Maintaining Large Software Repositories with SLINK

Alva L. Couch
Tufts University
Medford, MA 02155
couch@cs.tufts.edu

Greg Owen
Xerox Information Systems
Peabody, MA 02176
gowen@xis.xerox.com

Abstract

The traditional practice of installing a mix of software packages in a single filesystem tree creates several problems when the installation tree has a long lifetime. As software is replaced and deleted, it is difficult to completely remove any package, so that over time the tree gradually becomes corrupted with files of unclear function. Because all installers work in the same tree, they can unknowingly undo the work of other installers in ways that are difficult to detect.

Many special-purpose repository management tools solve these problems by allowing installers to segregate software into disjoint package trees while presenting them to the user in a union tree constructed from directories and symbolic links. Packages can then be added, deleted, and installed with no risk of affecting other packages. Most of these tools impose a rigid structure on the package hierarchy and support many features, such as software distribution.

SLINK is a general-purpose UNIX utility that incrementally constructs arbitrary symbolic unions of UNIX directory trees from normal UNIX directories and symbolic links. When used as a repository management tool, it imposes few limits on package structure and location, is very easy to learn to use, and supports a more fluid and flexible administrative protocol than that of special-purpose tools. Using SLINK and appropriate administrative protocols, one can avoid repository corruption, maintain multiple user environments, customize read-only filesystems such as CD-ROMS, and construct tiered hierarchies of local and imported repositories.

1 Introduction

SLINK is a UNIX utility that can incrementally construct a symbolic union of any number of UNIX directory trees. This union appears to contain all the files and directories in the trees, but is actually constructed from normal UNIX directories and symbolic links. Each file in a subtree of the union is accessible by following exactly one symbolic link from the union tree. The union can be built in a tree containing files that augment or shadow files in the trees to be combined. SLINK utilizes an understandable and predictable set of primitive operations to form the union by incrementally modifying the directory structure of the union tree while preserving existing file structure.

We use SLINK to provide the user with a unified view of a collection of software packages that are actually installed in disjoint package trees similar to those of NIST Depot[4], CMU Depot[2], UMASS Depot-lite[5], and Xhier[6]. Unlike these specialized tools for configuration management, SLINK is a general-purpose UNIX utility that can create unions for a variety of purposes. SLINK cannot copy files, manipulate mount tables, or manipulate settings of variables in the user's environment, and only manipulates the filesystems on a single machine. But SLINK is also not subject to the limitations of comprehensive configuration management packages. *Any* well-defined union can be created, either by an administrator or a normal user.

There are no structural requirements for SLINK's package trees other than those imposed by local administrative procedure. Our procedure separates a package into three kinds of directories:

1. Files the user should be able to access, such as commands, libraries, and user documentation. These are included in the user's view of the system.

2. Files the software package should be able to access, such as databases, private commands, and private libraries. These are hidden from the user but must be present for package operation.

3. Files not needed for normal system operation, such as installation documentation and source code. These are hidden and subject to deletion.

Only directories of the first type need be combined; users should see one master command directory to be included in their paths, one documentation hierarchy to be included in their manpaths, and one library to be included in their load library paths. Components of these directories can appear anywhere within package trees and have any name whatever. If commands do not naturally appear in a reserved directory in a package tree, one can instruct SLINK to incorporate individual commands into the union one at a time.

Each program or coordinated system of programs is installed in its own package together with all files it needs in order to function. One convenient exception to this rule is that databases shared by several packages, such as the Emacs `info` tree, are installed in their own packages so they will outlive deletion or revision of any packages that use them.

Packages can be installed anywhere within filesystems, though we tend to install them in trees where the first level is a software category and the second level contains all packages in that category. This keeps us from unknowingly installing two copies of the same package. Programs in a package have no knowledge of any unions containing the package, and reference package files solely through pathnames within the package tree.

Unlike other configuration management tools, SLINK manipulates files and directories rather than configuration objects. This is admittedly less desirable than managing packages as objects, particularly when packages must be copied to other machines. SLINK makes no provisions for parts of objects other than files, such as variables that must be set in the user's environment. But likewise, SLINK does not require package trees to have a rigid, predefined structure as objects. SLINK's simplicity allows packages to be installed and brought into service more quickly on a single machine. SLINK can simulate flexible, object-based installation of package trees with per-package configuration files, though this is not required and not typically used in our practice.

