Chapter

5

Packaging System: Tools and Fundamental Principles

Contents

As a Debian system administrator, you will routinely handle `.deb` packages, since they contain consistent functional units (applications, documentation, etc.), whose installation and maintenance they facilitate. It is therefore a good idea to know what they are and how to use them.

This chapter describes the structure and contents of "binary" and "source" packages. The former are `.deb` files, directly usable by `dpkg`, while the latter contain the source code, as well as instructions for building binary packages.

5.1. Structure of a Binary Package

The Debian package format is designed so that its content may be extracted on any Unix system that has the classic commands `ar`, `tar`, and `gzip` (sometimes `xz` or `bzip2`). This seemingly trivial property is important for portability and disaster recovery.

Imagine, for example, that you mistakenly deleted the `dpkg` program, and that you could thus no longer install Debian packages. `dpkg` being a Debian package itself, it would seem your system would be done for… Fortunately, you know the format of a package and can therefore download the `.deb` file of the *dpkg* package and install it manually (see sidebar "dpkg, APT and ar" page 72). If by some misfortune one or more of the programs `ar`, `tar` or `gzip/xz/bzip2` have disappeared, you will only need to copy the missing program from another system (since each of these operates in a completely autonomous manner, without dependencies, a simple copy will suffice). If your system suffered some even more outrageous fortune, and even these don't work (maybe the deepest system libraries are missing?), you should try the static version of `busybox` (provided in the *busybox-static* package), which is even more self-contained, and provides subcommands such as `busybox ar`, `busybox tar` and `busybox gunzip`.

TOOLS

dpkg, APT and ar

`dpkg` is the program that handles `.deb` files, notably extracting, analyzing, and unpacking them.

APT is a group of programs that allows the execution of higher-level modifications to the system: installing or removing a package (while keeping dependencies satisfied), updating the system, listing the available packages, etc.

As for the `ar` program, it allows handling files of the same name: `ar t` *`archive`* displays the list of files contained in such an archive, `ar x` *`archive`* extracts the files from the archive into the current working directory, `ar d` *`archive file`* deletes a file from the archive, etc. Its man page (`ar(1)`) documents all its other features. `ar` is a very rudimentary tool that a Unix administrator would only use on rare occasions, but admins routinely use `tar`, a more evolved archive and file management program. This is why it is easy to restore `dpkg` in the event of an erroneous deletion. You would only have to download the Debian package and extract the content from the `data.tar.gz` archive in the system's root (`/`):

```
# ar x dpkg_1.17.23_amd64.deb
# tar -C / -p -xzf data.tar.gz
```

BACK TO BASICS

Man page notation

It can be confusing for beginners to find references to "`ar(1)`" in the literature. This is generally a convenient means of referring to the man page entitled `ar` in section 1.

Sometimes this notation is also used to remove ambiguities, for example to distinguish between the `printf` command that can also be indicated by `printf(1)` and the `printf` function in the C programming language, which can also be referred to as `printf(3)`.

chapter 7, "Solving Problems and Finding Relevant Information" page 134 discusses manual pages in further detail (see section 7.1.1, "Manual Pages" page 134).

Have a look at the content of a `.deb` file:

```
$ ar t dpkg_1.17.23_amd64.deb
debian-binary
control.tar.gz
data.tar.gz
$ ar x dpkg_1.17.23_amd64.deb
$ ls
control.tar.gz  data.tar.gz  debian-binary  dpkg_1.17.23_amd64.deb
$ tar tzf data.tar.gz | head -n 15
./
./var/
./var/lib/
./var/lib/dpkg/
./var/lib/dpkg/parts/
./var/lib/dpkg/info/
./var/lib/dpkg/alternatives/
./var/lib/dpkg/updates/
./etc/
./etc/logrotate.d/
./etc/logrotate.d/dpkg
./etc/dpkg/
./etc/dpkg/dpkg.cfg.d/
./etc/dpkg/dpkg.cfg
./etc/alternatives/
$ tar tzf control.tar.gz
./
./conffiles
./postinst
./md5sums
./prerm
./preinst
./control
./postrm
$ cat debian-binary
2.0
```

As you can see, the `ar` archive of a Debian package is comprised of three files:

- `debian-binary`. This is a text file which simply indicates the version of the `.deb` file used (in 2015: version 2.0).
- `control.tar.gz`. This archive file contains all of the available meta-information, like the name and version of the package. Some of this meta-information allows package management tools to determine if it is possible to install or uninstall it, for example according to the list of packages already on the machine.
- `data.tar.gz`. This archive contains all of the files to be extracted from the package; this is where the executable files, documentation, etc., are all stored. Some packages may use other compression formats, in which case the file will be named differently (`data.tar.bz2` for bzip2, `data.tar.xz` for XZ).

5.2. Package Meta-Information

The Debian package is not only an archive of files intended for installation. It is part of a larger whole, and it describes its relationship with other Debian packages (dependencies, conflicts, suggestions). It also provides scripts that enable the execution of commands at different stages in the package's lifecycle (installation, removal, upgrades). These data are used by the package management tools but are not part of the packaged software; they are, within the package, what is called its "meta-information" (information about other information).

5.2.1. Description: the `control` File

This file uses a structure similar to email headers (as defined by RFC 2822). For example, for *apt*, the `control` file looks like the following:

```
$ apt-cache show apt
Package: apt
Version: 1.0.9.6
Installed-Size: 3788
Maintainer: APT Development Team <deity@lists.debian.org>
Architecture: amd64
Replaces: manpages-it (<< 2.80-4~), manpages-pl (<< 20060617-3~), openjdk-6-jdk (<< 6
       b24-1.11-0ubuntu1~), sun-java5-jdk (>> 0), sun-java6-jdk (>> 0)
Depends: libapt-pkg4.12 (>= 1.0.9.6), libc6 (>= 2.15), libgcc1 (>= 1:4.1.1), libstdc
       ++6 (>= 4.9), debian-archive-keyring, gnupg
Suggests: aptitude | synaptic | wajig, dpkg-dev (>= 1.17.2), apt-doc, python-apt
Conflicts: python-apt (<< 0.7.93.2~)
Breaks: manpages-it (<< 2.80-4~), manpages-pl (<< 20060617-3~), openjdk-6-jdk (<< 6
       b24-1.11-0ubuntu1~), sun-java5-jdk (>> 0), sun-java6-jdk (>> 0)
Description-en: commandline package manager
 This package provides commandline tools for searching and
 managing as well as querying information about packages
 as a low-level access to all features of the libapt-pkg library.
 .
 These include:
```

```
  * apt-get for retrieval of packages and information about them
    from authenticated sources and for installation, upgrade and
    removal of packages together with their dependencies
  * apt-cache for querying available information about installed
    as well as installable packages
  * apt-cdrom to use removable media as a source for packages
  * apt-config as an interface to the configuration settings
  * apt-key as an interface to manage authentication keys
Description-md5: 9fb97a88cb7383934ef963352b53b4a7
Tag: admin::package-management, devel::lang:ruby, hardware::storage,
 hardware::storage:cd, implemented-in::c++, implemented-in::perl,
 implemented-in::ruby, interface::commandline, network::client,
 protocol::ftp, protocol::http, protocol::ipv6, role::program,
 role::shared-lib, scope::application, scope::utility, sound::player,
 suite::debian, use::downloading, use::organizing, use::searching,
 works-with::audio, works-with::software:package, works-with::text
Section: admin
Priority: important
Filename: pool/main/a/apt/apt_1.0.9.6_amd64.deb
Size: 1107560
MD5sum: a325ccb14e69fef2c50da54e035a4df4
SHA1: 635d09fcb600ec12810e3136d51e696bcfa636a6
SHA256: 371a559ce741394b59dbc6460470a9399be5245356a9183bbeea0f89ecaabb03
```

BACK TO BASICS

RFC — Internet standards

RFC is the abbreviation of "Request For Comments". An RFC is generally a technical document that describes what will become an Internet standard. Before becoming standardized and frozen, these standards are submitted for public review (hence their name). The IETF (Internet Engineering Task Force) decides on the evolution of the status of these documents (proposed standard, draft standard, or standard).

