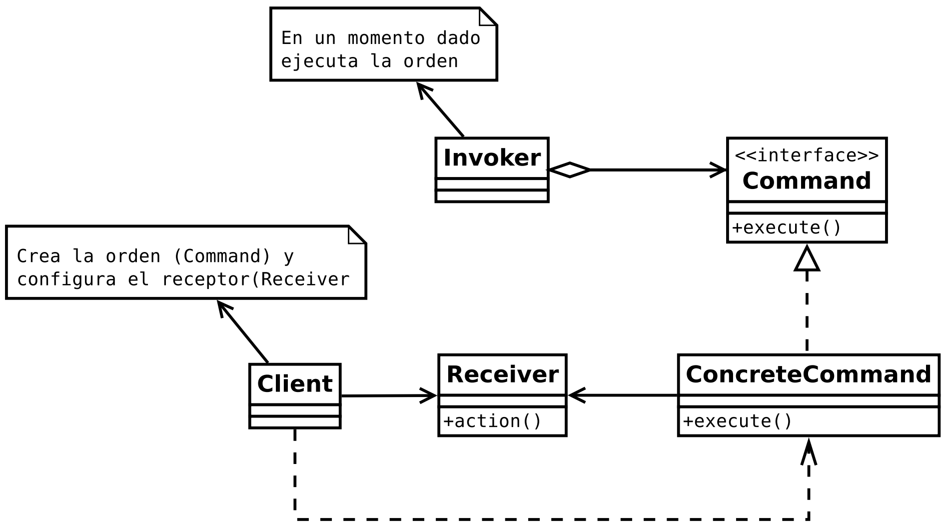

Figura 6.6: Diagrama de clases del patrón command.

La razón por la que utilizar este patrón no es otra que implementar todas las operaciones necesarias para construir los paquetes que como se han visto son:

Start Es la operación que enciende una máquina virtual con una arquitectura para realizar la reconstrucción del paquete.

Update Actualiza los repositorios de la máquina virtual para tener las versiones actualizadas.

Upgrade Realiza la actualización de paquetes de la máquina virtual y mantiene el entorno de construcción limpio y actualizado.

CreateSnapshot Genera una imagen de la máquina virtual para poder devolverla a su estado original una vez que la construcción del paquetes se ha terminado.

SourcePackage Descarga el código fuente del paquete que se pretende reconstruir.

BuildDependencies Instala las dependencias de construcción del paquete que se pretende reconstruir.

BuildPackage Operación que reconstruye el paquete para la arquitectura que se se está ejecutando en la máquina virtual.

RevertSnapshot Devuelve la máquina virtual a su estado original y actualizado permitiendo así que se pueda realizar otra construcción de otro paquete en un entorno limpio y actualizado.

DeleteSnapshot Elimina la imagen que se ha creado de la máquina virtual, ya que ya no es necesaria y de esta forma ahorrar espacio en memoria secundaria.

TurnOff Operación por la cual se apaga la máquina virtual de una determinada arquitectura.

Otra ventaja en el uso de este patrón es que se pueden utilizar cualquiera de estar ordenes de forma individual o crear una orden compuesta por varias órdenes, lo que daría lugar a una macro de órdenes. Por ejemplo, para construir el paquete P se necesitan n órdenes, por lo que se generará una orden de construcción para el paquete P que a su vez tiene todas las sub órdenes necesarias para construirlo.

En el caso particular de este proyecto es interesante implementar la operación *redo()* para rehacer una orden que no se ha podidio realizar.

El uso de este patrón, permitirá en un futuro realizar transacciones a nivel de «Command» es decir, que cada uno de los «Command» se convierta en una transacción y convertir así la unidad atómica de un «CompositeCommand» a un «Command». De momento esto no se hará, pero se deja abierta la posibilidad.

6.5.3 Diseño

Lo primero es añadir los métodos que modificarán el estado de los atributos al fichero slice original:

```
module DebBuilder{
  enum StateType{
    needsBuild,
    building,
    uploaded,
    depWait,
    bdUnstallable,
    failed,
    notForUs,
    installed };

  struct Package{ string name; string arch; StateType state; };

  interface Command{ ["freeze:write"]void execute(); };

  interface PackageOp{ ["freeze:write"]void setPackage(Package p);
  };
};
```

Listado 6.19: DebBuilder.ice

El módulo `DebBuilder` es el principal del programa, donde están definidos los estados por los que pasará un paquete a lo largo de su periodo de construcción. Lo que quiere decir cada uno de esos estados está definido en los estados de *wanna-build*[6].

Para implementar el patrón *composite-command* se ha creado un fichero `slice` adicional 6.20 donde se han añadido las clases y los datos persistentes.

Por último hay dos interfaces, la interface `Command`, que será la que implemente luego la clase `ConcreteCommand` en el listado 6.20, y que se encarga de ejecutar con el método

[6]/urlhttp://www.debian.org/devel/buildd/wanna-build-states.en.html

`execute()` cada una de las órdenes.

La interface `PackageOp` es la que permite al cliente decirle al servidor que paquete quiere construir y para que arquitectura.

Ambas interfaces tienen los metadatos `["freeze:write"]` que son los encargados de decirle a freeze que esos métodos son los que alteran los atributos persistentes.

Se han colocado las clases en un módulo diferente llamado `CommandType` y en un fichero fuente distinto 6.20

```
#include <DebBuilder.ice>
#include <Ice/Identity.ice>


module CommandType{

  class CompositeCommand;


  class ConcreteCommand implements DebBuilder::Command{
    Ice::Identity id;
    CompositeCommand* parent;
  };

  ["cpp:virtual"] class SimpleCommand extends ConcreteCommand{
    string architecture;
    string operation;
    string packageName;
    string snapshotName;
  };


  sequence<ConcreteCommand> CommandSeq;

  ["cpp:virtual"] class CompositeCommand extends ConcreteCommand
     implements DebBuilder::PackageOp{

    DebBuilder::Package debPackage;
    CommandSeq commands;
    CommandSeq redo;

    //methods
    ["freeze:write"]void add(ConcreteCommand cc); void destroy();
  };
};
```

Listado 6.20: CommandType.ice

La clase `ConcreteCommand` es la clase padre de la cual heredan las clases hijas `SimpleCommand` y `CompositeCommand`. Los objetos `SimpleCommand` serán cada una de las órdenes que será necesario ejecutar para construir un paquete Debian, mientras que los objetos `CompositeCommand` serán objetos que podrán estar compuestos de otros objetos `ConcreteCommand` y que por lo tanto podrán, a su vez, estar compuestos de otros objetos y así sucesivamente.

Esta diseño se puede apreciar mejor en el diagrama de clases (INCLUIR FIGURA))

6.5.4 Implementación

El método `add()` tiene como objetivo añadir objetos *ConcreteCommand* al objeto *CompositeCommand* de construcción de paquetes. Para alojar los objetos se utiliza un contenedor de tipo vector añadiendo los elementos en orden inverso para eliminarlos desde el final. Estos elementos antes de ser ejecutados se copian a otro contenedor de tipo vector para implementar la operacion `redo()`. De modo que antes de ejecutar cualquier orden se guarde esta y cuando suceda un fallo, la operación que se haga en `redo()` sea la que no se ha podido realizar.

En el siguiente listado se muestra el bucle que se implementa en el metodo `execute()` de la clase `CompositeCommand`, que se encarga de ejecutar cada uno de los "SimpleCommand.º ÇompositeCommand"que tenga el ÇompositeCommand".

```
[language = C++,
caption = {CommandTypeI.cpp},
label = code:executeCompsite]

while(!commands.empty()){

  redo.push_back(commands.back());
  commands.back()->execute();
  commands.pop_back();

}
```

El método `execute()` de la clase `SimpleCommand` ejecutará unas ordenes predefinidas. Estas ódenes son las explicadas anteriormente y las que están programadas en los script en bash que se realizaron en la iteración 6.3

Capturando señales en C++

Es importante recibir señales o interrupciones de teclado para que el servidor actúe en consecuencia y detenga la ejecución.

Para hacer el manejo de señales más fácil, ICE tiene la clase `Ice::Application` la cual encapsula en manejo de señales independientiente de la plataforma y proporcionadas por `IceUtil::CtrlCHandler`. Esto permite cerrar el programa sin errores en la recepción de una señal con la ventaja de utilizar el mismo código fuente en todos los sistemas operativos donde se ejecute.

Las funciones disponibles son:

- `destroyOnInterrupt()`: Esta función crea un `IceUtil::UtilCtrlCHandler` que **destruye** el comunicador cuando una de las señales monitorizadas llega. Este es el

comportamiento por defecto.

- `shutdownOnInterrupt()`: Esta función crea un `IceUtil::UtilCtrlCHandler` que **apaga** el comunicador cuando una de las señales monitorizadas llega.
- `ignoreOnInterrupt()`: **Ignora** las señales
- `callbackOnInterrupt()`: Esta función configura `Iceutil::Application` para que invoque `interruptCallback` cuando llega una señal, dando así la responsabilidad a la subclase para el manejo de la señal.
- `holdInterrupt()`: Esta función hace que se retenga la señal.
- `releaseInterrupt()`: Restaura la señal a la disposición anterior. Cualquier señal que llegue después, `holdInterrupt()` se entregará al llamar a `releaseInterrupt()`.
- `interrupted()`: Devuelve `true` si una señal causa que el comunicator se apague, y si no, `falla`.
- `interruptCallback()`: Una subclase sobreescribe esta función para responder a la señal.

Se elige `CallbackOnInterrupt()` por ser la que se adapta a las necesidades y se crea un método 6.21 para manejar la señal:

```
"Interrupcion!" << std::endl; _exit(i); }
```

Listado 6.21: interruptCallBack

6.5.5 Pruebas

Las pruebas realizadas durante esta iteración consisten en:

- Generar el *evictor* en ICE.
- Comprobar que el *evictor* registra los objetos correctamente.
- Parar el sistema y comprobar que al volver a empezar el *evictor* ha guardado los valores y se puede continuar con los valores antes de que el sistema se parase.

6.5.6 Entrega

En este incremento se ha dotado al sistema de persistencia. Ahora cuando ocurre un fallo, ya sea propiciado por alguna persona o no, el sistema es capaz de seguir construyendo el paquete que estaba construyendo antes del fallo.

Esta característica puede ser utilizaba en posteriores iteraciones para añadir persistencia en la generación de árboles de dependencia.

6.6 Iteración 4: El repositorio de paquetes Debian

Con toda la funcionalidad que se han añadido en las iteraciones anteriores, es necesario también disponer de un repositorio de paquetes Debian donde se puedan subir los paquetes y que estos se utilicen para construir las dependencias de cada uno de los paquetes.

Con el repositorio se consigue que todos los paquetes estén en un mismo sitio alojados, lo que permitirá que se pueda subir un paquete y todas sus dependencias a un mismo lugar, bien para construir otros paquetes o bien para que se utilicen para el trabajo diario.

En el apéndiceB existe un manual de los pasos a seguir para instalar un repositorio de paquetes Debian paso a paso.

6.6.1 User stories

Tras realizar las reuniones oportunas se decide comenzar con el trabajo de la iteración, para lo cual se seleccionan las siguientes historias de usuario:

- **Id:** 4.0
  - **Como:** Usuario.
  - **Quiero:** Un repositorio de paquetes Debian.
  - **De modo que:** Los paquetes construidos se puedan subir al repositorio y ser accesibles por el resto de usuarios.
  - **Notas:** Este requisito lleva consigo la instalación de un servidor.
  - **Prioridad:** Alta.
- **Id:** 4.1
  - **Como:** Usuario.
  - **Quiero:** Que el repositorio soporte arquitecturas i386 y amd64.
  - **De modo que:** Pueda servir paquetes para esas arquitecturas y sus usuarios puedan usarlos.
  - **Notas:** Inicialmente se empieza con estas dos, luego se podrán añadir más arquitecturas.
  - **Prioridad:** Alta.
- **Id:** 4.2
  - **Como:** Usuario.
  - **Quiero:** Que al repositorio solamente puedan subir paquetes los «developers».
  - **De modo que:** si el repositorio no conoce al firma del paquete no permita subirlo.
  - **Notas:** Utilizar firmado de claves gpg.
  - **Prioridad:** Alta.
- **Id:** 4.3

- **Como:** Usuario.
- **Quiero:** Hacer el repositorio accesible a los usuarios.
- **De modo que:** puedan utilizar los paquetes construidos para su arquitectura.
- **Notas:** Se requiere de un servidor, por ejemplo. *Apache*.
- **Prioridad:** Alta.

- **Id:** 4.4
  - **Como:** Usuario.
  - **Quiero:** Las firmas de cada una de las máquinas virtuales estén en el anillo del sistema.
  - **De modo que:** el sistema distribuido pueda «conocer» esas máquinas como confiables.
  - **Notas:** Las claves de las máquinas virtuales deben estar en el anillo del sistema, para que este las conozca y puedan confiar en esas claves.
  - **Prioridad:** Alta
- **Id:** 4.5
  - **Como:** Usuario.
  - **Quiero:** Que el repositorio esté firmado.
  - **De modo que:** Los usuarios puedan saber que los paquetes son confiables por medio de las firmas.
  - **Notas:** De esta forma al instalar paquetes del repositorio el sistema no avisará de que los paquetes están sin firmar.
  - **Prioridad:** Alta.

6.6.2 Análisis

La mejor forma de tener un repositorio a día de hoy es instalar y configurar uno que ya nos proporciona Debian. Hay distintas herramientas que sirven para montar un repositorio Debian, una de ellas es reprepro [Lin12], que permite crear un repositorio Debian no oficial con las características de uno oficial.

Hay otras herramientas para repositorios [7] que pueden cubrir estas necesidades o incluso otras, pero en este caso, se usará *reprepro* como la herramienta elegida para montar el repositorio Debian por las siguientes razones:

- Ofrece firmado de paquetes.
- Soporte para múltiples arquitecturas.

[7] Página web con las distintas herramientas que ofrece Debian para configurar un repositorio de paquetes `https://wiki.debian.org/HowToSetupADebianRepository`.

- Subida de paquetes.
- Cofiguración para las distintas ramas de Debian.
- Se está familiarizado con reprepro ya que en ARCO se usa este sistema.

Para poder tener acceso al repositorio de paquetes se necesita de un servidor, en este caso se han barajado dos posibilidades, *Apache* [Fou14] y *Cherokee* [Doc13]

Ambos son muy parecidos. Ambos son libres por lo que son compatibles con la licencia del proyecto, pero la principal diferencia que ha hecho inclinar la balanza sobre uno de ellos, es la utilización de recursos. *Cherokee* es mucho más ligero que *Apache*, y dado que solo se necesita montar un servidor para servir paquetes, se opta por *cherokee*.

6.6.3 Diseño

El servidor se instalará en un computador que se encuentra dentro de la misma red que el resto de computadores que forman el sistema distribuido. Se ha elegido una distribución Debian GNU/Linux y que tenga soporte para arquitecturas *i386* y *amd64*. El resto de arquitecturas podrán ser añadidas a posteriori. Estas dos arquitecturas son las más usadas en Debian.

6.6.4 Implementación

La implementación consiste en la realización de unos scripts que aportaran funcionalidad para la configuración y puesta en marcha del repositorio.

El archivo *distributions* del repositorio es donde se almacena la información. Para añadir las diferentes arquitecturas para las que se va a dar soporte se añade a la configuración del repositorio:

```
Architectures: i386 amd64 source
```

En el apéndice B existe un manual de como crear un repositorio Debian con todos los archivos de configuración y los pasos necesarios para ello.

Subida de paquetes

Cuando en la iteración 6.3 se añadió la funcionalidad para construir paquetes en las máquinas virtuales, no se implementó la parte de subida al repositorio. En este momento, con el repositorio funcionando, es cuando se añaden las órdenes necesarias para subir paquetes al repositorio.

El script tendría una nueva operación:

```
  sourcePackage)
      echo source
      dupload <nombe-del-paquete>.changes
```

Listado 6.22: dupload

Por lo tanto el conjunto de todas las operaciones una vez añadidas las correspondientes operaciones quedaría así:

```
        start)
            echo start
            start_machines \$VM
            check\_state \$HOST
            ;;
        update)
            echo update
            sshpass -p 'passwd' ssh -o StrictHostKeyChecking=no $USER@$HOST $update
            ;;

        upgrade)
            echo upgrade
            sshpass -p 'passwd' ssh -o StrictHostKeyChecking=no $USER@$HOST $upgrade
            ;;

        createSnapshot)
            echo create_snapshot
            create_snapshot \$VM \$3
            ;;

        sourcePackage)
            echo source
            sshpass -p 'passwd' ssh -o StrictHostKeyChecking=no $USER@$HOST $sourcep
            ;;

        buildDependencies)
            echo buildDep
            sshpass -p 'passwd' ssh -o StrictHostKeyChecking=no $USER@$HOST $buildDep
            ;;

        buildPackage)
            echo build
                        DIRSRC="$(sshpass -p 'passwd' ssh -o StrictHostKeyChecking=no
                            $USER@$HOST $directory)"
                        build="cd $DIRSRC;\
                        debuild -us -uc";

                        echo $build
            sshpass -p 'passwd' ssh -o StrictHostKeyChecking=no $USER@$HOST $build
            ;;

        revertSnapshot)
            echo revert
            revert_snapshot $VM $3
            ;;

        deleteSnapshot)
            echo delete_snapshot
            delete_snapshot $VM $3
            ;;

        turnOff)
            echo turnOff
            virsh shutdown $VM
            ;;

        *);;
    esac
}
```

Listado 6.23: Órdenes para construir paquetes

6.6.5 Pruebas

La creación de un repositorio y la modificación de los scripts encargados de realizar la construcción de los paquetes precisan de nuevas pruebas para garantizar el correcto funcionamiento.