SLINK implements a form of 'translucent' union similar to that implemented by Sun's Translucent File System (TFS)[7], Sun's Spring File System (SFS)[3], or Linux's Inherited File System (IFS)[1]. This allows us to form a union of various trees with selected files in the trees replaced by custom versions. Compared to these systems, SLINK's unions are relatively static in character; in each of the other filesystems one can create shadow files in the union at any time without knowing that it is a union, whereas in SLINK one must consciously create the shadow files before forming the union around them. SLINK has the advantage, however, that it implements translucency without daemon or kernel modifications, so that its translucency is portable to almost every UNIX-based system.

1.1 Avoiding filesystem rot

Like all other package management tools, SLINK prevents the syndrome of 'filesystem rot' that can occur when a single tree of installed software with a long lifetime is maintained by many administrators on a revolving basis[1]. While maintaining the tree, installers make many uncoordinated and undocumented changes to its structure. These changes leave files of unknown function in the tree. These files are probably not needed, but cannot be deleted without unacceptable risk of disabling unknown programs. Thus the tree gradually 'rots' by containing an increasing number of useless files whose removal is infeasible. Once a tree 'rots', the only reasonable way to remove the useless files is to reinstall all packages in a clean tree.

SLINK solves this problem by allowing installed packages to remain in separate installation trees. SLINK allows the user to think of all files as one coherent tree organized by file type, while in actuality the files are organized by package. The true pathname of a file is documentation of its package and function. A package can be reliably removed by deleting its tree and SLINK's links into it, so that one can add and remove packages arbitrarily often without corrupting the package tree.

1.2 Employing more installers

Historically, it has been difficult to employ more than one software installer per machine, or even to maintain several versions of one software package on one machine at the same time. One installer can easily undo the work of another, and installing a newer

[1] This is analogous to Fredrick Brooks' 'software rot' that occurs when one program is maintained by many programmers.

version of a package usually disables the older version by employing the same filenames as the previous version.

SLINK allows installers to work in package trees they own or share as a group. Installers can work without root privileges and with less potential to interfere with one another. This also allows safe use of untrained installers, even on production systems. Installation conflicts are automatically detected by SLINK as conflicts in the union tree. Incorrectly installed or obsolete packages can be deleted by removing the subtrees in which they reside. This means that a single coordinating administrator with control of SLINK can manage a virtually unlimited number of unskilled installers in maintaining a large-scale software repository.

There are three kinds of installers: those who have root privileges, those who have control over union trees, and those without privileges. The directories SLINK constructs in the union are owned by and writable to a specified UNIX group. Membership in that group gives one the ability to make modifications to the union tree. This establishes a training path for new installers, who start out without any privilege, attain control over unions once they understand them, and finally obtain root privileges when they have mastered both other levels. This structure allows us to manage a university site by utilizing a large pool of inexperienced, enthusiastic, and relatively inexpensive student installers while training them in a useful skill.

1.3 Maintaining environments

Often one site will serve the needs of several classes of users. Novice users can be confused by the availability of advanced commands. New and untested software should be made available to experienced users first, and older versions kept online for users who cannot yet employ the new versions for whatever reason.

SLINK solves this problem by allowing one to construct several distinct user environments as unions of subsets of the available packages. Many revisions of a package can concurrently reside in one repository as part of different user environments. For example, we maintain a beta-test environment in which new software appears first, a standard environment for normal users, and an archival environment of old packages that have been replaced but may be useful to someone. Users can, at their discretion, take their chances with new software or use old software that we consider obsolete, simply by putting the appropriate directories in their search paths.

1.4 Modifying read-only media

Many times we have a CD-ROM that could be used as a local filesystem if only we could modify selected files on it. Or perhaps there is a remote, stable, read-only repository that would serve our needs if only we could modify a few files within it. Although one solution is to copy the CD-ROM or remote partition into a writable filesystem and change it, SLINK provides an alternative that uses much less writable space at the cost of increasing the overhead of accessing a file in the read-only system.