RFC 2026 defines the process for standardization of Internet protocols.

➨ http://www.faqs.org/rfcs/rfc2026.html

Dependencies: the Depends Field

The dependencies are defined in the Depends field in the package header. This is a list of conditions to be met for the package to work correctly — this information is used by tools such as `apt` in order to install the required libraries, in appropriate versions fulfilling the dependencies of the package to be installed. For each dependency, it is possible to restrict the range of versions that meet that condition. In other words, it is possible to express the fact that we need the package *libc6* in a version equal to or greater than "2.15" (written "`libc6 (>=2.15)`"). Version comparison operators are as follows:

- <<: less than;
- <=: less than or equal to;
- =: equal to (note that "2.6.1" is not equal to "2.6.1-1");

- >=: greater than or equal to;

- >>: greater than.

In a list of conditions to be met, the comma serves as a separator. It must be interpreted as a logical "and". In conditions, the vertical bar ("|") expresses a logical "or" (it is an inclusive "or", not an exclusive "either/or"). Carrying greater priority than "and", it can be used as many times as necessary. Thus, the dependency "(A or B) and C" is written `A | B, C`. In contrast, the expression "A or (B and C)" should be written as "(A or B) and (A or C)", since the Depends field does not tolerate parentheses that change the order of priorities between the logical operators "or" and "and". It would thus be written `A | B, A | C`.

➨ `http://www.debian.org/doc/debian-policy/ch-relationships.html`

The dependencies system is a good mechanism for guaranteeing the operation of a program, but it has another use with "meta-packages". These are empty packages that only describe dependencies. They facilitate the installation of a consistent group of programs preselected by the meta-package maintainer; as such, `apt install` *`meta-package`* will automatically install all of these programs using the meta-package's dependencies. The *gnome*, *kde-full* and *linux-image-amd64* packages are examples of meta-packages.

DEBIAN POLICY

Recommends, Suggests, and Enhances fields

The Recommends and Suggests fields describe dependencies that are not compulsory. The "recommended" dependencies, the most important, considerably improve the functionality offered by the package but are not indispensable to its operation. The "suggested" dependencies, of secondary importance, indicate that certain packages may complement and increase their respective utility, but it is perfectly reasonable to install one without the others.

You should always install the "recommended" packages, unless you know exactly why you do not need them. Conversely, it is not necessary to install "suggested" packages unless you know why you need them.

The Enhances field also describes a suggestion, but in a different context. It is indeed located in the suggested package, and not in the package that benefits from the suggestion. Its interest lies in that it is possible to add a suggestion without having to modify the package that is concerned. Thus, all add-ons, plug-ins, and other extensions of a program can then appear in the list of suggestions related to the software. Although it has existed for several years, this last field is still largely ignored by programs such as `apt` or `synaptic`. Its purpose is for a suggestion made by the Enhances field to appear to the user in addition to the traditional suggestions — found in the Suggests field.

DEBIAN POLICY

`Pre-Depends`, a more demanding `Depends`

"Pre-dependencies", which are listed in the "`Pre-Depends`" field in the package headers, complete the normal dependencies; their syntax is identical. A normal dependency indicates that the package in question must be unpacked and configured before configuration of the package declaring the dependency. A pre-dependency stipulates that the package in question must be unpacked and configured before execution of the pre-installation script of the package declaring the pre-dependency, that is before its installation.

A pre-dependency is very demanding for `apt`, because it adds a strict constraint on the ordering of the packages to install. As such, pre-dependencies are discouraged unless absolutely necessary. It is even recommended to consult other developers on debian-devel@lists.debian.org before adding a pre-dependency. It is generally possible to find another solution as a work-around.

Conflicts: the Conflicts field

The Conflicts field indicates when a package cannot be installed simultaneously with another. The most common reasons for this are that both packages include a file of the same name, or provide the same service on the same TCP port, or would hinder each other's operation.

`dpkg` will refuse to install a package if it triggers a conflict with an already installed package, except if the new package specifies that it will "replace" the installed package, in which case `dpkg` will choose to replace the old package with the new one. `apt` always follows your instructions: if you choose to install a new package, it will automatically offer to uninstall the package that poses a problem.

Incompatibilities: the Breaks Field

The Breaks field has an effect similar to that of the Conflicts field, but with a special meaning. It signals that the installation of a package will "break" another package (or particular versions of it). In general, this incompatibility between two packages is transitory, and the Breaks relationship specifically refers to the incompatible versions.

`dpkg` will refuse to install a package that breaks an already installed package, and `apt` will try to resolve the problem by updating the package that would be broken to a newer version (which is assumed to be fixed and, thus, compatible again).

This type of situation may occur in the case of updates without backwards compatibility: this is the case if the new version no longer functions with the older version, and causes a malfunction in another program without making special provisions. The Breaks field prevents the user from running into these problems.

Provided Items: the Provides Field

This field introduces the very interesting concept of a "virtual package". It has many roles, but two are of particular importance. The first role consists in using a virtual package to associate a

generic service with it (the package "provides" the service). The second indicates that a package completely replaces another, and that for this purpose it can also satisfy the dependencies that the other would satisfy. It is thus possible to create a substitution package without having to use the same package name.

VOCABULARY

Meta-package and virtual package

It is essential to clearly distinguish meta-packages from virtual packages. The former are real packages (including real `.deb` files), whose only purpose is to express dependencies.

Virtual packages, however, do not exist physically; they are only a means of identifying real packages based on common, logical criteria (service provided, compatibility with a standard program or a pre-existing package, etc.).

Providing a "Service" Let us discuss the first case in greater detail with an example: all mail servers, such as *postfix* or *sendmail* are said to "provide" the *mail-transport-agent* virtual package. Thus, any package that needs this service to be functional (e.g. a mailing list manager, such as *smartlist* or *sympa*) simply states in its dependencies that it requires a *mail-transport-agent* instead of specifying a large yet incomplete list of possible solutions (e.g. `postfix | sendmail | exim4 | …`). Furthermore, it is useless to install two mail servers on the same machine, which is why each of these packages declares a conflict with the *mail-transport-agent* virtual package. A conflict between a package and itself is ignored by the system, but this technique will prohibit the installation of two mail servers side by side.

