SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration

Alva L. Couch
Tufts University

SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration

Alva L. Couch – Tufts University

ABSTRACT

We manage several large UNIX program repositories through a community effort of volunteerism and advocacy. Our effort requires a carefully crafted interplay between administrative policy and tools that operate within the limits of that policy. Rather than restricting administrators' actions, our tools reinforce their own use by making it easier and more effective to comply with policy than to dissent. Our tool SLINK provides a small number of commands that aid in synthesizing user environments from sets of disjoint software package trees. SLINK's commands, while more powerful than typical UNIX commands, refuse to violate predefined policy restrictions, thus protecting the user environment even from mistakes of root users. Our administrative policy and SLINK allow us to employ an arbitrarily large number of volunteer administrators without degrading system behavior or utilizing large amounts of staff time.

Introduction

We maintain a medium-scale, heterogeneous, departmental, academic UNIX network (50 stations, 1000 users) for the Department of Electrical Engineering and Computer Science at Tufts University. Our academic mission requires us to provide the most current versions of multitudes of software packages for Sun, DEC, and SGI systems. Our program repositories contain hundreds of programs, fill 10 gigabytes, are constantly being revised and updated, and encompass almost every form of software from freeware for simple document processing to complex commercial systems for simulation and computer-aided design.

We accomplish this mission with almost no staff involvement, due to a novel repository management approach based upon community involvement, user empowerment, administrator volunteerism, and software advocacy. Our shortage of administrative staffing is offset by availability of untrained but trustworthy, competent, and enthusiastic student and faculty help. Conscious of the many problems in scaling a standard managerial approach to student help [7], we developed instead an approach in which maintainers form a loosely coupled community bound together by policy much resembling a community charter. This policy is supported by tools that make it easier to comply with policy than to dissent. Thus we maintain system integrity and consistency not by force, but by making compliance the path of least resistance.

Previous approaches we tried for repository management caused tremendous frustration for us. Simply utilizing */usr/local* as intended led to constant problems requiring administrative intervention. Student workers often installed conflicting versions of the same package so that neither version worked. We never quite knew who installed what or when. Due to rapid turnover of software and the need to constantly update to new revisions, we were always breaking one software package or another, with no ability to control changes or to test software before installation. Worst, */usr/local* frequently filled up with files of unclear, undocumented function that we could not delete without unacceptable risk of breaking other, unknown programs.

Six years ago, we began working on solving this problem. We started by trying to install all programs in separate subtrees of identical structure, and wrote a very simple tool to build an image of the union of all subtrees that users could employ. This tool was a complete failure. Very little of the software we use would adapt to being installed in a 'standard' way, and the tool remained mostly unused. We spent more time adapting software to this standard form than we would have spent cleaning up a typical */usr/local*. We learned from bitter experience that any approach we use has to be very flexible and tolerant of deviance in software packages.

This flexibility required that we adopt a different philosophy about the role of tools in our maintenance strategy than is typical in modern practice. Most tools for repository management *enforce* standards by refusing to function unless requirements are met. Our administrators could not function in such an enforced environment, because the software we were trying to maintain was too difficult to adapt. Because of our site's volatility, budget, and need to keep up-to-date, we do not have the time to fight with software packages. Any strategy we adopt must allow us to break the rules instead of wasting time trying to comply.

If we allow breaking the rules, there must then be a way to maintain order in the ensuing chaos. We accomplish this by creating tools that, instead of limiting the administrator, make it easier to install software in the preferred way than in other ways. These tools do not enforce, but *reinforce* our policies. A tool that reinforces policy works whether or not that policy is followed, but 'works better' when policy is followed than when it is ignored. This means that everyone, whether compliant or not, will be able to perform useful work, but that compliant administrators will receive more positive reinforcement in the form of completed work.