To do this, we first construct a tree containing all the files we wish to add or change. We position this tree where the CD-ROM should appear to be mounted but mount the CD-ROM elsewhere, and make the tree of changes a union of itself and the CD-ROM. Changed files in the union are available directly, while an unmodified file in the read-only filesystem is accessed by following one symbolic link from the union tree to the read-only tree.

This technique can also be used to add files to an incomplete installation of UNIX when there is not enough space on the local disk to include all of `/usr`. To accomplish this we mount a complete copy from a remote location and make the local copy a union of itself and the remote one.

1.5 Creating tiered repositories

Often satellite UNIX systems do not have the disk space to make copies of all the software in centralized repositories. In this case, we would like to make the most commonly used programs available on a local filesystem, while allowing access to all other programs via the network file system (NFS) or some other file-sharing mechanism. If the central repository has a suitable form, we can use SLINK and NFS to integrate it into our local repository without making copies.

For each satellite machine, we copy commonly used packages into a local package tree. We mount the original package tree under a different, arbitrary name and make the local tree a union of itself and its remote source. Since SLINK will not replace files with links, this gives us a directory that looks like it contains all the packages available on either machine. This process can be used to incorporate packages from any number of machines.

This union does not contain a user environment. We form the user environment by a second symbolic union that combines selected packages, both local and remote. SLINK reduces paths of multiple symbolic links to single links, so the combined steps of both these operations cost one symbolic link traversal per user file access, whether that file is local or remote.

2 Implementation

From the preceding capabilities, SLINK may seem a complex program, but it is actually implemented as a fairly short PERL[8] program with assorted diagnostic and maintenance functions. The program's main function is to construct symbolic links that make one filesystem tree appear to be part of another. Additional functions include removing that appearance, removing links that are no longer valid, and optimizing the resulting structures to employ a minimal number of links and link traversals.

2.1 Links and slinks

SLINK implements a generalization of a symbolic link called a 'slink'. While a symbolic link `A->B` means that `A` is another name for `B`, a 'slink' `A=>B` is an *assertion* that the contents of `B` are also contents of `A`. If `B` is a file, then `A=>B` is implemented as a normal symbolic link `A->B`. If `B` is a directory, then `A=>B` means that each node `B/n` in `B` can be accessed as `A/n`, but does not constrain `A` from containing other nodes not in `B`. If `B` is a symbolic link, then it is expanded to the fully qualified name of the file or directory it represents, and the above rules then apply. `A` and `B` are called the *image* and *target* of the slink. A directory tree containing images is called an *image tree*.

A slink can be physically implemented in many ways at SLINK's discretion. If we install the files

```
/loc/foo/bin/
   foo
   fob
```

and wish for them to be available in `/local/bin`, we can construct the slink

```
/local => /loc/foo
```

by forming the link

```
/local -> /loc/foo/
             bin/
               foo
               fob
```

or the link

```
/local/
   bin -> /loc/foo/bin/
             foo
             fob
```

or the directory of links

```
/local/
   bin/
     foo -> /loc/foo/bin/foo
     fob -> /loc/foo/bin/fob
```

The links actually formed depend upon the preexisting structure of the image tree. The slink specifies the highest-level symbolic link that can potentially implement the desired behavior.

Two schemes for creating an image tree are *confluent* if exactly the same files are visible under both schemes using pathnames not containing '`..`'. Literally, the schemes 'flow together' to the same places. A link to a directory is confluent with a directory of links if both allow access to exactly the same files. All of the above schemes are confluent provided that the files and directories on the right hand side are the entire contents of `/loc`. Schemes are not confluent if one scheme makes files visible that are invisible in the other. The schemes above would not be confluent if there was another file `/loc/foo/bin/hide` or `/loc/foo/hide` that would be visible in some schemes but not in others.

2.2 Constructing slinks

SLINK constructs unions incrementally by selectively refining single symbolic links into confluent directories of links. In implementing `A=>B`, if no `A` exists it becomes a normal symbolic link `A->B`, where nonexistent directories in `A`'s path are created as required. However, if `A` is already a link to another directory, SLINK replaces it with a directory of links to the contents of that directory. Any time SLINK encounters a slink from an existing directory `A` to a directory `B`, it descends into both and attempts to form slinks from the contents of `A` the contents of `B`. This process continues recursively until the entirety of `B` appears in `A`. SLINK stops this refinement process if it detects an inconsistency, such as a file that should be a directory or vice versa. We call the process of refining links into confluent directories of links *brachiation* and say SLINK *brachiates* the image as needed.