- El servidor está conectado y funcionando.
- El servidor tiene instalado el paquete «reprepro».
- El servidor devuelve un *ping*.

Todas estas pruebas se añaden en el directorio `test` de la raíz del proyecto.

Además de todo ello, la propia herramienta *reprepro* ofrece información de lo que sucede por lo que también se utiliza el log como prueba para comprobar el correcto funcionamiento del sistema.

6.6.6 Entrega

El incremento que se ha realizado en esta iteración es tener un repositorio de paquetes Debian funcionando. Con un repositorio, además de facilitar las pruebas, se permite servir todos los paquetes a los usuarios de una forma muy sencilla, ya que lo único que se tiene que hacer es añadir la dirección del repositorio al fichero `/etc/sources.list`.

En principio el repositorio permite tener dos arquitecturas (i386 y amd64) y las fuentes de los paquetes. Para posteriores versiones de desarrollo se puede dar soporte a más arquitecturas como `ARM`.

6.7 Iteración 5: El manager para organizar la construcción de paquetes

Documentar la parte de la construccion de paquetes datos que conforman un paquete. archivos slice, metodos, etc.

Para continuar con el desarrollo del proyecto, se necesita algo que organice lo que pase dentro del sistema. Algún componente que se encargue de generar el árbol de dependencias de un paquete y que sepa que es lo que puede hacerse en cada momento con los paquetes, como por ejemplo, qué paquetes enviar a construir y cuales no.

En el capítulo 4 se explicaron todas las partes que formaban parte del sistema distribuido. Esta iteración se centrará en el desarrollo del «Manager», encargado de generar el árbol de dependencias de un paquete, tener las firmas de los nodos en su anillo, y de muchas cosas más.

6.7.1 User Stories

Las historias de usuario a desarrollar en esta iteración son las siguientes:

- **Id:** 5.0
  - **Como:** Usuario.
  - **Quiero:** Generar el árbol de dependencias de construcción de un paquete Debian.
  - **De modo que:** Que se construyan todo los paquetes necesarios para que construir el paquete demandado previamente
  - **Notas:** Puede que las dependencias de construcción de un paquete no existan, y entonces hay que construirlas también.
  - **Prioridad:** Alta.
- **Id:** 5.1
  - **Como:** Usuario.
  - **Quiero:** Conocer las dependencias de construcción de un paquete Debian.
  - **De modo que:** Puedan ser añadidos o no al árbol de dependencias de construcción
  - **Notas:** Ninguna
  - **Prioridad:** Normal.
- **Id:** 5.2
  - **Como:** Usuario.
  - **Quiero:** Saber si un paquete se encuentra ya disponible en el repositorio.
  - **De modo que:** No se construya más de una vez.
  - **Notas:** Construir un paquete dos veces puede desembocar en una gran pérdida de tiempo.

- **Prioridad:** Normal.

- **Id:** 5.3
  - **Como:** Usuario.
  - **Quiero:** Saber si un paquete se puede construir.
  - **De modo que:** Se genere la orden de construcción.
  - **Notas:** Un paquete se puede construir si existen sus dependencias de construcción en el repositorio. Si no existiesen habría que añadirlas al árbol y construirlas también
  - **Prioridad:** Alta.

- **Id:** 5.4
  - **Como:** Programador.
  - **Quiero:** Implementar persistencia.
  - **De modo que:** Si sucede un fallo el sistema pueda recuperarse.
  - **Notas:** Aplicar la misma técnica que en iteraciones anteriores con los *evictor*.
  - **Prioridad:** Alta.

- **Id:** 5.5
  - **Como:** Programador.
  - **Quiero:** Hacer una búsqueda DFS.
  - **De modo que:** Encontrar los nodos hoja, que serán los paquetes que se tengan que construir.
  - **Notas:** Se hará en orden inverso para recorrer el árbol al revés. anteriores con los *evictor*.
  - **Prioridad:** Alta.

- **Id:** 5.6
  - **Como:** Programador.
  - **Quiero:** Conocer los computadores conectados a la red.
  - **De modo que:** Se les pueda enviar trabajo.
  - **Notas:** Ninguna
  - **Prioridad:** Alta.

6.7.2 Análisis

Un paquete suele tener dependencias de construcción, esas dependencias deben estar instaladas en el sistema donde se va a realizar la construcción. Si las dependencias no se encuentran en el repositorio, se deberán construir previamente y subirlas al repositorio para que estén listas cuando se realice la construcción del paquete en cuestión.

Véase con un ejemplo en la figura 6.7 donde se pone un ejemplo del árbol de dependencias de construcción para el paquete *icebuilder-p1*.

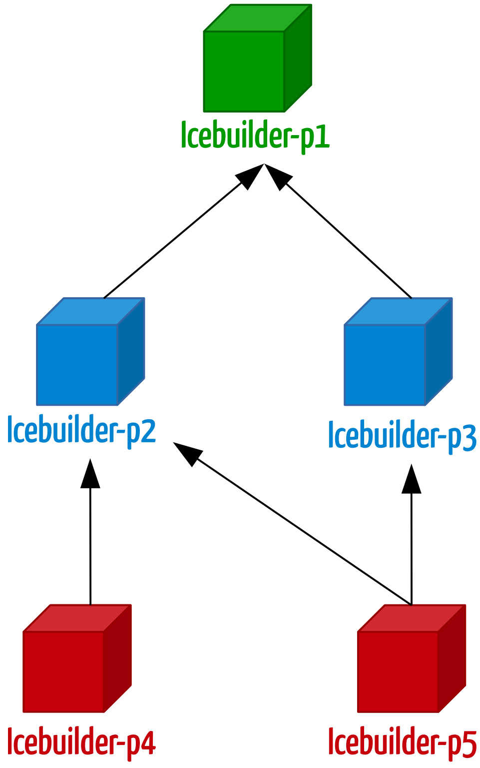

Figura 6.7: Árbol de dependencias

Supóngase el paquete *icebuilder-p1* para el cual sus dependencias de construicción son *icebuilder-p2* y *icebuilder-p3*. A su vez, *icebuilder-p2* tiene como dependencias de construcción a *icebuilder-p4* y *icebuilder-p5*. Y por último, *icebuilder-p3* tiene como dependencia de construcción a *icebuilder-p5*, que como se acaba de mencionar es también dependencia de construcción de *icebulder-p2*.

Al ir añadiendo dependencias se puede observar que se crea una estructura de árbol donde el nodo raíz es el paquete que se pretende construir y los nodos hijos son sus dependencias de construcción.

Como se ha necesita que las dependencias de construcción estén instaladas, es lógico que se construyan las dependencias empezando por los nodos hoja más profundos, enviando a los Workers las dependencias por que son los nodos hoja que forman el árbol.

Siguiendo con el caso del ejemplo, primero se enviarán a construir *icebuilder-p5* y *icebuilder-p4*, en la siguiente pasada, *icebuilder-p2* y *icebuilder-p3*, y por último *icebuilder-p1*.

Este proceso se podría realizar mediante un algoritmo de búsqueda en profundidad o DFS.

Los encargados de realizar la construcción de los paquetes serán los *Workers*.

Como no se sabe que computador va a construir cada paquete, se necesita de un lugar donde, una vez construidos, los paquetes se almacenen con el fin que se puedan instalar mediante APT las dependencias de construcción. Para albergar los paquetes se utilizará un repositorio de paquetes Debian, concrétamente *reprepro*, que ya se preparó para tal fin en la sección 6.6. En el apéndice B, se explica paso por paso las instrucciones para un uso básico de un repositorio.

La herramientda dose

Para la realización de este incremento del proyecto, hay que resolver un problema que surge a raiz de la generación del árbol de dependencias, y es que se necesita conocer si un paquete es construible para una determinada arquitectura. Que un paquete sea construible o no depende de sus dependencias de construcción y del paquete fuente. Para que un paquete sea construible, por ejemplo, para *i386*, tienen que estar instalas las dependencias de construcción, es decir, subidas en el repositorio para la arquitectura *i386*, junto el paquete fuente del paquete que se pretende construir. Si no estuviesen las dependencias de construcción habría que generarlas previamente, subirlas al repositorio, etc. Se trata pues de no duplicar esfuerzos e intentar que las construcciones de lo paquetes se hagan una sola vez.

Tras llevar a cabo una pequeña investigación sobre si hay algo ya hecho al respecto, se llega a dos opciones. En Debian, y más concretamente, en la *Debian autobuilder network*, que se explicaba en el capítulo de antecedentes 2, se utiliza una herramienta llamada *dose-debcheck* para comprobar si un paquete se puede instalar en el repositorio. Que un paquete se pueda instalar quiere decir, comprobar los efectos que provoca en el resto de paquetes ya subidos al repositorio el introducir un paquete nuevo, si rompe relaciones o genera conflictos. Un paquete se puede instalar si no genera conflictos ni rompe dependencias entre paquetes.

dose-debcheck forma parte del proyecto *dose* dentro del cual hay otras herramientas para saber, por ejemplo, si un paquete se puede construir, esta herramienta es *dose-builddebcheck* y para el caso que ocupa podría resolver prefectamente el problema.

La otra opción es que hacer desde cero un programa a medida para saber si un paquete es se puede construir o no. Esta opción tendría la ventaja de poder hacerlo a medida, aunque por contra se perdería tiempo en la implementación cuando ya hay algo que está propbado y que funciona resolviendo el problema.

Estudiadas las diferentes opciones, se decide usar *dose-builddebcheck* para resolver el problema, por ser una herramienta que se usa en Debian, está probada y funcionando, además desarrollar otra igual no aporta un gran beneficio ya que se obtendría el mismo resultado y con más esfuerzo

El paquete equivs

El paquete *equivs* es un paquete que crea paquetes denominados «tontos», esto quiere decir que por si mismos no hacen nada, no aportan una funcionalidad como puede aportar otro paquete normal. Sin embargo, se utilizan para crear paquetes para instalar dependencias o para engañar al sistema cuando hay problemas con las dependencias.

Un ejemplo del uso de esta herramienta serían los meta paquetes, donde en un solo paquetes instala, por ejemplo, el escritorio GNOME al completo.

Paquetes de prueba

Hará falta crear paquetes para probar todo, Utilizando el paquete **equivs** explicado en la sección 6.7.2, se crearán unos paquetes para recrear el ejemplo del árbol de dependencias comentado anteriormente.

Lo que hace equivs es generar un archivo `debian/control` con el nombre que se le dé por parámetro, y a partir de ahí generar el paquete.

6.7.3 Diseño

En iteraciones anteriores se ha hecho uso de la persistencia a través de *Freeze*. En el*manager* se seguirá usando persistencia para ir quitando o añadiendo paquetes del árbol de dependencias.

Por ser una estructura arbórea se utilizarán los patrones *composite-command*para su representación.

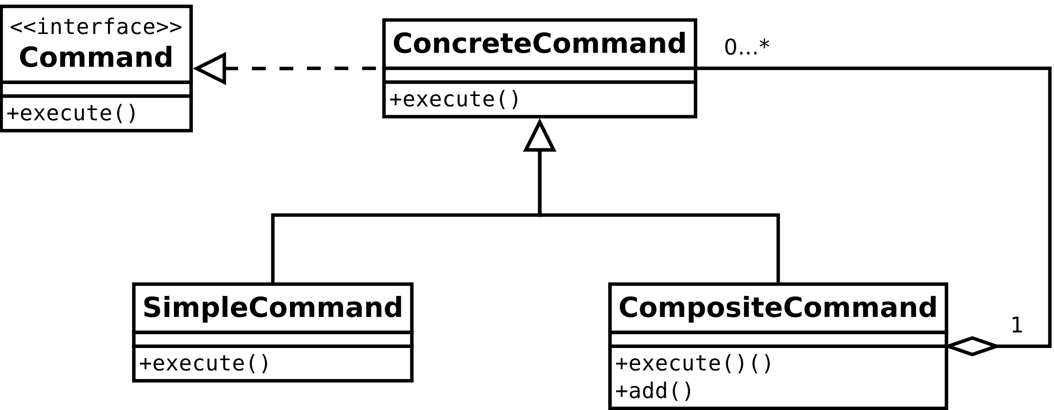

Figura 6.8: Diagrama UML composite-command

El fichero slice de diseño del sistema distribuido queda de la siguiente forma:

```
#include <DebBuilder.ice>
#include <Ice/Identity.ice>
```

```
module CommandType{

  class CompositeCommand;


  class ConcreteCommand implements DebBuilder::Command{
    Ice::Identity id;
    CompositeCommand* parent;
  };

  ["cpp:virtual"] class SimpleCommand extends ConcreteCommand{
    string architecture;
    string operation;
    string packageName;
    string snapshotName;
  };


  sequence<ConcreteCommand> CommandSeq;

  ["cpp:virtual"] class CompositeCommand extends ConcreteCommand
      implements DebBuilder::PackageOp{

    DebBuilder::Package debPackage;
    CommandSeq commands;
    CommandSeq redo;

    //methods
    ["freeze:write"]void add(ConcreteCommand cc); void destroy();
    int generateDependencyTree(DebBuilder::PackageInfo p, int depth)
        ;
    void findLeafsDFS(out LeafPackages theLeafPackages);
  };
};
```

Listado 6.24: Slice para el manager

Se añaden dos métodos nuevos, *generateDependencyTree* será el encargado de generar el árbol de dependencias de un paquete dado. El método *findLeadDFS* recorrerá el árbol para encontrar los nodos hoja.

6.7.4 Implementación

Un paquete virtual es aquel que aparece en el archivo control bajo la etiqueta «Provides» de otro paquete.

Para el desarrollo de esta iteración se va a utilizar la herramienta *dose-builddepcheck*, que se usa en la «Debian autobuilder network» con el fin de conocer si un paquete es construible dado un repositorio, o lo que es lo mismo, si sus dependencias de construcción se satisfacen. Además de comprobar si un paquete es construible, puede hacerlo en entornos curzados para distintas arquitectecturas por medios de los paquets fuente disponibles en el repositorio.

La salida de esta herramienta se ofrece codificada con el formato YAML que puede leerse

fácilmente con un parser[8]. *Yaml-cpp* está licenciado bajo licencia MIT [9], la cual es compatible con lo que la FSF entiende por software libre y por lo tanto es usable dentro del ámbito del proyecto. Por ello, se elige usar el parser en lugar de hacer un programa expresamente para ello.