Background

A *program repository* is a filesystem wherein programs to be executed on a network are stored. Repositories can be formatted so that programs within them are ready to run, and mounted on remote systems via the network file system (NFS) or some similar mechanism. Repositories can also consist of information to be copied, through some mechanism, to remote hosts before being executed. Our repositories are ready to run, NFS-mounted by nearby hosts, and can be copied to remote hosts, one package at a time.

Repository management refers to the activity of maintaining such a repository over time, which comprises adding and deleting programs and revisions of programs, and distributing software changes to remote systems where applicable. A management strategy should define appropriate *change control*, whereby the user's software environment is changed in an orderly, predictable, and reversible fashion as a result of software updates. The strategy should also avoid repository corruption or rot [5], in which the repository fills up with old files that have no function but cannot be efficiently located and deleted.

Many tools, including vendor-supplied software installation scripts, manage software repositories and avoid repository rot through a *package* structure [8]. Each software distribution occupies a separate filesystem tree called a package. Typically all such package trees exhibit parallel structure, so that each has a section for user commands, libraries, and documentation the user might need. Subject to obeying interdependencies between packages, package subtrees can be independently copied to remote hosts to provide custom software environments on each host. But simply copying them is not enough; to utilize the package the user must have access to those commands, libraries, and documentation.

The most common way these packages are made available to a user is through *environment synthesis*. The various packages are combined through some mechanism into a coherent user environment having single program, library, and manual directories that can then be included in each user's search paths. Usually this is accomplished through symbolic links [1, 3, 8, 9, 10, 11, 12], but it may be accomplished by copying files, as is done in several vendor-supplied software distribution systems.

Many developers have extended the basic distribution framework of Depot [8], whose basic capabilities include package definition, distribution, and environment synthesis using symbolic links. Although initially Depot expected all packages to have a predefined structure and obey particular naming conventions, now the structure of each package, including a list of files the user should see, may be declared within a file in each package. In Depot and its relatives [1, 3, 10, 12], changes are controlled and the user environment is protected from errors by a transaction commitment process, in which the proposed new environment is checked for lack of conflicts before the old environment is erased and a new one created in its place. A conflict occurs when one file in the environment has two possible definitions, usually due to an installer error. This strategy requires that the environment, as a filesystem subtree, be completely under the control of the software tool that destroys and recreates it.

Repository distribution tools like Cicero [2] (and some more recent variants of the Depot approach) take transaction commitment one step further, by giving the administrator control over the sequence of individual software installation transactions and the ability to undo transactions to recover from installation errors. This means that Cicero is not limited to installing packages in designated filesystem subtrees, but can safely undertake changes of a broader scale, such as changing files in */etc*.

SLINK

Our tool SLINK [5, 6] concentrates on simplifying the process of environment synthesis. SLINK is a freely available Perl5 script and library that is portable to a majority of UNIX systems. It has no distribution capabilities like Cicero or Depot relatives, and instead operates on a tree of packages that has already been distributed to the target host by some other distribution mechanism, such as NFS or rdist [4]. The version of SLINK described in this paper has been substantially modified from that described in [5], mainly by improving declaration file syntax and by adding the virtual protection mechanism for filesystems described below.

SLINK differs from other approaches in several important ways. SLINK attempts to make the environment synthesis task as simple as possible for an administrator to perform, so that administrators can be trained very quickly to modify environments. SLINK operates mainly by interpreting a single configuration file that tells it what should be true of the target system. One can create package-specific configuration files but this is not required. SLINK constructs environments *incrementally*, by making the changes necessary to install one package at a time, rather than creating an environment from scratch every time as in most Depot-like strategies. SLINK does not need to maintain complete control over the environment, but can merge its changes into an existing environment as easily as it can build a new one.

SLINK has five basic commands that control the user environment by specifying parallel structure between the package trees in which administrators install software and the image trees that normal users utilize. Each command is a structural assertion about the similarity between two UNIX filesystem subtrees, that specifies either that the contents of one be contained or not be contained within another. Commands recursively update the structure of filesystem subtrees to arbitrary depth, so that whole file hierarchies can be duplicated with a single command. These commands are general-purpose and can be used for a variety of replication tasks other than repository maintenance.