Continuing the example, suppose we have already implemented

```
/local => /loc/foo
```

by creating the single symbolic link

```
/local -> /loc/foo/
             bin/
               foo
               fob
```

Suppose we add the file /loc/bar/bin/bar and implement the slink

```
/local => /loc/bar
```

SLINK starts by trying to link /local to /loc/bar. /local is already a link, so SLINK brachiates it into a directory containing the link

```
/local/
   bin -> /loc/foo/bin/
             foo
             fob
```

and descends into both /local and /loc/bar with the goal of making everything in /loc/bar appear in /local. While trying to link /local/bin to /loc/bar/bin it discovers that /local/bin is already a link and brachiates it into a directory containing the links

```
/local/
   bin/
     foo -> /loc/foo/bin/foo
     fob -> /loc/foo/bin/fob
```

Then SLINK descends into /local/bin and /loc/bar/bin with the goal of making everything in /loc/bar/bin appear to be in /local/bin. This it accomplishes by forming a symbolic link from bar to /loc/bar/bin/bar. Now the union tree has the structure:

```
/local/
   bin/
     bar -> /loc/bar/bin/bar
     foo -> /loc/foo/bin/foo
     fob -> /loc/foo/bin/fob
```

SLINK only brachiates when absolutely necessary. If we install the file

```
/loc/fun/man/man1/fun.1
```

and implement the slink

```
/local => /loc/fun
```

then since /local is already a directory, SLINK will descend into /local and /loc/fun with the goal of making everything in /loc/fun appear in /local. SLINK will accomplish this by linking man to /loc/fun/man. Now the union tree has the structure:

```
/local/
   bin/
     bar -> /loc/bar/bin/bar
     foo -> /loc/foo/bin/foo
     fob -> /loc/foo/bin/fob
   man   -> /loc/fun/man
                man1/
                  fun.1
```

SLINK provides a view of files in many target trees at the expense of maintaining a fluid directory structure that can change without notice. SLINK will modify any links or directories in the image tree to suit its purposes, but will not replace a regular file in the image tree with a directory or link. Thus one may first build an image of files and utilize SLINK to surround them with structure from other filesystems.

2.3 Removing slinks

Conversely, if SLINK is asked to remove a slink by deleting the normal links that implement it, the removal may leave a directory of links that is exactly equivalent with a link to a directory. SLINK can then *condense* confluent directories back into links. While brachiation proceeds recursively downward, the condensing algorithm proceeds from the leaves of the filesystem upward in the hope that condensing lower directories into links will uncover higher level confluences to condense. Unlike brachiation, however, condensing is not performed automatically when removing a link. It is an optimization operation that is not necessary for proper functioning of SLINK, and is potentially dangerous in a live environment because it deletes directories that could potentially be the current directories of running processes.

Continuing the example, suppose we remove the slink

```
/local => /loc/foo
```

This removes the links

```
/local/
   bin/
     foo -> /loc/foo/bin/foo
     fob -> /loc/foo/bin/fob
     foz -> /loc/foo/bin/foz
```

so that the directory /local/bin, which now contains the single link

```
/local/
   bin/
     bar -> /loc/bar/bin/bar
```

is confluent with the link

```
/local/
   bin -> /loc/bar/bin/
             bar
```

If SLINK is asked to condense confluent directories to links, it will replace the directory `/local/bin` with this link. SLINK will not condense further, because `/local` now contains

```
/local/
   bin -> /loc/bar/bin/
             bar
   man -> /loc/fun/man/
             man1/
               fun.1
```

whose structure is not confluent with any single link. Condensing does not occur if the directory contains anything except links, like a regular file, or if there is a file in the target directory that is not presently visible in the image directory and would be made visible by the condensing operation.