La creación de cada uno de los paquetes con *equivs* se explica a continuación. Los nombres son los mismos que los del árbol de la figura 6.7 por lo que lo único que hace falta es rellenar los campos con lo que corresponda en cada caso según la política de Debian [JS12].

```
Source: icebuilder-p1
Section: misc
Priority: optional
Maintainer: Jose Luis Sanroma Tato <josel.sanromatato@gmail.com>
Build-Depends: icebuilder-p2, icebuilder-p3
Homepage: http://www.icebuilder.org
Standards-Version: 1.0

Package: icebuilder-p1
Architecture: any
Description: Dummy package which father is no one
This is a dummy package to test the icebuilder generator dependencies tree.
This package has no functionality but provides the dependencies in order
to build a test environment.
.
If you are testing icebuilder, make sure that you use this package properly
```

Listado 6.25: Paquete Debian *icebuilder-p1*.

```
Source: icebuilder-p2
Section: misc
Priority: optional
Maintainer: Jose Luis Sanroma Tato <josel.sanromatato@gmail.com>
Build-Depends: icebuilder-p4, icebuilder-p5
Homepage: http://www.icebuilder.org
Standards-Version: 1.0

Package: icebuilder-p2
Architecture: any
Description: Dummy package which father is icebuilder-p1
This is a dummy package to test the icebuilder generator dependencies tree.
This package has no functionality but provides the dependencies in order
to build a test environment.
.
If you are testing icebuilder, make sure that you use this package properly
```

Listado 6.26: Paquete Debian *icebuilder-p2*.

```
Source: icebuilder-p3
Section: misc
Priority: optional
Maintainer: Jose Luis Sanroma Tato <josel.sanromatato@gmail.com>
Build-Depends: icebuilder-p5
Homepage: http://www.icebuilder.org
Standards-Version: 1.0

Package: icebuilder-p3
```

[8]Un «parser» no es más que un analizador sintáctico.
[9]`http://opensource.org/licenses/mit-license.php`

```
Architecture: any
Description: Dummy package which father is icebuilder-p1
This is a dummy package to test the icebuilder generator dependencies tree.
This package has no functionality but provides the dependencies in order
to build a test environment.
.
If you are testing icebuilder, make sure that you use this package properly
```

Listado 6.27: Paquete Debian *icebuilder-p3*.

```
Source: icebuilder-p4

Section: misc
Priority: optional
Maintainer: Jose Luis Sanroma Tato <josel.sanromatato@gmail.com>
Build-Depends:
Homepage: http://www.icebuilder.org
Standards-Version: 1.0

Package: icebuilder-p4
Architecture: any
Description: Dummy package which father is icebuilder-p2
This is a dummy package to test the icebuilder generator dependencies tree.
This package has no functionality but provides the dependencies in order
to build a test environment.
.
If you are testing icebuilder, make sure that you use this package properly.
```

Listado 6.28: Paquete Debian *icebuilder-p4*.

```
Source: icebuilder-p5
Section: misc
Priority: optional
Maintainer: Jose Luis Sanroma Tato <josel.sanromatato@gmail.com>
Build-Depends:
Homepage: http://www.icebuilder.org
Standards-Version: 1.0

Package: icebuilder-p5
Architecture: any
Description: Dummy package which father is icebuilder-p3 and icebuilder-p2
This is a dummy package to test the icebuilder generator dependencies tree.
This package has no functionality but provides the dependencies in order
to build a test environment.
.
If you are testing icebuilder, make sure that you use this package properly
```

Listado 6.29: Paquete Debian *icebuilder-p5*.

Para generar de forma automática los paquetes se tiene un Makefile, donde las órdenes de construcción para los paquetes creados con equivs se realizan mediante:

```
equivs-build -f icebuilder-p5
```

Con el *flag* `-f` se consigue que se generen el paquete completo con fuentes incluido. Cabe destacar que si en el control que genera equivs no se le pone el campo `Build-depends`, equivs generará un paquete cuya dependencia de construcción sea `debhelper`. Esto ha ocasionado algún problema hasta que se ha descubierto, ya que al generar paquetes que apriori

no debería tener dependencias de construcción como son *icebuilder-p5* y *icebuilder-p4*, de repende apareceiese debhelper como dependencia de construcción. La solución a este pequeño inconveniente ha sido poner el campo `Build-Depends` y dejarlo vacío, por eso está vacío en los ejemplos que se han listado arriba.

6.7.5 Pruebas

Las pruebas que se efectúan durante esta iteración son:

- Generar el árbol de dependencias de un paquete.
- Conocer las dependencias de construiicción de un paquete dado y ver si se puede construir.
- Obtener los nodos hoja del árbol de construcción de paquete.
- Conocer los computadores conectados a la red.

6.7.6 Entrega

Tras esta iteración se ha añadido un nuevo componente al sistema, el «manager», el cual es el encargado de generar los árboles de dependencias de cada uno de los paquetes Debian que se deben construir. Además se ha añadido persistencia y cada una de las operaciones que se hacen en ese árbol, están acompañadas de una transacción que garantiza que si no se ha realizado la transacción se repita de nuevo.

Tras finalizar las reuniones oportunas se da por bueno el prototipo y se continúa con la siguiente iteración.

6.8 Iteración 6: Worker para realizar la construcción de paquetes

En la iteración 6.3 se construyó un entorno limpio para construir los paquetes Debian. Ese entorno limpio se validó para formar parte del sistema. En esta iteración los esfuerzos van dirigidos a la integración de esos entornos limpios en forma de «Workers» dentro del sistema distribuido para construir los paquetes en los distintos computadores.

Recuerde que los «workers» son los computadores que reciben los paquetes del «Manager». Es un computador que se encuentra en la misma red o en otra y, cada uno de los «workers» pueden tener una o varias máquinas virtuales instaladas (dependiendo siempre de las características del computador), donde se llevará a cabo la construcción de los paquetes.

Tener los «workers» incluidos y funcionando en el sistema distribuido será uno de los últimos pasos del desarrollo del proyecto.

6.8.1 User Stories

Las historias de usuario que se van a incluir en esta iteración tras las reuniones realizadas son las siguientes:

- **Id:** 6.0
  - **Como:** Usuario.
  - **Quiero:** Tener unos «workers».
  - **De modo que:** Realicen el trabajo de construcción de paquetes en el sistema distribuido.
  - **Notas:** Los «workers» los computadores que construirán los paquetes.
  - **Prioridad:** Alta.
- **Id:** 6.1
  - **Como:** Usuario.
  - **Quiero:** Que la construcción en el «worker» se pueda parar.
  - **De modo que:** Se pueda reanudar en otro momento si es necesario.
  - **Notas:** Serviría por si hay algún error y se quiere. restaurar el sistema.
  - **Prioridad:** Alta.
- **Id:** 6.2
  - **Como:** Usuario.
  - **Quiero:** Usar persistencia.
  - **De modo que:** Pueda guardar el estado de los workers.
  - **Notas:** Esto permitirá poder recuperar el sistema de un fallo, o simplemente pararlo porque sea necesario.
  - **Prioridad:** Alta.

- **Id:** 6.3
  - **Como:** Programador.
  - **Quiero:** Que un «worker» pueda construir paquetes usando más de una máquina virtual al mismo tiempo.
  - **De modo que:** se pueda aprovechar los cores de los microprocesadores maś nuevos para construir paquetes en paralelo.
  - **Notas:** Los computadores con más prestaciones podrán construir un mayor número de paquetes al mismo tiempo.
  - **Prioridad:** Normal.
- **Id:** 6.4
  - **Como:** Programador.
  - **Quiero:** Impementar persistencia con un «evictor» de Ice.
  - **De modo que:** Pueda manejar el desalojo de objetos automáticamente.
  - **Notas:** Aumentará la simpleza de la implementación.
  - **Prioridad:** Alta.
- **Id:** 6.5
  - **Como:** Programador.
  - **Quiero:** enviar una orden de construcción que envuelva todas las operaciones.
  - **De modo que:** Se meta todo en una transacción.
  - **Notas:** Se permite con ello que la transacción se lleve a cabo o no, si no se lleva a cabo se reintentará.
  - **Prioridad:** Alta.

6.8.2 Análisis

Partiendo de las historias de usuario que se han seleccionado tras las reuniones, y dado que los «workes» son otra parte distinta del sistema, se necesitará añadir nuevas interfaces a los ficheros SLICE.

Como se hizo en la iteración 6.5 será necesario implementar persistencia en cada uno de los «workers».

En este caso el sistema elegido para implementar la persistencia también es el *Transactional evictor* que se encuentra en *Freeze*. Esta decisión se ha tomado por las siguientes razones:

- Por seguir teniendo un sistema transaccional donde si no se lleva a cabo la transacción se deshace y se vuelve a intentar.
- Recuperable frente a fallos.

- La implementación de persistencia con los evictor es automática y por lo tanto la más sencilla.

Además de todo ello, será necesario añadir máquinas virtuales al worker, con el fin de tener al menos dos arquitecturas para añadirlas al sistema.

6.8.3 Diseño

Para añadir los workers es necesario modificar los ficheros SLICE e incluso añadir nuevos para implementar la persistencia.

Lo primero es añadir la interfaz `Worker` al fichero SLICE que nos permitirá tener los «Workers» dentro de ICE. El método exec, lo que hará será ejecutar la orden de construir el paquete, más delante se verá la implementación.

```
interface Worker{
  ["freeze:write"]void exec(CommandType::CompositeCommand p);
};
```

Listado 6.30: worker.ice

Después, se añaden los datos a un fichero SLICE diferente que serán los que almacene freeze:

```
#include <Worker.ice>

module WorkerPersistent{
  class Job implements Work::Worker{
    CommandType::CompositeCommand package;
  };
};
```

Listado 6.31: workerPersistent.ice

El dato que se guardará en la base de datos es un objeto de tipo `CommandType::CompositeCommand`. Este objeto se pasará por valor para que se realicen las operaciones oportunas para construir los paquetes.

Con estos cambios ya están creadas las correspondientes clases, por lo que es necesario implementar los métodos como corresponda.

Para añadir las múltiples arquitecturas a cada uno de los «Workers» será necesario como mínimo dotarlos de una máquina virtual con arquitectura *i386* y otra con arquitectura *amd64*.

6.8.4 Implementación

La puesta en ejecución de las nuevas funcionalidades que se han añadido en el diseño se reflejan en distintos aspectos.

Por un lado el `Worker` solamente recibirá objetos de tipo `CommandType::CompositeCommand` por **valor** y no por referencia. Esto permite que se guarde en la base de datos por si sucede algún error, que sea posible volver a empezar la construcción por ahí.

Los objetos `CompositeCommand` siguiendo el diagrama de clases, tienen un método `execute()` que es donde se llevarán a cabo las órdenes oportunas para construir los paquetes.

```
void
CommandType::CompositeCommandI::execute(const Ice::Current& current){

  IceUtil::Mutex::Lock lock(compositeMutex);

  std::cout << "CompositeCommandI: " << debPackage.name << std::endl;
  CommandSeq::iterator it = commands.begin();

  Ice::Identity id=current.id;
  std::string type=commands.back()->ice_id();

  if(type==CompositeCommand::ice_staticId()){
    CompositeCommandPrx cPrx;
    cPrx=CompositeCommandPrx::checkedCast(current.adapter->createProxy
        (commands.back()->id));
    cPrx->execute();
    commands.pop_back();
    cPrx->destroy();

  }
  else{

    while(!commands.empty()){

      commands.back()->execute();
      commands.pop_back();
    }
  }
}
```

Listado 6.32: operación execute

Las órdenes para construir los paquetes están dentro de objetos de tipo `SimpleCommand` y almacenadas en un contenedor de tipo `vector`, por lo que el algoritmo consiste en recorrer el vector poco a poco y ejecutando cada una de las órdenes.

Instalación de máquinas virtuales

Para añadir las máquinas virtuales al «Worker» se utiliza un script en `Bash` que utiliza los ficheros *preseed.cfg* que se construyeron en la 6.3 haciendo una instalación por red de

Debian GNU/Linux. Se le proporcionan datos como la IP que usará la máquina virtual y la dirección MAC, junto con el nombre de la máquina virtual.

```
name=i386

virt-install \
    --debug \
    --name=$name \
    --ram=512 \
            --arch=i386 \
    --disk ./$name-i386.qcow2,format=qcow2,size=5,sparse=true \
            --location=http://ftp.debian.org/debian/dists/stable/main/installer-
                i386 \
    --network=network=icebuilder,mac=00:AA:00:00:00:11 \
            --extra-args="\
    auto=true priority=critical vga=normal hostname=$name \
    url=http://arco.esi.uclm.es/~joseluis.sanroma/preseed.cfg"
```

Listado 6.33: script que instala una máquina virtual

En el apéndice A existe un manual de como instalar un «worker» con todos los pasos a seguir y los paquetes necesarios.

6.8.5 Pruebas

Una vez más, la prueba de que todo funciona se realiza construyendo un paquete Debian. Es *log* de la construcción del paquete la prueba en sí. Como ya se ha explicado anteriormente este *log* lo proporcionan las propias herramientas de Debian e indican donde ha fallado la construcción de un paquete y cual ha sido la razón del fallo. en caso de que la construcción del paquete fuese satisfactoria se tiene el paquete creado, es decir, el archivo *.deb* con todos los archivos asociados a él.

En la figura 6.9 están los paquetes construidos para arquitecturas *i386* y para *amd64* de un intento satisfactorio.

Las máquinas virtuales se han instalado correctamente y ya forman parte de cada uno de los *workers*. Para simplificar, solamente se han instalado unas pocas máquinas virtuales. En la figura 6.10 está el resultado y su puesta en marcha.

6.8.6 Entrega

El incremento que ha sufrido el sistema con esta iteración no es otro que añadir la posibilidad a los nodos de construir los paquetes por ellos mismos dentro de los entornos limpios basados en máquinas virtuales que se construyeron en la iteración 6.3

Se valida el incremento y se procede con la siguiente iteración.

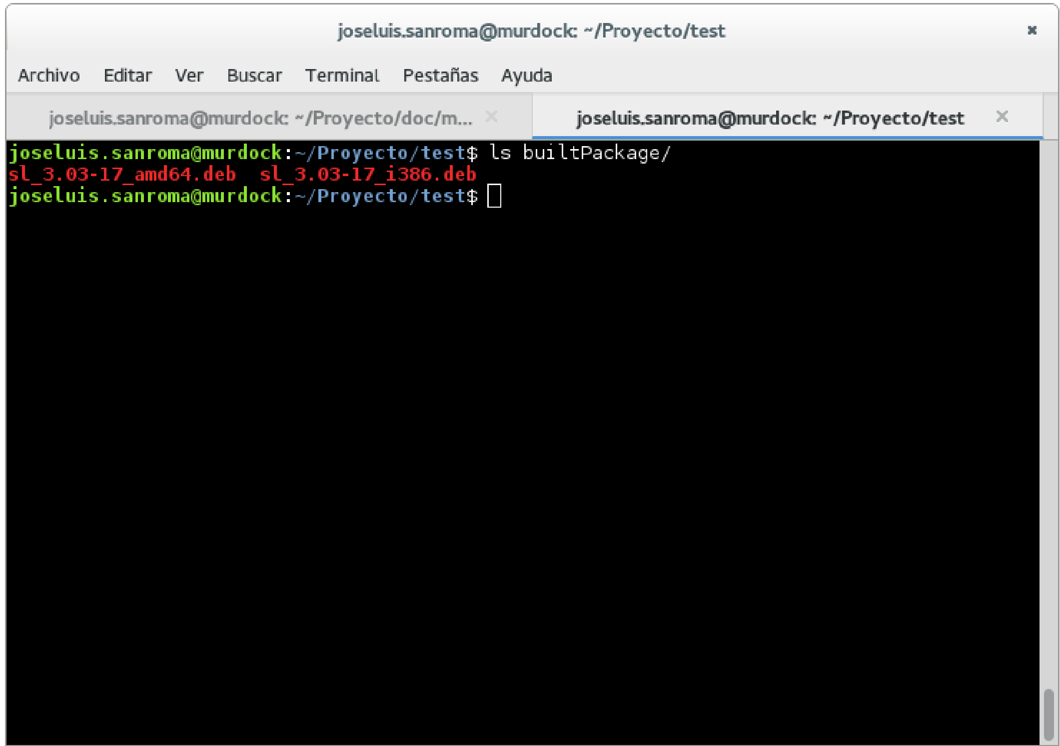

Figura 6.9: Paquetes construidos para dos arquitecturas

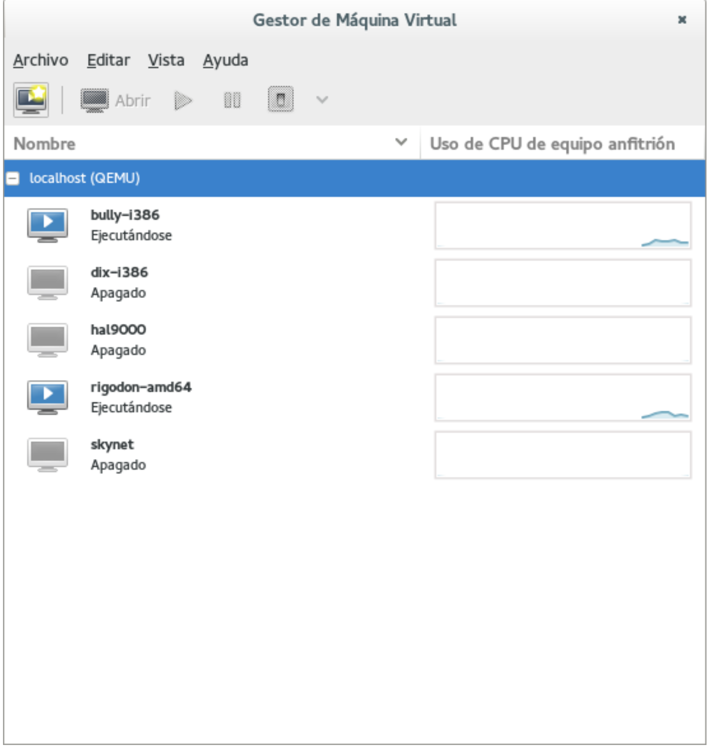

Figura 6.10: Máquinas virtuales instaladas

6.9 Iteración 7: Despliegue con IceGrid

En esta última iteración del prototipo, se va a proceder al despliegue del mismo en un entorno real. Aunque esta no es una iteración de desarrollo de software como tal, se tratará de la misma forma que las anteriores.

