

Backporting and Packaging in Debian

7. Juni 2004

Note légale

Dieser Beitrag ist lizenziert unter der UVM Lizenz für Freie Inhalte.

Zusammenfassung

Die Eigenheiten des Software- und Releasemanagements des Debian-Projekts stellen vor allem Ein- und Umsteiger immer wieder vor große Herausforderungen. Dieser Vortrag versucht, Klarheit über die Funktionsweise des Software- und Releasemanagements in Debian vermitteln und die sich aus der Adaption dieser Methoden ergebenden Vorteile aufzuzeigen.

Nach einer Einführung in die Konventionen und die Funktionsweise des Software- und Releasemanagements in Debian wird die Vorgehensweise und die dabei verwendeten Werkzeuge bei der Paketierung anhand eines Backports exemplarisch dargestellt. Anschließend werden einige der aus einer Adaption dieser Methodik ergebenden Vorteile vorgestellt. Mit einer kurzen Beschreibung der Struktur und Funktionsweise der Debian Repositories endet der Vortrag.

Der Vortrag basiert zu weiten Teilen auf meinem Beitrag zum DebianDay @ LinuxTag 2003 (siehe auch [Link](#)), geht aber spezifischer auf die Paketierung und die sich daraus ergebenden organisatorischen Vorteile und Risiken ein.

1 Introduction

This paper is about backporting and packaging in a Debian GNU/Linux environment. In order to provide a proper understanding, I'll explain the basic underlying general concepts. For those new to Debian a short overview of the Debian Project is given as well.

At first, the general concepts of software distribution (SD) will be explained. After a short introduction into the structure and goals of the Debian Project, structure and mode of operation of Debian Packages are explained as well as backports and why they may be useful.

1.1 Software Distribution: An Overview

Commonly, the term "software distribution" refers to

- Methods of distributing data to one or more machines

This includes all of the infrastructure needed to transfer the data being distributed, for example removable media, networking hardware and software, as well as the necessary storage for such data as distribution points, repositories, and databases.

- Methods of distribution control

The more complex part – build environments, testing environments, versioning, archiving, auditing, monitoring and all the respective infrastructures.

Keep in mind that SD does not only focus on distributing applications to client computers. It includes delivery of any data from one or more locations to one or more different locations whether the data are application binaries or perhaps stock data for a database needed by sales personnel.

1.2 Pros and Cons of SD

The usual "Why should I?" argument. Of course, it always depends on your specific environment, but the larger or the more distributed your environment, organizational structure and physical locations are, the more you may benefit from SD.

As soon as you're using any software, you need to maintain it. In addition to other maintenance tasks, you may have to apply patches or fixes to your software, independent of the underlying cause. You also may have to manage more complex major software upgrades or changes, including changes in file or data formats.

And as if those tasks weren't tough enough already, the larger and the more complex your infrastructure grows, the more complex and dangerous it will be to accomplish them. SD may help you, although nobody would seriously guarantee your success or failure.

The main advantages of SD are:

- **Manageability:** You gain significant control over the data that is distributed
- **Accountability:** You are able to assign specific data depending on various information (e.g., user names, DNS records)
- **Reliability:** The reliability of your infrastructure may increase because of a less complex distribution and thus fewer errors and less misconfiguration may occur

Sounds nice, doesn't it? But there are some major disadvantages in SD:

- **Discipline:** As soon as you've implemented SD, you are bound to it, as any exception may be fatal
- **Complexity:** Some SD frameworks are quite complex and they have to be maintained as well as their infrastructure
- **Processes:** Additional business processes may be needed for proper operations of an SD framework

You see, the pros and cons just about balance each other out. This means that only you can evaluate the advantages and disadvantages for your specific environment.

1.3 SD - How it Works

Now as we know what SD means and where we have to look for pros and cons, I'd like to explain the common concepts in SD. As I'm talking about packaging, I'll stick on a software package being built and distributed from now on. But keep in mind, that SD covers not only software packages but any data as well.

Usually a need for specific data on specific locations builds the very beginning in SD. The data in question shall be any software which shall be delivered to some but not all computers in an organizational infrastructure spread over several physical location all over the world. This example may be somewhat unusual but it best describes one of the main motivations to use SD mechanisms: Imagine you're travelling all through your organizations locations solely for the task of installing an application, it is much more efficient to let specialized automatons do the work which is what SD does.

After having determined all participating parties such as networks, computers and the like, the software in question needs to be packaged. That means the binaries, necessary configuration data and maybe some metadata need to be assembled into a deployable package.

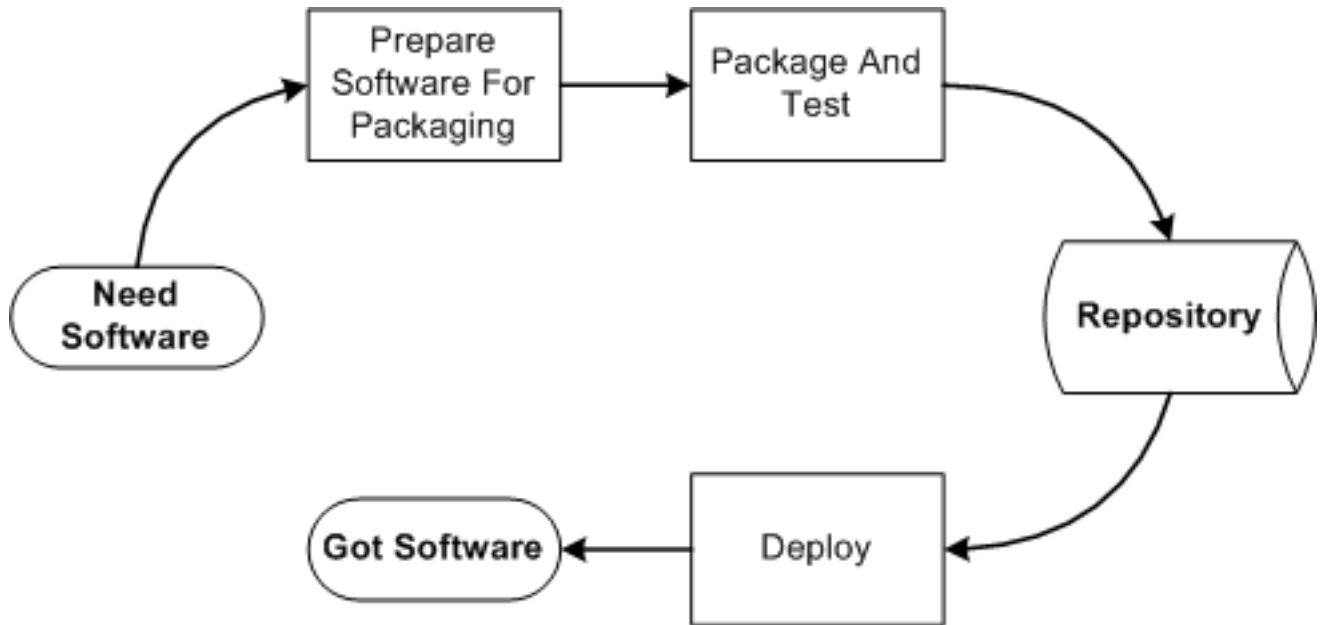
Of course those packages have to pass thorough tests both locally and remote to ensure proper operations. Local test can only reveal errors that occur at the specific workstation, they don't say anything about errors occurring during deployment or at installation on different computers.

After passing the tests the packages need to be stored somewhere before being distributed. Usually a so called Repository, a specialized structured storage, is used for this purpose. Reaching this point, preparation has ended and distribution itself may begin.

There are quite a lot of different methods delivering packages to their targets but all of them rely on two main concepts which will be explained later. At least the packages will be delivered using any method. As mentioned at the beginning, only specific machines at specific locations shall receive the package.

At the end of the process described the software needs to be installed and configuration informations need to be written. The installation usually consists of copying or moving files from the package to their destinations,

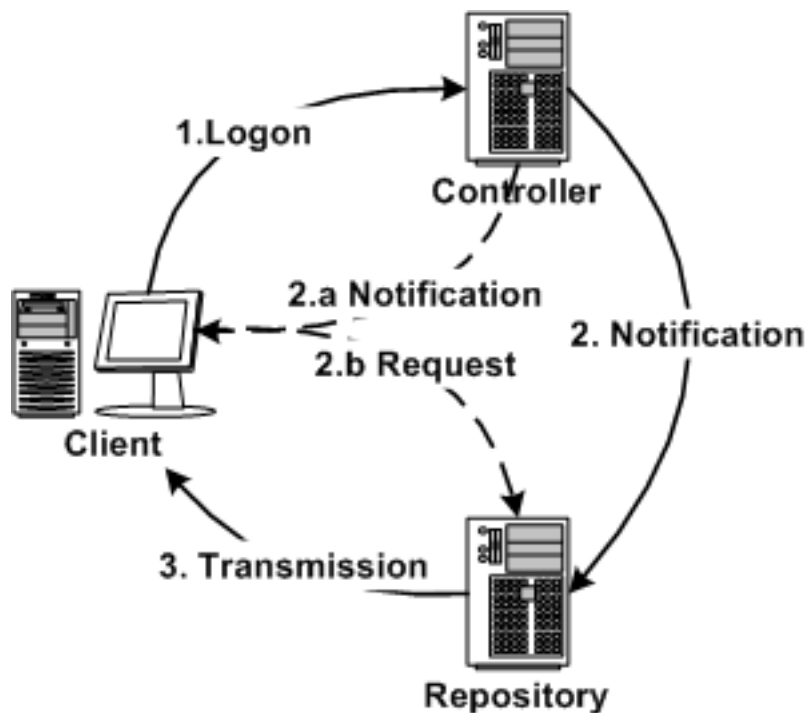
but writing configuration information is a much more complex task. The SD Framework needs to know if there's a corresponding configuration present on the system or not, where to find or write the configuration data to and so on. Package and configuration handling separate the better SD Frameworks from the not-so-good.



As said above, there are two main concepts of delivering the built packages, the so called "Push Distribution" and the so called "Pull Distribution". Both of them have not much in common and both of them have their advantages and disadvantages which shall not be discussed here because such a discussion would leave the scope of the document. Repository and Information-DB respectively Controller on the following figures are separated by purpose since there is no need to provide them at the same physical or logical location although it is often done.

1.3.1 The Push Distribution

The Push Distribution, sometimes referred to as "Active Distribution", is a very powerful concept since there is no need of any user interaction. It is initiated by a specific client side process, usually a so called agent running as a system daemon process. This concept is widely used by commercial, closed source System Management Frameworks.



The agent communicates with a corresponding centralized Distribution Manager who knows of all distributions scheduled for the specific client. This method is widely used by commercial, closed source systems management frameworks.

At startup the agent logs in to the Distribution Manager or Controller (1). This usually happens at system boot time so only informations concerning this machine can be exchanged.

The Controller then checks his Information Database for any new schedules concerning this client. If there are any new schedules listed, the controller sends a corresponding notification either to the Repository (2) or to the client (2.a).

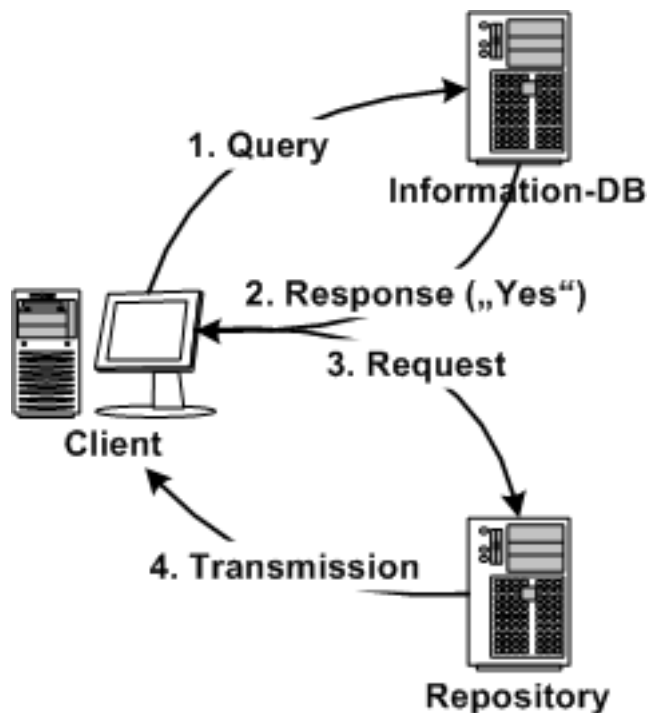
In case the repository got notified about the client's schedules it checks for the package(s) to deliver and, if present, starts transmission according to the schedule (3). Else the client itself contacts the repository (2.b) and requests delivery of the scheduled packages.

As soon as delivery ended the Agent starts and controls the installation including the configuration of the software delivered (if needed).

The same process happens when a user logs in to the client computer. This time the Agent transmits user specific informations, usually UID and GID, and the Controller checks for any schedules specific to that user.

1.3.2 The Pull Distribution

The passive or pull distribution is initiated at client side, queries the information database for any actions waiting to be performed and performs those actions demanded in the response to the query. This is the most common distribution method in GNU or OpenSource environments like Linux and *BSD.



The Pull Distribution usually is initiated by the local user who queries the Information-DB for available updates or new packages (1). The Information-DB then answers accordingly (2) and the Client has to determine if there are any updates available for local packages. Often the decision to update or install or not to is taken by the local user currently logged in.

After having decided what packages of those available shall be installed or updated, the client sends a request to the Repository (3) and the Repository delivers the packages to the client (4). NB: Steps 3 and 4 are usually handled by FTP.

After reception of the packages, extraction, installation and configuration are handled by the client, similar to the Push Distribution.

1.3.3 Controllers

This section is about the controlling instances in the process of SD, called *Controller* for the Push Distribution model or *Information-DB* for the Pull Distribution model. For the convenience both of them will be called Controller in this section.

The main task of a Controller is to decide who gets which package, provide these informations and keep them current. In the Push Distribution model this data varies from client to client, in the Pull Distribution model the data is the same for all clients. Apart from that the information managed by both of them is nearly identical:

- What: Package, Configuration, Data, etc.
- Which: Version, Date, etc.
- Where: Storage Path in Repository
- Who: Access Control Informations

This information is needed by the Pull Distribution client respectively by the Push Distribution Controller to determine whether a client is allowed to get what he requests and which Packages the Repository provides and where they can be found.

The Controller (both Pull Distribution's Information-DB and Push Distribution's Controller) itself receives its information by the Developers uploading Packages to the Repository and feeding the Controller with corresponding informations.

1.3.4 Repositories

There's no magic about Repositories. A Repository is a structured storage. In the context of Software Distribution a Repository keeps packages to be delivered in a machine-readable format. The main task of a Repository is to provide a solid and reliable infrastructure to store and deliver packages. Repositories are passive components in a Software Distribution Framework. That means they don't do anything by their own initiative but react on specific requests.

Most Repositories consist of a traditional File Server with one or more network services to provide access to the Repository. The services provided usually are FTP, NFS or HTTP.

When a client requests a Package, he queries the Repository if the Package in question is available. In reaction to the query the Repository sends an appropriate answer to the client. If the requested Package is available at the Repository the client connects to the Repository using any of the offered network services and retrieves the Package.

On the other side a Developer having built (and of course tested) a new Package simply connects to the Repository and uploads his new Package. Afterwards the Developer has to provide the Controller with certain information concerning his new Package. Without it the Repository couldn't know the Package and therefore couldn't provide any information about it.

1.4 Advanced Topics

There are some advanced topics and ideas concerning Software Distribution which I would like to spotlight.

Some of the commercial Software Distribution Frameworks store configuration and access control information in directories. I currently know no GNU or other Open Source Project who does so even though it would be an interesting feature especially in mid-ranged or large infrastructures.

Similar has to be said about the integration of Software Distribution and RDBMS. Even though all SD Frameworks use a kind of database, only some use a regular RDBMS such as PostgreSQL.

The main advantages of integration SD in LDAP-Directories and/or RDBMS are a higher scalability, flexibility and even portability. As I already said, there is no need to run Repository and Controller/Information-DB at the same physical location as these components may easily be distributed all across the networking infrastructure without any impacts on the functionality. Indeed the impacts on performance and reliability may be noticeable since a failure tolerant concept may easily implemented.

Another large and important aspect is security. Especially in the Open Source community everyone simply trusts the distributors and their distribution points. Whenever you do an *apt-get update* && *apt-get upgrade* you trust foreign infrastructure such as DNS servers, web ad file servers. You actually have neither evidence nor a reliable but easy to use mechanism to verify whether your connection reaches the host you intended it to or whether the packages you download are those they claim to be. Of course the distributors and vendors do what they can do to keep this trust and its foundation. But that's it or better that seems to be all. Especially in Debian there are some Developers trying to build methods and software allowing the user to verify origin and integrity of the Packages they retrieve. *apt-secure* is one example and there may be some others which I don't know but none of them has reached a level of simplicity that allows a user or an admin to simply use *apt-get update -verify* && *apt-get upgrade -verify* or similar.