2.4 Links and 'true' names

Target trees often themselves contain links. In making any link from image to target, SLINK first converts the target's name to a full pathname devoid of symbolic links. A link to the target node points to this 'true' pathname of the node, so that referencing the file from the image traverses only one level of symbolic links regardless of the linked structure of the target. This allows us to make unions of unions without incurring substantial link traversal overhead. Continuing the example, `/local/man` points to `/loc/fun/man`, so that the slink

```
/beta/man => /local/man
```

will be implemented with the symbolic link

```
/beta/
   man -> /loc/fun/man
             man1/
               fun.1
```

True names are only computed for the particular node a new link should reference. A link to a directory containing links is not optimized by brachiating the image link into a directory of links. It is more important to limit brachiation than to limit symbolic link traversal overhead. The latter simply costs us execution time, while the former consumes filesystem inodes, a somewhat more limited resource. Continuing the example, the slink

```
/gamma => /local
```

will be implemented as

```
/gamma -> /local
             bin -> /loc/bar/bin/
                       bar
             man -> /loc/fun/man/
                       man1/
                         fun.1
```

even though `/local` contains links `bin` and `man`. The alternative brachiation

```
/gamma/
   bin -> /loc/bar/bin
             bar
   man -> /loc/fun/man
             man1/
               fun.1
```

will not be constructed, even though it would incur less link traversal overhead.

3 Using SLINK

SLINK's configuration file `slink.conf` is a list of slinks to maintain. This file allows SLINK to check the global consistency of a union scheme, and allows errors to be corrected by changing the configuration and reinvoking SLINK. Each line in the file has the form

```
{target} {image}
```

which specifies one slink

```
{image} => {target}
```

This syntax makes SLINK consistent with '`ln -s`' in the case where the target is a regular file. Our example scheme is

```
/loc/bar /local
/loc/fun /local
/local/man /beta/man
/local /gamma
```

We can optionally specify the roots of relative addresses for each column, so that the above scheme could be written as

```
+ /loc /local
bar .
fun .
+ /beta /local
man man
/local /gamma
```

Slinks in the configuration file are created or updated in the order that they appear in the configuration file, so that later assertions will override earlier ones in the case of conflicts.

The configuration file syntax also supports delegation. The syntax

```
+/loc/foo/foo.conf /loc/foo /local
```

tells SLINK to include slinks from the package configuration file `/loc/foo/foo.conf`, where addresses of targets are relative to `/loc/foo` and addresses in the image are relative to `/local`. This allows a package to flexibly control its own image.

SLINK can be invoked in several ways. One can construct, update, or validate a set of images either in one step or incrementally. SLINK automatically detects errors in specification, such as conflicting assertions that tell one link to point to two files. SLINK can detect and correct inefficient image structure or irregular conditions within the image tree, such as a regular file hidden amongst the links. SLINK maintains several output files, including a list of all links made by SLINK, a running log of all actions taken by SLINK, and a list of errors encountered in implementing slinks.

4 Limitations

SLINK is in no sense a comprehensive translucency tool. There are many limits to its use, including limitations on relative references, limitations in specifying SLINK's behavior, and workarounds for insuring proper function of automounting daemons.

4.1 Relative references

SLINK cannot insure correct function of relative symbolic links from a package tree to its image tree. To understand this, suppose that we make a translucent version of `/usr` that replaces `/usr/sbin/sendmail` with a custom version, where the original copy of `/usr` is mounted as `/lusr`. Suppose that in the original `/lusr` there is a link

```
/usr/lib/sendmail -> ../sbin/sendmail
```

and that the translucent copy `/usr` has a link

```
/usr/lib -> /lusr/lib
```

Then a reference to `/usr/sbin/sendmail` will retrieve the new version, whereas a reference to `/usr/lib/sendmail` will retrieve the old one, because the relative reference will be evaluated relative to the target tree instead of the image tree. One workaround is to override all relative links of this kind in the image tree, creating the link

```
/usr/lib/sendmail ->
  /lusr/sbin/sendmail
```

or even

```
/usr/lib/sendmail -> ../sbin/sendmail
```

in the image tree. This behavior shows why we define confluence in terms of paths not containing '`..`'.