Será necesario mantener un balanceo de carga entre los equipos conectados al sistema. Uno de los objetivos hacía referencia a la facilidad de uso de este sistema distribuido, por lo tanto ese será uno de los objetivos de esta iteración.

6.9.1 User Stories

Las *user stories* son las siguientes.

- **Id:** 7.0
  - **Como:** Usuario.
  - **Quiero:** Balanceo de carga.
  - **De modo que:** La carga del sistema se reparta entre todos los computadores conectados.
  - **Notas:** Permitirá aprovechar mejor la capacidad de cómputo de todo el sistema distribuido.
  - **Prioridad:** Normal.
- **Id:** 7.1
  - **Como:** Programador.
  - **Quiero:** transparencia de localización.
  - **De modo que:** No necesite saber nada salvo qué computadores están conectados.
  - **Notas:** Ninguna.
  - **Prioridad:** Alta.
- **Id:** 7.2
  - **Como:** Usuario.
  - **Quiero:** Dar de alta en el sistema nodos de forma sencilla
  - **De modo que:** Cualquier usuario pueda dar de alta su nodo para que entre a formar parte del sistema.
  - **Notas:** Ninguna.
  - **Prioridad:** Normal.
- **Id:** 7.3
  - **Como:** Usuario.
  - **Quiero:** tener más información de los paquetes

- **De modo que:** pueda conocer su estado, versión, arquitectura, etc
- **Notas:** Se añaden más campos a la inforamción del paquete con el fin de tener más información de los mismos.
- **Prioridad:** Alta

6.9.2 Análisis

IceGrid es una plataforma para la gestión de sistema distribuidos heterogéneos. Permite la localización y activación de los servicios de las aplicaciones ICE. Además también existe *IcePatch* un servicio de despliegue de software.

El uso de esta herramienta y no otra es debido a está dentro de ICE y resulta familiar ya que se ha utilizado en diferentes proyectos en el laboratorio ARCO.

IceGrid proporciona, entre otras prestaciones, activación automática de objetos, transparencia de localización y balanceo de carga.

Para que *IceGrid* pueda funcionar correctamente, depende de una base de datos que se llama *IceGrid Registry*, la cual contiene información sobre todos los objetos conocidos en el sistema, las aplicaciones desplegadas, los nodos disponibles, etc.

Ejecutando el servicio *IceGrid Node* en cada uno de los nodos *IceGrid* sabrá qué computadores están disponibles para ejecutar una aplicación. Este servicio resolverá el problema de conocer los computadores disponibles dentro de la red.

Para administrar todas estas cosas *IceGrid* proporciona dos herramientas. La primera de ellas es *icegridadmin* que se usa por medio de una terminal e introduciendo órdenes que se desean ejecutar. La segunda es *icegrid-gui*, que permite hacer lo mismo pero mediante una interfaz gráfica.

Dentro de *IceGrid* conviene tener claros algunos conceptos:

Servidor Es un programa que se ejecuta en un Nodo, El programa puede ser tanto un cliente como un servidor.

Nodo Es una instancia de *IceGrid Node*. Identifica un nodo del sistema, pero no tiene por qué ser un computador ya que en un computador pueden ejecutarse más de un *IceGrid Node*.

Aplicación Son los objetos y servicios que conforman la aplicación distribuida, por ejemplo, canales de eventos, etc.

Adaptador de objetos Contiene los datos de un adaptador de objetos utilizado en un servidor ICE.

6.9.3 Diseño

Se va a tratar de construir un caso mínimo que permita hacer funcionar todo lo hecho anteriormente. Se utilizarán dos «Workers», que serán donde se construirán los paquetes Debian. Estos a su vez, tendrán instaladas máquinas virtuales que permitirán construir paquetes tanto para arquitectura *i386* como para *amd64*.

En resumen, lo que se necesita en cada nodo es:

- Computador don Debian GNU/Linux rama *testing*.
- Máquina virtual *i386*.
- Máquina virtual *amd64*.
- Zeroc Ice instalado.

La nomenclatura utilizada por *IceGrid* para los nodos sería la misma que la que se utiliza con los «Workers», un nodo podría ser un «Worker» y en cada computador puede haber varios «Workers».

6.9.4 Implementación

Para configurar *IceGrid* correctamente se necesita como mínimo un *Registry* y los nodos o «Workers» correctamente configurados.

La configuración se realiza por medio de ficheros en texto plano en los que los valores se configuran en un formato «clave=valor». Una de las primeras cosa que se necesitará es un *proxy* al servicio *Locator*.

```
Ice.Default.Locator=IceGrid/Locator -t:tcp -h 192.168.1.123 -p 4061
```

Listado 6.34: Configuración del Locator

La dirección IP tendría que ser reemplazada por la que corresponda a cada caso.

Se necesita un nodo que será donde se ejecute el *Registry*:

```
IceGrid.Node.CollocateRegistry=1
```

Listado 6.35: Configuración del *Registry*

```
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.Data=db/registry

IceGrid.Registry.PermissionsVerifier=IceGrid/NullPermissionsVerifier
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsVerifier
```

Listado 6.36: Configuración del *Registry* (continuación)

El directorio `db/registry` indica el donde se guardarán los datos del *Registry*. Las dos líneas que hacen referencia a los permisos se deben poner porque el *Registry* está controlado por por un verificador de permisos.

Los nodos de *IceGrid* deben tener un nombre que los identifique y un directorio donde se almacenarán los datos del nodo en cuestión.

```
Ice.Default.Locator=IceGrid/Locator -t:tcp -h 192.168.1.123 -p 4061
IceGrid.Node.Data=db/server
IceGrid.Node.Name=Worker1
IceGrid.Node.Endpoints=tcp
```

Listado 6.37: Configuración del «Worker1»

Así queda el primer nodo correctamente configurado. y faltaría el segundo, cuyo fichero de configuración es análogo al anterior.

```
IceGrid.Node.Data=db/server
IceGrid.Node.Name=Worker2
IceGrid.Node.Endpoints=tcp
```

Listado 6.38: Configuración del «Worker2»

Tras realizar todas estas configuraciones, queda lanzar los nodos y la herramienta icegrid-gui. Que se hace con una *Makefile*

```
all: start

start: create managerNode clientNode workerNode1 locator

locator: managerNode
	icegrid-gui --Ice.Config=locator.cfg &

workerNode1: managerNode
	icegridnode --Ice.Config=workerNode1.cfg &


clientNode: managerNode
	icegridnode --Ice.Config=clientNode.cfg &

managerNode:
	icegridnode --Ice.Config=managerNode.cfg &
	sleep 2

create:
	mkdir -p db/manager
	mkdir -p db/registry
	mkdir -p db/client
	mkdir -p db/worker1

destroy: clean
	killall icegridnode


clean:
	$(RM) -rf db/*
	$(RM) *~
```

Listado 6.39: Makefile para lanzar los nodos de *IceGrid*

Simplemente para lanzar todo lo realizado hasta ahora basta con escribir en un terminal:

```
$ make start
```

Listado 6.40: Órden para lanzar los nodos junto con *Icegrid*.

Con esto quedan los nodos correctamente configurados y algo similar a la figura 6.11.

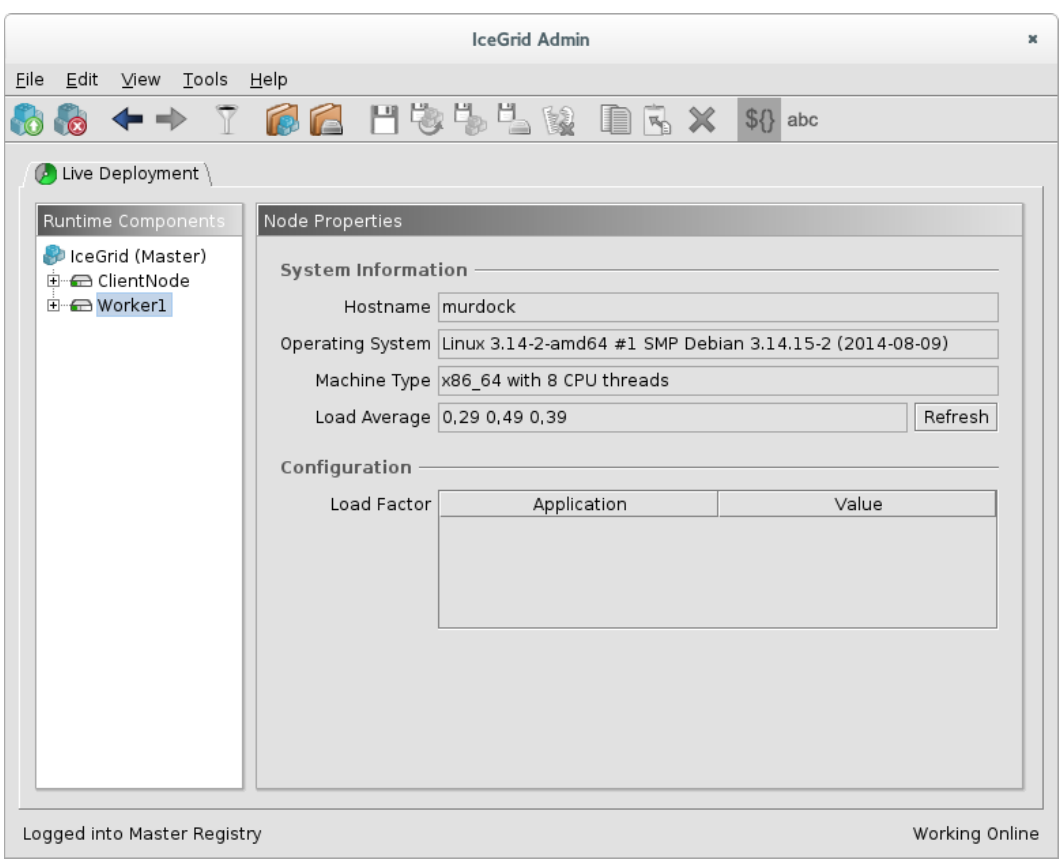

Figura 6.11: *IceGrid* con los nodos del sistema.

En la ruta relativa `src/icebuilder/icegrid` del repositorio y del CD adjunto se pueden encontrar estos ficheros de configuración para lanzar los nodos.

6.9.5 Pruebas

Las pruebas de esta iteración se realizan comprobando «in situ» en la herramienta *icegrid-gui* que todos los nodos están correctamente configurados. Cuando un nodo está conectado al sistema aparece marcado como verde. Además se procede a la construcción de un paquete Debian que arroja un log de toda la construcción.

6.9.6 Entrega

Tras esta iteración la entrega del prototipo una vez aprobados los cambios, consiste en tener todas las partes del sistema distribuido en una aplicación de *IceGrid* para una sencilla gestión a través de su interfaz gráfica. Se han añadido los nodos mínimos para que el sistema funcione.

7 Conclusiones y trabajo futuro

En el presente capítulo se muestran los objetivos que se han alcanzado con este proyecto. También se proponen mejoras para el mismo que se han propuesto en el Google Summer Of Code[1] del año 2014 o alguna comentadas con los integrantes del proyecto Debian. Por último se hace referencia a la participación de este proyecto en el CUSL y el premio obtenido en la fase final.

7.1 Conclusiones

Las conclusiones a las que se ha llegado tras el desarrollo de este proyecto y cumpliendo con los objetivos propuesto inicialmente son las siguientes:

- Diseño de una arquitectura distribuida P2P para donar ciclos de CPU tipo BOINC (Berkeley Open Infrastructure for Network Computing), construcción de paquetes Debian, construcción compartida en un entorno dado y monitorización del proceso, además de ofrecer balanceo de carga y transparencia de localización para saber que computadores están encendidos, y destinar la construcción de paquetes Debian a aquellos computadores que tengan menos carga de CPU.
- Desarrollo de nodos de construcción basados en máquinas virtuales en lugar de entornos «chroot» para facilitar la disponibilidad de los recursos (la construcción de paquetes se puede parar, reanudar o migrar a otro nodo compatible). Además, un único computador puede ejecutar varios nodos con el fin de construir paquetes para distintas arquitecturas.
- Implementación de construcción de paquetes mediante transacciones, que gracias a ellas, el sistema puede recuperarse frente a fallos tanto intencionados (cuando un trabajador termina su jornada laboral y apaga su equipo) como no intencionados (como apagones de luz).
- Repositorios «backports» personalizados impulsados por las necesidades de los usuarios en lugar de la política de Debian.

[1] El *Google Summer of Code* es un programa en el que Google remunera a estudiantes que participan en proyectos de software libre, más información en su página web `https://www.google-melange.com/gsoc/homepage/google/gsoc2014`

- Utilización de patrones de software. Gracias a los patrones se han utilizado soluciones ya probadas apara resolver algunos problemas. Además de esta ventaja, los patrones de software ofrecen facilidad para crear obras derivadas, y en este caso, gracias al uso de los patrones «composite» y «command» será más fácil reutilizar el código fuente de este proyecto.

- Reutilización de software. La ventajas de la reutilización de software se han visto reflejadas en numerosos aspectos. En este proyecto se han utilizado programas de software libre para cubrir necesidades que ya estaban cubiertas. Esto ha permitido utilizar herramientas que ya estaban probadas y que funcionaban, lo cual ha permitido reducir el tiempo de implementación.

7.2 Conclusiones personales

Siempre he querido que mi PFC estuviese relacionado con el software libre, además de incluir aspectos de redes y sistemas distribuidos que son unos de los campos por los que más interés tengo y sobre los que más me gusta aprender. Desde que empezó este proyecto he aprendido también nuevos lenguajes de programación que me han permitido enriquecer mi formación.

He aprendido como se empaqueta software y como se distribuye, así como los entresijos que existen con sus dependencias y los sistemas tan complejos como Debian. Ahora se un poco mejor como funciona Debian.

Me he dado cuenta en todo este tiempo que lo que he aprendido es una ínfima parte de lo que hay fuera y no ha hecho más que abrirme los ojos para seguir aprendiendo.

Además de todo ello, he podido estar en contacto con gente tan peculiar como la del proyecto Debian, que también han contribuido con este proyecto, aportando su granito de arena en forma de ideas cuando este proyecto se presentó al GSoC. También gracias al CUSL he estado en contacto con compañeros de otras universidades presentando nuestros proyectos en al fase final que se llevó a cabo en Sevilla

Por todo ello, considero que este proyecto se ha ajustado a lo que yo buscaba, y es el trabajo que más he disfrutado haciendo en toda la carrera.

7.3 Trabajo futuro

Un sistema de este tipo debe estar abierto a cambios y mejoras, y más aún cuando de un proyecto de software libre se trata. Por ello, en el último año de desarrollo este proyecto sirvió de base para proponer otro proyecto para el *Google Summer of Code*. Estas ideas forman parte del trabajo futuro que se puede desarrollar con este proyecto y alguna de ellas también ha sido propuesta por personas pertenecientes al proyecto Debian, con las cuales se

ha tenido contacto a través de su lista de correo[2].

7.3.1 Infraestructura distribuida y oportunista para construir paquetes Debian confiables

Esta mejora fue propuesta como proyecto para el Google Summer of Code[3] del año 2014.

Algunos usuarios pueden necesitar ciertas características en Debian que solamente están presentes en su rama *unstable* o incluso en *experimental*. Por ejemplo, un desarrollador de la biblioteca *Orca* podría necesitar la versión más nueva del paquete `Zeroc-ice`, pero no necesita actualizar todo el sistema a la rama *unstable*. A menudo, los desarrolladores reconstruyen los paquetes desde los fuentes de *unstable* o *experimental* en un entorno *stable* con pbuilder y crean repositorios no oficiales para esos paquetes.

Este proyecto proporciona una manera fácil de poder tener esos repositorios denominados «backports» desde Debian.

Se trataría de diseñar un modelo de confianza, que podría estar basado en claves GPG, y aceptado por usuarios de Debian y/o Debian developers, en el que ellos mismos donen ciclos de CPU para construir los paquetes que posteriormente se subirán a los repositorios.

7.3.2 Soporte a más arquitecturas

Debian ofrece soporte para muchas arquitecturas de forma oficial. Este proyecto por el momento, solamente ofrece soporte para dos de ellas, que son *amd64* y *i386*. La mejora que se plantea aquí es obvia, dar soporte a más arquitecturas como por ejemplo *powerpc* o *ARM*. Será necesario realizar un estudio de cuál es el estado en el que se encuentra la emulación de procesadores de otras arquitecturas con *Qemu* e ir añadiendo en la medida de lo posible máquinas virtuales con esas arquitecturas a cada uno de los nodos.