DEBIAN POLICY

List of virtual packages

For virtual packages to be useful, everyone must agree on their name. This is why they are standardized in the Debian Policy. The list includes among others *mail-transport-agent* for mail servers, *c-compiler* for C programming language compilers, *www-browser* for web browsers, *httpd* for web servers, *ftp-server* for FTP servers, *x-terminal-emulator* for terminal emulators in graphical mode (`xterm`), and *x-window-manager* for window managers.

The full list can be found on the Web.

➨ `http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt`

Interchangeability with Another Package The Provides field is also interesting when the content of a package is included in a larger package. For example, the *libdigest-md5-perl* Perl module was an optional module in Perl 5.6, and has been integrated as standard in Perl 5.8 (and later versions, such as 5.20 present in *Jessie*). As such, the package *perl* has since version 5.8 declared Provides:libdigest-md5-perl so that the dependencies on this package are met if the user has Perl 5.8 (or newer). The *libdigest-md5-perl* package itself has eventually been deleted, since it no longer had any purpose when old Perl versions were removed.

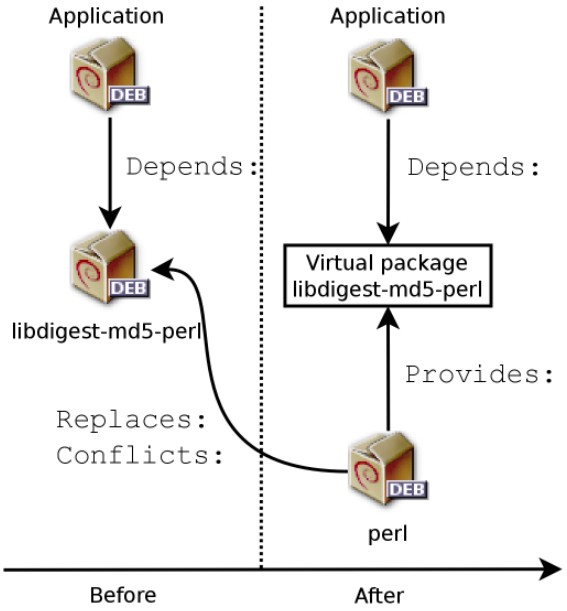

Figure 5.1 *Use of a Provides field in order to not break dependencies*

This feature is very useful, since it is never possible to anticipate the vagaries of development, and it is necessary to be able to adjust to renaming, and other automatic replacement, of obsolete software.

BACK TO BASICS

Perl, a programming language

Perl (Practical Extraction and Report Language) is a very popular programming language. It has many ready-to-use modules that cover a vast spectrum of applications, and that are distributed by the CPAN (Comprehensive Perl Archive Network) servers, an exhaustive network of Perl packages.

➨ `http://www.perl.org/`

➨ `http://www.cpan.org/`

Since it is an interpreted language, a program written in Perl does not require compilation prior to execution. This is why they are called "Perl scripts".

Past Limitations Virtual packages used to suffer from some limitations, the most significant of which was the absence of a version number. To return to the previous example, a dependency such as Depends:libdigest-md5-perl (>=1.6), despite the presence of Perl 5.10, would never be considered as satisfied by the packaging system — while in fact it most likely is satisfied. Unaware of this, the package system chose the least risky option, assuming that the versions do not match.

This limitation has been lifted in *dpkg* 1.17.11, and is no longer relevant in Jessie. Packages can assign a version to the virtual packages they provide with a dependency such as Provides:libdigest-md5-perl (=1.8).

Replacing Files: The Replaces Field

The Replaces field indicates that the package contains files that are also present in another package, but that the package is legitimately entitled to replace them. Without this specification, `dpkg` fails, stating that it can not overwrite the files of another package (technically, it is possible to force it to do so with the --force-overwrite option, but that is not considered standard operation). This allows identification of potential problems and requires the maintainer to study the matter prior to choosing whether to add such a field.

The use of this field is justified when package names change or when a package is included in another. This also happens when the maintainer decides to distribute files differently among various binary packages produced from the same source package: a replaced file no longer belongs to the old package, but only to the new one.

If all of the files in an installed package have been replaced, the package is considered to be removed. Finally, this field also encourages `dpkg` to remove the replaced package where there is a conflict.

GOING FURTHER

The Tag field

In the *apt* example above, we can see the presence of a field that we have not yet described, the Tag field. This field does not describe a relationship between packages, but is simply a way of categorizing a package in a thematic taxonomy. This classification of packages according to several criteria (type of interface, programming language, domain of application, etc.) has been available for a long time. Despite this, not all packages have accurate tags and it is not yet integrated in all Debian tools; `aptitude` displays these tags, and allows them to be used as search criteria. For those who are repelled by `aptitude`'s search criteria, the following website allows navigation of the tag database:

➨ `http://debtags.alioth.debian.org/`

5.2.2. Configuration Scripts

In addition to the `control` file, the `control.tar.gz` archive for each Debian package may contain a number of scripts, called by `dpkg` at different stages in the processing of a package. The Debian Policy describes the possible cases in detail, specifying the scripts called and the arguments that they receive. These sequences may be complicated, since if one of the scripts fails, `dpkg` will try to return to a satisfactory state by canceling the installation or removal in progress (insofar as it is possible).

GOING FURTHER

dpkg's database

All of the configuration scripts for installed packages are stored in the `/var/lib/dpkg/info/` directory, in the form of a file prefixed with the package's name. This directory also includes a file with the `.list` extension for each package, containing the list of files that belong to that package.

The `/var/lib/dpkg/status` file contains a series of data blocks (in the format of the famous mail headers, RFC 2822) describing the status of each package. The information from the `control` file of the installed packages is also replicated there.

In general, the `preinst` script is executed prior to installation of the package, while the `postinst` follows it. Likewise, `prerm` is invoked before removal of a package and `postrm` afterwards. An update of a package is equivalent to removal of the previous version and installation of the new one. It is not possible to describe in detail all the possible scenarios here, but we will discuss the most common two: an installation/update and a removal.

CAUTION **Symbolic names of the scripts**

The sequences described in this section call configuration scripts by specific names, such as `old-prerm` or `new-postinst`. They are, respectively, the `prerm` script contained in the old version of the package (installed before the update) and the `postinst` script contained in the new version (installed by the update).

TIP **State diagrams**

Manoj Srivastava made these diagrams explaining how the configuration scripts are called by `dpkg`. Similar diagrams have also been developed by the Debian Women project; they are a bit simpler to understand, but less complete.