4.2 Destructive operations

SLINK will modify any links or directories in an image to suit its purposes, but will not touch files found in the image. There are two cases where this default behavior is not desirable. If we are making an image that already contains symbolic links, such as `/usr`, the previously defined links should be left alone. These are mostly relative links that define relationships between `/usr` and `/var`. If we want a pure image without shadowing, files should be replaced with links. Currently SLINK supports neither of these operations, and we have been forced to write custom copies to obtain this behavior.

We could implement SLINK so that it only modifies symbolic links it created, and only condenses directories it made. These conventions would make more sense, but we approach this change with some concern. These modifications would force SLINK to maintain a database of its actions separate from the filesystem itself, so that the loss of the database would disable SLINK. This represents unacceptable risk to us.

4.3 Automount trees

Some older automounters not integrated into the UNIX kernel mount NFS filesystems in a reserved directory `/tmp_mnt` and make symbolic links from each actual mount point into this temporary directory. If SLINK tries to create an image of an automounted filesystem, and is allowed to resolve links into their true pathnames, it will create links into this reserved tree. This will have the effect of disabling automounting, as the automounter will only mount nodes accessed by their mount names, not nodes accessed by the names under which the mount is implemented. The workaround is to inform SLINK of all such reserved trees so it will avoid making links into them. This is done by editing a list at the beginning of the script. This modifies computation of true names so that links pointing into the automounter's reserved space are not followed in computing the names.

4.4 Ill-formed unions

Of course, SLINK cannot construct an infinite structure, such as a union that contains itself. Unfortunately, it will try to do so. One must avoid specifying such slinks. This is the only limitation on the unions SLINK creates.

5 Future work

Many new features await implementation. SLINK can remove slinks as easily as it creates them. The configuration file syntax

```
!  {target} {image}
```

will soon direct SLINK to remove a slink, insuring that no files in the image came from the target. Now one must move the target to another name and ask SLINK to clean up the link tree by removing links to nonexistent objects.

SLINK can theoretically build automounter databases while building unions of local and remote filesystems. Rather than explicitly mounting the remote filesystem as in the example above, one would ask SLINK to determine the optimal pattern of mounts according to some criterion of optimality to be determined. Since a mount has the character of a link, this fits nicely into slink's syntax; one could specify the target as `host:target`.

SLINK is just beginning to be used to construct modified images of read-only filesystems, and many features could aid that process. Whereas now the modified files must be known beforehand, it is easy to write a variant of SLINK that can modify the image incrementally, replacing one file at a time with a modified copy in the image. This would implement true translucency of the image, which would have the added benefit of allowing individual users to construct personal, translucent views of remotely mounted filesystems exactly like TFS. This technique could also be used in prototyping changes to a filesystem before making the changes permanent. SLINK could also easily automatically shadow all relative links in target trees, working around the limitation cited above.

Of course, it would be wonderful if SLINK could distribute target trees to remote locations, but that seems a better task for the distribution systems designed for the task. We think SLINK should simply be integrated into existing distribution packages. This will keep its syntax simple and easy to learn, and generally increase the utility of the tool.

6 Conclusions

We have used SLINK on a variety of platforms, including SunOs, Solaris, ULTRIX, OSF/1, OSF/3, and Linux. Once tried at a site its usage has spread like a virus; administrators immediately see its advantages and how to utilize it. There is essentially no learning curve for trained installers, and untrained installers can learn safely as they work.

We discovered in previous implementations of SLINK that any form of administrative protocol enforced by a configuration tool decreases its usability. Previous versions enforced tree naming schemes and a predetermined package object structure. But packages are supplied to us with no such conventions in place. Even if a package can easily be installed in a private tree, the structure of its tree is not guaranteed to match whatever conventions our tool enforces. We spent more time forcing the package trees to look appropriate as objects than in installing the packages. This waste of time led to frustration, avoidance of the tool, and cavalier corruption of the image tree by frustrated installers. We think the object-based view supported by most configuration management tools is too idealistic for the packages we must install, and prefer the more loosely defined file and directory-based paradigm that SLINK supports, especially when maintaining isolated machines.

In promoting SLINK we have had two major problems. It is always difficult to convince new installers to abide by the fairly rigid installation protocols needed to insure clean filesystems. In order for SLINK to be worth the trouble of learning to install things differently, one has to have a system that requires maintenance over a long lifetime. Systems that will be rebuilt rather than maintained do not require SLINK. Once installers understand this we have no further attitude problems.