7.3.3 Dependencias con repositorios

Este proyecto ofrece unos repositorios «backports» en el que las versiones de los paquetes no son las que corresponden oficialmente con esa rama en Debian. En el respositorio los paquetes que se construyen pueden generar conflictos con algunos paquetes especiales como *libmcpp*. Si se necesita una versión concreta, por ejemplo *libmcpp*, para un paquete de este proyecto, pero el paquete se quiere construir para rama *stable* y en esta no está *libmcpp*, se tendría un gran problema porque se tendrían que actualizar muchos paquetes del sistema.

En la figura 7.1 se describe el problema. En el repositorio situado a la derecha está el paquete *libmccp-2.7.1* que es el que se tendría instalado en el sistema comunmente. Sin embargo, se ha construido *libmcpp-2.7.2* en el repositorio *backports*. En el momento en

[2]Lista de correo del *Google Summer of Code en Debian*`http://lists.alioth.debian.org/pipermail/soc-coordination/`

[3]Página web general del Google Summer of Code `http://www.google-melange.com/`

el que se quiera instalar ese paquete, se actualizaría a la versión más nueva, generando el conflicto.

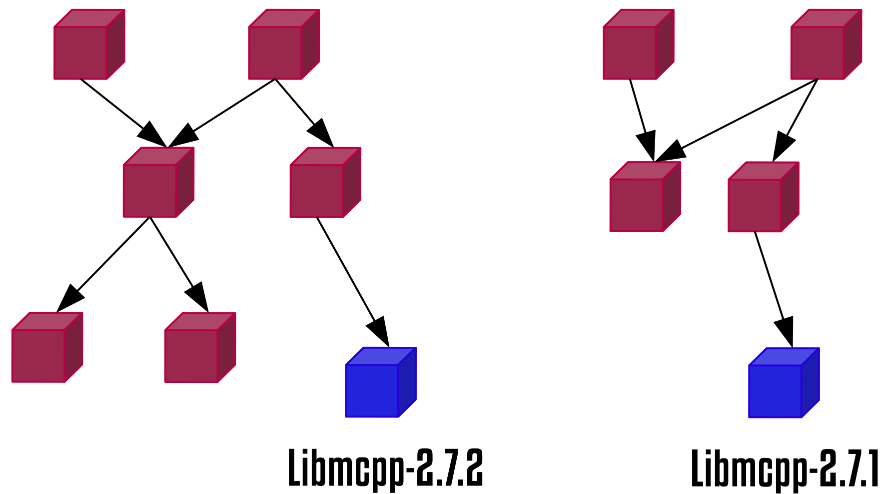

Figura 7.1: Dependencias con otros repositorios.

Este problema se podría resolver creando repositorios para determinadas versiones de los paquetes.

7.4 Premios y honores

El CUSL (Concurso Universitario de Software Libre), es un concurso de desarrollo de software, hardware y documentación técnica libre en el que pueden participar estudiantes universitarios de primer, segundo y tercer ciclo; así como estudiantes no universitarios de bachillerato, grado medio y superior del ámbito estatal español. La VIII Edición se llevó a cabo durante el curso 2013 - 2014. En él se inscribieron 80 proyectos y más de 120 estudiantes de todas las universidades de España. Este proyecto fue seleccionado como proyecto finalista y fue presentado en la Escuela Técnica Superior de Ingeniería Informática de Sevilla durante los días 14, 15 y 16 de Mayo del año 2014 donde recibió el **«Premio al mejor proyecto innovador»** (Figura 7.2).

Figura 7.2: Premio al mejor proyecto innovador

ANEXOS

Manual de usuario. Instalación de un Nodo

A.1 Introducción

EL presente manual describe los pasos necesarios para la instalación de un nodo en el sistema **icebuilder**, un sistema distribuido para construcción de paquetes Debian. Un nodo del sistema distribuido está compuesto de un computador que a su vez tiene máquinas virtuales con una distribución Debian GNU/Linux versión *testing* para construir paquetes Debian.

A.1.1 Estructura del manual

Este manual se divide en 4 secciones:

- ¿Qué es IceBuilder Node?
- Requisitos
- Instalación
- Configuración

Cada uno de los capítulos describe las acciones necesarias explicando con detalle todos los pasos a seguir para configurar un nodo que contiene a dos máquinas virtuales de arquitecturas una *i386* y otra *amd64*.

A.2 ¿Qué es IceBuilder Node?

IceBuilder Node es un nodo del sistema distribuido para construir paquetes debian que se encarga de realizar la reconstrucción de paquetes. Genera los paquetes para las distintas arquitecturas para las cuales se quieren construir los paquetes ya existentes y una vez construidos, subirlos al repositorio (Esta opción se implementará en posteriores versiones de desarrollo).

IceBuilder Node está formado por un computador que a su vez aloja una o varias máquinas virtuales donde se realiza la construcción de paquetes, comprobación de paquetes una vez construido, congelación de estado de máquina virtual y muchas más operaciones.

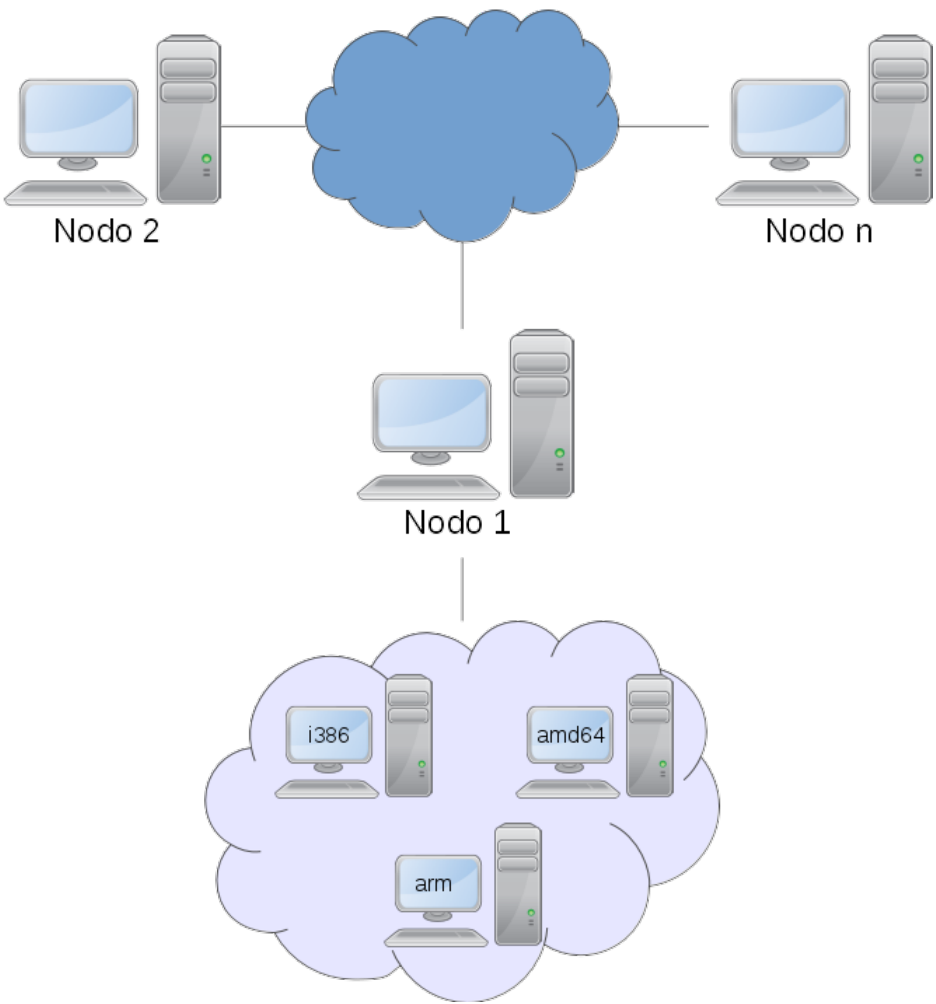

Figura A.1: Esquema del sistema

A.2.1 ¿Por qué Icebuilder Node?

Mientras se desarrollaban las primeras iteraciones de **icebuilder** y tras realizar algunas pruebas, se llegó a la conclusión de que ofrecía más ventas utilizar máquinas virtuales en lugar de *pbuilder* (ver sección 6.3). Con las máquinas virtuales se tienen sistemas limpios donde realizar la construcción.

Al utilizar máquinas virtuales se tiene una opción que puede resolver el problema que se tiene cuando un paquete tarda mucho tiempo en construirse y el computador se tiene que apagar. Esto se soluciona con la utilización de *snapshots*. Una *snapshot* o, traducido al español, una imagen, es la operación por la cual se toma un estado en un instante *t* cualquiera de la máquina virtual y se congela, pudiendo en cualquier momento ir a ese estado de congelación y, empezar o continuar desde ese estado la tarea que se estuviese desarrollando la próxima vez que se encienda el ordenador o la máquina virtual. Esta operación también tiene la ventaja de que se puede revertir y volver a un estado anterior cuando se requiera.

Una de las ventajas que tiene utilizar este sistema es que se pueden generar paquetes uti-

lizando la emulación de algunos procesadores y que permitiría construir paquetes para un mayor número de arquitecturas que si se usase *pbuilder*. De esta forma, se generaliza la solución al problema dando soporte a más arquitecturas. Otra ventaja es que la máquina emulada, puede resultar en ocasiones más rápida que la propia máquina real. Se puede utilizar de ejemplo el famoso *Raspberry Pi*, en el cual la construcción de un paquete tardaría mucho más que en una máquina virtual emulando ese procesador en un PC de escritorio.

A.3 Requisitos

Los requisitos que se detallan en esta sección hacen referencia tanto a software como a hardware.

A.3.1 Requisitos de hardware

Los requisitos de hardware indican el hardware mínimo que tiene que tener un computador para poder instalar un nodo y que este funcione.

- Conexión a internet de banda ancha.
- Al menos 2GB de RAM.
- Procesador que soporte virtualización para *KVM*.
- Mínimo 10 GB de espacio libre en la partición.

A.3.2 Paquetes necesarios

Para la instalación y configuración de IceBuilder node se necesitan los siguientes paquetes:

- libvirt
- virtinst
- virt-viewer
- virt-manager
- sshpass
- libguestfs-tool
- guestfish

Todos ellos hacen referencia a la API libvirt por la cual se pueden gestionar máquinas virtuales con comandos y de forma casi automática.

Para facilitar la instalación de todos esos paquetes, se ha hecho un paquete que los instala, cuyo nombre es *icebuilder-tools*:

```
# aptitude install icebuilder-tools
```

Por defecto los comandos de libvirt solamente los puede manejar el usuario **root**, por ello es necesario que se ajuste al entorno donde se vaya a utilizar el sistema. En este caso se le dará permiso al grupo de usuarios ïcebuilder". Para ello es necesario editar el fichero `/etc/libvirt/libvirtd.conf` y cambiar el grupo de usuarios de la línea

```
unix_sock_group = "libvirt"
```

y configurarlos para que el grupo ïcebuilder"de usuarios puedan interactuar con las máquinas virtuales.

```
unix_sock_group = "icebuilder"
```

Una vez realizado esto, es necesario reiniciar el servicio con

```
# service libvirt-bin restart
```

A.3.3 Requisitos de ejecución

Dada la naturaleza de libvirt, es altamente recomendable añadir lo siguiente al .bashrc para no tener que ejecutarlo continuamente cada vez que se inicia el computador

```
export LIBVIRT_DEFAULT_URI=qemu:///system
```

Se necesita crear una red virtual [1] para interactuar con las máquinas virtuales y un directorio[2] donde se alojarán las máquinas virtuales que en este caso, se creará en la ruta relativa del sistema `/home/usuario/.icebuilder`. Este directorio debe existir en el sistema donde se realice la instalación. Además deberás introducir la ruta **absoluta** en el archivo src/xml/pool-icebuilder.xml entre las etiquetas `<path>`, tal y como se muestra en el siguiente ejemplo:

```
<pool type="dir">
<name>icebuilder</name>
<target>
  <path>/home/usuario/.icebuilder</path>
</target>
</pool>
```

Listado A.1: Configuración de red virtual.

Ejecuta ahora el script *create_stuff.sh* para que se configuren tanto la red como el directorio donde se instalarán las máquinas virtuales.

[1]Formato XML para redes con libvirt
[2]Formato XML para almacenamiento con libvirt

A.3.4 El archivo preseed

Preseed[3] ofrece un mecanismo para responder a las preguntas que se realizan durante la instalación de Debian sin tener que introducir las respuestas manualmente mientras se ejecuta la instalación. Gracias a esto, es posible automatizar completamente la mayoría de instalaciones e incluso añadir algunas características que no están disponibles durante la instalación normal, como puede ser la instalación de algunos paquetes o la configuración de sudo por defecto.

El fichero pressed.cfg ya ha sido configurado de forma óptima para la instalación del sistema.

A.4 Instalación

Este es el punto en el que se instalan las máquinas virtuales en un computador. El manual sigue un ejemplo en el cual se instala una máquina virtual de cada una de las arquitecturas.

A.4.1 Instalación de máquinas virtuales

Para proceder a instalar las máquinas virtuales se ha creado un script que automatiza esta tarea, está en `src/install_vm.sh`. Este script genera automáticamente una instalación mínima del sistema perativo Debian GNU/Linux rama testing de arquitectura i386 y amd64 y que tiene todos los paquetes necesarios para poner en marcha el nodo. Debe ejecutarse desde el directorio donde se crearán las máquinas virtuales, que en el ejemplo es `/home/usuario/.icebuilder`:

```
./install_vm.sh
```

Si solamente se desa instalar **una** de las dos arquitecturas (i386 o amd64), basta con pasarla como parámetro al script y automáticamente se generará:

```
./install_vm.sh <arquitectura>
```

¡Importante! Si se detectase algún error durante la instalación es recomendable ejecutar la instalación por separado de ambas máquinas virtuales.

Al ejecutar el script, se realiza una descarga de internet y luego aparece la ventana del *virt-manager* donde se puede seguir el proceso de la instalación. En esta visión no se podrá ver más que unos gráficos con la carga de trabajo que tiene cada una de las máquinas virtuales, por lo que no aporta nada interesante por el momento.

[3]Más información en su página web

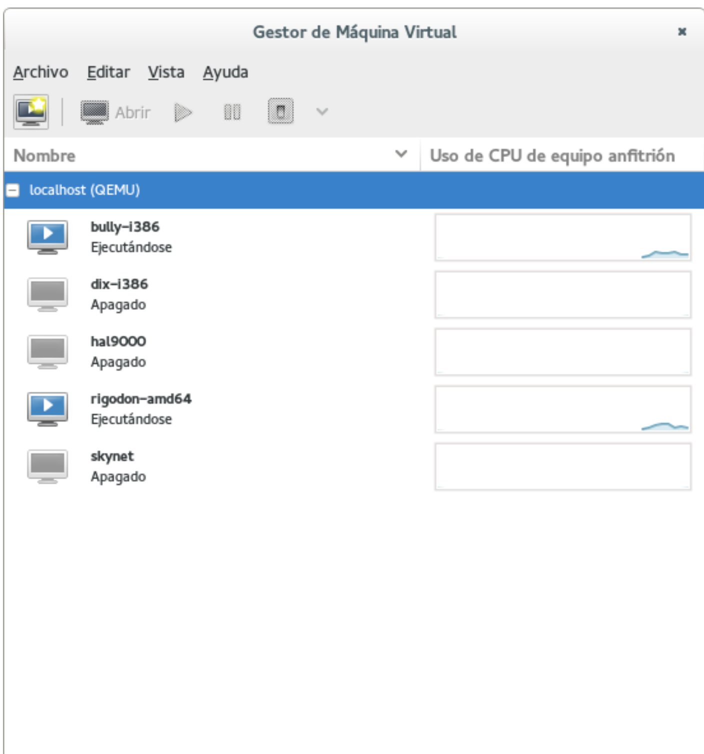

Figura A.2: Máquinas virtuales.

Cabe la posibilidad de ver con más detalle el proceso, seleccionando una máquina virtual y pinchando sobre el botón «*open*» del menú de virt-manager.

Lo que se verá es la típica instalación de Debian pero donde el usuario final no tiene que hacer nada porque todo ya se ha configurado automaticamente.

Cuando todo el proceso de instalar una o ambas máquinas virtuales ha finalizado, las máquinas se apagan y se puede continuar con su configuración.

A.5 Configuración

Esta es la sección referida a la configuración de las máquinas virtuales. Si solamente se ha instalado una máquina virtual, no tiene por qué utilizar *clusterssh*. Pase directamente al siguiente punto, configuración de *sudo* A.5.2.