➨ `https://people.debian.org/~srivasta/MaintainerScripts.html`

➨ `https://wiki.debian.org/MaintainerScripts`

Installation and Upgrade

Here is what happens during an installation (or an update):

1. For an update, `dpkg` calls the `old-prerm upgrade` *`new-version`*.
2. Still for an update, `dpkg` then executes `new-preinst upgrade` *`old-version`*; for a first installation, it executes `new-preinst install`. It may add the old version in the last parameter, if the package has already been installed and removed since (but not purged, the configuration files having been retained).
3. The new package files are then unpacked. If a file already exists, it is replaced, but a backup copy is temporarily made.
4. For an update, `dpkg` executes `old-postrm upgrade` *`new-version`*.
5. `dpkg` updates all of the internal data (file list, configuration scripts, etc.) and removes the backups of the replaced files. This is the point of no return: `dpkg` no longer has access to all of the elements necessary to return to the previous state.
6. `dpkg` will update the configuration files, asking the user to decide if it is unable to automatically manage this task. The details of this procedure are discussed in section 5.2.3, "Checksums, List of Configuration Files" page 83.
7. Finally, `dpkg` configures the package by executing `new-postinst configure` *`last-version-configured`*.

Package Removal

Here is what happens during a package removal:

1. `dpkg` calls `prerm remove`.
2. `dpkg` removes all of the package's files, with the exception of the configuration files and configuration scripts.
3. `dpkg` executes `postrm remove`. All of the configuration scripts, except `postrm`, are removed. If the user has not used the "purge" option, the process stops here.
4. For a complete purge of the package (command issued with `dpkg --purge` or `dpkg -P`), the configuration files are also deleted, as well as a certain number of copies (`*.dpkg-tmp`, `*.dpkg-old`, `*.dpkg-new`) and temporary files; `dpkg` then executes `postrm purge`.

VOCABULARY

Purge, a complete removal

When a Debian package is removed, the configuration files are retained in order to facilitate possible re-installation. Likewise, the data generated by a daemon (such as the content of an LDAP server directory, or the content of a database for an SQL server) are usually retained.

To remove all data associated with a package, it is necessary to "purge" the package with the command, `dpkg -P` *`package`*, `apt-get remove --purge` *`package`* or `aptitude purge` *`package`*.

Given the definitive nature of such data removals, a purge should not be taken lightly.

The four scripts detailed above are complemented by a `config` script, provided by packages using `debconf` to acquire information from the user for configuration. During installation, this script defines in detail the questions asked by `debconf`. The responses are recorded in the `debconf` database for future reference. The script is generally executed by `apt` prior to installing packages one by one in order to group all the questions and ask them all to the user at the beginning of the process. The pre- and post-installation scripts can then use this information to operate according to the user's wishes.

TOOL

`debconf`

`debconf` was created to resolve a recurring problem in Debian. All Debian packages unable to function without a minimum of configuration used to ask questions with calls to the `echo` and `read` commands in `postinst` shell scripts (and other similar scripts). But this also means that during a large installation or update the user must stay with their computer to respond to various questions that may arise at any time. These manual interactions have now been almost entirely dispensed with, thanks to the `debconf` tool.

`debconf` has many interesting features: it requires the developer to specify user interaction; it allows localization of all the strings displayed to users (all translations are stored in the `templates` file describing the interactions); it has different frontends to display the questions to the user (text mode, graphical mode, non-interactive); and it allows creation of a central database of responses to share the same configuration with several computers… but the most important is that it is now possible to present all of the questions in a row to the user, prior to starting a long installation or update process. The user can go about their business while the system handles the installation on its own, without having to stay there staring at the screen waiting for questions.

In addition to the maintainer scripts and control data already mentioned in the previous sections, the `control.tar.gz` archive of a Debian package may contain other interesting files. The first, `md5sums`, contains the MD5 checksums for all of the package's files. Its main advantage is that it allows `dpkg --verify` (which we will study in section 14.3.3.1, "Auditing Packages with `dpkg --verify`" page 386) to check if these files have been modified since their installation. Note that when this file doesn't exist, `dpkg` will generate it dynamically at installation time (and store it in the dpkg database just like other control files).

`conffiles` lists package files that must be handled as configuration files. Configuration files can be modified by the administrator, and `dpkg` will try to preserve those changes during a package update.

In effect, in this situation, `dpkg` behaves as intelligently as possible: if the standard configuration file has not changed between the two versions, it does nothing. If, however, the file has changed, it will try to update this file. Two cases are possible: either the administrator has not touched this configuration file, in which case `dpkg` automatically installs the new version; or the file has been modified, in which case `dpkg` asks the administrator which version they wish to use (the old one with modifications, or the new one provided with the package). To assist in making this decision, `dpkg` offers to display a "`diff`" that shows the difference between the two versions. If the user chooses to retain the old version, the new one will be stored in the same location in a file with the `.dpkg-dist` suffix. If the user chooses the new version, the old one is retained in a file with the `.dpkg-old` suffix. Another available action consists of momentarily interrupting `dpkg` to edit the file and attempt to re-instate the relevant modifications (previously identified with `diff`).

GOING FURTHER

Force dpkg to ask configuration file questions

The `--force-confask` option requires `dpkg` to display the questions about the configuration files, even in cases where they would not normally be necessary. Thus, when reinstalling a package with this option, `dpkg` will ask the questions again for all of the configuration files modified by the administrator. This is very convenient, especially for reinstalling the original configuration file if it has been deleted and no other copy is available: a normal re-installation won't work, because `dpkg` considers removal as a form of legitimate modification, and, thus, doesn't install the desired configuration file.

GOING FURTHER

Avoiding the configuration file questions

dpkg handles configuration file updates, but, while doing so, regularly interrupts its work to ask for input from the administrator. This makes it less than enjoyable for those who wish to run updates in a non-interactive manner. This is why this program offers options that allow the system to respond automatically according to the same logic: `--force-confold` retains the old version of the file; `--force-confnew` will use the new version of the file (these choices are respected, even if the file has not been changed by the administrator, which only rarely has the desired effect). Adding the `--force-confdef` option tells dpkg to decide by itself when possible (in other words, when the original configuration file has not been touched), and only uses `--force-confnew` or `--force-confold` for other cases.

These options apply to dpkg, but most of the time the administrator will work directly with the aptitude or apt-get programs. It is, thus, necessary to know the syntax used to indicate the options to pass to the dpkg command (their command line interfaces are very similar).

```
# apt -o DPkg::options::="--force-confdef" -o DPkg::options
       ::="--force-confold" full-upgrade
```

These options can be stored directly in apt's configuration. To do so, simply write the following line in the `/etc/apt/apt.conf.d/local` file:

```
DPkg::options { "--force-confdef"; "--force-confold"; }
```

Including this option in the configuration file means that it will also be used in a graphical interface such as aptitude.

5.3. Structure of a Source Package

5.3.1. Format

A source package is usually comprised of three files, a `.dsc`, a `.orig.tar.gz`, and a `.debian.tar.gz` (or `.diff.gz`). They allow creation of binary packages (`.deb` files described above) from the source code files of the program, which are written in a programming language.

The `.dsc` (Debian Source Control) file is a short text file containing an RFC 2822 header (just like the `control` file studied in section 5.2.1, "Description: the `control` File" page 74) which describes the source package and indicates which other files are part thereof. It is signed by its maintainer, which guarantees authenticity. See section 6.5, "Checking Package Authenticity" page 121 for further details on this subject.