There's still a lot work to be done as ideas are many ... ;)

2 About Debian

In August 1993 Ian Murdock announced a new Operating System Distribution called "Debian" supposed to be more user-friendly, more flexible and easier in installation as well as in administration and that "will contain the most up-to-date of everything." (Ian Murdock in his initial announce on c.o.l.d). OK, the Project may have missed the latter goal in the stable Distribution as of now ;)

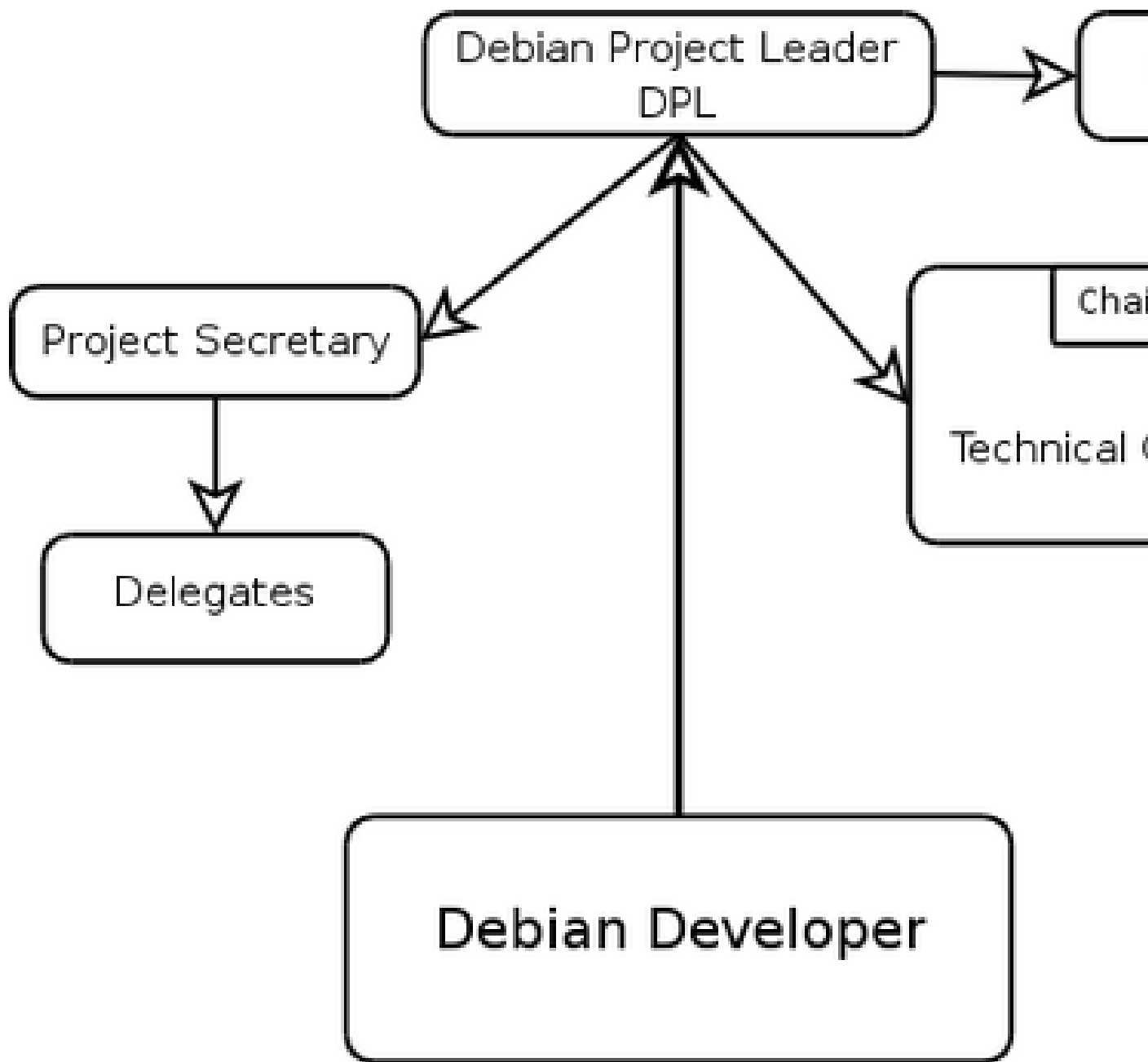
Frank Ronneburg offers a very good overview (in german) of the history of the Debian Project at <http://www.debiananwenderhandbuch.de/wasistdebiangu.html#debiangeschichte> . Also available in other languages is the official history at <http://www.debian.org/doc/manuals/project-history/> <http://www.debian.org/doc/manuals/project-history/> .

2.1 The Project Structure

I would like to point out here that I would like this distribution to develop in the same way as much of the rest of Linux has developed. In other words, I want everyone to **contribute** to this effort and not simply use something that one man or team has put together. This distribution will be improved by the Linux community as a whole, and I will simply serve as the coordinator of the effort.”(Ian Murdock, 27.08.1993 in comp.os.linux.development)

This cited, there’s not much left to be said about the Debian Project. The currently more than 900 Debian Developers building the Distribution form the core of the Project. All formal or informal roles and structures only exist in a (well working) attempt to coordinate and concentrate the efforts made to ensure the high level of quality, security and diversity of the Debian Distributions.

The Debian Constitution defines roles, responsibilities and processes of finding decisions in the Debian Project. The Debian Project Leader represents the project to the outside and manages it inside, the Technical Committee who makes final decisions on technical questions as necessary, the Project Secretary who’s actually doing most of the administrative work to be done in the project such as conducting votes or interpreting/disputing the Debian Constitution. And there are many DDs voluntarily maintaining other parts such as support and management of the infrastructure, distribution management and so on.



A more comprehensive list of roles and responsibilities can be found at <http://www.debian.org/intro/organization> <<http://www.debian.org/intro/organization>>

2.1.1 Debian Developers

As of April 2nd, 2004 there are 908 approved Debian Developers working on the Debian Distributions (<http://lists.debian.org/debian-devel-announce/2004/debian-devel-announce-200404/msg00004.html> <<http://lists.debian.org/debian-devel-announce/2004/debian-devel-announce-200404/msg00004.html>>). They are the ones who do the work to be done thus they are the ones who make the decisions.

The process of making decisions is described in detail in the Debian Constitution. In short, every Developer may demand a decision on any aspect of the Debian Project. He simply has to state his demand in a written form, the Resolution, which then has to be discussed. After a minimum discussion period all Debian Developers are called to vote on the Resolution.

Taking into account that more than 900 people are allowed to demand Resolutions, one could guess that there are so many Resolutions waiting to get voted that there could be any time left either for further

development or for voting. In fact, as every Debian Developer is responsible for the Packages he maintains and as he may make any decision concerning his Packages there are only very few Resolutions being discussed or voted.

2.1.2 Officers

The Debian Project Leader (DPL) is the official representative of the Debian Project. Internally, the DPL has to manage the Project and define its vision. He is elected once a year for a one-year period. The current DPL is Martin Michlmayr.

The Debian Technical Committee consists of at least four up to eight members including one chairman. They're responsible for the technical development of the Debian Distribution. They may make final decisions as a last resort when there is now chance for a broad consense. The members of the Technical Committee are appointed by the DPL after being recommended by the Technical Committee. Membership in the Technical Committee is not limited by time but members may be removed or replaced. The Chairman of the Technical Committee is elected by its members. The current Committee members are Raul Miller, Manoj Srivasta, Guy Maor, Bdale Garbee, Wichert Akkerman and Jason Gunthorpe. The current Chairman is Ian Jackson.

The Debian Project Secretary is responsible for taking votes amongst the Developers, determining identity and number of the Debian Developers and adjudicating and disputing the Debian Constitution. The Debian Project Secretary is appointed by the DPL and the current Debian Project Secretary for a one-year period. The current Debian Project Secretary is Manoj Srivasta.

2.2 Goals and Consequences

Structure and goals of the Debian Project cause heavy impact on the way work on the Distribution is done.

2.2.1 Goals of the Debian Project

The main objectives of the Debian Project, as described by Ian Murdock in 1993, 1994 and later and as defined in the Debian Social Contract may be summarized as follows:

- Open Development

Inspired by the Linux Kernel development, Ian Murdock founded the Debian Project in an attempt to demonstrate the ability of an open community to develop a high-quality distribution. Similar to Linus Thorvalds he realized the necessity to establish consistent cooperation management to ensure continuous development on a high-quality level.

- Free Software

The Debian Project and his Developers always have been engaged no only in the Debian Project but in the Open Source Community. Foundations of important Institutions were initialized or supported by Debian Developers, all software developed for "proprietary" purposes of the Debian Distribution has been published under a license considered as "free" by the Debian Project (mostly one of the GNU Licenses) and all amendments on software used in the Debian Distribution have been fed back to the original authors.

- Standards Compliance

For the purpose of open development, free software and integration with the Open Source community, the Debian Project also tried to follow open and accepted standards or if such standards hadn't been defined yet to contribute in their development. For example the Debian Project contributed to the former FSSTDN (now FHS).

- "Ease Of Use"

Usability was one of the core aspects, Ian Murdock wanted to realize with the Debian Distribution. The powerful and mature package management software "dpkg" was released just under one year after the initial announcement of Debian GNU/Linux. Since, much has changed but usability and manageability still are one of the major "features" of the Debian Distributions.

2.2.2 Impacts On The Distribution

Having summarized the main objectives, we'll have a look on the consequences those objectives drive on the distribution.

The constantly growing number of Debian Developers in conjunction with a huge number of packages being maintained slows down the release cycle. The more packages have to be maintained and QA'ed and the more Developers have to make decisions on their packages the harder it is not to loose track of the next release.

Additionally since any bugs not originating by the Debian Package are fed back to upstream"(the original author), the time spent waiting for a fix by upstream (or having upstream merged the patch supplied) often interferes the release cycle too. And of course the personal capabilities of the responsible maintainer have an impact - don't forget that they are doing their work voluntarily, one maintainer may have time enough to reproduce and fix any bug reported within a few days, another one may not.

Continuity is another keyword. The stable"Debian Distributions are meant to be production-ready therefore no new features are added to a stable distribution. Only security patches (through security.debian.org) and bug fixes are applied onto the stable distributions.

What may look like an antagonism in times of rapid development and constantly growing number of features is indeed a real benefit in production environments. As production needs usually evolve slowly (more slowly than most commercial software manufacturers would like) there's in fact no need to always have the newest (and untested, unaudited, ...) features available but there's always a huge need of reliability and continuity especially in organizational environments whether they are non-profit or for-profit. Basically it's the same philosophy as in "Never touch a running system!"

Quality goals such as usability, standards compliance and so on are strictly followed. Every Package incoming is checked against the Debian Policy which defines the standards used in the Debian Project and altered accordingly if it doesn't comply with it. The Debian Policy is of high value for every Debian user either end-user or administrative user. For example it defines where to expect configuration files and how to handle them, it describes the structure of Debian Packages and so on. In addition there are many other documents describing the various aspects of the Debian Distributions such as the Installation Guide, the Debian Reference or the Debian Developers Reference.

2.3 Structure Of The Debian Distributions

The Debian Distributions have a sophisticated structure which I explain in this section. With regard to build your own or customize any existing Package, knowledge of the structure is essential unless you can cope with a messed-up package database.

2.3.1 Releases

The Debian project permanently maintains three so called Distributions, sometimes there appear one or two additional Distributions for a short period of time:

Tabelle 1: Overview of Debian Distributions

Distribution	Name	Description
stable	woody	Current Production Release
testing	sarge	Upcoming Production Release
unstable	sid	Preparation Stage for Upcoming testing
experimental	N/A	Temporary Distribution
frozen	N/A	Temporary Distribution

The stable"Distribution is the current production release, codenamed "woody", the "testing"Distribution will be the next stable"Distribution and is codenamed "sarge". sid is a bit special since this Distribution never changes its codename and therefore usually is referred to as sidänd not as unstable".

sid is a preparation stage for "testing"where new software versions and the corresponding Packages are being tested. After matching certain criteria, the Package moves to "testingänd replaces the old version there.

Software and Packages in this Distribution are considered faulty *per se* hence the name "unstable". Sometimes new Software enters "experimental" before entering "unstable". This is to test the Package(s) as well as the Software roughly for general functionality.

"testing" is a preparation stage, Packages and Software included in this Distribution are tested and found bugs are fixed either by the original author of the Software, the so called Upstream, or by the Package Maintainer as long as it's a bug in the Package and not in the Software. As soon as all or nearly all bugs are fixed this Distribution enters "frozen", a stage where no code changes are allowed except for critical bugfixes and where the entire Distribution gets thoroughly tested.

"stable" is the previous "testing" which passed "frozen". It consists of a fixed set of Packages, no new versions of a Software enter this Distribution, only security updates or extreme-critical patches may make their way into "stable" as they are backported to the software version used in "stable". Therefore Software versioning doesn't mean much in Debian.

2.3.2 Sections

The Packages in a Distribution are grouped into so called Sections. Each Section has its meaning and the Packages are assorted following certain criteria:

Tabelle 2: Overview of Sections

Name	
main	Packages in this Section must comply with the Debian Free Software Guidelines and must not depend on non-free Packages.
contrib	Packages in this Section must comply with the Debian Free Software Guidelines and must not depend on non-free Packages.
non-free	Packages that do not comply with the Debian Free Software Guidelines or are encumbered by patents or other restrictions.
non-US	Packages are considered non-free but contain (cryptographic) code, use patented algorithms with a restrictive license.

As it is important to choose a correct "location" for the Packages you're going to build, knowledge of the Debian "sorting rules" may help you in making your decision.

The Packages in "main" build something like the "core" Debian GNU/Linux Distribution as they meet all of the ambitious criteria the Project has defined in its Policy and the Debian Free Software Guidelines (DFSG). They are "free" as they may be freely redistributed, have their source code available to anyone without any obstacles, allow modifications and derivations and so on. So if the Software (or data) you maybe intend to package is published under one of the licenses considered as "free" by the Debian Project and does not require any Package outside "main", neither to build nor to execute, this Section may be the right one.

Packages in "contrib" essentially meet the same criteria as Packages in "main" do with the exception that they may depend on a Package outside of "main" or on Software or Packages outside the official Debian archive. Thus usually free software that depends in any way on "non-free" or "contrib" such as add-ons, wrapping Packages or similar go into "contrib". If you're planning to build a Debian Package of your Software, check if it depends either at compile-time or at run-time on any Packages outside of "main" as well as if the License you chose is considered "free" according to the DFSG.

"non-free" contains Software that are considered not "free" as defined in the DFSG. That's quite easy, if a Software doesn't fit into "main", "contrib" or "non-US" it probably has to be put into "non-free".

"non-US" is special. Primarily Software whose distribution or export from inside the US is restricted or prohibited for any reason but whose distribution from outside the US is not may go into "non-US". This mostly applies to cryptographic software but there are some other Packages that went into "non-US" for non-cryptographic reasons. Additionally, "non-US" doesn't contain any Packages but contains the Sections "main", "contrib" and "non-free". In there Packages can be found that can't go into "main", "contrib" or "non-free" because of the restrictions mentioned. Check carefully, if the Software you're going to package uses any algorithms that are patented in the US (but not in your home country) or cryptographic algorithms or Software whose export from inside the US is prohibited. If so, your Package has to go into "non-US/(main|contrib|non-free)".

2.3.3 Subsections

For the purpose of simplified Package handling, Packages in the Sections "main", "contrib" and "non-free" are grouped into further Subsections such as "admin" for administrative utilities, "devel" for development applications, libraries, etc. or "web" for www-related stuff.

Currently they are:

Tabelle 3: Overview of Subsections

Name	Description
admin	Administrative utilities
base	The Debian base system
comm	Programs for faxmodems and other communication devices
contrib	Programs which depend on software not in Debian
devel	Utilities and programs for software development
doc	Documentation and specialized programs for viewing documentation
editors	Text editors and word processors
electronics	Programs for working with circuits and electronics
embedded (not in woody)	Software for embedded systems
games	Games, toys and fun programs
gnome (not in woody)	GNOME applications, utilities, libraries
graphics	Utilities to create, view and edit graphics files
hamradio	Software for hamradio operators
interpreters	Interpreters for interpreted languages
kde (not in woody)	KDE applications, utilities, libraries
libs	Collection of software routines
libdevel (not in woody)	Libraries, development files, headers, etc.
mail	Programs to write, send and route email messages
math	Numeric analysis and other mathematics-related software
misc	Miscellaneous software
net	Programs to connect to and provide various services
news	Usenet client and servers
non-US	Programs stored outside US due to export restrictions
non-free	Programs which are not free software
oldlibs	Obsolete libraries
otherosfs	Emulators and software to read foreign filesystems
perl (not in woody)	Perl and related software
python (not in woody)	Python and related software
science	Software for scientific work
shells	Command shells and alternative console environments
sound	Utilities to play and record sound
tex	The TeX typesetting system
text	Text processing utilities
utils	Various system utilities
web	Web browsers, servers, proxies and other tools
x11	The X window system and related software

2.3.4 Priorities

In addition to the Sections the Debian Software Management knows so called Priorities that name the importance of the Package in question:

As assigning a Priority to a Package influences the entire Software Management System it is important to know about the Priorities in order to make the correct decision when building your own Packages.