A.5.1 Configuración de máquinas virtuales

Por defecto la mayoría de los paquetes están listos e instalados en las máquinas virtuales, incluso se tiene creado un usuario por defecto que puede utilizar el comando `sudo`. Este usuario tiene por nombre `arco` y por contraseña `arco`. Además, se necesita instalar algún paquete más en el computador anfitrión, ya que la conexión entre el anfitrión y las máquinas virtuales se realizad por *SSH*.

```
# aptitude install sshpass clusterssh
```

El paquete *clusterssh* permite ejecutar el mismo comando en n máquinas conectadas por *SSH*. Viene muy bien para no tener que repetir tareas en ambas máquinas virtuales. Con el uso de *cssh* se hará todo de una sola vez. Se verá más adelante su uso en la subsección A.5.2 aunque es opcional.

A.5.2 Configuración de sudo

Si solamente tiene una máquina virtual no utilice *clusterssh*, puede utilizar *ssh* o meterse normalmente en la máquina virtual para realizar la configuración.

Una vez que *clusterssh* está instalado en el sistema anfitrión, lo que hay que hacer ahora es configurar las máquinas virtuales para que no soliciten la contraseña cada vez que se utilice la orden `sudo`. Por lo tanto, mediante el siguiente comando se procede a conectarse a las dos máquinas virtuales instaladas previamente.

```
cssh icebuilder@IP1 icebuilder@IP2
```

Donde `IP1` y `IP2` son las *IP* correspondientes a cada una de las máquinas virtuales, por defecto 192.168.122.11 y 192.168.122.12.

```
sudo nano /etc/sudoers
```

Y se añade al archivo para que no pida la contraseña cada que vez que se usa el comando `sudo` la línea `%icebuilder ALL=NOPASSWD:ALL`:

```
Defaults        env_reset
Defaults        mail_badpass
Defaults        secure_path="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
    :/bin"

# Host alias specification

# User alias specification

# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL

# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) ALL
%icebuilder   ALL=NOPASSWD:ALL
# See sudoers(5) for more information on "#include" directives:

#includedir /etc/sudoers.d
```

Listado A.2: Configuración de *sudo*.

A.5.3 Configuración de repositorios

Por defecto las instalación de Debian marca unos repos con el nombre en clave de la distribución testing (actualmente *wheezy* y proximamente *Jessie*).

Se procede a cambiar los repositorios para que apunten a *testing* y evitar así problemas con las nomenclaturas. De esta forma, las máquinas virtuales siempre serán rama *testing* de Debian.

Editar el fichero `/etc/apt/sources.list`

```
sudo nano /etc/apt/sources.list
```

Y se sustituye *wheezy* por testing, quedando el archivo de la siguiente forma:

```
deb http://ftp.es.debian.org/debian/ testing main
deb-src http://ftp.es.debian.org/debian/ testing main

deb http://security.debian.org/ testing/updates main
deb-src http://security.debian.org/ testing/updates main
```

Listado A.3: Configuración de repositorios.

Además hay que añadir los repositorios de las otras dos ramas de Debian, «stable» y «unstable»:

```
#TESTING
deb http://ftp.es.debian.org/debian/ testing main
deb-src http://ftp.es.debian.org/debian/ testing main

deb http://security.debian.org/ testing/updates main
deb-src http://security.debian.org/ testing/updates main

#UNSTABLE
deb http://ftp.debian.org/debian/ unstable main
deb-src http://ftp.debian.org/debian/ unstable main

#STABLE
#deb http://ftp.debian.org/debian/ stable main
#deb-src http://ftp.debian.org/debian/ stable main

# deb http://security.debian.org/ stable/updates main
# deb-src http://security.debian.org/ stable/updates main
```

Listado A.4: Configuración de repositorios (continuación).

Para que los paquetes de la rama «testing» tengan preferencia sobre los demás hace falta configurar *apt-pinning*. Con esto se pretende que, salvo que se indique lo contrario, por defecto se instalen paquetes de «testing».

Para realizar la configuración basta con editar el fichero `/etc/apt/preferences` dentro de la máquina virtual:

```
sudo nano /etc/apt/preferences
```

Los números indican prioridad, es decir, cuanto más alto, más prioridad tiene. Como se ha dicho antes, se le da la máxima prioridad a «testing».

```
Package: *
Pin:release a=testing
Pin-Priority: 900

Package: *
Pin:release a=unstable
Pin-Priority: 800

Package: *
Pin:release a=stable
Pin-Priority: 700
```

Listado A.5: Configuración de *apt-pinning*.

A.5.4 Configuración de Grub

Este paso es opcional y sirve para desactivar el tiempo de espera que tiene Grub para iniciar las máquinas virtuales, ya que *libvirt* no sabe a priori cuando una máquina virtual está lista para realizar las conexiones ssh. En los scripts de construcción de paquetes ya se tiene en cuenta que las máquinas virtuales pueden tardar un tiempo en encenderse, por lo que no es necesario ajustarlo a tiempo 0 aunque si recomendable.

Por ello, se edita el fichero */boot/grub/grub.cfg* cambiando el `tiemout` de 5 a 0.

Una vez se tiene todo esto, ya no es necesario realizar ninguna tarea más en la máquina virtual, ya que serán los scripts los que se encarguen de realizar todas las tareas.

A.5.5 Prueba y ejecución

Para comprobar que todo funciona correctamente, se ha realizado un caso de prueba que a su misma vez sirve para construir el paquete *sl* para las arquitecturas *i386* y *amd64*.

Desde el directorio *test* dentro del repositorio solamente haría falta ejecutar la orden:

```
$ make test
```

Para comprobar el resultado, dentro del directorio `test/builtpackage` estarán los paquetes que se acaban de construir.

Creación de un repositorio de paquetes Debian

UNA pieza clave para distribuir el software son los repositorios de paquetes, por eso, en este apéndice se llevará a cabo la instalación y configuración de un repositorio de paquetes Debian con la herramienta *reprepro*[Lin12] con soporte para subida y firmado de paquetes.

Reprepro [Lin12] es una herramienta para administrar respositorios de paquetes Debian. Puede albergar paquetes subidos manualmente o descargados de otros repositorios. No se necesita ningún servidor de bases de datos puesto que los «checksum» y los paquetes se almacenan en un archivo de la base de datos *BerkeleyDB*.

Hay muchas más herramientas para la gestión de repositorios [1]. En el caso que ocupa a este proyecto se ha elegido reprepro por resultar un entorno familiar ya que se ha trabajado previamente con él.

El repositorio se ha instalado en una máquina virtual con el fin de estar aislado del resto de máquinas y debido a que no se dispone de un gran número de computadores.

Se deja a decisión del lector elegir otra herramienta para la gestión de los repositorios, pero tenga en cuenta que esta ya está probada durante el desarrollo del proyecto.

B.1 Instalación

Se requiere de una instalación previa de Debian GNU/Linux, versión «stable» recomendada, por ser la que más ofrece de entre las tres ramas. Se procede al uso de las órdenes habituales en cualquier distribución de Debian como *aptitude* o *apt-get* para la instalación:

```
# apt-get install reprepro
```

```
# aptitude install reprepro
```

[1] Herramientas para la gestión de repositorios Debian

B.2 Configuración

Se supone que el repositorio se va a crear en `/var/repo`, por lo tanto, es necesario crear los siguientes directorios:

```
$ mkdir -p /var/repo/conf
$ mkdir -p /var/repo/incoming
```

El directorio `conf` tendrá los archivos necesarios para la configuración de reprepro. El directorio `incoming` se utilizará a modo de cola de entrada de los paquetes que suban los developers.

B.2.1 Configuraración de conf/distributions

En este fichero se establecen las reglas por las cuales se actualizan los paquetes en el repositorio.

Lo primero es crear el archivo `conf/distributions` con el siguiente contenido:

```
Origin: Example Server
Label: skynet
Suite: stable
Codename: stable
Architectures: i386 amd64 source
Components: main
Description: Developers repository
Pull: stable-pull

Origin: Exmple Server
Label: skynet
Suite: unstable
Codename: sid
Architectures: i386 amd64 source
Components: main
Description: Developers repository
SignWith: PGP-ID-123
Uploaders: ./uploaders
```

Listado B.1: Fichero `conf/distributions`

El repositorio en cuestión tendrá dos ramas, *estable* y *usntable*. Mediante la propiedad *Pull* define una regla (*stable-pull*)por la cual se actualizará la rama *stable* a partir de *sid*.

La rama *unstable* será donde por defecto suban los paquetes los developers autorizados para ello. El campo *Signwith* especifica el identificador de la firma PGP con la que se generarán los archivos `Release.gpg`, esto es lo que se conoce comúnmente como «firmar el repositorio». El o los usuarios que tengan privilegios de modificar el repositorio, deben tener esta clave en su anillo de claves.

Es necesario que se genere una clave para el repositorio, lea el apéndice dedicado a la generación de claves GPG con el fin de generar una clave para el repositorio. Las claves en este documento tendrán valores arbitrarios.

Por último, la propiedad «Uploaders» especifica la ubicación de un archivo con las firmas de los developers autorizados por el repositorio.

```
allow * by key PGPID-DEVELOPER1
allow * by key PGPID-DEVELOPER2
allow * by key PGPID-DEVELOPERN
```

Permite a los developers que tienen esas claves subir los paquetes al repositorio.

B.2.2 Configuración de conf/pulls

Como se ha mostrado previamente en el archivo de configuración de *reprepro*, la rama *stable* se actualizará por medio de la regla *stablle-pull*

```
Name: stable-pull
From: sid
Components: main
Architectures: i386 amd64 source
FilterList: purge packages.old
```

Listado B.2: Fichero `conf/pulls`

Uno de los campos importantes son las reglas de filtrado *FilterList*. En el caso de este repositorio, mediante el comando `purge`, que indica en qué tiempo de actualización se eliminan los archivos que no aparecen dentro de *packages.old*. El nombre del archivo puede ser cualquiera, en este caso se pone el ejemplo de paquetes que llevan más de una semana sin actualizarse, y por lo tanto como no se han actualizado en ese tiempo, pasan a stable. Este archivo es generado por un programa externo.

B.2.3 Configuración conf/incoming

Por último hay que gestionar la cola de subida de paquetes al repositorio.

```
Name: sid-process
IncomingDir: incoming
TempDir: /tmp
Default: sid
Allow: unstable>sid UNRELEASED>sid
Cleanup: on_deny on_error
```

Listado B.3: Fichero `conf/incoming`

Los paquetes por defecto se subirán a la rama «sid». Si los paquetes no han sido firmados, lo han sido por alguien extraño cuya clave no está en el repositorio ,o se produce un error durante la subida, se borrarán de la lista con la propiedad *Cleanup*. La propiedad *Allow* permite que los paquetes en cuyo *changelog* se utilice «unstable» o «UNRELEASE» vayan a *sid*. Esto es muy común si se está probando el paquete.

B.3 Nociones básicas del uso del repositorio

A continuación, se muestran algunas órdenes para la gestión del repositorio que se suponen ejecutadas desde el directorio raiz donde se encuentra el repositorio, es decir `/var/repo`

Importar paquetes .deb

```
$ reprepro -V includedeb sid path/to/*.deb
```

La ruta `path/to*.deb` hace referencia a la ruta absoluta o relativa en donde se encuentra el .deb que se quiere subir al repositorio.

Eliminar un paquete

```
$ reprepro -V remove sid package_name
```

Procesar cola de envíos

```
$ reprepro -V processincoming sid-process
```

B.4 Hacer el repositorio accesible

Hacer el repositorio accesible significa que otros usuarios podrán ver su contenido y descargar los paquetes de él. Al mismo tiempo, otros usuarios que tengan el rol de «developers» podrán subir los paquetes utilizando las claves que se configuraron previamente en la sección B.2.1.

B.4.1 Exportar la clave

Para saber que los paquetes son confiables es necesario exportar la clave GPG del repositorio, de este modo, esa clave podrá ser copiada por los usuarios que quieran usar el repositorio y sabrán que los paquetes con esa firma serán confiables.

```
$ gpg --export --armor KEY-123 > key.asc
$ mv key.asc /var/repo
```

B.4.2 Importar la clave

La clave del repositorio se importa para que cada usuario la tenga en su anillo de claves, de esta forma podrá subir los paquetes si es «developer» y actualizar con `apt-get update` sin obtener errores.

```
$ wget -O- http://example.com/repo/key.asc | sudo apt-key add -
$ sudo apt-get update
```

B.5 Utilizando el repositorio

En esta sección se mostrarán las órdenes que tienen que seguir los «developers» o mantenedores de paquetes para subir un paquete al repositorio.

B.5.1 Firmado de paquetes

Una vez que el repositorio está accesible, los «developers» podrán subir paquetes. Para ello tras generar el paquete, antes de nada hay que firmarlo con la clave del «developer» para que el repositorio permita su subida.

```
$ design -kKEY1234 paquete_1.0-1_amd64.changes
```

B.5.2 Subida de paquetes al repositorio

Debian ofrece herramientas para subir los paquetes al repositorio como son **dupload** o **dput**.

dupload

Es un paquete y un script que automatiza la subida de paquetes al repositorio, Para dejar constancia de la subida y enviar un e-mail sobre la subida del paquete. Todo ello se puede configurar fácilmente. Con la edición del fichero `dupload.conf` se simplifica el proceso:

```
  package config;

 $default_host = "host_del_repo";

 $cfg{'host\_del\_repo'} = {
 fqdn => "example.com",
 login => "repo_user",
 method => "scpb",
 incoming => "/var/repo/incoming/",

# The dinstall on ftp-master sends emails itself
 dinstall_runs => 1,
};
```

Listado B.4: Fichero `.dupload.conf`

Para subir el paquete solamente hay que ejecutar:

```
$ dupload --to host_del_repo paquete_1.0-1_amd64.changes
```

dput

Este paquete es muy similar a *dupload* pero hace las cosas de diferente manera. Además, tiene algunas opciones más como puede ser la comprobación de la clave GPG antes de subir el paquete o usar `dinstall`[2] después de subir el paquete. Para configurarlo es necesario

[2] `dinstall` es el demonio encargado de actualizar los paquetes en el repositorio Debian

editar el fichero `.dput.cfg`

```
[myrepo]
fqdn = myrepo.server.com
method = scp
login = myuser
incoming = /var/repo/incoming
post_upload_command = ssh myrepo.server.com reprepro -b /var/repo/ -V processincoming
     sid-process
```

Listado B.5: Fichero .dput.cfg

Para subir el paquete basta con ejecutar:

```
$ dput myrepo paquete_1.0-1_amd64.changes
```

B.5.3 Añadir más arquitecturas

En cualquier momento puede haber necesidad de añadir una nueva arquitectura al repositorio. Este caso supone que se requiere de una arquitectura *armel*. Lo primero es añadir la arquitectura *armel* en el campo *Architecture* del fichero `conf/distributions`:

```
Origin: Example Server
Label: skynet
Suite: stable
Codename: stable
Architectures: i386 amd64 armel source
Components: main
Description: Developers repository
Pull: stable-pull

Origin: Exmple Server
Label: skynet
Suite: unstable
Codename: sid
Architectures: i386 amd64 armel source
Components: main
Description: Developers repository
SignWith: PGP-ID-123
Uploaders: ./uploaders
```

Una vez añadida la arquitectura se ejecuta reprepro con el argumento `flood` seguido de la `rama` y la arquitectura que se acaba de añadir:

```
$ reprepro flood <rama> <arquitectura>
```

Se crearán automáticamente los directorios necesarios para dar soporte a la nueva arqutiectura *armel* que se podrá utilizar con normalidad.

Construcción de un paquete Debian

EL paquete Debian es una parte muy importante del desarrollo de este proyecto, de hecho, todo gira en torno a él. Aunque para documentar todo lo referente al paquete Debian ya está la *Debian Policy* [JS12] y la *guía del nuevo desarrollador de Debian* [RA13], se mostrará a modo resumido todos los aspectos considerados más importantes del paquete Debian que pueden resultar de ayuda para entender el presente documento. No obstante, se recomienda al lector que quiera conocimientos más profundos leer los documentos citados anteriormente.

Se podría hablar de dos tipos de paquetes Debian, el oficial, que es el que se encuentra en los repositorios oficiales de Debian y sigue la política de Debian, y el no oficial, que es el que no se encuentra en los repositorios oficiales de Debian y se puede distribuir por los medios que cada uno considere oportunos. Un paquete no oficial puede estar construido de la forma correcta como uno oficial, pero la diferencia radica en que no ha sido subido a Debian por ningún «Debian Developer».