Example 5.1 *A .dsc file*

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

Format: 3.0 (quilt)
Source: zim
```

```
Binary: zim
Architecture: all
Version: 0.62-3
Maintainer: Emfox Zhou <emfox@debian.org>
Uploaders: Raphaël Hertzog <hertzog@debian.org>
Homepage: http://zim-wiki.org
Standards-Version: 3.9.6
Vcs-Browser: http://anonscm.debian.org/gitweb/?p=collab-maint/zim.git
Vcs-Git: git://anonscm.debian.org/collab-maint/zim.git
Build-Depends: debhelper (>= 9), xdg-utils, python (>= 2.6.6-3~), libgtk2.0-0 (>=
       2.6), python-gtk2, python-xdg
Package-List:
 zim deb x11 optional arch=all
Checksums-Sha1:
 ad8de170826682323c10195b65b9f1243fd75637 1772246 zim_0.62.orig.tar.gz
 a4f70d6f7fb404022c9cc4870a4e62ea3ca08388 14768 zim_0.62-3.debian.tar.xz
Checksums-Sha256:
 19d62aebd2c1a92d84d80720c6c1dcdb779c39a2120468fed01b7f252511bdc2 1772246 zim_0.62.
       orig.tar.gz
 fc2e827e83897d5e33f152f124802c46c3c01c5158b75a8275a27833f1f6f1de 14768 zim_0.62-3.
       debian.tar.xz
Files:
 43419efba07f7086168442e3d698287a 1772246 zim_0.62.orig.tar.gz
 725a69663a6c2961f07673ae541298e4 14768 zim_0.62-3.debian.tar.xz

-----BEGIN PGP SIGNATURE-----
Version: GnuPG v2
Comment: Signed by Raphael Hertzog

iQEcBAEBCAAGBQJUR2jqAAoJEAOIHavrwpq5WFcH/RsdzCHc1oXXxHitU23hEqMj
T6ok29M1UFDJDowMXW75jQ1nT4WPUtvEGygkCHeoO/PvjEvB0sjU8GQlX+N9ddSB
aHfqfAYmVhADNGxrXQT5inZXUa8qGeeq2Sqf6YcWtsnuD56lDbvxkyf/XYopoIEl
oltfl05z/AI+vYsW482YrCz0fxNAKAvkyuPhDebYI8jnKWeAANoqmKpsNc/HYyvT
+ZiA5o57OiGdOKT6XGy3/FiF3dkHiRY8lXW7xdr1BbIgulwl9UmiUNwuxwOYbQO7
edtjiTJqOaFUA0x1zB/XGv5tHr1MjP8naT+kfVoVHTOox51CDbeu5D3DZY4imcY=
=Wtoa
-----END PGP SIGNATURE-----
```

Note that the source package also has dependencies (Build-Depends) completely distinct from those of binary packages, since they indicate tools required to compile the software in question and construct its binary package.

CAUTION

Distinct namespaces

It is important to note here that there is no required correspondence between the name of the source package and that of the binary package(s) that it generates. It is easy enough to understand if you know that each source package may generate several binary packages. This is why the `.dsc` file has the `Source` and `Binary` fields to explicitly name the source package and store the list of binary packages that it generates.

CULTURE

Why divide into several packages

Quite frequently, a source package (for a given software) can generate several binary packages. The split is justified by the possibility to use (parts of) the software in different contexts. Consider a shared library, it may be installed to make an application work (for example, *libc6*), or it can be installed to develop a new program (*libc6-dev* will then be the correct package). We find the same logic for client/server services where we want to install the server part on one machine and the client part on others (this is the case, for example, of *openssh-server* and *openssh-client*).

Just as frequently, the documentation is provided in a dedicated package: the user may install it independently from the software, and may at any time choose to remove it to save disk space. Additionally, this also saves disk space on the Debian mirrors, since the documentation package will be shared amongst all of the architectures (instead of having the documentation duplicated in the packages for each architecture).

PERSPECTIVE

Different source package formats

Originally there was only one source package format. This is the `1.0` format, which associates an `.orig.tar.gz` archive to a `.diff.gz` "debianization" patch (there is also a variant, consisting of a single `.tar.gz` archive, which is automatically used if no `.orig.tar.gz` is available).

Since Debian *Squeeze*, Debian developers have the option to use new formats that correct many problems of the historical format. Format `3.0 (quilt)` can combine multiple upstream archives in the same source package: in addition to the usual `.orig.tar.gz`, supplementary `.orig-`*`component`*`.tar.gz` archives can be included. This is useful with software that is distributed in several upstream components but for which a single source package is desired. These archives can also be compressed with `bzip2` or `xz` rather than `gzip`, which saves disk space and network resources. Finally, the monolithic patch, `.diff.gz` is replaced by a `.debian.tar.gz` archive containing the compiling instructions and a set of upstream patches contributed by the package maintainer. These last are recorded in a format compatible with `quilt` — a tool that facilitates the management of a series of patches.

The `.orig.tar.gz` file is an archive containing the source code as provided by the original developer. Debian package maintainers are asked to not modify this archive in order to be able to easily check the origin and integrity of the file (by simple comparison with a checksum) and to respect the wishes of some authors.

The `.debian.tar.gz` contains all of the modifications made by the Debian maintainer, especially the addition of a `debian` directory containing the instructions to execute to construct a Debian package.

TOOL

Decompressing a source package

If you have a source package, you can use the `dpkg-source` command (from the *dpkg-dev* package) to decompress it:

```
$ dpkg-source -x package_0.7-1.dsc
```

You can also use `apt-get` to download a source package and unpack it right away. It requires that the appropriate `deb-src` lines be present in the `/etc/apt/sources.list` file, however (for further details, see section 6.1, "Filling in the `sources.list` File" page 100). These are used to list the "sources" of source packages (meaning the servers on which a group of source packages are hosted).

```
$ apt-get source package
```

5.3.2. Usage within Debian

The source package is the foundation of everything in Debian. All Debian packages come from a source package, and each modification in a Debian package is the consequence of a modification made to the source package. The Debian maintainers work with the source package, knowing, however, the consequences of their actions on the binary packages. The fruits of their labors are thus found in the source packages available from Debian: you can easily go back to them and everything stems from them.

When a new version of a package (source package and one or more binary packages) arrives on the Debian server, the source package is the most important. Indeed, it will then be used by a network of machines of different architectures for compilation on the various architectures supported by Debian. The fact that the developer also sends one or more binary packages for a given architecture (usually i386 or amd64) is relatively unimportant, since these could just as well have been automatically generated.

5.4. Manipulating Packages with `dpkg`

`dpkg` is the base command for handling Debian packages on the system. If you have `.deb` packages, it is `dpkg` that allows installation or analysis of their contents. But this program only has a partial view of the Debian universe: it knows what is installed on the system, and whatever it is given on the command line, but knows nothing of the other available packages. As such, it will fail if a dependency is not met. Tools such as `apt`, on the contrary, will create a list of dependencies to install everything as automatically as possible.

NOTE

`dpkg` or `apt`?

`dpkg` should be seen as a system tool (backend), and `apt` as a tool closer to the user, which overcomes the limitations of the former. These tools work together, each one with its particularities, suited to specific tasks.

5.4.1. Installing Packages

`dpkg` is, above all, the tool for installing an already available Debian package (because it does not download anything). To do this, we use its -i or --install option.

Example 5.2 *Installation of a package with dpkg*