Tabelle 4: Overview of Priority Levels

Name	Description
required	Highest Priority, needed to build a working base system
important	Commonly expected software
standard	Standard software, text-only system
optional	Any optional components, e.g. X, TeX, etc.
extra	Lowest Priority

The Priorities determine how important a specific Package is in order to have a proper functioning system. This influences decisions and suggestions made by either dpkg or apt concerning Packages being installed or removed.

Packages marked "required" are essential. Removal of any required Package most probably makes your system unusable. The core binary Packages are marked "required".

"important" Packages are those expected on any UNIX or UNIX-like system. If the expectation is that an experienced Unix person who found it missing would say "What on earth is going on, where is foo?", it must be an 'important' package." (Debian Policy, Chap. 2.5). Additionally, Packages that are needed in order to achieve a usable and well running system are also marked "important". Note: Software like Emacs, the X Windows System or similar are *not* marked "important"!

The priority "standard" is applied to packages needed to "provide a reasonably small but not too limited character-mode system" (Debian Policy, Chap. 2.5).

"optional" Packages are those neither essential to the base system nor commonly expected on an UNIX system. Most of the Packages are marked as "optional". There's one important rule for Packages being marked as "optional": They shall not conflict with each other.

The "extra" priority determines Packages that conflict with Packages marked as one of the higher priorities, have special requirements or may only be useful in a specific environment.

Determining the Priority of a Package is quite easy because there are only few rules to follow (indeed, usually every Backport is marked as "extra" in order not to tangle the Debian Software Management):

1. Packages must not depend on packages with lower priorities, excluding build dependencies
2. Packages marked as "optional" should not conflict with each other.

That's all. Of course the criteria described above should be met too.

2.4 The Release Cycle

Finally I'd like to explain the Debian release process a little to offer a better understanding why some things happen and others don't.

After a new, not yet packaged, version of a software has been released by upstream (its author or project) the responsible Maintainer needs to get, build and test it first. Then he will build new Debian Packages (either source, binary or both) and upload them into an incoming queue where it eventually will be accepted by the Debian Archive Maintainers (most of the latter is automated but that doesn't matter).

The new packages enter either "experimental" or "unstable" first, depending on whether it is a completely new software or major changes have been implemented with the new release of the software (such go into "experimental") or whether it is an update or minor version change (such go into "unstable"). After being accepted into unstable, the new packages need to meet certain criteria concerning number of severe bugs, availability for several platforms and so on. An exact list is available at the "testing" pages (<http://www.debian.org/devel/testing> <<http://www.debian.org/devel/testing>>).

As explained earlier, packages in stable are more or less static. Security fixes are applied by the Debian Security Team and only fixes of major bugs really make their way into stable. There are some other ways to implement minor changes into stable but these aren't of interest now. In fact, those versions that were in testing the day the freeze was declared remain untouched for the rest of the "life" of the specific "stable" Distribution. That's why the "visible" version numbers often seem to be outdated - no new feature releases but security related fixes have been incorporated since the release.

The three different Debian Distributions often raise questions about the distribution to use. Feature sets versus stability, security updates versus versioning - it's hard to decide for someone not familiar with the way software makes through the Debian Distributions.

That is to say that there's no official support, especially no official security support for any other Debian Distribution than "stable". And the reason is quite simple.

As I said, any software entering Debian goes into unstable first. Software there is considered unstable and buggy per se but by nature the most current security and bug fixes by upstream are already implemented. After a while, these packages may enter testing where they stay until the next release when testing becomes stable. As no new feature releases (usually, there are exceptions) make their way into stable, security related modifications always do. In addition the Debian Security Team as well as the Debian Developers constantly monitor the corresponding sources of information - mailing lists, forums, groups and so on - to insure appropriate reaction as prompt as possible.

Knowing all this, it should be obvious that the only Debian Distribution used in production environments should be "stable" since it is the only one with guaranteed security support. The fact that many of the software versions seem to be outdated is irritating because at least security fixes are always implemented into the packages - unless you never made an "apt-get update && apt-get upgrade".

3 Using Debian GNU/Linux

At this point I'd like to introduce a name change: I'll further use the term Software Management because it fits better to the capabilities the Debian software offers. Software Management is not just a buzzword, it includes SD as described above as well as Configuration Management, Release Management and some other topics. The technical concepts of controlled data distribution remain the same only the scope is enlarged.

3.1 Debian Software Management

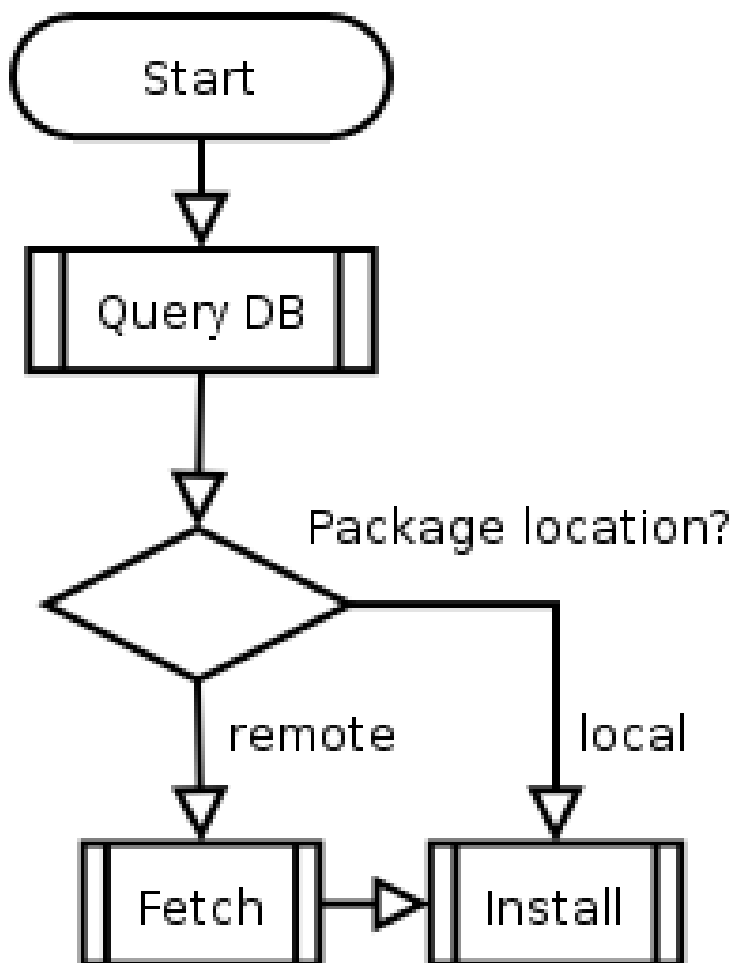
Debian implements the concept of Pull Distribution while keeping a consolidated local replica of the Information-DB in use. This means, all SD tasks are initiated at client side and up-to-date information about available packages or updates have to be requested from a remote Information-DB. Information about the packages being installed, upgraded or removed are kept locally as well as informations where to find them.

To update the local Information-DB the client contacts the repositories known to him (by configuration) and requests their databases - the file Packages(.gz) located in dists/<release>/<section>/binary-<architecture>/ for the Binary Packages or the file Sources(.gz) located in dists/<release>/<section>/source/ for the Source Packages. These files are merged in /var/lib/dpkg/available, the local Information-DB. Note, that other tools may use different and/or additional Information-DBs, e.g. apt maintains its own as well as aptitude does. I will refer to the databases maintained by dpkg as the local Information-DB.

In order to install a new package the local Information-DB is queried for its status (installed, not installed, configured, etc.), its dependencies and the repository where it can be found. If the package is neither installed nor its configuration is marked as broken the installation may continue. If it is installed and its configuration is marked broken the reconfiguration process is triggered. This process consist of the configuration steps described later but only the configuration files are generated. If there's no other version of the package installed the package will either be installed immediately if it is locally available or first fetched from the repository and then installed. The latter happens when any of the available front-ends as aptitude, dselect, synaptic, etc. are used because they provide the ability to automatically fetch needed packages from remote repositories whereas dpkg was developed for local only use.

One aspect the Software Management has to consider are dependencies. If there are any other packages this package depends on (requires them either to run or to build), the management software has to install the missing packages in order to ensure that all dependencies are properly met. Those dependencies are called "backward dependencies". The opposite is named "forward dependencies packages that depend on the package in question. Backward dependencies are important during package removal.

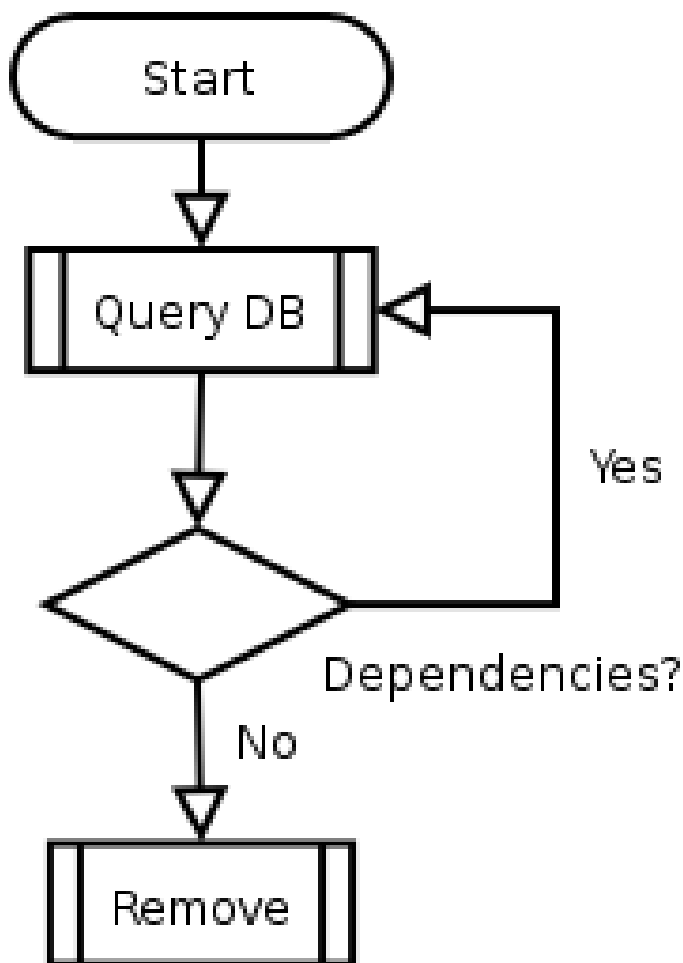
All of the available tools in Debian offer some automation to a certain degree: dpkg simply complains about the dependencies not met and refuses to install the package, dselect offers the opportunity to add the missing packages, others act different. But none will (by default) install a package whose dependencies are not satisfied to a minimum. Dependencies are not considered at package removal or reconfiguration.



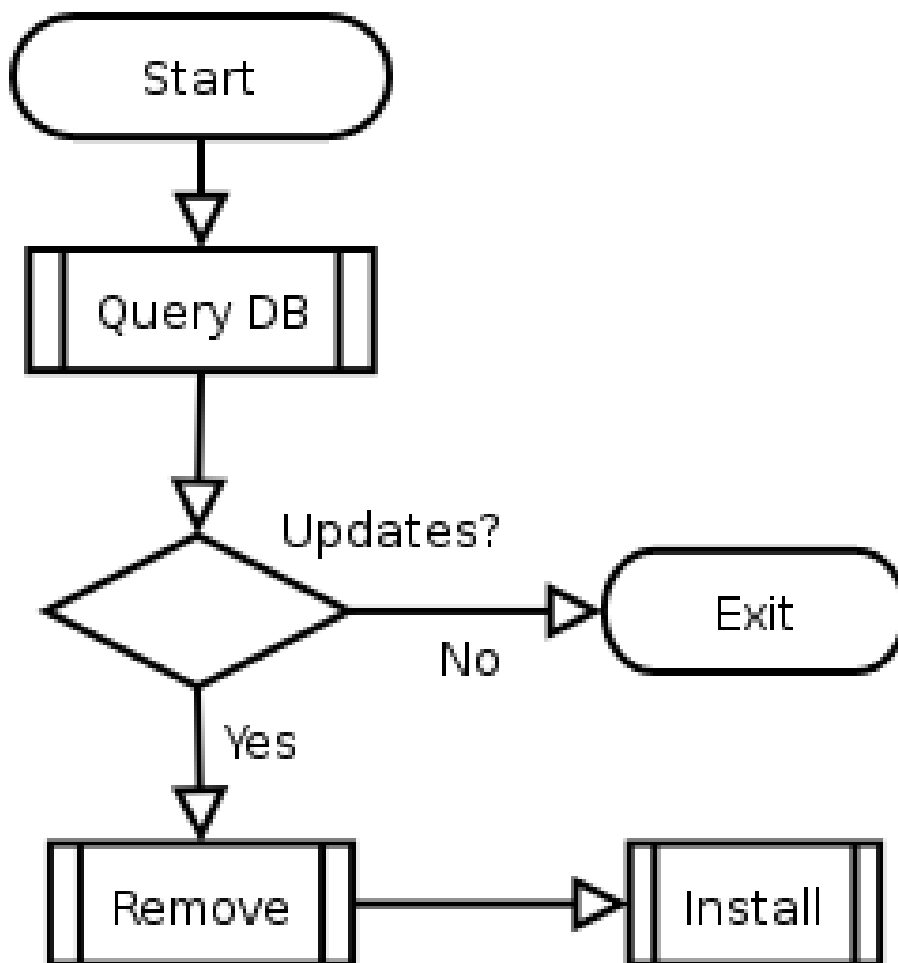
Removal of packages is a little more complex because of the backward dependencies that have taken into account. The Debian software management tools offer the same range of automation here as well as they do for package installation. Most of them remove all packages having a strong dependency to the package being removed. This behavior can be overridden although this is highly deprecated because it breaks the dependencies and may leave the system in an unusable state.

On request of removal the local database is queried for other packages possibly depending on the package in question. If there are some this subprocess runs into recursion until all dependencies have been resolved. If there are no dependencies to consider the package will be removed, else the dependant packages will be removed before the original package.

There's one peculiarity in Debian: You can "remove" or "purge" packages. If you "remove" a package, its binaries will be deleted but all configuration information, configuration files and debconf database entries, for example, remain. If you "purge" a package, its binaries and configuration information will be deleted, sometimes, in case of databases, for example, even data you entered will be deleted too. Knowledge about this is essential and may often prevent confusion and frustration.



Upgrading a package is, generally speaking, a combination of removal and installation of a package. After querying the Information-DB for available upgrades the package removal process of the old version is run. Then the installation process of the new package is run and finally the software is configured using the configuration process of the new package version.



As you can see, Software Management in Debian is quite an easy job to be done - as long as one focusses on official Debian sources it is very easy and there is a wide variety of tools available to simplify this process even more by automation and integration. Nevertheless with a little knowledge about the main concepts and ideas it may also quite easy to use and to manage it in a customized environment.

3.1.1 User Interfaces

The more common user interfaces to the Debian Software Management are:

- dpkg - CLI, doesn't provide any automaton to fetch a package remotely
- dselect - Graphical console-frontend to dpkg with the ability to fetch packages remotely
- apt-get - CLI, able to operate fully automated including remote package retrieving
- synaptic - Full-featured GUI for the GNOME Desktop Environment
- kpackage - Full-featured GUI for the KDE Desktop Environment

3.2 Debian Packages

Debian Packages mainly consist of a compressed archive containing either the binaries or the sources, additional configuration scripts and the documentation if not packaged seperately. There are four kinds of Debian Packages, three of them will be discussed in this paper. The so called Binary Packages are the most widely used since they contain the compiled binaries, the Source Packages containing application and package source, the

Virtual Packages consisting merely of dependencies and the Tasks that are a special kind of virtual Packages and since won't be discussed here.

Some data is handled in a special way in the Debian Project: Shared libraries, documentations, development headers, etc. This data is packaged separately to distinguish between end-user packages, developer packages and system packages. These kinds of packages are too special to be discussed in detail here. Instead, for further and more detailed information I'd like to refer to the Debian Documentation. In general, the differences between such special packages and regular Debian Binary Packages is marginal but the impact of these differences is important and, in case of shared libraries, for example, quite complex and hard to explain shortly.

