

# Linux package management

Wichert Akkerman

wichert@cistron.nl

July 8, 2000

## Abstract

This paper discuss package management technology for Linux. A brief history of the various methods of software distribution is given, including the development of package managers like dpkg and rpm. Then we discuss the various issues involved in modern package management and the development of a new package format.

## 1 Distribution methods

Over the years there has been a huge improvement in the way that software has been shipped. Software used to be hand compiled, but it is now possible to install and configure most software with a single command.

### 1.1 Source distribution

Originally software was only shipped as source in a tar-archive. If you wanted to install it you had to download the source using FTP, unpack the archive and try to build it (see figure 3). This involved reading the documentation on configuring it, modifying the Makefile and possibly other files, compiling and then installing. This is a tedious process during which it is very easy to make an error that won't be noticed until it is too late. At some point, several people noticed that a lot of similar work being repeatedly. It was decided that a general framework was needed to automate much of this work; this would provide a uniform interface to configure basic options and check the buildsystem: `autoconf` was born. Installing a pro-

gramming that uses `autoconf` is generally simpler (see figure 4): after unpacking the source archive you run the `configure` script with the build time options you want, and it will take care of figuring out what kind of system you are compiling on (and for) and setting the right variables in the right places.

### 1.2 Binary distribution

With more and more software becoming available for Linux and the high speed of development installing software from source became impractical: it is easy to make an error, and due to the rapid release cycles a lot of time needed to be spent just on keeping a system up to date. To resolve this big packages like `gcc`, `libc` and `XFree86` were also distribution in precompiled form. Installing those became a simple matter of downloading the archive and extracting it (see figure 5).

### 1.3 Packages

Although installing using precompiled binaries is very simple, there were still a couple of problems with this method:

- There is no way to keep track of what software is installed
- There might be undetected conflicts between software
- Some software needs other software to run; checking such dependencies can be a tedious job.

To resolve this problems the concept of *packages* and *package management systems* was introduced. A package is a basically an archive that contains both

precompiled binaries ready for installation and meta-data that describes the contents of the archive. A package management system is a system which keeps track of the packages that are installed in the system and can install, upgrade and remove packages.

Thought there have been many advances in package management systems, the various package formats have not change a lot since they were introduced. The first package managers were little more than a wrapper around `tar` which also kept track of which packages were installed, while current systems are capable of upgrading complete systems and fetching packages on demand if needed.

Since packages are not stand-alone objects it became necessary to define relations between them. This led to the definition of *dependencies* and *conflicts*. With a dependency you can state that a package needs another package in order to do its work. An example: in order to use `autoconf` you need to have `m4` and `perl` installed. Conflicts accomplish the opposite: some packages simply can not be installed at the same time. An example is `sendmail` and `smail`: you can not have two mail transfer agents running at the same time on your system. If you install one you will have to remove the previous one.

With the ability to express dependencies and conflicts thing were fine for a while, but after a while a new problem arose: it became increasingly difficult to manage multiple alternatives for a package. Lets take mail transport agents to show what is meant here: some packages need a MTA in order to function (`cron` to send its reports, `inn` to send daily status reports, etc.). When there were only two MTAs those packages needed to depend on either `sendmail` or `smail`. However more MTAs appeared, and every time a new MTA was packaged the dependencies for all packages that need a mail transport had to be updated to also mention the new MTAs. You can see the complexity in figure 1.

To resolve this the concept of a virtual package was introduced: a virtual package represents an interface or functional-

ity. Other package can *provide* a virtual package to state that they implement that interface or functionality. This way multiple packages can say that provide the interface. To resolve the above problem all MTA packages only needed to say that they provide the virtual package `mail-transport-agent`. Packages that need a MTA can then depend on `mail-transport-agent` instead of a listing all possible MTAs. If you look at figure 2 you see that the result is much cleaner.

After all these improvements there was still a nuisance: you needed to have all the package on your system, either by mirroring an archive or using a CD. The next logical extensions to the package system was introduced: on demand downloading. To do this a database that lists all available packages is put on the site that hosts the archive (which could be a FTP site, CD or website). The package system can download that database in order to maintain its own database of available packages. Using that database it knows exactly what packages you can install and where it should download them from. Using this installing becomes an almost trivial task (see figure 7).

## 2 A new package format

For a while now Linux has had two popular package formats: the `rpm` format as made by Red Hat, and the `deb` format as created by Debian. The feature-set supported by the formats and different package managers (`rpm` and `dpkg`) differ in some areas. This has been problematic since it means vendors have to support both formats. As a result there has been a desire to create a common package format that makes it possible to use a package on all Linux distributions. While existing formats like `deb` and `rpm` work fine, there are a few missing features that we will need:

- internationalization support

- multiple architecture and OS support
- multiple payloads

In order to resolve these issues it was decided that a new design format should incorporate the features of `deb` and `rpm` as well as a number of new features that will be needed in the future. This new (as yet unnamed) package format is scheduled to replace the existing formats and provide a common package format for Linux for the next decade.