```
# dpkg -i man-db_2.7.0.2-5_amd64.deb
(Reading database ... 86425 files and directories currently installed.)
Preparing to unpack man-db_2.7.0.2-5_amd64.deb ...
Unpacking man-db (2.7.0.2-5) over (2.7.0.2-4) ...
Setting up man-db (2.7.0.2-5) ...
Updating database of manual pages ...
Processing triggers for mime-support (3.58) ...
```

We can see the different steps performed by `dpkg`; we know, thus, at what point any error may have occurred. The installation can also be effected in two stages: first unpacking, then configuration. `apt-get` takes advantage of this, limiting the number of calls to `dpkg` (since each call is costly, due to loading of the database in memory, especially the list of already installed files).

Example 5.3 *Separate unpacking and configuration*

```
# dpkg --unpack man-db_2.7.0.2-5_amd64.deb
(Reading database ... 86425 files and directories currently installed.)
Preparing to unpack man-db_2.7.0.2-5_amd64.deb ...
Unpacking man-db (2.7.0.2-5) over (2.7.0.2-5) ...
Processing triggers for mime-support (3.58) ...
# dpkg --configure man-db
Setting up man-db (2.7.0.2-5) ...
Updating database of manual pages ...
```

Sometimes `dpkg` will fail to install a package and return an error; if the user orders it to ignore this, it will only issue a warning; it is for this reason that we have the different --force-* options. The `dpkg --force-help` command, or documentation of this command, will give a complete list of these options. The most frequent error, which you are bound to encounter sooner or later, is a file collision. When a package contains a file that is already installed by another package, `dpkg` will refuse to install it. The following messages will then appear:

```
Unpacking libgdm (from .../libgdm_3.8.3-2_amd64.deb) ...
dpkg: error processing /var/cache/apt/archives/libgdm_3.8.3-2_amd64.deb (--unpack):
 trying to overwrite '/usr/bin/gdmflexiserver', which is also in package gdm3 3.4.1-9
```

In this case, if you think that replacing this file is not a significant risk to the stability of your system (which is usually the case), you can use the option --force-overwrite, which tells `dpkg` to ignore this error and overwrite the file.

While there are many available --force-* options, only --force-overwrite is likely to be used regularly. These options only exist for exceptional situations, and it is better to leave them alone as much as possible in order to respect the rules imposed by the packaging mechanism. Do not forget, these rules ensure the consistency and stability of your system.

CAUTION

Effective use of `--force-*`

If you are not careful, the use of an option `--force-*` can lead to a system where the APT family of commands will refuse to function. In effect, some of these options allow installation of a package when a dependency is not met, or when there is a conflict. The result is an inconsistent system from the point of view of dependencies, and the APT commands will refuse to execute any action except those that will bring the system back to a consistent state (this often consists of installing the missing dependency or removing a problematic package). This often results in a message like this one, obtained after installing a new version of *rdesktop* while ignoring its dependency on a newer version of the *libc6*:

```
# apt full-upgrade
[...]
You might want to run 'apt-get -f install' to correct these
       .
The following packages have unmet dependencies:
  rdesktop: Depends: libc6 (>= 2.5) but 2.3.6.ds1-13etch7
         is installed
E: Unmet dependencies. Try using -f.
```

A courageous administrator who is certain of the correctness of their analysis may choose to ignore a dependency or conflict and use the corresponding `--force-*` option. In this case, if they want to be able to continue to use `apt` or `aptitude`, they must edit `/var/lib/dpkg/status` to delete/modify the dependency, or conflict, that they chose to override.

This manipulation is an ugly hack, and should never be used, except in the most extreme case of necessity. Quite frequently, a more fitting solution is to recompile the package that's causing the problem (see section 15.1, "Rebuilding a Package from its Sources" page 420) or use a new version (potentially corrected) from a repository such as the `stable-backports` one (see section 6.1.2.4, "Stable Backports" page 103).

5.4.2. Package Removal

Invoking `dpkg` with the -r or --remove option, followed by the name of a package, removes that package. This removal is, however, not complete: all of the configuration files, maintainer scripts, log files (system logs) and other user data handled by the package remain. That way disabling the program is easily done by uninstalling it, and it's still possible to quickly reinstall it with the same configuration. To completely remove everything associated with a package, use the -P or --purge option, followed by the package name.

Example 5.4 *Removal and purge of the* debian-cd *package*

```
# dpkg -r debian-cd
(Reading database ... 97747 files and directories currently installed.)
Removing debian-cd (3.1.17) ...
# dpkg -P debian-cd
(Reading database ... 97401 files and directories currently installed.)
Removing debian-cd (3.1.17) ...
Purging configuration files for debian-cd (3.1.17) ...
```

5.4.3. Querying dpkg's Database and Inspecting .deb Files

BACK TO BASICS
Option syntax

Most options are available in a "long" version (one or more relevant words, preceded by a double dash) and a "short" version (a single letter, often the initial of one word from the long version, and preceded by a single dash). This convention is so common that it is a POSIX standard.

Before concluding this section, we will study `dpkg` options that query the internal database in order to obtain information. Giving first the long options and then corresponding short options (that will evidently take the same possible arguments) we cite --listfiles *package* (or -L), which lists the files installed by this package; --search *file* (or -S), which finds the package(s) containing the file; --status *package* (or -s), which displays the headers of an installed package; --list (or -l), which displays the list of packages known to the system and their installation status; --contents *file.deb* (or -c), which lists the files in the Debian package specified; --info *file.deb* (or -I), which displays the headers of this Debian package.

Example 5.5 *Various queries with dpkg*

```
$ dpkg -L base-passwd
/.
/usr
/usr/sbin
/usr/sbin/update-passwd
/usr/share
/usr/share/lintian
/usr/share/lintian/overrides
/usr/share/lintian/overrides/base-passwd
/usr/share/doc-base
/usr/share/doc-base/users-and-groups
/usr/share/base-passwd
/usr/share/base-passwd/group.master
/usr/share/base-passwd/passwd.master
```