3.2.1 Binary Packages

A Binary Package consists of a single file with the suffix ".deb". It contains the binaries, configuration scripts and the documentation. Technically, a Binary Package is an archive containing two gzipped tarballs and an ASCII file:

```
> ar xf bash_2.05a-11_i386.deb
> ls -l
total 982
-rw-r--r-- 1 dea dea 478460 8 Apr 2002 bash_2.05a-11_i386.deb
-rw-r--r-- 1 dea dea 3484 14 Apr 11:27 control.tar.gz
-rw-r--r-- 1 dea dea 474783 14 Apr 11:27 data.tar.gz
-rw-r--r-- 1 dea dea 4 14 Apr 11:27 debian-binary
```

The ASCII file `debian-binary` contains a version number, needed by `dpkg`. The compressed archive `data.tar.gz` contains the data being deployed and the compressed archive `control.tar.gz` contains the configuration scripts and data:

```
> tar -tzvf data.tar.gz
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./bin/
-rwxr-xr-x root/root    511400  8 Apr 21:07 2002 ./bin/bash
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./usr/
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./usr/bin/
-rwxr-xr-x root/root     7104  8 Apr 21:07 2002 ./usr/bin/bashbug
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./usr/share/
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./usr/share/man/
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./usr/share/man/man1/
-rw-r--r-- root/root    63634  8 Apr 21:07 2002 ./usr/share/man/man1/bash.1.gz
-rw-r--r-- root/root     713  8 Apr 21:07 2002 ./usr/share/man/man1/bashbug.1.gz
-rw-r--r-- root/root     155  8 Apr 21:07 2002 ./usr/share/man/man1/rbash.1.gz
-rw-r--r-- root/root     517  8 Apr 21:07 2002 ./usr/share/man/man1/builtins.1.gz
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./usr/share/info/
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./usr/share/doc/
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./usr/share/doc/bash/
-rw-r--r-- root/root    10919  8 Apr 21:07 2002 ./usr/share/doc/bash/NEWS.gz
...
-rw-r--r-- root/root    1756  8 Apr 21:07 2002 ./usr/share/doc/bash/copyright
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./usr/share/bash_completion/
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./etc/
drwxr-xr-x root/root          0  8 Apr 21:07 2002 ./etc/skel/
```

```

-rw-r--r-- root/root          1093  8 Apr 21:07 2002 ./etc/skel/.bashrc
-rw-r--r-- root/root           509  8 Apr 21:07 2002 ./etc/skel/.bash_profile
drwxr-xr-x root/root           0  8 Apr 21:07 2002 ./etc/bash_completion.d/
-rw-r--r-- root/root          211  8 Apr 21:07 2002 ./etc/bash.bashrc
-rw-r--r-- root/root        65921  8 Apr 21:07 2002 ./etc/bash_completion
lrwxr-xr-x root/root           0  8 Apr 21:07 2002 ./bin/rbash -> bash
lrwxr-xr-x root/root           0  8 Apr 21:07 2002 ./bin/sh -> bash
lrwxr-xr-x root/root           0  8 Apr 21:07 2002 ./usr/share/man/man1/sh.1.gz -> bash.1.gz
>
> tar -xzvf control.tar.gz
drwxr-xr-x root/root           0  8 Apr 21:07 2002 ./
-rwxr-xr-x root/root        6048  8 Apr 21:07 2002 ./preinst
-rwxr-xr-x root/root         306  8 Apr 21:07 2002 ./postinst
-rwxr-xr-x root/root         133  8 Apr 21:07 2002 ./prerm
-rwxr-xr-x root/root         133  8 Apr 21:07 2002 ./postrm
-rw-r--r-- root/root          80  8 Apr 21:07 2002 ./conffiles
-rw-r--r-- root/root         728  8 Apr 21:07 2002 ./control
>

```

The `post*` and `pre*` files are used to control installation and removal of the package. They usually are shell skripts, used to place files somewhere, trigger `debconf` to build configuration files and so on. The file `conffiles` contains a simple list of configuration files needed by the packaged software and the file `control` contains the package description and some information needed by `dpkg` such as section, priority and dependencies.

3.2.2 Source Packages

There are two kinds of Source Packages, regular Source Packages consisting of a gzipped tarball from the original sources "`<name>_<version>.orig.tar.gz`", a descriptive file containing a list of files with their corresponding MD5 sums "`<name>_<version>-<revision>.dsc`" and a gzipped diff "`<name>_<version>-<revision>.diff.gz`" containing all the changes it needs to create the Debian source tree and special files from the original sources.

The so called Native Source Packages contain sources developed by the Debian Project usually to fit any needs special to the Debian Distributions. A Native Source Package contains of two files, the source archive "`<name>_<version>.tar.gz`" and the description "`<name>_<version>.diff.gz`" (note that the suffices differ).

It is obvious how to use these files without any further knowledge. Uncompress them, extract the source tarball and patch the sources using the "`diff`" file if applicable. This works fine but there are a few tools that can ease your life a lot.

"`dpkg-source`" provides an easy way to handle Debian Source Packages. After having utilised "`dpkg-source -x <name>_<version>-<revision>.dsc`" you'll find a "ready to go" source tree in your working directory. Similar but limited to Source Packages of your Distribution functionality is provided by "`apt-get source <name>`".

"`apt-get source`" does all this in a single call. It downloads the Source Package and unpacks it in your current working directory. But there is one restriction to consider: `apt-get` only uses the releases it is configured to. If you're using the "stable" Repositories you can only download "stable" sources, etc. A possible solution may be the use of pinning, but I wouldn't recommend it only for downloading sources not available in "your" repository.

Another commonly used tool is `wget`. Using `wget` (or `fetch` or similar) you're absolutely free in choosing the release you'd like to get the sources from.

The exact directory structure of a Debian Source Package varies because it's built on top of the original source tree. But the components special to Debian are always the same and can always be found in the "`debian`" subdirectory of the source tree. Below the "`debian`" subdirectory all data needed to build a Debian Package (besides the data being distributed with the package) can be found. There are some mandatory and some optional files:

- `debian/changelog`
Mandatory; contains the history of the package itself-
- `debian/control`

Mandatory; contains information such as build-dependencies, names and dependencies of the Binary Packages, etc.

- `debian/rules`

Mandatory; a regular Makefile containing compile-time configuration statements, pre- and post-compile processing and the like.

- `debian/files`

Optional; a list of files contained by the Debian Package(s) being built.

- `debian/preinst`

Mandatory; pre-installation script for use with the Binary Package.

- `debian/postinst`

Mandatory; post-installation script for use with the Binary Package.

- `debian/prerm`

Mandatory; pre-removal script for use with the Binary Package.

- `debian/postrm`

Mandatory; post-removal script for use with the Binary Package.

There are usually more files in the `debian/` subdirectory but the ones mentioned above are the ones you have to know about when you're going to build your own or customized packages. All of the standard files and their possible contents are defined in the Debian Policy.

3.2.3 Virtual Packages

Virtual Packages consist of dependencies. They are often used to group packages in order to provide a sort of "Convenience Package". KDE, for example, is provided as a Virtual Package named `kde`. This package depends on ("requires") the packages the KDE desktop environment consists of and provides a way for the end user to install a fully functional desktop environment (by installing `kde`) without having to bother about package dependencies.

Sometimes Virtual Packages are used to inform the local Information-DB of manually installed applications. In this case you simply build an empty package with the name and the corresponding version of the software you installed manually. There's also a tool called `equivs` available to aid you in building the virtual package.

3.2.4 Backports

Backports are packages either ported "back" from another release than the one currently used or customized by the end-user. By backporting, everyone can highly customize his Debian installation without having to leave or intercept the built-in software management tools.

The more common reasons to either use or build backports are:

- Customization.

Build-time configuration can be done in a "clean" manner without a need of applying dirty hacks or similar. "build-time" in this context refers to the build process of the package and the data being packaged as well.

- Up-to-dateness.

There are plenty of packages in the current stable release containing antiquated software. Since you're neither bound to the release you're using nor to make use of the "pinning" feature of `apt`, for someone in need of the most current version or features of a particular software, backporting may be an easy way to meet his needs.

There are some sites in the internet, offering backported Binary Packages as a convenience. For many users such repositories fully satisfy their requirements but since the backports offered there are just ported from "testing to unstable" to "stable" without any changes in their configuration they may not satisfy more sophisticated requirements.

But, everyone fetching packages from such sites implicitly trusts them and their respective operators. In addition, those sites often suffer from enhanced security measures comparable to those the Debian Project takes to ensure integrity and authenticity of their repositories.

3.3 Debian Repositories

The Debian repositories aren't complex. They simply consist of a hierarchical directory structure with some essential files besides the Binary and Source Packages. For historical reasons there are currently two different repository hierarchies maintained. An old one rooting in ancient, pre-woody times, and a new one that is in use since woody.

The old directory structure followed the schema "debian/dists/<release-codename>/<section>/binary-<architecture>/" for Binary Packages and "debian/dists/<release-codename>/<section>/source/" for Source Packages with symbolic links "stable", "testing" and "unstable" linked to their respective release names. Nowadays this hierarchy is used to store the archives' Information-DBs.

The new hierarchy uses a common package pool for all releases mainly to simplify mirroring of the Debian Archives. The pool is located in "debian/pool/<section>". Subdirectories named by the first letters or numbers of the packages aid in manual administration or usage of the archives (indeed, there are some subdirectories, especially those containing libraries which names consist of up to four letters and numbers). These subdirectories contain subdirectories named by the Source Package's name (e.g. "debian/pool/main/o/openldap2"). Additionally there are some special files for controlling purposes maintained in "debian/indices/". These files, the so called overrides, control the creation of the archives' local Information-DBs. They are used to override informations placed in the packages control fields. This is done by maintenance purposes and may be used during the build of a private repository too.

3.3.1 Tools

There are some tools available that can be used to build or operate a Debian Repository. But, with the exception of DAK, they were designed and built with different purposes. Namely, some of these tools are:

- `install` - a tool to install binaries in a given location, creating directories and setting permissions as necessary
- `apt-proxy` - a repository proxy to avoid multiple machines running Debian to connect separately to the same official mirror
- `apt-move` - a tool to move cached Debian Packages (by `dpkg`, `apt`, etc.) to a proper repository hierarchy
- `mini-dinstall` - "lightweight" version of `dinstall/katie` with less sanitary checks and without the need of a PostgreSQL database. `mini-dinstall` is not available in woody but will be included in sarge
- DAK aka `katie` aka `dinstall` - the original Debian Archive management suite is available as (open) source from `cvs.debian.org`. DAK is not (yet) packaged and thus neither available in any release

From those tools, `install`, `apt-proxy` and `mini-dinstall` are choices fitting the needs to build and maintain a small (private) or medium-sized repository. `apt-move` may be used for local repositories whereas DAK is best in large and complex environments similar to the Debian Archives because of its need of heavy customization.

4 Integration

As we now know about SD, the Debian Project, its Package Formats and Repository structure, we head on to how it is actually done. We start with a short Backporting HOWTO as an example of the packaging process. I chose backporting because I assume that this kind of package building process approximates the more common needs in production environments.

Next, I'll give an overview of how a Debian style repository may be set up and how a private repository could be integrated with the official Debian mirrors followed by some hints about the necessary client configuration. And in the end I'll cover some of the caveats in building, operating and using private repositories.

4.1 Building a Backport

In this Chapter I'll demonstrate the process of backporting an existing package. The application being backported is OpenLDAP, an open source Directory Service. Instead of simply backporting OpenLDAP from either testing or unstable to stable I use the current stable package to demonstrate how custom modifications are applied to an already packaged application are applied.

This is a real life example. Early this year I wanted to install OpenLDAP on my server. I ran aptitude, selected slapd and gasped as I saw an entire X-Window environment in the queue of packages supposed to be installed. Since I run my server with as less software as possible (he only shall do the jobs I pay him for and nothing else) a X-Window environment was the last I ever wanted to run, neither at that time nor today. And why should a directory service need a graphical desktop environment?

So I went on and had a look after the cause of this dependency. Soon I discovered that the ODBC interface caused the dependency even though I still had no clue why. Again, why should a *NIX ODBC library or toolkit require an entire X-Window environment?

My next step was to visit the homepages of both the OpenLDAP Project and the two *NIX ODBC developers ("libiodbc2" and "unixodbc") to look for any requirement possibly causing a dependency on XFree in Debian. I couldn't find anything so I suspected one of the two package maintainers possibly having incorporated any kind of "convenience" into their packages. I did some tests, found openldap2innocent and finally caught the dependency in both of the ODBC packages.

Because the packages (and upstream source distributions as well) include a sample application (originally meant for demonstration purposes), a small gtk+-GUI, both need - as a result of the chain of backward-dependencies - an entire X-Window environment.

Once I found the cause, the solution was easy. Since I had no use of a GUI to configure ODBC DSNs, I simply had to look for a way to avoid its compilation and inclusion into the binary distribution. So I installed the libiodbc2 Source Package and read debian/changelog, debian/control and debian/rules. Nothing there offered a hint how to exclude the GUI.

Next I read the upstream (original) README and INSTALL. Here I found the gtk+-GUI mentioned as an optional application but again, no hint how to exclude it. So I read Makefile.in (usually this is not reliable since sometimes this file is modified during the packaging process by maintainer scripts) and finally found what I had been looking for: An undocumented build-time option named "-disable-gui" (just the name of the option, no further explanations).

I gave it a try and it worked in my local sandbox. So I customized debian/rules among other files by adding this option and the package built without errors. I linted the package without errors. Upon installation no errors occurred and so I decided finally to consider this package ready for (my) production.

All this took me about a day or so and my OpenLDAP is still up and running. But what I wanted to point to is the ability to customize existing packages. It's not very expensive (in terms of effort, time and money) and you may get exactly what you wanted "your software to do.

4.1.1 Preparation

To build any package, either a backport or a new one, we have to set up the development environment at first. There's a number of applications needed in order to create backports or new packages either:

Tabelle 5: Overview of Necessary Packages

Compiler and associated tools	binutils, cpio, cpp, gcc, libc6-dev, make, patch
Package building tools	build-essentials, debconf-utils, debhelper, debian-test, devscripts, dh-make, dpkg-dev, fakeroot
Documentation	dahb (de), debian-policy, developers-reference, devhelp, new-maint-guide
Other tools	file, gpg, perl

Having installed these packages the development environment is set up and ready to be used. In addition, we need an internet connection to get the sources.

4.1.2 Building in a Chroot'ed Environment

Especially when different releases are used in your environment (e.g. unstable for workstations and stable for servers which seems to be quite common) it may be useful to build a chroot'ed environment especially for backporting or package building purposes.

Using a chroot'ed build environment keeps your daily's working environment clean (no unnecessary packages) and if you broke the chroot's configuration you can simply delete and rebuild it.

Fortunately, Debian eases this task a lot by providing "debootstrap", a utility designed for first-time setups. "debootstrap" installs everything needed to get a working Debian Operating Environment (kernel, system binaries, basic utilities) of a given release into a given location. Chapter 3.7 of the Debian Installation Manual provides a good overview on how to use debootstrap during installation but we focus on how it can be used to prepare a chroot'ed environment for packaging purposes.

At first we need to choose a place where the chroot will reside. I'll take "/chroot/<release>/" for our example. Next we have to run debootstrap with appropriate arguments. The general syntax of debootstrap is "debootstrap -arch <architecture> <release> <path> <mirror>/debian" (note the trailing "/debian", it is needed because the repositories reside in this subdirectory at the Debian Mirrors). As we're using the i386 architecture our debootstrap call would look like this:

```
# mkdir /chroot/woody
# debootstrap --arch i386 woody /chroot/woody http://ftp.debian.org/debian
```

Depending on the speed of the internet connection this process may take several minutes up to maybe half an hour.

As soon as debootstrap finished, we have to mount the /proc filesystem into the chroot and enter it:

```
# mount -t proc woody /chroot/woody/proc
# chroot /chroot/woody/ /bin/bash
```

To finish the setup we just have to configure the environment basically and install the packages needed for packaging. To do so we first run "dpkg-reconfigure console-data" to configure keyboard and terminal fonts and then "base-config" and answer the questions asked. In the end, "base-config" offers you the opportunity to run "tasksel" (to select some ready-to-run package collections) and dselect. Choose either at your own will.

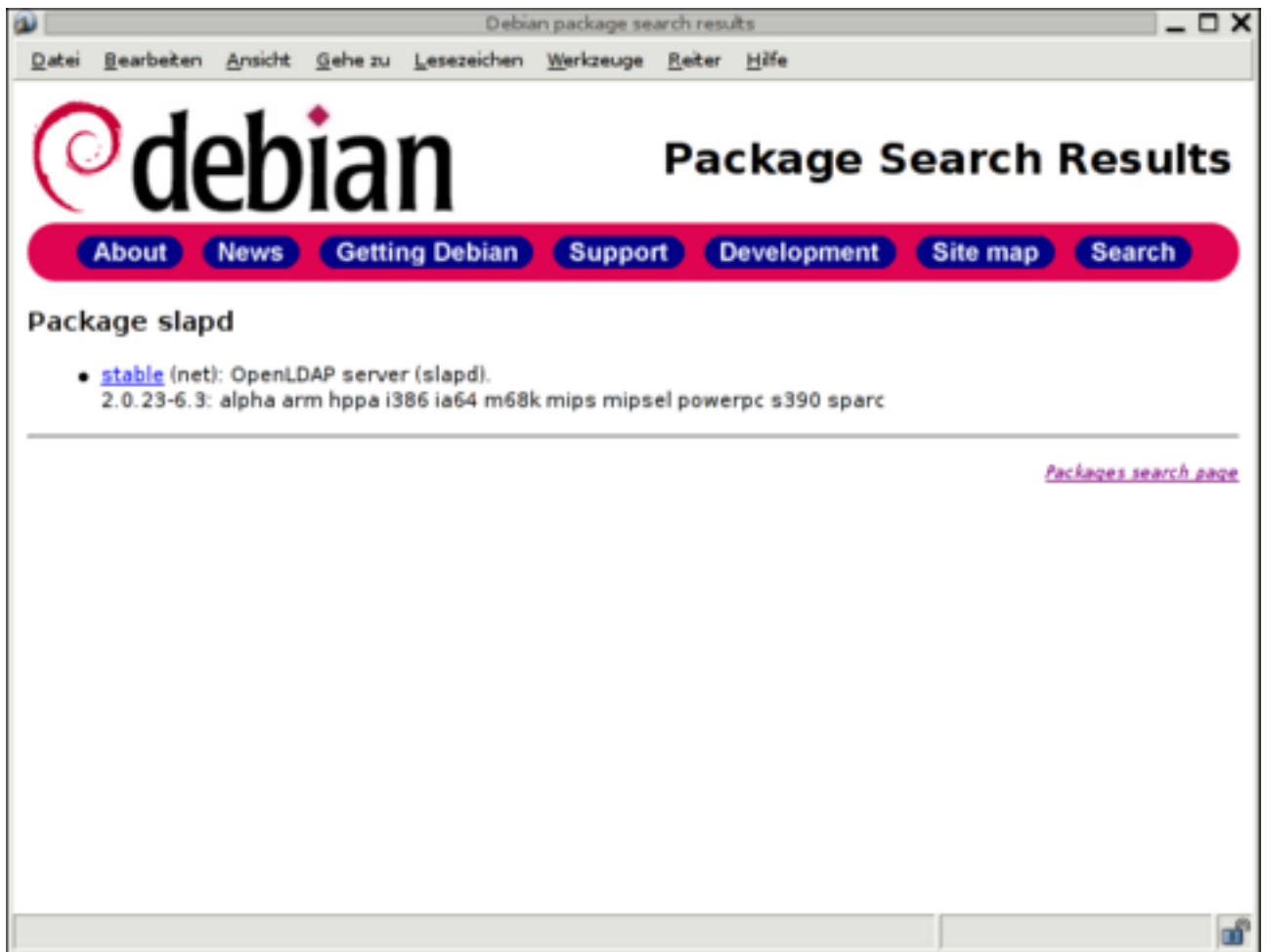
If you like your environment localized you may need to install the package locales. Note that you probably have to run "dpkg-reconfigure locales" after the installation. That's because of a bug in the package's configuration routines.