The design is not completely finished at this moment, but we have a good basis and are starting to write code to support it. The design is based on a set of goals that we felt must be met in order for the format to be accepted as a new standard packaging format:

- *Must be extractable using only traditional UNIX tools*

Sometimes you might want to use a package on a system which does not have a package manager installed (for example on a rescue disk). Being able to extract the package using standard tools that are always available (like `ar`, `tar`, `cpio` and `gzip`) can be very practical in those circumstances.

- *Must support all features of the `deb` and `rpm` formats.*

If we want to replace `deb` and `rpm` we must make sure that no functionality is lost.

- *Must support signatures and checksums*

Any format that wants to be taken seriously needs to support a way to verify its authenticity and correctness.

- *Must not be Linux-specific*

We would like to see this format accepted on other systems as well.

- *Must allow distribution-specific extensions*

Distributions differ, and some distributions will need options that others don't. In order to keep the package format simple those must be

kept outside the common area and put in a distribution-specific area.

- *Must be extensible*

We believe that it is impossible to predict what features will be needed in the future, so the format has to allow for extensions.

## 2.1 Global structure

The wrapper in which the data and metadata are stored is a `tar` archive which contains *payloads* with the actual data (see figure 9). A payload is a collection of four files (some of which are optional). The type of each file (metadata, scripts, data or signatures) is encoded in the filename. The version of the format used is also encoded in the name of the metadata-file. The files may be compressed to save space, using `gzip` or optionally `bzip`. This gives filenames like this:

```
meta-1-XXX
scripts-XXX.tar.gz
data-XXX.tar.gz
signatures-XXX
meta-1-YYY
data-YYY.tar.gz
signatures-YYY
```

The `XXX` and `YYY` used in the filenames are used to make sure there are no duplicate filenames in the archive. The `1` in the name `meta-1-XXX` indicates that we are using revision one of the packaging format to store the files for the `XXX` payload.

The order in which the files are stored is important: the metadata has to be the first file in each payload, as this is needed to get the version number.

## 2.2 Metadata section

We chose to use the XML-format since it is a very extensible and non-proprietary description format, well suited for inter application exchanges. It also allows us to easily add features such as translation and distribution-specific extensions.

The basic structure of the XML data is as follows:

```

<package>
  common information
distribution-specific information
</package>

```

Distribution-specific information will be specific using XML namespaces. The common information will consist of information that has been specified by the LSB. An example of XML metadata can be found in figure 8.

register init-scripts, etc. A new common format will not resolve those issues and magically make packages portable across distributions. For this the efforts like the Linux Standards Base (LSB) and the Filesystem Hierarchy Standard (FHS) are needed.

## 2.3 Scripts section

Many packages need to perform some specific tasks either before or after installation, upgrade or removal. These tasks are performed by *maintainer scripts*. In order to keep the filesystem clean these are not distributed in the data section but in a separate section.

## 2.4 Data section

The data section contains the files that will be installed in the filesystem. It is distributed as a *cpio* or *tar* archive, usually compressed with either *gzip* or *bzip2*.

There is a slight problem in that there are extensions to the *tar* format that not every implementation can handle. In order to avoid problems only a defined subset of *tar* features will be allowed (at the moment the only allowed extension is the GNU long filename extension).

## 2.5 Signature section

Digital signatures and checksums will be stores in a simple XML file. There are different signatures for the different files in a payload, and per file multiple signatures are possible. An example can be seen in figure 10.

## 2.6 Notes

An important thing to note is that most of the differences between distributions do not arise from the package format used, but from the policies in creating packages: where to put files, should documentation be compressed, how do you