C.1 Primeros pasos

Si se va a empaquetar algún programa, lo primero que hay que hacer es obtener una copia del programa para probarlo. A partir de aquí, la generación de un paquete Debian requiere nombrar a los ficheros de una determinada manera. Por ejemplo, la copia del programa, comprimido en formato `tar`: `package_version.tar.gz`

Lo siguiente es añadir las modificaciones para «debianizar» el programa:

- `package_version.orig.tar.gz`
- `package_version-revision.debian.tar.gz`
- `package_version.orig.dsc`

Y por último generar el programa: `package_version-revision_arch.deb`

Nótese que la palabra «package» debe ser el nombre del programa en cuestión y «version» el número de versión del programa original. «revision» es la revisión del paquete Debian, y por último «arch» es la arquitectura del paquete según la *Debian Policy* [JS12]

C.2 Estructura básica

Normalmente los paquetes Debian se construyen a partir de un código que no está desarrollado por el propio mantenedor del paquete. Es por ello habitual partir de algún archivo comprimido *tar.gz* donde se encuentra todo el código fuente. El programa `dh_make` ayuda a generar las plantillas de ficheros necesarios para construir el paquete. `dh_make` necesita conocer la información del mantenedor, por lo que es necesario proporcionar tanto el nombre completo (DEBFULLNAME) como el e-mail (DEBEMAIL). Es bueno añadir estas variables al `$HOME/.bashrc` para que no se tengan que escribir una y otra vez:

```
DEBFULLNAME="Nombre Apellidos"
DEBEMAIL="tue-mail@ejemplo.com"
export DEBULLFNAME DEBEMAIL
```

Una vez configurado es hora de ejecutar `dh_make` desde dentro del directorio del programa.

```
dh_make -f programa.tar.gz
```

Tras la ejecución de la orden anterior, se ha crea un directorio `debian` con muchos ficheros, muchos de ellos son plantillas. La «única» tarea que hay que hacer es completar toda la información requerida siguiendo la «Debian Policy».

C.2.1 Archivos necesarios en el directorio `debian`

A continuación se describirán los archivos importantes para el paquete. Cuando se refiera a algún archivo se hará mediante la ruta relativa al directorio que se acaba de crear con `dh_make`, es decir, si se refiere al archivo `debian/control` quiere decir que el archivo estará dentro del directorio `debian` y se llamará `control`. El `debian/copyright` es el archivo `copyright` que está dentro del directorio `debian` y así sucesivamente.

debian/control

Es uno de los ficheros más importantes del paquete *Debian*, es una especie de manifiesto donde se incluye información relevante para la el sistema de gestión de paquetes. Véase un archivo `debian/control` cualquiera, por ejemplo el del paquete *gentoo*.

```
Source: gentoo
Section: x11
Priority: optional
Maintainer: Innocent De Marchi <tangram.peces@gmail.com>
Build-Depends: debhelper (>= 9), dh-autoreconf, libgtk2.0-dev, libglib2.0-dev
Standards-Version: 3.9.3
Homepage: http://www.obsession.se/gentoo/

Package: gentoo
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Suggests: file
```

```
Description: fully GUI-configurable, two-pane X file manager
gentoo is a two-pane file manager for the X Window System. gentoo lets the
user do (almost) all of the configuration and customizing from within the
program itself. If you still prefer to hand-edit configuration files,
they're fairly easy to work with since they are written in an XML format.
.
gentoo features a fairly complex and powerful file identification system,
coupled to an object-oriented style system, which together give you a lot
of control over how files of different types are displayed and acted upon.
Additionally, over a hundred pixmap images are available for use in file
type descriptions.
.
gentoo was written from scratch in ANSI C, and it utilizes the GTK+ toolkit
for its interface.
```

Listado C.1: debian/control paquete Debian

En este fichero hay dos secciones diferenciadas, el paquete fuente y la información necesaria para su construcción se pueden ver en la sección `Source`. El paquete o los paquetes binarios están separados por una línea en blanco y están en la sección `Package`.

Hágase un repaso de los campos del paquete fuente donde si no se especifica lo contrario, el campo a rellenar no es obligatorio:

Source Es el nombre del paquete fuente y es obligatorio. Tanto en el `debian/control` como en el archivo `.dsc`, este campo debe tener solamente el nombre del paquete fuente.

Maintainer Nombre del mantenedor del paquete y su dirección de correo electrónico, este campo es obligatorio. El nombre aparece primero y luego el correo eĺectrónico entre <>.

Uploaders Nombre o nombres con sus respectivas direcciones de correo electrónico de los mantenedores del paquete. El formato es el mismo que en el campo anterior.

Section Este campo no es obligatorio pero si recomendable. Especifica un área en la cual clasificar el paquete [1]

Priority Este campo es recomendable y representa el nivel de importancia que tiene para el usuario que este paquete esté instalado. Estos niveles pueden ser: `required`, `important`, `standard`, `optional` ó `extra`. Para ver una descripción detallada deniveles de prioridad consultar el capítulo 2.5 de la «Debian Policy» [JS12]

Build-depends Es una lista con los nombres de los paquetes y en algunos casos sus versiones que son necesarios para construir el paquete. Estos paquetes deben estar instalados en el sistema a la hora de construir el paquete fuente para poder construir los paquetes binarios.

Standars-Version Recomendado, la versión más reciente de las normas con el que el paquete cumple.

[1] Lista completa de las secciones en «SID» http://packages.debian.org/unstable/

Homepage La dirección del sitio web para el paquete. Preferiblemente la web desde donde puede obtenerse el paquete fuente y cualquier documentación que pueda servir de ayuda. El contenido desde este campo es **solamente** la URL, sin nada más.

Vcs-Browser Los paquetes están desarrollados con controles de versiones. El propósito es indicar el repositorio público donde se está desarrollando el paquete fuente.

Campos del paquete binario:

Package Campo obligatorio en el que se pone el nombre del paquete binario, el que dará lugar al `.deb`. Este nombre deberá cumplir una serie de reglas dependiendo del lenguaje en que está escrito, su finalidad, etc.

Architecture Dependiendo del contexto, este campo obligatorio, puede incluir una de las siguiente opciones:

- Una única palabra identificando una de las arquitecturas descritas. Estos nombres pueden obtenerse con la orden `dpkg-architecture -L`.
- Una arquitectura comodín como puede ser `any` y que sirve para todas las arquitecturas listadas con la orden `dpkg-architecture -L`. Es la más usada.
- `all` lo que indica que el paquete es independiente de la arquitectura. Para que el lector se haga una idea, `all` se refiere a programas que funcionan en todas las arquitecturas tales como ficheros multimedia, manuales o programas escritos en lenguajes interpretados, etc.
- `source` lo cual indica un paquete fuente.

Section Este campo aunque no es obligatorio, si es recomendable. Tal y como en el paquete fuente, indica un área en donde ubicar le paquete.

Priority Recomendado. Indica como de importante para el usuario es tener el paquete instalado. La prioridad `optional` se utiliza para paquetes nuevos que no entran en conflicto con otros de prioridad `required`, `important` o `standar`

Depends y otras Son las dependencias binarias del paquete que describen sus relaciones con el resto de paquete. Básicamente son los paquete que se requiere que estén instalados en el sistema para que el paquete funcione. Una relación de las dependencias binarias puede verse en el capítulo 7.2 de la *Debian policy* [JS12]

Description Este campo es obligatorio. Contiene la descripción del paquete binario. Consiste en dos parte, una descripción corta y la descripción larga basándose en el siguiente formato y sin los símbolos <>.

Homepage La dirección del sitio web para el paquete. Preferiblemente la web desde donde puede obtenerse el paquete fuente y cualquier documentación que pueda servir de ayuda. El contenido desde este campo es **solamente** la URL, sin nada más.

Package-Type Campo simple donde se indica el tipo de paquete, `deb` para paquetes binarios y `udeb` para paquete micro binarios.

debian/changelog

Este fichero contiene una descripción muy breve de los cambios que el mantenedor del paquete hace a los ficheros específicos del paquete. No se describen los cambios que sufre el programa, eso corresponde a su autor y eso suele estar en un fichero `.changes`

```
miproyecto (1.0-1) unstable; urgency=low

  * Initial release (Closes: #nnnn)  <nnnn is the bug number of your ITP>

 -- John Doe <johnDoe@icebuilder.com>  Wed, 25 Dec 2013 14:02:25 +0100
```

Listado C.2: changelog paquete Debian

Cada vez que el mantenedor realice un cambio en el paquete debe crear una nueva entrada en el `debian/changelog`. Si el mantenedor hace cambio en la versión del paquete, se incrementará el número de versión del paquete. Cuando el mantenedor del paquete empaquete una nueva versión, el número de versión del paquete vuelve a empezar desde 1.

En l listado C.2 «miproyecto» es el nombre del paquete, y entre parentesis (1.0-1), del cual «1.0» es la versión del código fuente, mientras que «1» es la versión del paquete Debian.

debian/copyright

Este fichero contiene información sobre el autor del programa y las licencias que se utilizan en el programa y en cada una de sus partes. Debería indicar también quien es el autor y la licencia de los ficheros del paquete Debian.

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: miproyecto
Source: <url://example.com>

Files: *
Copyright: <years> <put author's name and email here>
<years> <likewise for another author>
License: GPL-3.0+

Files: debian/*
Copyright: 2013 John Doe <johnDoe@icebuilder.com>
License: GPL-3.0+

License: GPL-3.0+
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
.
This package is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
.
```

```
 You should have received a copy of the GNU General Public License
 along with this program. If not, see <http://www.gnu.org/licenses/>.
 .
 On Debian systems, the complete text of the GNU General
 Public License version 3 can be found in "/usr/share/common-licenses/GPL-3".

 # Please also look if there are files or directories which have a
 # different copyright/license attached and list them here.
 # Please avoid to pick license terms that are more restrictive than the
 # packaged work, as it may make Debian's contributions unacceptable upstream.
```

Listado C.3: copyright paquete Debian

debian/rules

Es el `Makefile` que debe inlcuir una serie de objetivos como : clean, binary, binary-arch, binary-indep y build.

Estas reglas objetivo serán ejecutadas por `dpkg-buildpackage` o `debuild` cuando se construya el paquete.

- `clean` (obligatorio): elimina todos los archivos generados, compilados o innecesarios del árbol de directorios de las fuentes.
- `build` (obligatorio): para la construcción de archivos compilados a partir de los archivos fuenteo o la construcción de documentos formateados.
- objetivo `build-arch` (obligatorio): para la compilación de las fuentes en programas compilados (dependientes de la arquitectura) en el árbol de directorios de la compilación.
- objetivo `build-indep` (obligatorio): para la compilación de las fuentes en documentos formateados (independientes de la aqruitectura) en el árbol de directorios de la compilación
- `install` (opcional): para la instalación en la estructura de directorios temporal para el directorio `debian` de los archivos para cada uno de los paquetes binarios. Si existe el objetivo `binary` dependerá de este.
- `binary` (obligatorio): para la construcción de cada uno de los paquetes binarios (combinando con los objetivos `binary-arch` y `binary-indep`).
- `binary-arch` (obligatorio): para la construcción de paquetes dependientes de la aquitectura (`Architecture: any`).
- `binary-indep` (obligatorio): para la construcción de paquetes independientes de la aquitectura (`Architecture: all`).
- `get-orig-source` (opcional): para obtener la versión más reciente de las fuentes originales desde el lugar de almacenaje del autor.

Cuando se utiliza `dh_make` se genera un `debian/rules` como el siguiente:

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

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

% :
	dh $@
```

Listado C.4: debian/rules paquete Debian

debhelper hace casi todo el trabajo aplicando las reglas por defecto por lo que la única tarea que habría que realizar sería la de sobreescribir dichos objetivos.

C.3 La construcción del paquete

Para construir un paquete, primero hay que asegurarse de que estén instalados:

- el paquete `build-essential`
- los paquetes listados en el campo `build-Depends` del `debian/control`
- los paquetes listados en el campo `build-Depends-Indep` del `debian/control`

Después se ejecuta la orden

```
$ dpkg-buildpackage
```

Para los paquetes no nativos Debian, en el directorio padre se podrán ver los archivos

- `paquete_1.0.orig.tar.gz` El código fuente original comprimido, se ha renombrado a ese nombre para seguir los estándares de Debian.
- `paquete_1.0-1.dsc` Es un resumen de los contenidos del paquete. Se genera a partir del `debian/control` y se usa cuando se descomprimen las fuentes con `dpkg-source`. Está firmado con GPG por lo que se puede saber de quien es el paquete.
- `paquete_1.0-1.debian.tar.gz` Este fichero contiene el directorio `debian` al completo. Todas las moficiaciones están en los archivos de parches **quilt** en el directorio `debian/patches`
- `paquete_1.0-1_i386.deb` Es el paquete binario completo. Se puede instalar o eliminar con `dpkg` como el resto de paquetes.
- `paquete_1.0-1_i386.changes` Este fichero describe todos los cambios hechos en la versión actual del paquete y lo utilizan algunos programas de FTP de Debian

C.4 Actualización del paquete

Es posible que el paquete deba ser actualizado por diversos motivos.

C.4.1 Nueva versión del paquete

Tras tener el paquete listo y dado que muchos usuarios lo están utilizando, pueden surgir bugs. Para modificar el paquete se utiliza la herramienta `dquilt`. Para aplicar una nueva modificación:

```
dquilt new nombre_modificacion.patch
```

Y para actualizar una modificación ya existente:

```
dquilt pop nombre_modificacion.pathc
```

Uan descripción más detallada de la orden `dquilt` puede encontrarse en el capítulo 9 de la *guía del nuevo mantenedor de Debian* [RA13]

C.4.2 Nueva versión del autor

Cuando el autor libera una nueva versión, el mantenedor del paquete, debe revisarla. Es importante ver los `changelogs`, *NEWS*, ... para saber que es lo que ha cambiado. Con la orden `diff` se puede ver rapidamente comparando los ficheros de la nueva versión con la del antiguo.

C.4.3 Nueva versión del programa fuente

Si se parte de un paquete bien empaquetado, habría que copiar el directorio `debian` de la versión anterior a la nueva para realizar las adaptaciones necesarias. Además de ello, habría que realizar algunas tareas como comprimir las fuentes en un nuevo `paquete_nueva_-version.orig.tar.gz`, actualizar el `changelog`, etc. Nuevamente hay más detalles sobre esto en el capítulo 9.4 de la *guía del nuevo mantenedor de Debian*c̃itemaintainer

Dependencias de un paquete Debian

Como se explica en el apéndice C, en el archivo `debian/control` existen numerosos campos con diferentes significados. Uno de esos campos, es el *depends*, el cual describe las relaciones que existen entre los paquetes Debian. Otro de los campos importantes para este proyecto es el de *arquitecture*, donde se dice expresamente para que arquitecturas se tiene que construir el paquete.

Hay mucha más información en la Debian Policy [JS12] acerca de las dependencias de paquetes, sus campos y posibles situaciones que se pueden dar en cada caso. Como el objetivo de este documento no es explicar la *debian policy*, se hará un pequeño resumen que permita comprender mejor las relaciones entre los paquetes Debian. Por ello primero se explicará la sintaxis de las relaciones y más adelante las relaciones en sí.

Es muy importante también, conocer la diferencia entre **paquete fuente** y **paquete binario**. El paquete fuente es aquel que aparece en primer lugar en el archivo `debian/control`. Es todo el software que maneja el mantenedor del paquete, es decir el código fuente original junto con el directorio *debian*.

El paquete **binario** es lo que se instala en el sistema mediante *apt* o *aptitude*. Son aquellos que están después de la descripción del paquete fuente en el archivo `debian/control`

Del paquete *sl* se tendría el siguiente fichero *debian/control* en el listado D.1:

```
Source: sl
Section: games
Priority: optional
Maintainer: Hiroyuki Yamamoto <yama1066@gmail.com>
Build-Depends: cdbs, debhelper (>= 9), libncurses5-dev
Standards-Version: 3.9.3
Homepage: http://www.tkl.iis.u-tokyo.ac.jp/~toyoda/index_e.html

Package: sl
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: Correct you if you type ‘sl’ by mistake
Sl is a program that can display animations aimed to correct you
if you type ’sl’ by mistake.
SL stands for Steam Locomotive.
```

Listado D.1: Fichero *debian/control* del paquete *sl*.

Del cual la información del paquete fuente sería:

```
Source: sl
Section: games
Priority: optional
Maintainer: Hiroyuki Yamamoto <yama1066@gmail.com>
Build-Depends: cdbs, debhelper (>= 9), libncurses5-dev
Standards-Version: 3.9.3
Homepage: http://www.tkl.iis.u-tokyo.ac.jp/~toyoda/index_e.html
```

Listado D.2: Información del paquete fuente *sl*.

Del mismo modo que la información del paquete binario se obtendría de:

```
Package: sl
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: Correct you if you type ‘sl’ by mistake
Sl is a program that can display animations aimed to correct you
if you type ’sl’ by mistake.
SL stands for Steam Locomotive.
```

Listado D.3: Información del paquete binario *sl*.

D.0.4 Repositorio de paquetes Debian

Un repositorio de paquetes es un archivo ordenado donde se alamacenan todos los paquetes (sean binarios o no) y con una estructura definida y constanmenete actualizado.

El sources.list

En los sistemas Debian los repositorios vienen indicados en el archivo `/etc/apt/sources.list`, en este fichero cada linea representa a un repositorio y la rama en la que se encuentran los paquetes. En Debian estas ramas pueden ser `stable`, `testing` y `unstable`.

