Package management

over and above

vendor solutions

Peter Samuel

<peter@uniq.com.au>

Presented to AUUG NSW Chapter

18 June 1998

The problems

"The vendor’s version of package *X* doesn’t suit my needs. How do I install a new version without overwriting the vendor’s version?"

"The vendor doesn’t include package *X* in the operating system distribution. How can I install a third party version of package *X* in an intelligent manner?"

"My users rely on version *n.m* of package *X*. How can I test a new version in the production environment without impacting on the existing users?"

"How can I cut over to the new version of package *X* with minimal impact on my users?"

"I’m upgrading package *X*. Some of the files in the old version are not in the new version. How can I tell which files to remove?"

The problems

"The new version failed. How can I rollback to the stable version of package *X* with minimal impact on my users?"

"Most of my users are happy with the new version of package *X*, however there is some legacy software that relies on the previous version. How do I support this legacy in an intelligent manner?"

The solutions?

Replace the vendor’s version of package *X* with the version that suits your needs. However, the remaining problems will still exist.

The most common approach is to install the new package in a section of the file system away from the vendor’s area of influence. This area is traditionally

```
/usr/local
```

Executables are installed in `/usr/local/bin`, libraries in `/usr/local/lib`, etc.

However, this leads to an equally cluttered set of directories in which it is almost impossible to establish package ownership from one file to another.

The solutions?

Install each package in its own directory. The directory should encapsulate the complete package.

Choose a common installation directory and install all packages under this directory.

```
/usr/local/pkgs/apache-1.3b5
/usr/local/pkgs/cu-sudo-1.5.3
/usr/local/pkgs/elm-2.4.25
/usr/local/pkgs/ghostscript-5.10
/usr/local/pkgs/gcc-2.7.2.1
. . .
```

Now users can run programs from each separate package directory.

```
/usr/local/pkgs/ghostscript-5.10/bin/gs file.ps
/usr/local/pkgs/gcc-2.7.2.1/bin/gcc -c prog.c
```

The solutions?

Users can now add relevant directories to their **$PATH, $MANPATH** and **$LD_LIBRARY_PATH** environment variables.

```
PATH=:/usr/local/ghostscript-5.10/bin:\
  /usr/local/gcc-2.7.2.1/bin:\
  /usr/local/sudo-1.5.3/bin:\
  /usr/local/xv-3.10a/bin:\
  /usr/local/pine-3.96/bin:
  ...
```

However these variables will quickly become very large indeed. Some shells will fail to handle such large variables.

Asking users to manage complex path variables can often mean asking too much. 🙂

The solutions?

Use symbolic links so that the publicly available executables and libraries appear under a common directory.

```
/usr/local/bin/gs –>
        /usr/local/pkgs/ghostscript-5.10/bin/gs
/usr/local/bin/gsnd –>
        /usr/local/pkgs/ghostscript-5.10/bin/gsnd
/usr/local/bin/gcc –>
        /usr/local/pkgs/gcc-2.7.2.1/bin/gcc
/usr/local/bin/g++ –>
        /usr/local/pkgs/gcc-2.7.2.1/bin/g++
/usr/local/lib/libstc++ –>
        /usr/local/pkgs/gcc-2.7.2.1/lib/libstc++
```

Users can now have shorter, stable paths:

```
PATH=/usr/local/bin
MANPATH=/usr/local/man
LD_LIBRARY_PATH=/usr/local/lib
```

Administrators can install new versions of a package and test them without impacting on existing users.

The final solution

All that is needed now

is something to manage

the symbolic links

in an intelligent manner!

Availability and competing products

graft – version 2.2

ftp://ftp.uniq.com.au/pub/tools/graft

stow – version 1.3.2

http://www.gnu.ai.mit.edu/software/stow/stow.html
ftp://archie.au/gnu/stow-1.3.2.tar.gz

depot – version 5.13

http://andrew2.andrew.cmu.edu/depot/depot.html

encap – version 1.2

http://uiarchive.uiuc.edu/encap
ftp://uiarchive.uiuc.edu/pub/encap/encap.tar.gz

Graft

graft: To insert (a graft) in a branch or stem of another tree; to propagate by insertion in another stock; also, to insert a graft upon.

To implant a portion of (living flesh or skin) in a lesion so as to form an organic union.

To join (one thing) to another as if by grafting, so as to bring about a close union.

How does it work?

Every package is compiled and installed so that internal references are made to the package installation directory.

Instead of installing the package files in /usr/local/bin and /usr/local/lib etc, you install them in /usr/local/pkgs/package-version.

Many packages use a configure script. Instead of choosing the default installation directory, choose a package specific directory.

```
./configure --prefix=/usr/local/pkgs/gcc-2.7.2.1
./configure --prefix=/usr/local/pkgs/perl-5.00402
```

gcc version 2.7.2.1 will now look for its specific files in /usr/local/pkgs/gcc-2.7.2.1/lib etc and perl version 5.004-02 will look for its specific files in /usr/local/pkgs/perl-5.00402/lib etc.

Similarly for gcc version 2.8.1 and perl version 5.005 etc.

How does it work?

Grafting packages

Graft the real package directories into the common area.

```
graft -i gcc-2.7.2.1
graft -i perl-5.00402
```

Users can now run /usr/local/bin/gcc and /usr/local/bin/perl

How does it work?

Ungrafting packages

Ungraft the old real package directories from the common area.

```
graft -d gcc-2.7.2.1
```

You are now ready to *graft* in a new version of the package.

Pruning packages

Graft’s *prune* mode moves original package files aside.

```
graft -p pine-3.96
```

This locates conflicting files and renames them. It is useful when transitioning from *non-grafted* to *grafted* systems.

Overriding default locations

The default package repository and target locations can be overridden on the command line.

```
graft -i -t /opt/local ssh-1.2.25
graft -i -t /usr/local/bin /opt/SUNWspro/SC4.2/bin
```

The first example *grafts* ssh-1.2.25 from the default package repository into the non default target directory /opt/local.

The second example *grafts* the contents of the fully qualified directory /opt/SUNWspro/SC4.2/bin into the non default target directory /usr/local/bin.

Note: This procedure also shows how to perform a partial *graft*.

How does it work?

Excluding package components

A subdirectory tree may be excluded from *grafting* by adding a .nograft file to the directory. If **Graft** sees this file it will bypass that directory and any subdirectories it contains.

Individual files in a directory can be excluded by listing their names in the file .graft-exclude. Typical examples of such exclusions would be

```
License
README
relnotes.txt
```

Note: These files are ignored during *ungrafting*. This ensures that the entire package will be successfully *ungrafted*.

How does it work?

Conflict resolution

Graft considers the following cases to be conflicts.

Source Object	**Target Object**
directory	not a directory
file	directory
file	file
file	symbolic link to something other than the source object

Graft will stop *grafting* when it encounters a conflict.

Graft will not stop *ungrafting* or *pruning* when it encounters a conflict. This maximises the amount of deletion or renaming that can be performed.

Conflicts can be resolved by using either a .nograft or a .graft-exclude file or by using a partial *graft*.

Caveats

- **Graft** does not do dependency mapping
- **Perl** and **Graft** have a symbiotic relationship that may need one off bootstrapping – depending on your operating system
- **Graft** will not make directories which are automount points
- the underlying file system must support symbolic links