```
/usr/share/man
/usr/share/man/pl
/usr/share/man/pl/man8
/usr/share/man/pl/man8/update-passwd.8.gz
/usr/share/man/ru
/usr/share/man/ru/man8
/usr/share/man/ru/man8/update-passwd.8.gz
/usr/share/man/ja
/usr/share/man/ja/man8
/usr/share/man/ja/man8/update-passwd.8.gz
/usr/share/man/fr
/usr/share/man/fr/man8
/usr/share/man/fr/man8/update-passwd.8.gz
/usr/share/man/es
/usr/share/man/es/man8
/usr/share/man/es/man8/update-passwd.8.gz
/usr/share/man/de
/usr/share/man/de/man8
/usr/share/man/de/man8/update-passwd.8.gz
/usr/share/man/man8
/usr/share/man/man8/update-passwd.8.gz
/usr/share/doc
/usr/share/doc/base-passwd
/usr/share/doc/base-passwd/users-and-groups.txt.gz
/usr/share/doc/base-passwd/changelog.gz
/usr/share/doc/base-passwd/copyright
/usr/share/doc/base-passwd/README
/usr/share/doc/base-passwd/users-and-groups.html
$ dpkg -S /bin/date
coreutils: /bin/date
$ dpkg -s coreutils
Package: coreutils
Essential: yes
Status: install ok installed
Priority: required
Section: utils
Installed-Size: 13855
Maintainer: Michael Stone <mstone@debian.org>
Architecture: amd64
Multi-Arch: foreign
Version: 8.23-3
Replaces: mktemp, realpath, timeout
Pre-Depends: libacl1 (>= 2.2.51-8), libattr1 (>= 1:2.4.46-8), libc6 (>= 2.17),
       libselinux1 (>= 2.1.13)
Conflicts: timeout
Description: GNU core utilities
 This package contains the basic file, shell and text manipulation
 utilities which are expected to exist on every operating system.
 .
```

```
 Specifically, this package includes:
 arch base64 basename cat chcon chgrp chmod chown chroot cksum comm cp
 csplit cut date dd df dir dircolors dirname du echo env expand expr
 factor false flock fmt fold groups head hostid id install join link ln
 logname ls md5sum mkdir mkfifo mknod mktemp mv nice nl nohup nproc numfmt
 od paste pathchk pinky pr printenv printf ptx pwd readlink realpath rm
 rmdir runcon sha*sum seq shred sleep sort split stat stty sum sync tac
 tail tee test timeout touch tr true truncate tsort tty uname unexpand
 uniq unlink users vdir wc who whoami yes
Homepage: http://gnu.org/software/coreutils
$ dpkg -l 'b*'
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-aWait/Trig-pend
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name                    Version          Architecture     Description
+++-=======================-================-================-================================

un  backupninja             <none>           <none>           (no description
      available)
ii  backuppc                3.3.0-2          amd64            high-performance,
      enterprise-grade system for backin
un  base                    <none>           <none>           (no description
      available)
un  base-config             <none>           <none>           (no description
      available)
ii  base-files              8                amd64            Debian base system
      miscellaneous files
ii  base-passwd             3.5.37           amd64            Debian base system
      master password and group files
[...]
$ dpkg -c /var/cache/apt/archives/gnupg_1.4.18-6_amd64.deb
drwxr-xr-x root/root         0 2014-12-04 23:03 ./
drwxr-xr-x root/root         0 2014-12-04 23:03 ./lib/
drwxr-xr-x root/root         0 2014-12-04 23:03 ./lib/udev/
drwxr-xr-x root/root         0 2014-12-04 23:03 ./lib/udev/rules.d/
-rw-r--r-- root/root      2711 2014-12-04 23:03 ./lib/udev/rules.d/60-gnupg.rules
drwxr-xr-x root/root         0 2014-12-04 23:03 ./usr/
drwxr-xr-x root/root         0 2014-12-04 23:03 ./usr/lib/
drwxr-xr-x root/root         0 2014-12-04 23:03 ./usr/lib/gnupg/
-rwxr-xr-x root/root     39328 2014-12-04 23:03 ./usr/lib/gnupg/gpgkeys_ldap
-rwxr-xr-x root/root     92872 2014-12-04 23:03 ./usr/lib/gnupg/gpgkeys_hkp
-rwxr-xr-x root/root     47576 2014-12-04 23:03 ./usr/lib/gnupg/gpgkeys_finger
-rwxr-xr-x root/root     84648 2014-12-04 23:03 ./usr/lib/gnupg/gpgkeys_curl
-rwxr-xr-x root/root      3499 2014-12-04 23:03 ./usr/lib/gnupg/gpgkeys_mailto
drwxr-xr-x root/root         0 2014-12-04 23:03 ./usr/bin/
-rwxr-xr-x root/root     60128 2014-12-04 23:03 ./usr/bin/gpgsplit
-rwxr-xr-x root/root   1012688 2014-12-04 23:03 ./usr/bin/gpg
[...]
$ dpkg -I /var/cache/apt/archives/gnupg_1.4.18-6_amd64.deb
```

```
 new debian package, version 2.0.
 size 1148362 bytes: control archive=3422 bytes.
    1264 bytes,    26 lines      control
    4521 bytes,    65 lines      md5sums
     479 bytes,    13 lines   *  postinst             #!/bin/sh
     473 bytes,    13 lines   *  preinst              #!/bin/sh
 Package: gnupg
 Version: 1.4.18-6
 Architecture: amd64
 Maintainer: Debian GnuPG-Maintainers <pkg-gnupg-maint@lists.alioth.debian.org>
 Installed-Size: 4888
 Depends: gpgv, libbz2-1.0, libc6 (>= 2.15), libreadline6 (>= 6.0), libusb-0.1-4
        (>= 2:0.1.12), zlib1g (>= 1:1.1.4)
 Recommends: gnupg-curl, libldap-2.4-2 (>= 2.4.7)
 Suggests: gnupg-doc, libpcsclite1, parcimonie, xloadimage | imagemagick | eog
 Section: utils
 Priority: important
 Multi-Arch: foreign
 Homepage: http://www.gnupg.org
 Description: GNU privacy guard - a free PGP replacement
  GnuPG is GNU's tool for secure communication and data storage.
  It can be used to encrypt data and to create digital signatures.
  It includes an advanced key management facility and is compliant
  with the proposed OpenPGP Internet standard as described in RFC 4880.
[...]
```

GOING FURTHER

Comparison of versions

Since dpkg is the program for handling Debian packages, it also provides the reference implementation of the logic of comparing version numbers. This is why it has a `--compare-versions` option, usable by external programs (especially configuration scripts executed by dpkg itself). This option requires three parameters: a version number, a comparison operator, and a second version number. The different possible operators are `lt` (strictly less than), `le` (less than or equal to), `eq` (equal), ne (not equal), `ge` (greater than or equal to), and `gt` (strictly greater than). If the comparison is correct, dpkg returns 0 (success); if not, it gives a non-zero return value (indicating failure).

```
$ dpkg --compare-versions 1.2-3 gt 1.1-4
$ echo $?
0
$ dpkg --compare-versions 1.2-3 lt 1.1-4
$ echo $?
1
$ dpkg --compare-versions 2.6.0pre3-1 lt 2.6.0-1
$ echo $?
1
```

Note the unexpected failure of the last comparison: for dpkg, `pre`, usually denoting a pre-release, has no particular meaning, and this program compares the alphabetic characters in the same way as the numbers (a < b < c ...), in alphabetical order. This

is why it considers "0pre3" to be greater than "0". When we want a package's version number to indicate that it is a pre-release, we use the tilde character, "~":

```
$ dpkg --compare-versions 2.6.0~pre3-1 lt 2.6.0-1
$ echo $?
0
```

5.4.4. dpkg's Log File

dpkg keeps a log of all of its actions in /var/log/dpkg.log. This log is extremely verbose, since it details every one of the stages through which packages handled by dpkg go. In addition to offering a way to track dpkg's behavior, it helps, above all, to keep a history of the development of the system: one can find the exact moment when each package has been installed or updated, and this information can be extremely useful in understanding a recent change in behavior. Additionally, all versions being recorded, it is easy to cross-check the information with the changelog.Debian.gz for packages in question, or even with online bug reports.

5.4.5. Multi-Arch Support

All Debian packages have an Architecture field in their control information. This field can contain either "all" (for packages that are architecture independent) or the name of the architecture that it targets (like "amd64", "armhf", ...). In the latter case, by default, dpkg will only accept to install the package if its architecture matches the host's architecture as returned by dpkg --print-architecture.

This restriction ensures that users do not end up with binaries compiled for an incorrect architecture. Everything would be perfect except that (some) computers can run binaries for multiple architectures, either natively (an "amd64" system can run "i386" binaries) or through emulators.

Enabling Multi-Arch

dpkg's multi-arch support allows users to define "foreign architectures" that can be installed on the current system. This is simply done with dpkg --add-architecture like in the example below. There is a corresponding dpkg --remove-architecture to drop support of a foreign architecture, but it can only be used when no packages of this architecture remain.