D.0.5 Campo architecture

En el fichero `debian/control` existe un campo que marca la arquitectura del paquete y para que arquitecturas puede ser construido. Este campo puede tener varios valores:

- Una única palabra indicando la arquitectura. Esta palabra, o mejor dicho esta arquitectura, debe estar entre las cadenas que se describen al ejecutar el comando `dpkg-architecture -L` en un terminal.
- Una arquitectura comodín identificando todas las arquitecturas a las que Debian da soporte. Para ello se utiliza la palabra `any`. `any` Es lo más común que se ve en los paquetes Debian.
- `all` indica que es un paquete de arquitectura independiente. Un ejemplo de este tipo de paquetes sería un *script* en *Bash*.

- source indica un paquete fuente.

En el fichero debian/control aparecerá una de las tres opciones, all, any o una lista de arquitecturas separadas por un espacio en blanco.

Especificando any en el campo architecture se indica que el paquete fuente no depende de ninguna arquitectura en particular, y por lo tanto debería compilarse bien en las otras. El paquete binario será específico para cualquier que sea la configuración de construcción de la máquina donde se realice.

Con all se indica que el paquete fuente construirá solamente paquetes independientes de arquitectura.

Si se especifica any all se indica que el paquete fuente no es dependiente de ninguna arquitectura en particular. El conjunto de paquetes binarios producidos incluirá al menos un paquete dependiente de la arquitectura y otro idependiente.

D.0.6 Sintaxis en los campos de relación

En un sistema de paquetes como el que se utiliza en Debian, existen relaciones entre los paquetes. Las relaciones se utilizar para indicar qué versión de un paquete se necesita o el nombre del mismo.

Las relaciones que se permiten en los paquetes Debian en cuanto a la versión son *estrictamente anterior*, *anterior o igual*, *exactamente igual*, *posterior o mayor* y *estrictamente mayor*. Estas relaciones se refieren tanto al paquete en cuestión como a la versión del mismo. Por ejemplo:

```
Package: mutt
Version: 1.3.17-1
Depends: libc6 (>= 2.2.1), exim | mail-transport-agent
```

Listado D.4: Relaciones entre las versiones de los paquetes.

Las relaciones pueden estar restringidas a un determinado tipo de arquitecturas, esto se indica entre corchetes «[]». Los corchetes contienen una lista **no vacía** de las arquitecturas (ref 11.1 debpolicy) separados por espacios en blanco. También pueden contener símbolos de exclamación.

Para las dependencias de construcción (build-depends, build-depends-indep, build-conflicts y build-conflict-indep), si la arquitectura del host no está incluida en la lista y no hay símbolos de exclamación, o está en la lista con un símbolo de exclamación que lo precede, el nombre del paquete y la versión asociada se ignorarán con el propósito de definir las relaciones.

Por ejemplo:

```
Source: glibc
Build-Depends-Indep: texinfo
Build-Depends: kernel-headers-2.2.10 [!hurd-i386],
hurd-dev [hurd-i386], gnumach-dev [hurd-i386]
```

Listado D.5: relaciones entre arquitecturas.

En este caso requiere `kernel-headers-2.2.10` en cualquier arquitectura menos *hurd-i386* y requiere `hurd-dev` y `gnumach-dev` solamente *hurd-i386*.

Para los campos de relaciones binarias, las restricciones en las arquitecturas solamente se admiten en el fichero `debian/control`. Cuando se genera el correspondiente paquete binario y su fichero `control`, la relación será omitida o incluida sin la restricción de arquitectura basada en la arquitectura del paquete binario. Esto quiere decir que las restricciones de arquitectura no deben ser usadas en los campos de relaciones ni en los paquetes independientes de arquitectura (architecture: all).

Por ejemplo:

```
Depends: foo [i386], bar [amd64]
```

Listado D.6: relaciones entre arquitecturas independientes.

Depende de `foo` cuando el paquete se construye para *i386* y depende de `bar` cuando se construye para *amd64*, y se omite para el resto de arquitecturas.

Si las restricciones de la arquitectura es tiene una |, esa alternativa se ignora por completo en arquitecturas que no cumplen la restricción. Por ejemplo:

```
Build-Depends: foo [!i386] | bar [!amd64]
```

Listado D.7: relaciones entre arquitecturas independientes.

Es equivalente a `foo` en *amd64*, a `bar` en i386, y a `foo|bar` para el resto de arquitecturas.

D.1 Dependencias en paquetes Debian

A continuación se explicarán las dependencias binarias de los paquetes Debian. Hay dependencias de varios tipos, para obtener información sobre todos los tipos de dependencias se aconseja consultar la política de Debian [JS12] en su capítulo destinado a tratar el tema de dependencias.

D.1.1 Dependencias binarias

Los paquetes en su fichero `control` tienen el campo para definir las relaciones con otros paquetes. Por ejemplo, pueden no ser instalados al mismo tiempo que otros paquetes o dependen de la presencia de otros.

Esto se hace utilizando campos `Depends`, `Pre-Depends`, `Recommends`, `Suggests`, `Enhances`, `Breaks` y `Conflicts`. Por el momento `Breaks` y `Conflicts` se verán más adelante.

Los siete campos se usan para declarar las dependencias que un paquete tiene sobre otro, salvo `Enhances` y `Breaks` aparecen en control del paquete binario.

El campo `Depends` tiene efecto cuando un paquete va a ser configurado. Esto consigue que un paquete no esté en el sistema en un estado «no configurado» mientras que sus dependencias no están satisfechas, y es posible que reemplace un paquete cuyas dependencias están satisfechas y que está correctamente instalado con una versión diferente cuyas dependencias no existen y no están satisfechas. cuando esto ocurre, el paquete se dejará sin configurar y no funcionará correctamente. Si es necesario se puede usar el campo `Pre-Depends` que tiene un efecto parcial cuando un paquete está siendo desempaquetado. Los otros tres campos se usan con *front-ends* como *apt-get*, *aptitude* o *dselect*.

Pueden darse casos de dependencias circulares, si esto sucede puede que un paquete no pueda confiar en sus dependencias ya que pueden estar configuradas antes que ellos mismos. Este tipo de dependencias se intentan evitar.

Aunque para un sistema que construye paquetes pueden resultar más interesantes los dos primeros campos, a continuación se explica el significado de los campos de dependencia para tener una visión global de su significado y su forma de utilizarlos:

- `Depends`: Declara una absoluta dependencia. Un paquete no se configurará a menos que todos los paquetes listados en este campo hayan sido correctamente configurados.
- `Recommends`: Declara una dependencia fuerte, pero no absoluta.
- `Suggests`: Es usado para decir que un paquete puede ser más útil que otro u otros. El uso de este campo indica que los paquetes de la lista están relacionados con este y tal vez pueden mejorar su funcionalidad, pero la instalación sin ellos se lleva a cabo sin problemas.
- `Enhances`: Es similar al anterior `Suggests` pero en la dirección contraria. Es usado para declarar que un paquete puede mejorar la funcionalidad de otro.
- `Pre-Depends`: Este campo es como `Depends` pero con la salvedad de que fuerza a `dpkg` a completar la instalación del paquete del que se declara la pre-dependencia:
  - Cuando un paquete tiene *Pre-Depends* y está a punto de ser desempaquetado, la pre-dependencia puede satisfacerse si el paquete del que depende está completamente configurada o incluso si el paquete/s del que depende está desempaquetado

o está en un estado de «medio-configurado». En este caso, ambos paquetes deben satisfacer la versión que aparece en el campo `Pre-Depends`

- Cuando está a punto de ser configurado, la pre-dependencia, será tratada como `Depends`. Se considerará que se satisface solamente si el paquete dependiente ha sido correctamente configurado. Sin embargo, a diferencia de `Depends` este caso no permite dependencias circulares. Si se encuentra la instalación se interrumpirá.

Si se quiere profundizar más aún en las dependencias entre paquete se recomienda la lectura del capítulo 7 de la Debian Policy [JS12].

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.

- I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

5. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

6. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

7. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English

version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

8. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

9. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

10. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

Bibliografía

[And12] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. Octubre, 2012.

[Aok12] Osamu Aoki. Debian Reference v2. http://www.debian.org/doc/manuals/debian-reference/, 2012.

[Bel13] Fabrice Bellard. Qemu Emulator User Documentation, 2013.

[Blo09] Eduar Bloch. maintaining Debian packages with Subversion. http://debianpaket.de/svn-buildpackage/index.html, 2009.

[CA12] Andreas Steil Christoph Arnold, Michel Rode. *KVM Best Practices*. ADMIN Magazine, edición 1, 2012.

[ClS09] Craig Small Christoph lameter y Bruce Sass. dh_make Manual page. http://man.he.net/man8/dh_make, Marzo 12, 2009.

[Com12] RS Components. *Raspberry Pi Start Guide*. RS Components, edición 1, 2012. url: http://d4c027c89b30561298bd-484902fe60e1615dc83faa972a248000.r12.cf3.rackcdn.com/supporting_materials/Raspberry%20Pi%20Start%20Guide.pdf.

[Cor13] Oracle Corporation. Oracle VM Virtual Box User Manual, 2013.

[Dav99] John E. Davis. *Jed Manual*, 1999. url: http://www.jedsoft.org/jed/doc/jedfuns.html.

[Deb] Debian. chroot. http://wiki.debian.org/chroot.

[Deb12] Debian. Wanna-build states: an explanation. http://www.debian.org/devel/buildd/wanna-build-states, Jun 7 2012.

[Deb14] Debian. *Debian package management*, 2014. url: https://www.debian.org/doc/manuals/debian-reference/ch02.en.html.

[Doc13] Cherokee Documentation. *Alvaro Lopez Ortega*. 2013. url: http://cherokee-project.com/doc.

[DV13] Óscar. Aceña D. Villa, F. Moya. *Computación Distribuida Heterogénena con ZeroC Ice*. Grupo Arco, 2013.

[Fal12] Luca Falavigna. Deb-o-matic Documentation, May 26, 2012.

[Fer10] Duncan Ferguson. Clusterssh documenation manpage, 2010.

[Fou13a] Free Software Foundation. GNU Hurd documentation. http://www.gnu.org/software/hurd/hurd.html, 2013.

[Fou13b] Python Foundation. *Python Language Reference (Version 3.3)*, 2013. url: http://docs.python.org/3.3/reference/.

[Fou14] Apache Software Foundation. *Apache HTTP Server Documentation*, 2014. url: http://httpd.apache.org/docs/2.4/.

[Gro13] The PostgreSQL Global Development Group. *PostgreSQL 9.3.1 Documentation*, 2013. url: http://www.postgresql.org/docs/9.3/interactive/index.html.

[Gue13] Guido Guenther. *git-buildpackage Manual*, 2013.

[Hes10] Joey Hess. Debhelper Manual page. http://man.he.net/man7/debhelper, Febrero, 2010.

[HS13] M. Henning y M. Spruiell. *Distributed Programming with Ice (Version 3.5.1)*. ZeroC Inc, 2013.

[HTT99] Hypertext Transform Protocol, HTTP/1.1. RFC 2616, 1999. url: http://www.ietf.org/rfc/rfc2616.txt.

[Inf13] Escuela Superior De Informática. Normativa del Proyecto Fin de Carrera. http://www.esi.uclm.es:8081/www/documentos/Normativas/NormativaPFC2007.pdf, 2013.

[Int11] Ecma International. *Javascript Language Specifitacion*, 2011. url: http://www.ecma-international.org/ecma-262/5.1/.

[Int13a] Intel. Intel Core i5 Specifications. http://ark.intel.com/products/65520, 2013.

[Int13b] Intel. Intel Core i7 Specifications. http://ark.intel.com/products/65719/Intel-Core-i7-3770-Processor-8M-Cache-up-to-3_90-GHz, 2013.

[JS12] Ian Jackson y Christian Schwarz. Debian Policy Manual, version 3.9.4.0, 2012-09-19. http://www.debian.org/doc/debian-policy/, Septiembre 2012.

[JS13] Ian Jackson y Christian Schwarz. Debian Developer's Reference 3.4.11. https://www.debian.org/doc/manuals/developers-reference/index.en.html, Septiembre 2013.

[Kni07] H Kniberg. *Scrum and XP from the trenches*. C4Media, edición 1ª, 2007.

[Koc10] Werner Koch. *Bash Reference Manual*. Free Software Foundation, Inc, edición 4.2 for bash version 4.2, 2010. url: http://www.gnupg.org/documentation/manuals/gnupg-devel/.

[Koc13] Werner Koch. *Using the GNU Privacy Guard*. Free Software Foundation, Inc, 2013. url: http://www.gnupg.org/documentation/manuals/gnupg-devel/.

[Lin12] Bernhard R. Link. Reprepro Manual. http://mirrorer.alioth.debian.org/reprepro.1.html, Jun 24, 2012.

[Lóp09] J. López. Redmine, gestión de proyectos Ruby on Rails. *Todo linux: la revista mensual para entusiastas de GNU/LINUX*, (104):21–24, 2009.

[MA14] Friedhelm Betz et al Mehdi Achour. PHP Manual. http://www.php.net/manual/en/, 2014.

[Mac05] M. Mackall. Mercurial v0. 1-a minimal scalable distributed SCM. *Linux-Kernel mailing list*, 2005.

[Mal99] Jordi Mallach. *Nano Command manual*, 1999. url: http://www.nano-editor.org/dist/v2.2/nano.html.

[Moo99] Bram Moolenaar. *Vi manual page*, 1999.

[NBM09] T. Dybå N. Brede More, T. Dungsyør. A teamwork model for understanding an agile team: A case study of a Scrum project. *Science Direct*, 2009.

[NT10] Christian Schwarz Niels Thykier, Richard Braakman. Lintian User's Manual. http://lintian.debian.org/manual/index.html, Febrero, 2010.

[OMG09] Object Management Group, 2009. url: http://www.omg.org/.

[OPF13] OpenFusion, 2013. url: http://www.prismtech.com/openfusion/technologies/corba.

[ORB09] ORBexpress, 2009. url: http://www.ois.com/Products/Communications-Middleware.html.

[PA13] Stefano Zacchiroli P. Abate, Roberto Cosmo. Dose-builddepcheck man page. http://www.edos-project.org, 12 2013.

[PAW13] Jussi Ronkainen Pekka Abrahamsson, Outi Salo y Juhani Warsta. Agile Software Development Methods: Review and analysis, 2013.

[Per04] Bruce Perens. Debian Social Contract and Debian Free Software guidelines. http://www.debian.org/social_contract, april 2004.

[Pla13] GNU Platform. GNU Compiler Collection. *Free and Open Source Software*, página 365, 2013.

[Pro13] Perl Project. *The Perl 5 language interpreter*, 2013. url: http://perldoc.perl.org/perl.html.

[Pro14] FreeBSD Project. *manualde FreeBSD*. FreeBSD Documentation Project, 2014.

[RA13] Josip Rodin y Osamu Aoki. Debian New Maintainers Guide. http://www.debian.org/doc/manuals/maint-guide/, Diciembre 2013.

[RFC07] OpenPGP Message Format, 2007. url: http://www.ietf.org/rfc/rfc4880.txt.

[RH11] Inc. Red Hat. The virtualization API. http://libvirt.org/index.html, Septiembre 2011.

[RMI09] Sun Microsystems Inc. Java Remote Method Invocation (Java RMI), 2009. url: http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html.

[sli14] Slice API Reference. http://doc.zeroc.com/display/Ice/Slice+API+Reference, 2014.

[SM88] R.M. Stallman y R. McGrath. *GNU make*, volume 2002. Free Software Foundation, Inc, 1988.

[SP14] R.M. Stallman y R.H. Pesch. *Debugging with GDB*. Free software foundation, 2014.

[Sta13] Richard Stallman. *GNU Emacs manual (version 24.3)*. Free Software Foundation, Inc, edición 17ª, 2013.

[Str13] B. Stroustrup. *C++ Programming Language*. Addison Wesley, edición 4ª, 2013.

[TAO09] TAO, 2009. url: http://www.cs.wustl.edu/~schmidt/TAO.html.

[Tea10] Debian Installation Team. *Debian GNU/Linux Installation Guide*. 2010. url: http://d-i.alioth.debian.org/tmp/en.amd64/apb.html.

[Uek07] Junichi Uekawa. Pbuilder User's Manual, May 27, 2007.

[Uek08] Junichi Uekawa. *Qemubuilder Manual*, 2008.

[Vil09] D. Villa. Infraestructura para la integración de redes de sensores y actuadores en entornos inteligentes, 2009.

[VMw13] Inc. VMware. VMware Debian 6 Documentation, 2013.

Este documento fue editado y tipografiado con $\LaTeX$
empleando la clase **arco-pfc** que se puede encontrar en:
`https://bitbucket.org/arco_group/arco-pfc`

[Respeta esta atribución al autor]