It is also difficult to convince administrators of potential sites that SLINK will not cause their systems to slow down due to link traversal overhead. In order to avoid symbolic link overhead, one has to design SLINK unions carefully. In particular, if overhead is a concern, one should not provide a union via NFS, which would cause symbolic links to be traversed via NFS calls. In practice however, we have no trouble with systems that do this, because the amount of symbolic link following is much smaller than one would naively predict.

The most limiting factor is SLINK's inability to resolve relative references to an image tree from one of its target trees. While this may seem esoteric, several things we would like to do become less possible because of it. In particular, unions built from `/usr` do not always work as expected.

One must also be vigilant of changes made to links in target systems, as SLINK must be invoked after any change that will modify a link in the image tree. SLINK is by nature a bottom-up filesystem building tool that does not automatically adapt to changes made to a target after it constructs its image tree. For example, if one modifies a link in the target, then since SLINK resolves the link to its destination, the image tree will remain pointing to the old location until SLINK is invoked again to repair the image link.

But all these concerns are less important than the ability to copy working, verified packages from one machine to another with absolute certainty that they will work on the other machine, that copying them will not corrupt the target machine, and that they can be deleted without corrupting the target machine. Because of SLINK, we need not designate a particular machine containing a software repository, because every machine containing a SLINK tree is one. This allows a fluid package migration strategy of finding a machine with the packages we want and simply copying them verbatim. We hope that future repositories will be designed to exploit this, so that software will be provided on the Internet in a form ready to SLINK and execute.

SLINK is a powerful tool for implementing translucent unions of UNIX trees. In the hands of a skilled administrator, it can be used to manage the work of a veritable crowd of less skilled installers. It requires a minimal amount of training to use, and enables everyone, including administrators, installers, and users, to work more efficiently and productively.

7 Availability

SLINK is written in PERL 4 and compatible with PERL 5 to be portable among most UNIX systems. It is available from anonymous ftp at **`ftp.eecs.tufts.edu`** in the directory **`/pub/slink`**. It is made freely available under the GNU public license.

8 Acknowledgements

We wish to thank all the people who participated in the design and beta testing of this tool. Prof. David Krumme saw the need for the tool and encouraged me to implement it, and participated in lengthy design discussions. Grant Taylor pointed out many flaws in the initial protocols for using SLINK, and initially tested SLINK on a wide variety of platforms. Our lengthy email discussions crystallized into reasonable administrative protocols. Prof. James Schmolze became the first person to learn our installation protocols from our documentation, and discovered the hard way that trees shared among packages, such as the emacs info tree, must be in their own packages. Finally, we would like to thank Larry Wall for Perl, without which this program would not have been practical to implement and would never have been written.

References

[1] Almesberger, Werner, "The Inherited File System", a software package supporting layered mounts of Linux filesystems.

[2] Colyer, Wallace and Walter Wong, "Depot: a tool for managing software environments", *LISA VI Proceedings*, 1992. pp. 153-158.

[3] Khalidi, Yousef A. and Michael N. Nelson, *Extensible File Systems in Spring*, Sun Microsystems Technical Report TR-93-18, Sun Microsystems, Inc., 1993.

[4] Manheimer, Kenneth, Barry Warsaw, Stephen N. Clark, and Walter Rowe, "The depot: a framework for sharing software installation across organizational and UNIX platform boundaries", *LISA IV Proceedings*, 1990. pp. 37-46.

[5] Rouillard, John P. and Richard B. Martin, "Depot-Lite: a mechanism for managing software", *LISA VII Proceedings*, 1994. pp. 83-92.

[6] Sellens, John, "Software Maintenance in a campus environment: the Xhier approach", *LISA V Proceedings*, 1991. pp. 21-44.

[7] Sun Microsystems, Inc., **tfs**(1), **tfs**(4), and **tfsd**(8), in the *SunOs 4.1 Manual Pages*, 1988.

[8] Wall, Larry, and Randall L. Schwartz, *Programming Perl*, O'Reilly and Associates, Inc. 1991.