```
# dpkg --print-architecture
amd64
# dpkg --print-foreign-architectures
# dpkg -i gcc-4.9-base_4.9.1-19_armhf.deb
dpkg: error processing archive gcc-4.9-base_4.9.1-19_armhf.deb (--install):
 package architecture (armhf) does not match system (amd64)
Errors were encountered while processing:
```

```
 gcc-4.9-base_4.9.1-19_armhf.deb
# dpkg --add-architecture armhf
# dpkg --add-architecture armel
# dpkg --print-foreign-architectures
armhf
armel
# dpkg -i gcc-4.9-base_4.9.1-19_armhf.deb
Selecting previously unselected package gcc-4.9-base:armhf.
(Reading database ... 86425 files and directories currently installed.)
Preparing to unpack gcc-4.9-base_4.9.1-19_armhf.deb ...
Unpacking gcc-4.9-base:armhf (4.9.1-19) ...
Setting up gcc-4.9-base:armhf (4.9.1-19) ...
# dpkg --remove-architecture armhf
dpkg: error: cannot remove architecture 'armhf' currently in use by the database
# dpkg --remove-architecture armel
# dpkg --print-foreign-architectures
armhf
```

NOTE

APT's multi-arch support

APT will automatically detect when dpkg has been configured to support foreign architectures and will start downloading the corresponding Packages files durings its update process.

Foreign packages can then be installed with `apt install` *`package`*`:`*`architecture`*.

IN PRACTICE

Using proprietary i386 binaries on amd64

There are multiple use cases for multi-arch, but the most popular one is the possibility to execute 32 bit binaries (i386) on 64 bit systems (amd64), in particular since several popular proprietary applications (like Skype) are only provided in 32 bit versions.

Multi-Arch Related Changes

To make multi-arch actually useful and usable, libraries had to be repackaged and moved to an architecture-specific directory so that multiple copies (targeting different architectures) can be installed alongside. Such updated packages contain the "Multi-Arch:same" header field to tell the packaging system that the various architectures of the package can be safely co-installed (and that those packages can only satisfy dependencies of packages of the same architecture). Since multi-arch made its debut in Debian *Wheezy*, not all libraries have been converted yet.

```
$ dpkg -s gcc-4.9-base
dpkg-query: error: --status needs a valid package name but 'gcc-4.9-base' is not:
       ambiguous package name 'gcc-4.9-base' with more than one installed instance

Use --help for help about querying packages.
$ dpkg -s gcc-4.9-base:amd64 gcc-4.9-base:armhf | grep ^Multi
Multi-Arch: same
Multi-Arch: same
```

```
$ dpkg -L libgcc1:amd64 |grep .so
/lib/x86_64-linux-gnu/libgcc_s.so.1
$ dpkg -S /usr/share/doc/gcc-4.9-base/copyright
gcc-4.9-base:amd64, gcc-4.9-base:armhf: /usr/share/doc/gcc-4.9-base/copyright
```

It is worth noting that Multi-Arch:same packages must have their names qualified with their architecture to be unambiguously identifiable. They also have the possibility to share files with other instances of the same package; `dpkg` ensures that all packages have bit-for-bit identical files when they are shared. Last but not least, all instances of a package must have the same version. They must thus be upgraded together.

Multi-Arch support also brings some interesting challenges in the way dependencies are handled. Satisfying a dependency requires either a package marked "Multi-Arch:foreign" or a package whose architecture matches the one of the package declaring the dependency (in this dependency resolution process, architecture-independent packages are assumed to be of the same architecture than the host). A dependency can also be weakened to allow any architecture to fulfill it, with the *package*:any syntax, but foreign packages can only satisfy such a dependency if they are marked "Multi-Arch:allowed".

5.5. Coexistence with Other Packaging Systems

Debian packages are not the only software packages used in the free software world. The main competitor is the RPM format of the Red Hat Linux distribution and its many derivatives. Red Hat is a very popular, commercial distribution. It is thus common for software provided by third parties to be offered as RPM packages rather than Debian.

In this case, you should know that the program `rpm`, which handles RPM packages, is available as a Debian package, so it is possible to use this package format on Debian. Care should be taken, however, to limit these manipulations to extract the information from a package or to verify its integrity. It is, in truth, unreasonable to use `rpm` to install an RPM on a Debian system; RPM uses its own database, separate from those of native software (such as `dpkg`). This is why it is not possible to ensure a stable coexistence of two packaging systems.

On the other hand, the *alien* utility can convert RPM packages into Debian packages, and vice versa.

COMMUNITY

Encouraging the adoption of `.deb`

If you regularly use the `alien` program to install RPM packages coming from one of your providers, do not hesitate to write to them and amicably express your strong preference for the `.deb` format. Note that the format of the package is not everything: a `.deb` package built with `alien` or prepared for a version of Debian different than that which you use, or even for a derivative distribution like Ubuntu, would probably not offer the same level of quality and integration as a package specifically developed for Debian *Jessie*.

```
$ fakeroot alien --to-deb phpMyAdmin-2.0.5-2.noarch.rpm
phpmyadmin_2.0.5-2_all.deb generated
$ ls -s phpmyadmin_2.0.5-2_all.deb
  64 phpmyadmin_2.0.5-2_all.deb
```

You will find that this process is extremely simple. You must know, however, that the package generated does not have any dependency information, since the dependencies in the two packaging formats don't have systematic correspondence. The administrator must thus manually ensure that the converted package will function correctly, and this is why Debian packages thus generated should be avoided as much as possible. Fortunately, Debian has the largest collection of software packages of all distributions, and it is likely that whatever you seek is already in there.

Looking at the man page for the `alien` command, you will also note that this program handles other packaging formats, especially the one used by the Slackware distribution (it is made of a simple `tar.gz` archive).

The stability of the software deployed using the `dpkg` tool contributes to Debian's fame. The APT suite of tools, described in the following chapter, preserves this advantage, while relieving the administrator from managing the status of packages, a necessary but difficult task.

Keywords

apt
apt-get
apt-cache
aptitude
synaptic
sources.list
apt-cdrom