Figure 1: Alternatives complicate dependencies

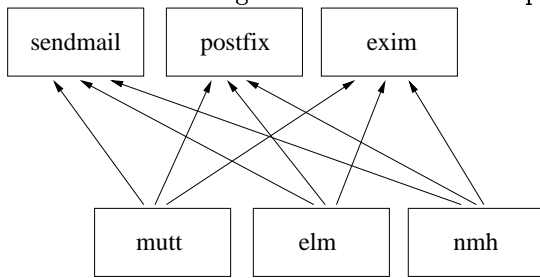


Figure 2: Virtual package help simplify alternatives

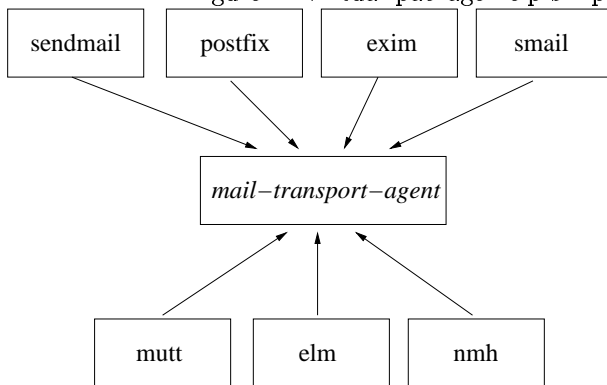


Figure 3: Old style software installation

```
% ftp somewhere
> cd /pub/linux
> bin
> get lilo.tar.gz
> quit
% gunzip lilo.tar.gz
% tar xf lilo.tar
% cd lilo
% more README
% vi Makefile
% make
% vi Makefile
% make clean ; make
% make install
```

Figure 4: Installing autoconfed software

```
% ftp ftp.gnu.org
> cd /gnu
> bin
> get grep-2.4.tar.gz
> quit
% gunzip grep-2.4.tar.gzx
% tar cf grep-2.4
% cd grep-2.4
% ./configure --prefix=/usr
% make
% make install
```

Figure 5: Installing precompiled binaries

```
% ftp tsx-11.mit.edu
> cd /pub/ALPHA/gcc/private
> bin
> get gcc-2.7.2.binary.tar.gz
> quit
% cd /
% tar xzf ~/gcc-2.7.2.binary.tar.gz
```

Figure 6: Installing using packages

```
% ftp ftp.redhat.com
> cd /pub/redhat/RedHat/I386/RPMS
> bin
> get lilo.rpm
> quit
% rpm -i lilo.rpm
```

Figure 7: Modern installing using apt-get

```
[tornado;~]-4# apt-get install gnome-stones
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  gnome-games-locale
The following NEW packages will be installed:
  gnome-games-locale gnome-stones
0 packages upgraded, 2 newly installed, 0 to remove and 1 not upgraded.
Need to get 127kB/325kB of archives. After unpacking 821kB will be used.
Do you want to continue? [Y/n]
```

Figure 8: XML metadata sample for mount

```
<package xmlns:rpm="http://www.rpm.org/v4.0"
  xmlns:deb="http://www.debian.org/dpkg/v2.0">
  <name>mount</name>
  <version>2.9t-1</version>
  <summary>Tools for mounting and manipulating filesystems</summary>
  <description>
    This package provides the mount(8), umount(8), swapon(8),
    swapoff(8), and losetup(8) commands.
  </description>
  <keywords>mount,umount,filesystem</keywords>
  <author>
    <name>Andries Brouwer</name>
    <email>aeb@cwi.nl</email>
    <nick>Andries</nick>
  </author>
  <packager>
    <name>Vincent Renardias</name>
    <email>vincent@debian.org</email>
  </packager>
  <home>ftp://sunsite.unc.edu:/pub/Linux/system/Misc/util-linux-2.5.tar.gz</home>
  <origin>Debian</origin>
  <license>GPL</license>
</package>
```

Figure 9: Basic format structure

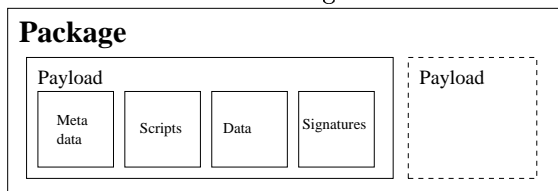


Figure 10: Signature data example

```
<signatures>
  <md5 file="metadata-1-1">db9557f3fdd690d07bc9f2682b7d0447</md5>
  <md5 file="data-1">732813edc871575e107dd8324ed417ce</md5>
  <pgp file="metadata-1-1">
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.0.1 (GNU/Linux)
Comment: For info see http://www.gnupg.org

iD8DBQA4w1bAP4VifPYwQCsRApV7AJ9xporhmU11+n0YgHFSVQJ++UtHsgCgwawT
ruLKGjuFPQnNmLamTU90TEI=
=Wg9K
-----END PGP SIGNATURE-----
  </pgp>
  <pgp file="metadata-1-1">
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.0.1 (GNU/Linux)
Comment: For info see http://www.gnupg.org

iD8DBQA4w1b1P4VifPYwQCsRAp0ZAJ9zBMy750g3RpoYceg2ULgNb1MynACfXk6u
3mKwj/m750YKblxz4jjeq1k=
=N9zL
-----END PGP SIGNATURE-----
  </pgp>
</signatures>
```