4.1.3 Getting the Sources

Having set up the development environment we need to get the package sources first. Official Debian Source Packages can either be fetched using apt-get source <packagename> or downloaded via ftp or http over the internet from the Debian Archives.

Note that it is common in Debian that the names of Source Packages do not equal those of the Binary Packages. In our example of OpenLDAP the Binary Packages are named slapd, "libldap2", "ldap-gateways" and "ldap-utils" so which one corresponds to the Source Package's name?

If you ever look for a package or a file the Debian Packages webpages at <http://packages.debian.org/> are the place to find them. Here you have the opportunity to either search the package directories for a specific package or search the contents of the packages for a specific file. In our case of OpenLDAP we use the package search and look for the name of one of the Binary Packages (e.g. slapd).



The search returns the releases in which the package is available and offers a link (named `stable` in this case) to the package description page. Following this link, a detailed description of the package, its dependencies and additional links are provided.



Above the name and version of the package its "Distribution"(release) and section are shown as links. Following the "Distribution"link you get a categorical overview of the current stable distribution. Following the section link you get a comprehensive listing of all packages of the corresponding (i.e. stable) release in this section. The red-colored word "[security]"indicates, that there has been a security update for this package.

At the bottom of the page we have the opportunity either to fetch the Binary Package for the architectures it is available or to follow some links providing additional and more detailed information. Below the caption "More Information on slapd" a link to the Debian Bug Tracking System (BTS) is provided as well as links to the Source Package's files, the corresponding changelog and the copyright file. Next to the name of the maintainer a link named "developer information for slapd"guides us to the Source Package's page.

Debian GNU/Linux -- slapd

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Werkzeuge Editor Hilfe

Download slapd

	alpha	arm	hppa	i386	ia64	m68k	mips	mipsel	powerpc	s390	sparc
	[list of files]	[list of files]	[list of files]	[list of files]	[list of files]	[list of files]	[list of files]	[list of files]	[list of files]	[list of files]	[list of files]
Sizes	787.6	656.9	737.3	592.7	1030.6	536.3	674.1	674.3	643.7	615.3	618.4
Installed sizes	2400	1840	1908	1768	3420	1596	2760	2764	1928	1860	1816

Size is measured in kBytes.

More information on slapd

Check for [bug reports](#) about slapd
Source Code: [\[dsc\]](#) [\[openldap2-2.0.23.orig.tar.gz\]](#) [\[openldap2-2.0.23-5.3.diff.gz\]](#)
View the [Debian changelog](#)
View the [copyright file](#)

[Wichert Akkerman](#) is responsible for this Debian package. See the [developer information for slapd](#).

Search for [other versions of slapd](#)

This page is also available in the following languages:
[English](#) [español](#) [français](#)
How to set [the default document language](#)

Back to: [Debian Project homepage](#) || [Packages search page](#)

See the Debian [contact page](#) for information on contacting us.

Last Modified: Sat, 14 Feb 2004 22:16:05 +0300

The Source Package's pages are of vital interest if you're going to backport a package from either testing or unstable to stable. Here you can find important additional information such as information on the maintainer(s) (Note, Wichert Akkerman isn't responsible anymore for this package, instead a small team around Torsten Landschoff has overtaken his duties), the BTS for the Source Package including an overview of the number of current bugs, you have the opportunity to subscribe to the Debian Package Tracking System (PTS) what you really should to keep up to date with the sources of your backports, links to the build logs for this package (very interesting), debcheck on stable and the lintian reports.

Overview of openldap2 source package

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Werkzeuge Reiter Hilfe

Overview of openldap2 source package

Jump to package (home page):

General Information	
Last version	2.1.29-2
Maintainer	Torsten Landschoff [mail]
Co-Maintainers	Roland Bauerschmidt [mail] Stephen Frost [mail]
Standards-Version	3.6.1
Priority & Section	important - net
Bugs Count	
All bugs	70
Release Critical	5
Important and Normal	42
Minor and Wishlist	23
Fixed and Pending	0
Subscription - Package Tracking System	
Subscribers count	11
<input type="text" value="Subscribe"/> your email	<input type="button" value="Send"/>

Problems
<ul style="list-style-type: none"> The package has not yet entered testing even though the 10-day delay is over. Check why.
Testing Status
<ul style="list-style-type: none"> 13 days old (needed 10 days) slapd (alpha, arm, hppa, i386, ia64, m68k, mips, mipsel, powerpc, s390, sparc) is (less) buggy! (1 <= 2) libldap2 (alpha, arm, hppa, i386, ia64, m68k, mips, mipsel, powerpc, s390, sparc) is buggy! (2 > 0) Not considered Depends: openldap2 cyrus-sasl2 (not considered)
Latest News
<ul style="list-style-type: none"> [2004-04-13] Accepted openldap2 2.1.29-2 (i386 source all) [2004-04-12] Accepted openldap2 2.1.29-1 (i386 source all) [2004-02-23] Accepted openldap2 2.1.26-1 (i386 source all) [2004-02-06] Accepted openldap2 2.1.25-1 (i386 source all) [2003-10-18] Accepted openldap2 2.1.23-1 (i386 source all) [2003-10-04] Accepted openldap2 2.1.22-3 (i386 source all) [2003-10-03] Accepted openldap2 2.1.22-2 (i386 source all) [2003-07-16] Accepted openldap2 2.1.22-1 (i386 source all) [2003-07-15] Accepted openldap2 2.1.21-2 (i386 source all) [2003-06-19] Accepted openldap2 2.1.21-1 (i386 source all) [2003-06-09] Accepted openldap2 2.1.17-3 (i386 source all) [2003-04-18] Accepted openldap2 2.1.17-1 (i386 source all)

At the right side of the page you have an excerpt of the package's history, status information and some statements about current problems with the package if there should be some present.

Since we just needed to know the name of the Source Package we don't need to go into more detail at this point. But I strongly recommend to browse the Debian Source Package's site, to forage the BTS and to subscribe to the corresponding PTS. The results of your curiosity may save you a lot of pain and work later on.

Now we know the name of the Source Package and as this backport is a customization of an existing stable package, we will get the sources by using apt-get:

```
> pwd
/home/dea
> apt-get source openldap2
Reading Package Lists... Done
Building Dependency Tree... Done
Need to get 1325kB of source archives.
Get:1 ftp://ftp.de.debian.org stable/main openldap2 2.0.23-6.3 (dsc) [763B]
Get:2 ftp://ftp.de.debian.org stable/main openldap2 2.0.23-6.3 (tar) [1303kB]
Get:3 ftp://ftp.de.debian.org stable/main openldap2 2.0.23-6.3 (diff) [20.9kB]
Fetched 1325kB in 37s (35.1kB/s)
dpkg-source: extracting openldap2 in openldap2-2.0.23
> ls -l
total 1308
drwxr-xr-x 11 dea dea 4096 Apr 28 14:10 openldap2-2.0.23
-rw-r--r-- 1 dea dea 20913 Jan 13 2003 openldap2_2.0.23-6.3.diff.gz
-rw-r--r-- 1 dea dea 763 Jan 13 2003 openldap2_2.0.23-6.3.dsc
-rw-r--r-- 1 dea dea 1302928 Feb 17 2002 openldap2_2.0.23.orig.tar.gz
> ls -l openldap2-2.0.23/debian/
```

```

total 100
-rw-r--r-- 1 dea dea 11372 Apr 28 14:10 changelog
-rw-r--r-- 1 dea dea 2253 Apr 28 14:10 control
-rw-r--r-- 1 dea dea 1402 Apr 28 14:10 copyright
-rw-r--r-- 1 dea dea 258 Apr 28 14:10 general.postinst
-rw-r--r-- 1 dea dea 170 Apr 28 14:10 general.preperm
-rw-r--r-- 1 dea dea 106 Apr 28 14:10 libldap2.conf
-rw-r--r-- 1 dea dea 349 Apr 28 14:10 libldap2.postinst
-rw-r--r-- 1 dea dea 261 Apr 28 14:10 libldap2.preperm
-rw-r--r-- 1 dea dea 101 Apr 28 14:10 libldap2.shlibs
-rwxr-xr-x 1 dea dea 8521 Apr 28 14:10 rules
-rw-r--r-- 1 dea dea 1870 Apr 28 14:10 slapd.conf
-rw-r--r-- 1 dea dea 298 Apr 28 14:10 slapd.conf
-rw-r--r-- 1 dea dea 4224 Apr 28 14:10 slapd.config
-rw-r--r-- 1 dea dea 1050 Apr 28 14:10 slapd.init
-rw-r--r-- 1 dea dea 2670 Apr 28 14:10 slapd.postinst
-rw-r--r-- 1 dea dea 317 Apr 28 14:10 slapd.postrm
-rw-r--r-- 1 dea dea 458 Apr 28 14:10 slapd.preperm
-rw-r--r-- 1 dea dea 3878 Apr 28 14:10 slapd.templates
-rw-r--r-- 1 dea dea 7312 Apr 28 14:10 slapd.templates.de
> _
> _

```

As apt-get finished downloading the Source Package it unpacks it into the current working directory. The name of the new directory containing the sources is following the convention <packagename>-<version>, in our case `./openldap2-2.0.23/`.

We cd into this directory and look around. What we see is the debian subdirectory next to the original sources. It contains several files of which especially `debian/changelog`, `debian/control` and `debian/rules` (in this order) are of interest.

```

> cd openldap2-2.0.23
> head -n 34 debian/changelog
openldap2 (2.0.23-6.3) stable-security; urgency=high

* Non-maintainer upload by the Security Team
* Disabled check for Berkeley DB thread support and enabled support
  unconditionally since a it seems like that check is broken by an
  compiler error on HPPA.

-- Torsten Landschoff <torsten@debian.org> Fri, 10 Jan 2003 01:54:12 +0100

openldap2 (2.0.23-6.2) stable-security; urgency=high

* Non-maintainer upload by the Security Team
* Build-Conflict with libbind-dev to use the proper resolver library
  everywhere and not let slapd immediately fail (see: Bug#112459).
* Disabled Berkeley DB thread support since it's not working on HPPA and
  was only enabled on half of the other architectures.

-- Martin Schulze <joe@infodrom.org> Fri, 20 Dec 2002 21:47:40 +0100

openldap2 (2.0.23-6.1) stable-security; urgency=high

* Non-maintainer upload by the Security Team
* Applied security patch from SuSE after a security audit

```

```
-- Martin Schulze <joe@infodrom.org> Thu, 19 Dec 2002 22:21:49 +0100
```

```
openldap2 (2.0.23-6) unstable; urgency=high
```

```
* Make slapd.config idempotent, so that calling it once (during
preconfiguration) and again (during postinst) doesn't break things.
Patch from Anthony Towns. Closes: Bug#137552).
```

```
-- Wichert Akkerman <wakkerma@debian.org> Sun, 14 Apr 2002 19:10:50 +0200
```

```
> _
```

As we can see, there is an explanation of what has been done during the security updates. As we scroll forward we can also read that nothing of particular interest (no major changes neither to the application nor to the package itself) happened in the past.

Next we'll have a look at `debian/control` since it defines the build-dependencies which are most interesting now. We can see that several packages are required to build the Binary Packages. Most of these are `*-dev` packages containing development headers (due to the dependency chains you still would have to install the corresponding applications) and some clearly devel-related packages. By the way, the beginning of the dependency chain causing me to backport this package is `"libiodbc2-dev"`.

The build-dependencies listed in this file may (and should) be used to check, if the listed packages are installed in your development environment. If they aren't you need to install them otherwise the package won't build.

The special variable `"${shlibs:Depends}"` used in the `"Depends"` stanza of the `slapd` package description will be substituted with the runtime dependencies by the helper scripts. Any additionally listed dependencies are defined by the maintainer and usually are related to either package (debconf) or daemon management (`psmisc`, `fileutils`).

```
> head -n 21 debian/control
```

```
Source: openldap2
```

```
Section: net
```

```
Priority: extra
```

```
Maintainer: Wichert Akkerman <wakkerma@debian.org>
```

```
Build-Depends: libdb3-dev, libwrap0-dev, libiodbc2-dev, patch, libsasl-dev, dpkg-dev (>= 1.16.1), libncurses5-dev, autoconf, debconf-utils
```

```
Build-Conflicts: libbind-dev
```

```
Standards-Version: 3.1.0.0
```

```
Package: slapd
```

```
Section: net
```

```
Priority: extra
```

```
Architecture: any
```

```
Depends: ${shlibs:Depends}, debconf (>= 0.2.50), fileutils (>= 4.0i-1), psmisc
```

```
Suggests: openldap-guide, ldap-utils
```

```
Conflicts: umich-ldapd, ldap-server
```

```
Provides: ldap-server
```

```
Description: OpenLDAP server (slapd).
```

This is the OpenLDAP (Lightweight Directory Access Protocol) standalone server (`slapd`). The server can be used to provide a standalone directory service and also includes the `slurpd` replication server and `centipede`.

```
> _
```

Our next step is to read `debian/rules`. Instead of pasting the entire file, I pick up the more important sections.

`debian/rules` is a regular Makefile and all its statements have to follow the corresponding rules. At the beginning we find the build-time options given to `./configure`. Here are the unchanged ones:

```
1 #!/usr/bin/make -f
2
3 rootdir      := $(shell pwd)
4 tmpdir      := $(rootdir)/debian/tmp
5 builddir    := $(rootdir)/debian/build
6
7 $(builddir)/Makefile:
8     mkdir -p $(builddir)
9     cd $(builddir) ; $(rootdir)/configure \
10         --enable-debug --enable-syslog --enable-proctitle \
11         --enable-cache --enable-referrals --enable-ipv6 \
12         --enable-local --with-cyrus-sasl --with-readline \
13         --with-threads --enable-slaped --enable-cleartext \
14         --enable-crypt --enable-passwd --enable-spaswd \
15         --enable-multimaster --enable-phonetic --enable-rlookups \
16         --enable-wrappers --enable-dynamic --disable-dnssrv \
17         --enable-ldap --enable-ldbm --enable-shell --enable-sql \
18         --enable-slurpd --enable-shared --without-tls \
19         --prefix=/usr --localstatedir=/var/lib \
20         --sysconfdir=/etc --libexecdir='$$${prefix}'/sbin \
21         --mandir='$$${prefix}'/share/man --with-subdir=ldap
22     $(MAKE) -C $(builddir) depend
23
```

The options passed to `./configure` are pretty self-explaining. Most of them deal with specific details of the OpenLDAP implementation but some don't. We will revert the options `-enable-ipv6` and `-without-tls` later.

As we proceed in reading `debian/rules` we can see the steps taken to separate the "monolithic source distribution into several functionally separated Binary Packages. At the end of the file the packages are actually built. I included this listing because it shows in great detail how the process of package creation actually works.

Everything happens below the `debian/` subdirectory of the extracted Source Package directory. After some cleanup a temporary directory is set up and passed as target to "make install" (ll. 36). After some general cleanup (ll. 37-45) the individual packages are prepared by moving the corresponding files into their "target directories" (ll. 47-160). Note, these target directories reside below `debian/` since they still are of temporary nature and removed after the individual packages have been built.

If all cleaning, building and moving has finished, the Binary Packages are built. At first, needed files and directories are prepared (ll. 162-193), the files are stripped (ll. 202-203), control information is generated (l. 204) and then the individual packages are built by running "dpkg -build" (l. 205). That's it.

```
24 build: $(builddir)/Makefile
25     $(MAKE) -C $(builddir)
26
27 binary: binary-arch
28
29 binary-indep:
30
31 binary-arch: build
32     rm -rf $(tmpdir) debian/ldap-gateways debian/ldap-utils
```

```

33     rm -rf debian/libldap2-dev debian/libldap2 debian/slappd
34
35     install -d -o root -g root -m 755 $(tmpdir)
36     $(MAKE) -C $(builddir) DESTDIR=$(tmpdir) install
37     mv $(tmpdir)/usr/bin/ud $(tmpdir)/usr/bin/ud-ldap
38     mv $(tmpdir)/usr/share/man/man1/ud.1 \
39         $(tmpdir)/usr/share/man/man1/ud-ldap.1
40     mv $(tmpdir)/usr/sbin/in.xfingerd $(tmpdir)/usr/sbin/in.ldapfingerd
41     mv $(tmpdir)/usr/share/man/man8/in.xfingerd.8 \
42         $(tmpdir)/usr/share/man/man8/in.ldapfingerd.8
43     find $(tmpdir) -type f -a -name \*.default | xargs rm
44     chmod -R u+w $(tmpdir)
45     chmod -R g-w $(tmpdir)
46
47 # Split off ldap-gateways
48     install -d -o root -g root -m 755 debian/ldap-gateways/usr/sbin
49     install -d -o root -g root -m 755 debian/ldap-gateways/usr/share/man/man8
50     install -d -o root -g root -m 755 debian/ldap-gateways/usr/share/ldap
51
52     set -e ; for cmd in go500 go500gw mail500 rp500 fax500 xrpcmp \
53         rcpt500 maildap in.ldapfingerd ; do \
54         mv $(tmpdir)/usr/sbin/$$cmd debian/ldap-gateways/usr/sbin ; \
55
[...]
```

```

65 # Split off ldap-utils
66     install -d -o root -g root -m 755 debian/ldap-utils/usr/bin
67     install -d -o root -g root -m 755 debian/ldap-utils/usr/share/man/man1
68     set -e ; for cmd in ldapsearch ldapmodify ldapdelete ldapmodrdn \
69         ldapadd ldappasswd ud-ldap ; do \
70         mv $(tmpdir)/usr/bin/$$cmd debian/ldap-utils/usr/bin/ ; \
71
[...]
```

```

161 # Build the packages
162     install -d -o root -g root -m 755 debian/libldap2/DEBIAN
163     install -p -o root -g root -m 644 debian/libldap2.conffiles \
164         debian/libldap2/DEBIAN/conffiles
165     install -p -o root -g root -m 644 debian/libldap2.shlibs \
166         debian/libldap2/DEBIAN/shlibs
167     install -p -o root -g root -m 755 debian/libldap2.postinst \
168         debian/libldap2/DEBIAN/postinst
169     install -p -o root -g root -m 755 debian/libldap2.preperm \
170         debian/libldap2/DEBIAN/preperm
171
172     set -e ; for pkg in ldap-gateways ldap-utils libldap2-dev ; do \
173         install -d -o root -g root -m 755 debian/$$pkg/DEBIAN ; \
174         sed -e "s/@PK@/$$pkg/g" debian/general.postinst > \
175             debian/$$pkg/DEBIAN/postinst ; \
176         sed -e "s/@PK@/$$pkg/g" debian/general.preperm > \
177             debian/$$pkg/DEBIAN/preperm ; \
178         chown root.root debian/$$pkg/DEBIAN/* ; \
179         chmod 755 debian/$$pkg/DEBIAN/* ; \
180     done
181
```

```

182     install -d -o root -g root -m 755 debian/slapd/DEBIAN
183     install -p -o root -g root -m 644 debian/slapd.conffiles \
184         debian/slapd/DEBIAN/conffiles
185
186     debconf-mergetemplate debian/slapd.templates.de debian/slapd.templates \
187         > debian/slapd/DEBIAN/templates
188     chmod 644 debian/slapd/DEBIAN/templates
189
190     set -e ; for f in config prerm postinst postrm ; do \
191         install -p -o root -g root -m 755 debian/slapd.$$f \
192             debian/slapd/DEBIAN/$$f ; \
193     done
194
195     set -e ; for pkg in ldap-gateways ldap-utils libldap2-dev libldap2 \
196         slapd ; do \
197         rm -f debian/substvars ; \
198         lst="$( test -d debian/$$pkg/usr/lib && find debian/$$pkg/usr/lib -ty
199             test -d debian/$$pkg/usr/bin && find debian/$$pkg/usr/bin -typ
200             test -d debian/$$pkg/usr/sbin && find debian/$$pkg/usr/sbin -t
201         test -n "$$lst" && ( \
202             strip --strip-unneeded --remove-section=.comment --remove-sect
203             dpkg-shlibdeps $$lst ) ; \
204         dpkg-gencontrol -isp -p$$pkg -Pdebian/$$pkg ; \
205         dpkg --build debian/$$pkg .. ; \
206     done
207

```

We still have to customize libiodbc2 (remember, there's the origin of the dependency chain causing XFree86 as a required dependency). Therefore we apt-get the sources too.

First, we have a look at debian/control. We see a build-dependency on libgtk1.2-dev which probably will result in a dependency on libgtk1.2 of the Binary Package. But that's not enough yet to claim it proven. So we will have to search further.

```

> apt-get source libiodbc2
Reading Package Lists... Done
Building Dependency Tree... Done
Need to get 377kB of source archives.
Get:1 ftp://ftp.de.debian.org stable/main libiodbc2 3.0.5-3 (dsc) [658B]
Get:2 ftp://ftp.de.debian.org stable/main libiodbc2 3.0.5-3 (tar) [374kB]
Get:3 ftp://ftp.de.debian.org stable/main libiodbc2 3.0.5-3 (diff) [2803B]
Fetched 377kB in 26s (14.3kB/s)
dpkg-source: extracting libiodbc2 in libiodbc2-3.0.5
> ls -l libiodbc2-3.0.5/debian/
total 44
-rw-r--r-- 1 dea    dea          600 Apr 28 16:04 README.Maintainer.patch.s390-confi
-rw-r--r-- 1 dea    dea       1546 Apr 28 16:04 changelog
-rw-r--r-- 1 dea    dea       1674 Apr 28 16:04 control
-rw-r--r-- 1 dea    dea       1082 Apr 28 16:04 copyright
-rw-r--r-- 1 dea    dea         19 Apr 28 16:04 libiodbc2-dev.examples
-rw-r--r-- 1 dea    dea        252 Apr 28 16:04 libiodbc2-dev.files
-rw-r--r-- 1 dea    dea         33 Apr 28 16:04 libiodbc2.docs
-rw-r--r-- 1 dea    dea         44 Apr 28 16:04 libiodbc2.examples
-rw-r--r-- 1 dea    dea         92 Apr 28 16:04 libiodbc2.files
-rw-r--r-- 1 dea    dea         30 Apr 28 16:04 libiodbc2.undocumented

```

```

-rwxr-xr-x  1 dea      dea          1542 Apr 28 16:04 rules
> head -n 5 debian/control
Source: libiodbc2
Section: libs
Priority: optional
Maintainer: Christian Hammers <ch@debian.org>
Build-Depends: debhelper (>= 3.0.0), libgtk1.2-dev
Standards-Version: 3.5.1
> _

```

A quick look into `debian/changelog` shows that a `iodbcadm-gtk` binary has been added in revision 3 of the package (l. 11). The name of this binary sounds like it could be originator of our dependency. But do we need it and if not, how do we exclude it?

```

1 libiodbc2 (3.0.5-3) unstable; urgency=low
2
3 * Corrected typo in package description.
4 * Added libgtk1.2-dev to build-depends. Closes: #140723
5
6 -- Christian Hammers <ch@debian.org>  Fri, 29 Mar 2002 17:12:55 +0100
7
8 libiodbc2 (3.0.5-2) unstable; urgency=low
9
10 * Corrected typo in package description.
11 * Added libgtk1.2-dev to build-depends. Closes: #140723
12
13 -- Christian Hammers <ch@debian.org>  Fri, 29 Mar 2002 17:12:55 +0100
14
15 libiodbc2 (3.0.5-1) unstable; urgency=low
16
17 * New upstream version. Closes: #113987
18   - now supports ODBC3
19   - added iodbc-config and iodbcadm-gtk binaries
20 * This library is supposed to be backwards compatible to the 2.x
21   versions.
22 * iodbcadm-gtk doesn't build correctly
23

```

Searching for any information about this `iodbcadm-gtk` thingy in the usual upstream files `README`, `INSTALL`, `NEWS`, `ChangeLog` and another file called `IAPA-PACKAGE` doesn't reveal anything else than it was added recently. The last step is to think about the way `./configure` works. The script `./configure` is usually built by `autoconf`. `autoconf` uses a file `configure.in` to generate `./configure` and, lucky enough, a file called `configure.in` is distributed with the Source Package. And here we find an undocumented option to disable the `gui` (l. 103) and a few lines later, the existence of `gtk-libraries` is checked. Seems as if this would be the option we've been looking for.

But still no information about the tasks `iodbcadm-gtk` shall fulfill nor if it is really needed or just a convenience tool. I assume for now that this tool is supposed to ease the task of creating DSN files and placing them in the right places. In consequence, this means that it should be possible to omit this application to the cost of less convenience. Fair enough for a server system.

```

97 ##
98 ## Check if we want to build the GUI applications and libraries ##
99 ##
100 #####
101 AC_ARG_ENABLE(gui,
102 [ --enable-gui          build GUI applications (default),
103   --disable-gui         build GUI applications],
104 [ case "${enableval}" in
105     yes) gui=true ;;
106     no)  gui=false ;;
107     *)  AC_MSG_ERROR(bad value ${enableval} for --enable-gui) ;;
108   esac
109 ],[gui=true])
110 AM_CONDITIONAL(GUI, [test x$gui = xtrue])
111
112
113 #####
114 ##
115 ## Check for GTK library functions ##
116 ##
117 #####
118 AM_PATH_GTK(1.2.3)
119 AM_CONDITIONAL(GTK, [test x$no_gtk = x])
120

```

As we found a possibility to disable the building of the GUI application we know now every change we have to apply.

4.1.4 Naming

Naming is maybe the most important topic concerning (but not only) backports. Why?

Imagine, you made a backport of package foo in version 2.0 revision 5. The resulting name would be "foo-2.0-5". You install the package and everything is fine. Until the next revision of the package is available. Especially if you're using one of the higher automated package management tools (e.g. aptitude, synaptic, dselect) you can get a situation where the tool silently upgrades the package to the "new" version, overriding all customizations you made. That's not desirable and therefore we have to mark the backport as a backport. But how?

Choosing a different name is not a good idea since this leads to confusion constantly increasing as the number of backports or own packages rises. It would also be hard to use any other (regular) package which depends on the (original of the) backported package because their dependencies wouldn't match the new name.

A different version number only delays the problem for a time. As soon as the upstream source reaches the version you chose not only the package management cannot distinguish between your backport and the current regular package but it is very much likely that the current regular package contains a different software than your backport does. The same goes for the revision numbers, they too should be left untouched.

Fortunately the Debian Package Management lets you the opportunity to append an arbitrary string after the last character or digit of the revision. For example, you could name the backported package foo something like "foo-2.0-5.BACKPORT". This wouldn't work since one of the few restrictions of the Debian Naming Conventions says that package names (including versioning, revisioning and extra information included) must be in lowercase. This would mean you could name the package "foo-2.0-5.backport" and it would work.

The advantages of appending an identifier behind the package revision are mainly better recognition by both tools and user and independence of future changes in version, revision or even name of the corresponding regular package. The main disadvantage is the resulting long name, often crippled by the package management tools. Most if not all of the reputable backporters do so. For example, Norbert Tretkowski appends his domainname "backports.org" to the revision or Adrian Bunk appends "bunk-<revision>" where "<revision>" identifies his backports' revision.

It's up to you to find an appropriate naming scheme, there are only very few limitations. But keep it consistently!

4.1.5 Applying Changes

Now we're going to apply the changes we collected in the previous section:

- Changes in openldap2-2.0.23:
 - Disable IPv6 support
 - Enable TLS
- Changes in libiodbc2-3.0.5:
 - Apply `-disable-gui` to omit `iodbcadm-gtk`

The changes in openldap2 are quite easy to apply, simply change the corresponding options passed to `./configure` in `debian/rules`:

```
--- debian.rules          Sun Feb 15 21:16:12 2004
+++ openldap2-2.0.23/debian/rules      Sun Feb 15 19:50:32 2004
@@ -8,14 +8,14 @@
     mkdir -p $(builddir)
     cd $(builddir) ; $(rootdir)/configure \
-         --enable-debug --enable-syslog --enable-proctitle \
+         --enable-cache --enable-referrals --disable-ipv6 \
+         --enable-cache --enable-referrals --enable-ipv6 \
         --enable-local --with-cyrus-sasl --with-readline \
         --with-threads --enable-slapd --enable-cleartext \
         --enable-crypt --enable-passwd --enable-spaswd \
         --enable-multimaster --enable-phonetic --enable-rlookups \
         --enable-wrappers --enable-dynamic --disable-dnssrv \
         --enable-ldap --enable-ldb --enable-shell --enable-sql \
-         --enable-slurpd --enable-shared --with-tls \
+         --enable-slurpd --enable-shared --without-tls \
         --prefix=/usr --localstatedir=/var/lib \
         --sysconfdir=/etc --libexecdir='$$prefix'/sbin \
         --mandir='$$prefix'/share/man --with-subdir=ldap
```

Because we only changed values of build-time options that do not result in additional build- or binary-dependencies we're done with this package.

In the case of libiodbc2, there are more changes to be applied. At first we have to add the option `-disable-gui` to the appropriate section in `debian/rules`. Then we have to customize the dependencies in `debian/control` and any other control file that may be affected by our customization.

Adding the option `-disable-gui` is easy and done by simply appending it to the corresponding line. Justifying `debian/control` is nearly as easy since we only have to remove any occurrence of `"libgtk1.2-dev"` or `"libgtk1.2"`. There's only one occurrence of `"libgtk1.2-dev"` in `"Build-Depends"` and no occurrence of `"libgtk1.2"` simple.

To find other files that need to be customized there are two ways: Either `grep` all files in the `"debian/"` subdirectory to strings containing `iodbcadm` or compile the binaries (not building the package) once using the default options, get a filelisting, clean up (remove the entire source directory and unpack the Source Package again) and compile the binaries again this time using the customized options. Comparing the filelisting from the `"default"` compilation with the files you got after compiling with the customized options will give you a listing with files not being built. These filenames have to be removed from the control files below `"debian/"`.

The former is quick, easy and unreliable. Since you don't know in advance which files will not be compiled using the customized build options, relying on the string `iodbcadm` will give you some results but you don't know if the results are complete. The latter is cumbersome and takes time but the result you get is complete and reliable.

I decided to do the latter but won't bother with an exhaustive description. What you have to do in such cases is to copy the options for `./configure` from `debian/rules`, run `./configure` with these options, run `make` and either analyze the Makefile for the installation procedure or just do a `ls -lR * > ../filelisting.txt`. Then you remove the entire source directory and unpack the sources again using `dpkg-source -x libiodbc2_3.0.5-3.dsc` (or whatever package you're backporting). Next, re-enter the source directory, modify `debian/rules`, copy the options and run `./configure` again, this time with the modified options. Run `make` and `ls -lR * > ../filelisting-cust.txt`. Run `diff -U filelisting.txt filelisting-cust.txt > files.diff` to get the list of files. The `grep "debian/"` for the names of those files that do not exist in your customized compilation and you should get a list of files and lines to change.

In case of `libiodbc` the files to change are `debian/rules`, `debian/control`, `debian/libiodbc2.files`, `debian/libiodbc2-dev.files` and `debian/libiodbc2.undocumented`. The `*.files` files each contain a listing of files of the corresponding Binary Package whereas the file `libiodbc2.undocumented` acts as a pointer to the special `undocumented` manpages, a manpage that simply says that there's no documentation yet available and that you're invited to write some.

4.1.6 Patch Management

Now we have applied the changes we wanted to. But how to handle them in the future? What to do after the next security update with the package? Since the number of changes in our example is small, it may be sufficient to remember the changes we made and apply them manually again. But this won't work anymore as soon as the number of backports or changes increases. Imagine a collection of ten backports incorporating approximately 25 changes resulting in 35 or 40 files to edit. This is already too much to remember and re-apply manually.

There are probably a lot of possible solutions available. One of them is to do it by using CVS. All you have to do is to run `diff` over the original and the modified file and commit the patch you get into your CVS. When a new package version or revision is available you just have to checkout the patch and apply it - in theory. Indeed, there is no way to get around thorough inspections of the changes incorporated by the package maintainer. It is always possible that one of the changes render your patches useless or even needless.

There are several nuances up to importing the entire Source Package and comparing it against your customized copy. The one described here is an easy and simple way to handle and manage your patches. It's up to you how far you go and what measures you (have to) take.

Another issue is not described here: Patching the upstream sources. This is not as easy as it may look at first sight. Since the automated package building process reverts any changes made outside the `debian/` subdirectory by deleting and re-extracting the sources, such changes need to be applied somehow during the package creation using Debian methods and tools.

In `woody` such patches usually reside either in the `debian/` subdirectory or in their own subdirectory below `debian/`. Their appliance is often managed by individual maintainer scripts called in `debian/rules`. This works but may cause severe problems if not properly managed and thoroughly re-checked with every new package revision.

The next release, `sarge`, provides a tool called `dpatch` that adds hooks to the package creation process which are supposed to be used for patching the upstream sources. All it needs is to be included in `debian/rules` and the patches placed in a special subdirectory below `debian/`. There's extensive documentation as well as example files available either in the package source or in `/usr/share/doc/dpatch`. I would strongly recommend using this tool since it is powerful yet easy to use.

4.1.7 Documentation

Not everybody likes it, even less do it. Documentation. But especially in backported environments it is vital to document every change you applied to either the packages or the sources. Even the smallest undocumented change may cause devastating troubles so you better keep at least records of what you've done in the changelog. Of course a more thorough and comprehensive documentation separate from the changelog is always a

good thing to have but for now I'll focus on the package's changelog and how to use it.

The package's changelog is located in "debian/changelog". We already saw it in an earlier subsection but now we need to change it in order to properly document the changes we made. The changelog's format is defined as follows in the Debian Policy:

```
package (version) distribution(s); urgency=urgency
    [optional blank line(s), stripped]
    * change details
      more change details
      [blank line(s), included in output of dpkg-parsechangelog]
    * even more change details
      [optional blank line(s), stripped]
-- maintainer name <email address>[two spaces] date
```

Here is what a Debian changelog may look like:

```
libiodbc2 (3.0.5-3) unstable; urgency=low

* Corrected typo in package description.
* Added libgtk1.2-dev to build-depends. Closes: #140723

-- Christian Hammers <ch@debian.org> Fri, 29 Mar 2002 17:12:55 +0100
```

Since the changelog is a plain text file we could edit it with any editor. But we don't because we use "dch", a tool that simplifies the editing of the changelog very much. As some of the data provided by the changelog entries (namely version, distribution, maintainer name and email) influences the automated process of package building this data must be correct. This is enforced by dch. For example, if you decide to use a version that does not comply with the Debian Policy and do so by manually editing the changelog your mistake will come up at the end of the package build process. Annoying and unnecessary. dch complains immediately if you try to use an incompatible naming scheme. Additionally, dch handles revision upgrades automatically as well as it launches your favorite editor to edit the entries in the changelog. The syntax is quite simple:

- Use "dch -i" to increment the revision of the package
- Use "dch -a" to append a new changelog entry to the current entries
- Use dch -v <version>-<revision>-<identifier>-<your revision>" to create a new version of the package

All shown combinations accept arbitrary text as a last argument passed to dch which will become the first changelog entry. If you run "dch -v" it will rename your working directory according to the version you specified unless you pass the option -preserve" to dch. Read the manpage for further details. Below is a sample session.

```
> pwd
/home/dea/libiodbc2-3.0.5
> head debian/changelog
libiodbc2 (3.0.5-3) unstable; urgency=low

* Corrected typo in package description.
* Added libgtk1.2-dev to build-depends. Closes: #140723

-- Christian Hammers <ch@debian.org> Fri, 29 Mar 2002 17:12:55 +0100
```

```
libiodbc2 (3.0.5-2) unstable; urgency=low
```

```
* Corrected typo in package description.
```

```
> dch -v 3.0.5-3.backport-1 Initial Backport
```

```
Warning: your current directory has been renamed to:
```

```
../libiodbc2-3.0.5-3.backport
```

```
> cd ../libiodbc2-3.0.5-3.backport/
```

```
> head debian/changelog
```

```
libiodbc2 (3.0.5-3.backport-1) unstable; urgency=low
```

```
* Initial Backport
```

```
-- Daniel E. Atencio Psille <dea@glux.atencio.info> Thu, 29 Apr 2004 17:12:47 +0200
```

```
libiodbc2 (3.0.5-3) unstable; urgency=low
```

```
* Corrected typo in package description.
```

```
* Added libgtk1.2-dev to build-depends. Closes: #140723
```

```
> _
```

Keep in mind that using the changelog is only half of your duties as a responsible backporter! You always should use and maintain a separate and comprehensive documentation either at a separate place, in the package or both.

Just one last word regarding the changelog before we proceed to the remaining steps. Entries in "debian/changelog" are strictly related to the package and its corresponding bugs, changes, etc. From time to time there are extensive discussions in debian-devel that always lead to the point that it is very hard to write good changelogs. Some hints to consider when writing a changelog entry:

- Be verbose enough to avoid misunderstandings but keep the entries short
- If you close any bugs, mention each of them and provide a short explanation why they have been closed
- If you change something in the upstream sources, mention it but provide more detailed information in the corresponding changelog
- Don't try to differ from entries worth being written and entries not worth it - there's no difference they all are worth to be written

4.1.8 Building

The packages are patched and everything is ready and waiting. So we're going to build the Binary Packages. In order to do so we first must ensure that our current working directory is the root of the Source Packages directory hierarchy.

Building of the package itself is easy. Just make sure you cd'ed into the root of the Source Package's directory tree and call "dpkg-buildpackage -rfakeroot". The sources are compiled and the Binary Package, a Source Package (because we changed the version) and some other control files are generated in the directory one level above your current working directory.

```
> ls -l ../*. {deb,dsc,changes,gz}
```

```
-rw-r--r-- 1 dea dea 65352 Apr 29 18:57 ldap-gateways_2.0.23-6.backport-1_
-rw-r--r-- 1 dea dea 86792 Apr 29 18:57 ldap-utils_2.0.23-6.backport-1_i38
-rw-r--r-- 1 dea dea 1790456 Apr 29 18:57 libldap2-dev_2.0.23-6.backport-1_i
-rw-r--r-- 1 dea dea 186484 Apr 29 18:57 libldap2_2.0.23-6.backport-1_i386.
```

```

-rw-r--r--    1 dea    dea          20913 Jan 13  2003 openldap2_2.0.23-6.3.diff.gz
-rw-r--r--    1 dea    dea           763 Jan 13  2003 openldap2_2.0.23-6.3.dsc
-rw-r--r--    1 dea    dea           482 Apr 29  18:53 openldap2_2.0.23-6.backport-1.dsc
-rw-r--r--    1 dea    dea        1319078 Apr 29  18:53 openldap2_2.0.23-6.backport-1.tar
-rw-r--r--    1 dea    dea           1334 Apr 29  18:57 openldap2_2.0.23-6.backport-1_i386
-rw-r--r--    1 dea    dea        1302928 Feb 17  2002 openldap2_2.0.23.orig.tar.gz
-rw-r--r--    1 dea    dea        609700 Apr 29  18:57 slapd_2.0.23-6.backport-1_i386.deb
> _

```

As we see, the old Source Package files have been kept and new ones were created according the new versioning. And we see all Binary Packages carrying our identifier "backport" in their names.

There are a few options of "dpkg-buildpackage" that might be of special interest because the control what will be built and how it will be built:

- `r<rootcommand>` specifies the command to switch to a superuser account such as `sudo` or "fakeroot"
- `b` tells "dpkg-buildpackage" to build only the Binary Packages
- `us` and `uc` tell dpkg-buildpackage not to sign the Source Package or the ".changes" file, respectively

Since "dpkg-buildpackage" starts with first checks on build-dependencies and similar, does not install anything and stops at every severe error it can be used safely. Usually the error reports are verbose enough to locate the error but if you reached this stage you shouldn't encounter any errors anymore.

4.1.9 Building the Source

What may sound confusing at first sight may become useful if you're going to feed your repository not only with your backports but also with the corresponding (backported) Source Packages: Backporting the sources. Indeed, this simply means, that you build a *.diff.gz that can be used in conjunction with the corresponding *.orig.tar.gz (which you already have) and *.dsc (which you built using "dpkg-buildpackage" without the -böption).

Doing so is simple, just run "dpkg-source -b <modified_source_directory> <corresponding>.orig.tar.gz" as shown in the following example:

```

> dpkg-source -b openldap2-2.0.23-6.3.backport openldap2_2.0.23.orig.tar.gz
dpkg-source: warning: .orig.tar.gz name openldap2_2.0.23.orig.tar.gz is not
<package>_<upstreamversion>.orig.tar.gz (wanted openldap2_2.0.23-6.3.backport.orig.tar.gz)
dpkg-source: building openldap2 using existing openldap2_2.0.23.orig.tar.gz
dpkg-source: building openldap2 in openldap2_2.0.23-6.3.backport-1.diff.gz
dpkg-source: building openldap2 in openldap2_2.0.23-6.3.backport-1.dsc
> ls -ld openldap2*
drwxr-xr-x    11 dea    dea          4096 Apr 29  18:46 openldap2-2.0.23-6.3.backport
-rw-r--r--    1 dea    dea        20978 May  2  16:51 openldap2_2.0.23-6.3.backport-1.di
-rw-r--r--    1 dea    dea           536 May  2  16:51 openldap2_2.0.23-6.3.backport-1.ds
-rw-r--r--    1 dea    dea          20913 Jan 13  2003 openldap2_2.0.23-6.3.diff.gz
-rw-r--r--    1 dea    dea           763 Jan 13  2003 openldap2_2.0.23-6.3.dsc
-rw-r--r--    1 dea    dea        1302928 Feb 17  2002 openldap2_2.0.23.orig.tar.gz
> _

```

This way you get a complete Debian Source Package of your Backport which can be distributed using the standard Debian mechanisms. The warning issued is because of the naming scheme we chose at the beginning. The Debian tools expect only digits and hyphens as parts of a version and complain on versions or revisions containing letters. That's a regular behavior and may be safely ignored in our special case.

4.1.10 Testing

After having built the package we need to test it. There's a variety of testing degrees you may perform, simple ones that merely check the package structure, more advanced ones which also try to install and remove the package up to sophisticated ones that not only install and remove the package but also take a look on the application or data packaged as well as if there are any traces (wanted and unwanted) of the package after removal. Which one you choose is up to you or your organization's requirements and standards.

Since testing is a very complex theme and in backporting we rely heavily on the work done by the Debian Developers, I'll discuss only simple tests for now. There is many extensive documentation available in the web, especially in the Debian Developers Reference and the Debian Policy.

There is as usual a wide variety of tools available to test Debian Packages. I will demonstrate "lintian" since it probably is the most convenient for end-user's needs (even though it might not meet the Debian Developer's needs).

"lintian" can be used "the easy way" but can also do thorough checks on the packages. The "easy way" consists of the installation and a call "lintian <packagename> | lintian-info" and may look like this (since OpenLDAP consist of multiple Binary Packages I used "lintian *.deb | lintian-info"):

```
> lintian *.deb | lintian-info
W: slapd: postinst-uses-db-input
N:
N:   It is generally not a good idea for postinst scripts to use debconf
N:   commands like db_input. Typically, they should restrict themselves to
N:   db_get to request previously acquired information, and have the config
N:   script do the actual prompting.
N:
W: libldap2: postinst-unsafe-ldconfig
N:
N:   The postinst script calls ldconfig unsafely. The postinst must only
N:   call ldconfig when given the argument "configure".
N:
N:   Refer to Policy Manual, chapter 9 for details.
N:
W: libldap2: prerm-calls-ldconfig
N:
N:   The prerm script calls ldconfig. Calls to ldconfig should only be in
N:   postinst and postrm scripts.
N:
N:   Refer to Policy Manual, chapter 9 for details.
N:
W: ldap-utils: package-contains-hardlink
N:
N:   hardlinks are bad m'kay, don't use hardlinks.
N:
E: ldap-gateways: binary-without-manpage maildap
N:
N:   Each binary in /usr/bin, /usr/sbin, /bin, /sbin, or /usr/games, must
N:   have a manual page.
N:
N:   Note, that though the 'man' program has the capability to check for
N:   several program names in the NAMES section, each of these programs
N:   must have its own manual page (a symbolic link to the appropriate
N:   manual page is sufficient) because other manual page viewers such as
N:   xman or tkman don't support this.
N:
```

```
N: Refer to Policy Manual, section 13.1 for details.
N:
E: ldap-gateways: binary-without-manpage xrpcmp
> _
```

Some warnings ("W:") have been issued and there are also two errors (E:). Both errors occur because there are binaries without corresponding manpages. Since this is something we can't really change and doesn't affect the functionality at all we can either patch the Source Package to contain symlinks to `undocumented(1)` (a special manpage saying that there's no documentation available) or ignore the errors. The same applies for the warnings issued. Since we didn't touch the package structure we're not "responsible" for them - but, of course, we could follow the recommendations and correct the warning's causes.

What happened in the background when we ran `lintian *.deb | lintian-info`? At first, all files with a suffix of `.deb` were queued for inspection. Then lintian dissected them in a so called "laboratory". Then it checks by comparing the package and its contents against the Debian Policy and prints the results to stdout. Note, lintian is not a bug hunting tool and never has been designed for. Its purpose is to check Debian Packages against the Debian Policy.

4.1.11 Maintenance Tasks

Maintaining a package or backport includes tasks like watching upstream for new releases or bugfixes and incorporating them into the corresponding package. `upstream` in this context does not only mean the Debian `upstream`, the author of the software itself, but also the Debian distribution for which the package has been backported.

There are quite a lot of tools and scripts available, most of them in the package `devscripts`, of which I'll describe one in more detail.

`uupdate` can be used to update an existing Source Package with an archive or patches from `upstream`. I have been using `uupdate` to apply self made patches to the upstream sources of another package I've been backporting. It works fine, even though I consider its use as risky since you don't know what you're actually doing as long as you haven't studied upstreams corresponding documentation (README, ChangeLog, NEWS, etc.) thoroughly. Have a look at the manpage and decide whether to use it or not.

Another important task is to monitor bug reports for all the packages you've backported. This is quite easy but results in a lot of reading. All you have to do is to subscribe to the Package Tracking System (PTS) of the corresponding package. This can be done at `packages.qa.debian.org` on the Source Package's website. Having acknowledged your subscription you will receive an email whenever something happens to either the Source Package or the corresponding Binary Package(s). This includes new uploads by the maintainer as well as NMUs, new bug reports and changes to bug reports and the like. Most of the mails may be irrelevant to a backporter but the ones that aren't are of high value because you're informed very early and can use the time to react.

4.1.12 Building a New Debian Package

There are some Differences between backporting an existing and building a new Debian Package from scratch. Mainly, those differences happen at the beginning of the process:

- Decisions

Upon backporting an existing package you don't have to come to certain decisions, e.g. in what section the package has to be put into, or whether it should be split up into several packages and so on. If you're going to build a new package you will have to and there are no tools, only humans to aid you in this process.

- Initial Preparation

If you're building a new Debian Package all you have is the original source code. You will have to prepare the sources and build the structures needed to build a Debian Package. You also have to provide all the information needed to install, configure and remove the package properly. There is a tool available, `dh-make` aids you in performing this task by doing most of the initial jobs automatically and providing you with a set of (more or less empty) template files.

- Testing

You should be prepared to invest a noteworthy amount of time into testing the package itself, next to testing the application or data being packaged. There are some tools available to aid you, lintian for example, but by design they can't find every mistake or bug in the package.

- Maintenance

A Backporter just needs to monitor the package sources he backported and as they change re-apply his changes, build and distribute the package again. He may do so because he can rely on the package maintainer keeping the package itself in proper constitution.

A package maintainer has to monitor the original data sources and if they change, test them (maybe they wouldn't even compile), adopt the changes, test again and often correct bugs in his package sources originating by the upstream changes (configuration changes, build-time options and so on). In addition, he could have to operate and maintain his own bug tracking system to coordinate bugs related to his package as well as bugs related to the upstream sources.

You see, there's a lot more work to be done if you're actively maintaining packages than you'd have to if you're backporting existing packages.

With the Debian New Maintainers Guide there's a good beginner's documentation available. For those already familiar with the fundamentals of package creation the Debian Developers Reference offers a lot of tips and tricks as well as best practices.

4.2 The Repository

Backporting or customizing existing Debian Packages or even creating new Debian packages from scratch is oftenly quite easy. But what to do when you have all those nice packages at your DevEnv's workstation? You need to bring them to your organization's computers in any way.

Building and using a repository is the solution of choice. A Debian Repository is a well-formed structured storage for Debian Packages, both Source and Binary Packages. It consists of the "Pool" and a control structure, both visibly represented by the directory structure of the Repository. In the previous Chapter I spoke about the structure of Debian Repositories, so I'll omit the more general topics and focus on the implementation specific parts.

4.2.1 Strategies

It is important to choose a strategy on how to operate a private repository of Debian Packages since a later change in the implementation after a time of operations can lead to disproportional expenses. There are two main strategies to choose between, each of them flexible enough to be customized to meet each organization's needs:

- One private repository containing everything needed in your organization's environment, a mixture of the official Debian mirrors and your own packages
- One private repository containing only your packages, any other packages are fetched from their respective public repositories, usually done through an apt-proxy

Both of them have strong advantages and disadvantages. The former gives you fine-grained control over the packages you deploy and thus on what data will be installed on the client machines (it would even possible to specify certain repositories to specific machines) but it involves a continuous synchronization with the external repositories (especially the official ones since you probably don't want to miss the security updates) and incorporation of their contents into your repository.

The latter relies on the official repositories as the main package source(s) and focusses on providing your own packages to the client machines thus avoiding the need to maintain incorporation of external repositories content. The main disadvantages of this strategy are less control over the contents (compared to the former strategy), a strong need of name and version control (to avoid collisions with packages in external repositories) and the dependency on the reliability and structure of the external repositories.

As usual, it's up to you to find the appropriate strategy depending on both requirements and needs of your organization. There are several aspects to take into account, expenses, security, control, reliability, availability, consistency and the like. And the assigned values of each and every aspect differ from organization to organization.

For the sample repository being built I assume that it will be used to hold and provide several backports or self-built Binary and Source Packages. In addition the official Debian mirrors will be used directly for all other packages. This assumption may be slightly off reality but it is sufficient to show how to build and operate a private repository.

4.2.2 The Setup

There are three tasks to accomplish in order to set up a repository:

1. Build the Repository's directory structure
2. Place the packages in their corresponding directories
3. Build the local Information-DB

During normal operation, steps 2 and 3 have to be processed each time new or altered packages have to be incorporated into the repository.

Depending on the repository strategy you chose you could use a tool like apt-move or mini-dinstall to accomplish this task, I will do this for now using a more manual way:

```
# for debdir in main contrib non-free
do
  install -o root -g staff -m 755 -d /var/local/debian/pool/$debdir
  install -o root -g staff -m 755 -d /var/local/debian/dists/woody/$debdir/binary-i386
  install -o root -g staff -m 755 -d /var/local/debian/dists/woody/$debdir/source
done; \
\
cp -a /var/local/debian/dists/woody /var/local/debian/dists/sarge; \
cp -a /var/local/debian/dists/woody /var/local/debian/dists/sid; \
\
ln -s /var/local/debian/dists/woody /var/local/debian/dists/stable; \
ln -s /var/local/debian/dists/sarge /var/local/debian/dists/testing; \
ln -s /var/local/debian/dists/sid /var/local/debian/dists/unstable; \
\
install -o root -g staff -m 755 -d /var/local/debian/indices
# _
```

Note that I (intentionally) left out the non-US section. You could either incorporate it into the main directory structure (as implicitly done in the example) or in its own directory structure which the probably would be located in "pool/non-US" and "dists/<release>/non-US" respectively. Technically there is no reason to do it either way, but there may be legal reasons to do so (probably the same that forced the Debian Project to distinguish non-US from the other sections).

Step 1 is accomplished, the basic directory structure is built and ready to accept contents. Of course the subdirectory structure below "pool/<section>" needs to be built but I suggest to create these directories as they are needed. Since we already produced a number of packages we will place them in their corresponding directories:

```
# install -o root -g staff -m 755 /var/local/debian/pool/main/o/openldap2
# install -o root -g staff -m 755 /var/local/debian/pool/main/libi/libiodbc2
```

```
# mv openldap2_2.0.23-6.backport-1.{dsc,diff.gz} openldap2_2.0.23.orig.tar.gz \
*ldap*.deb slapd*.deb /var/local/debian/pool/main/o/openldap2/
# mv libiodbc2_3.0.5-3.backport-1.{dsc,diff.gz} libiodbc2_3.0.5.orig.tar.gz \
libiodbc2*.deb /var/local/debian/pool/main/libi/libiodbc2/
# _
```

Having done so, we accomplished Step 2. The repository is "fed" and we can head on to Step 3, the creation of the repository's local Information-DB.

4.2.3 The Information-DB

The repository's local Information-DB is scattered over several files (paths and names are taken from an official Debian Archive):

- "debian/dists/<release>/Contents-<arch>(gz)
Contains table of files and in which packages they can be found
- "debian/dists/<release>/Release
Contains information about the release and MD5 checksums and SHA1 signatures of the Index files
- "debian/dists/<release>/<section>/binary-<arch>/Packages(gz)
Contains detailed information of each Binary Package in this branch
- "debian/dists/<release>/<section>/binary-<arch>/Release
Contains release information
- "debian/dists/<release>/<section>/source/Release"
Contains release information, similar to "binary-<arch>/Release"
- "debian/dists/<release>/<section>/source/Sources.gz"
Contains detailed information of each Source Package in this Branch, similar to "binary-<arch>/Packages.gz"
- "debian/indices/override.<release>.<section>.gz
Contains specific control information that overrides the corresponding one in the Binary Package
- "debian/indices/Maintainers(gz)"
Contains a listing of each package and its maintainer
- "debian/indices/md5sums.gz"
Contains a listing of each file in the Debian Archive and its corresponding MD5 hash

Most of these files are optional in order to achieve a working repository, only the index files "Packages.gz" and "Sources.gz" and the "Release" files (except "debian/dists/<release>/Release" which is a special file) are needed. All the other files are for control and other purposes.

Just to demonstrate the process of the local Information-DB creation we will create the "override", "Packages.gz", "Sources.gz" and the "Release" files manually even though there's a tool designed for this very purpose available: `apt-ftparchive`, part of the `apt-utils` package.

As already mentioned, the "override" file contains information supposed to override the corresponding information in the Binary Package itself: "While most information about a package can be found in the control file, some must be filled in by the distribution czars rather than by the maintainer, because they relate to the arrangement of files for release rather than the actual dependencies and description of the package. This information is found in the override file." (man 8 `dpkg-scanpackages`)

Each entry in the "override" file consists of three or four blank delimited fields: "package", "priority", "section" and "maintainerinfo" (optional). We will use all four fields because we want to ensure, that we as the repository's operator are responsible for the packages (even though we probably will forward to the Debian Maintainer any bugs filed).

To create the override file (which will be placed in `"/var/local/debian/indices/override.binary"` for the Binary Packages and in `"/var/local/debian/indices/override.sources"` for the Source Packages) we have to collect the names of the packages first, and assign appropriate values to the three remaining fields afterwards. The same will act as the override file for the Source Packages (but this behavior may . For example, our `override.binary` file would look like this:

```
# debian/indices/override.binary -- override file
#
# package      priority      section      maintainerinfo
ldap-gateways  extra         net          Daniel E. Atencio Psille <dea@atencio.de>
ldap-utils    extra         net          Daniel E. Atencio Psille <dea@atencio.de>
libiodbc2     extra         libs         Daniel E. Atencio Psille <dea@atencio.de>
libiodbc2-dev extra         devel       Daniel E. Atencio Psille <dea@atencio.de>
libldap2      extra         libs         Daniel E. Atencio Psille <dea@atencio.de>
libldap2-dev  extra         devel       Daniel E. Atencio Psille <dea@atencio.de>
slapd         extra         net          Daniel E. Atencio Psille <dea@atencio.de>
```

Except the maintainer info which we assigned the name and email address of the repository operator (or whoever may be responsible in your organization) all original informations were kept. As we don't want to create a new Debian based distribution but just customize or add a few (or a lot of) packages there's no need to alter the assigned section or priority of any package.

Next, we will create the `"Packages.gz"` and `SSources.gz` index files. These files are the ones that control the behavior of the clients by pointing them to the correct files and providing all the information the client-side software needs to choose and fetch the packages. To create them we use `"dpkg-scanpackages"` for the Binary Packages and `"dpkg-scansources"` for the Source Packages. They both accept basically the same options and do the same things: They scan the packages for specific information, collect it and print it out in a defined format. They both expect two mandatory options, the root of the pool relative to the root of the server's root (I'll explain this in a minute, for now this is the `"debian/"` you see in your `sources.list` appended to the server's URI) and the path to the override file. By default, both tools will print their output to `stdout` so we have to redirect it into the corresponding files. To simplify the operation, it's best to `cd` into the repositories' root (i.e. `/var/local/debian`) and perform the operations relative to this path:

```
# cd /var/local/debian
# dpkg-scanpackages pool/main indices/override.binary > ./Packages
* Unconditional maintainer override for ldap-gateways *
!! Package ldap-gateways has 'Section: net', but file is in 'o/openldap2' !!
* Unconditional maintainer override for ldap-utils *
!! Package ldap-utils has 'Section: net', but file is in 'o/openldap2' !!
* Unconditional maintainer override for libiodbc2 *
!! Package libiodbc2 has 'Section: libs', but file is in 'libi/libiodbc2' !!
* Unconditional maintainer override for libiodbc2-dev *
!! Package libiodbc2-dev has 'Section: devel', but file is in 'libi/libiodbc2' !!
* Unconditional maintainer override for libldap2 *
!! Package libldap2 has 'Section: libs', but file is in 'o/openldap2' !!
* Unconditional maintainer override for libldap2-dev *
!! Package libldap2-dev has 'Section: devel', but file is in 'o/openldap2' !!
* Unconditional maintainer override for slapd *
!! Package slapd has 'Section: net', but file is in 'o/openldap2' !!
```

Wrote 7 entries to output Packages file.

```
# gzip -9 Packages && mv Packages.gz dists/woody/main/binary-i386/
# dpkg-scansources pool/main indices/override.binary > ./Sources
```

```
# gzip -9 Sources && mv Sources.gz dists/woody/main/source/
# _
```

The warnings are issued because dpkg-scantools checks the overridden Section value against the sub-directory it finds the corresponding package. They do not affect the functionality of the archive we're building.

To complete the local Information-DB we just have to create the "Release" files in "dists/woody/main/binary-i386" and in "dists/woody/main/source" (because we don't have any packages in the other sections). To create those files we open the favorite editor and type the appropriate information manually:

```
# vi dists/woody/main/binary-i386/Release
Archive: stable
Version: 3.0r2
Component: main
Origin: atencio.de
Label: atencio.de
Architecture: i386
# vi dists/woody/main/source/Release
Archive: stable
Version: 3.0r2
Component: main
Origin: atencio.de
Label: atencio.de
Architecture: source
# _
```

The names of the fields are quite self-explanatory. Basically they identify the release ("Archive"), the section ("Component") and – these are the fields you will have to customize to represent your organization – the creator ("Origin") and his or his organization's name ("Label").

If you want some extra reliability to your repository you may create another "Release" file in "dists/<release>". This one looks similar like the ones we just created ("Archive" exchanged by "Suite", "Component" extended to "Components", new fields "Description" and "Codename"), but contains additional MD5 and SHA1 checksums for all files below "dists/release" thus providing a facility to verify the files' integrity. I recommend you to have a look at the original Debian Release files at one of the official mirrors, below is a "quick and dirty" example with only MD5 checksums:

```
# cat dists/woody/Release
Origin: Debian
Label: Debian
Suite: stable
Version: 3.0r2
Codename: woody
Date: Thu, 20 Nov 2003 18:57:17 UTC
Architectures: alpha arm hppa i386 ia64 m68k mips mipsel powerpc s390 sparc
Components: main contrib non-free
Description: Debian 3.0r2 Released 20th November 2003
MD5Sum:
 6b3c8358c71d257cadf6aa8d0a7cfc9d dists/woody/main/binary-i386/Packages.gz
 5a47fb8ba4d769a5c2b29celf1e3cf59 dists/woody/main/binary-i386/Release
 54c73dc6bd08782c4d168d7f4bd18318 dists/woody/main/source/Sources.gz
 9ff49c6bflab3c7048486bbd18f49c48 dists/woody/main/source/Release
#_
```

Now we're finished with the setup of the repository. All we need is a service to use for transmission. The most convenient ways are to use either a ftp server or a http server (or both) with their roots pointing to (in this example) `"/var/local/"`, respectively.

4.3 The Clients

There's not very much to be done at client side, just adjust the configuration file `"/etc/apt/sources.list"` to point to your repository. Depending on the strategy you chose when you built your repository you may either have to replace or expand the list of repositories with your repository. That's it, the next time either `apt-get`, `dselect`, `synaptic` or the like is run it will use your repository.

5 Summary

Usually, organizations require their hard- and software to be "cost-effective" (indeed, they're usually demanding that soft- and hardware shall do all kind of jobs at no or nearly no cost). The same applies to management infrastructure, whether it's about perimeter management, asset management or software management.

Regarding software management, this means that a software management infrastructure should accomplish at least the most important or valuable "core" tasks at as less cost as possible. In addition, the (monetary) balance between the "cheap and frequent core tasks and the less frequent but expensive optional tasks should be at least even.

The tools and methods developed and provided by the Debian Project and its voluntary contributors may provide valuable help in accomplishing the management's requirements without missing the organization's needs. The flexibility of the Debian Packages and repository infrastructure gives an organization a bunch of possibilities in managing the software environment.

6 Acknowledgements

I would like to thank all those who helped and supported me but especially the following:

- Alexander Winston for proof-reading
- Martin "JoeySSchulze for his support
- Last year's audience: Their interest in backporting motivated me to offer this year's proposal
- Brian May: He gave me valuable hints on katie
- The entire `debianhowto.de` team

7 Appendix

Here's a not at all complete and comprehensive list of resources in relation to this paper's topic.

7.1 Web Resources

Here are some hopefully useful resources in the web.

7.1.1 Debian Pages

Debian web pages:

- The Debian FAQ: <http://www.debian.org/doc/FAQ/> <<http://www.debian.org/doc/FAQ/>>
- Short Overview of Debian Distributions: <http://www.debian.org/releases/> <<http://www.debian.org/releases/>>

- Shell skript to check the integrity of Debian Packages: <http://people.debian.org/~ajt/apt-check-sigs> <<http://people.debian.org/~ajt/apt-check-sigs>>
- The Debian Policy: <http://www.debian.org/doc/debian-policy/> <<http://www.debian.org/doc/debian-policy/>>
- A Brief History of Debian: <http://www.debian.org/doc/manuals/project-history/> <<http://www.debian.org/doc/manuals/project-history/>>
- The Debian Constitution: <http://www.debian.org/devel/constitution> <<http://www.debian.org/devel/constitution>>
- The "testing" Pages: <http://www.debian.org/devel/testing> <<http://www.debian.org/devel/testing>>
- debconf Specifications: http://www.debian.org/doc/packaging-manuals/debconf_specification.html <http://www.debian.org/doc/packaging-manuals/debconf_specification.html>
- Debian Installation Manual: <http://www.debian.org/releases/stable/installmanual> <<http://www.debian.org/releases/stable/installmanual>>
- Lintian Site: <http://lintian.debian.org/> <<http://lintian.debian.org/>>
- Lintian Manual: <http://lintian.debian.org/manual/index.html> <<http://lintian.debian.org/manual/index.html>>

7.1.2 Other Debian Related Pages

Third Party Web Resources:

- Ian Murdock's Announcement: <http://groups.google.com/groups?selm=CBusDD.MIK%40unix.portal.com> <<http://groups.google.com/groups?selm=CBusDD.MIK%40unix.portal.com>>
- Ian Murdock's Article at Debian's 10th Anniversary: <http://www.linuxplanet.com/linuxplanet/editorials/4959/1/> <<http://www.linuxplanet.com/linuxplanet/editorials/4959/1/>>
- Debian Anwenderhandbuch (de): <http://www.debiananwenderhandbuch.de/> <<http://www.debiananwenderhandbuch.de/>>
- Vortrag "Backporting & Repositories" (de), DebianDay @ LinuxTag 2003: <http://www.atencio.de/lt2k3/> <<http://www.atencio.de/lt2k3/>>
- debianhowto.de, HOWTOs about Debian on servers (en, de): <http://www.debianhowto.de/> <<http://www.debianhowto.de/>>
- Short (and erroneous) description of katie configuration: <http://lists.debian.org/debian-devel/2002/debian-devel-200212/msg00735.html> <<http://lists.debian.org/debian-devel/2002/debian-devel-200212/msg00735.html>>
- Debian Repository HOWTO: <http://www.isotton.com/debian/docs/repository-howto/> <<http://www.isotton.com/debian/docs/repository-howto/>>
- My LinuxTag pages: <http://www.atencio.de/lt2k3> <<http://www.atencio.de/lt2k3/>> / and <http://www.atencio.de/lt2k4/> <<http://www.atencio.de/lt2k4/>>

7.1.3 Debian History

Historical stuff about Ian Murdock, Debian, etc.:

- Ian Murdock, Aug. 1993, Announcement of Debian GNU/Linux: <http://groups.google.com/groups?selm=CBusDD.MIK%40unix.portal.com>
- Ian Murdock, Oct. 1994, "Overview Of The Debian GNU/Linux System": <http://www.linuxjournal.com/article.php?sid=2841>
- Ian Murdock, Aug. 2003, "Debian: A Brief Retrospective": <http://www.linuxplanet.com/linuxplanet/editorials/4959/2/>

7.1.4 Backports

"Ready-To-Use" Backports:

- [apt-get.org](http://www.apt-get.org/), Directory of private Backports with search: <http://www.apt-get.org/>
- [backports.org](http://www.backports.org/), Several Backports, ready to use with apt, maintained by Norbert Trekowski (de): <http://www.backports.org/>

7.2 Further Reading

Here are some printed resources:

- Ganten, Peter H., Alex, Wulf, 2. überarb. Aufl. 2004, "Debian GNU/Linux-PowerPack", Springer Verlag Berlin/Heidelberg, ISBN: 3-540-43250-7, <http://www.springeronline.com/sgw/cda/frontpage/0,10735,1-146-22-2301081-0,00.html>