The command

```
link /loc/lang/perl/bin /local/bin
```

says that */local/bin* contains the contents of */loc/lang/perl/bin*, plus perhaps more files, by utilizing symbolic links to point from */local/bin* to */loc/lang/perl/bin*. The command

```
unlink /loc/lang/perl/bin /local/bin
```

undoes the *link* command, specifically unlinking any links in */local/bin* that happen to point to corresponding files in */loc/lang/perl/bin*. The *link* and *unlink* commands implement environment synthesis, including both installation and deletion of software. They operate recursively on whole filesystem subtrees, so that one can install *all* of *perl*'s files in */local* by typing

```
link /loc/lang/perl /local
```

This is done by optimal use of symbolic links; for details, see [5]. This syntax has changed from that described in [5], due to difficulty in reading the original syntax.

The current version of SLINK also includes commands for normal copying of files from one place to another. The command

```
copy /loc/lang/perl/bin /local/bin
```

does the same thing as *link*, except that it copies files rather than linking them, while the command

```
uncopy /loc/lang/perl/bin /local/bin
```

removes files that are identical (in all respects, including owner, mode, timestamps, and contents) to those in */loc/lang/perl/bin*. This replaces *cp -r*, *tar*, and other more primitive ways of copying files. However, copying only works perfectly for root users, and of course does not preserve owner or group of each file when invoked by normal users.

The commands *link*, *unlink*, *copy*, and *uncopy* function more like structural assertions than commands, and do nothing if the desired condition already exists. The commands *unlink* and *uncopy* only undo exactly matching *link* and *copy* commands; these will not delete files in the user's environment that happen to have the same name as files in the package tree but were replicated from another package or source. This means that *unlink* and *uncopy* can be repeated even if the affected files are replaced by files of the same name but from newer revisions; *unlink* and *uncopy* will not destroy these newer files.

The command

```
destroy /local/bin
```

will unconditionally destroy */local/bin* if allowed. This is mainly a cleanup command.

As SLINK does nothing if a series of command assertions are already true, it can safely be used to verify or change the configuration of systems while users are using them. SLINK may, in the course of an incremental change, turn a symbolic link into a directory of links, but will never do the reverse by default, so that no user can be deprived of a current directory through SLINK's actions.

Virtual protections

SLINK's commands are very powerful and dangerous in the hands of the uninitiated if left unrestricted. In particular, a root user could destroy the whole disk image by typing '*destroy /*', or corrupt the */usr* partition by typing '*link / /usr*'. Such powerful commands must be tempered with limits so that SLINK is safe to use. This is done through a *virtual protection* scheme that augments the normal *physical* UNIX filesystem protection scheme. The virtual protection scheme is new to SLINK 5.0; in the previous version protection directives applied only to individual commands. We realized through bitter experience that this was an unsafe practice.

A virtual protection is a voluntary mechanism implemented by a software tool to protect against unwise actions that are perhaps physically possible. While physical protections are enforced as usual, by UNIX, virtual protections are enforced by SLINK itself. In the performance of its duties SLINK will refuse to modify any virtually protected subtree. Of course, the administrator can work around this by making changes manually, given enough physical privilege. But this is considerably more difficult than using SLINK commands.

Our virtual protection scheme is designed to be as simple as possible while supporting common administrative operations for repository management and filesystem maintenance. There are currently five levels of virtual protection we support:

1. freeze – prohibit all changes to this filesystem.
2. protect – allow only the addition of files, links, and directories.

3. relink – allow the removal and recreation of symbolic links.
4. redirect – allow the removal and and recreation of links and directories consisting only of links.
5. replace – allow any change, including removing files or directories containing files.

These protections are the simplest scheme we could design within the context of SLINK's mission. The reasons for protections of *freeze*, *protect*, and *replace* are obvious to any administrator, allowing one to completely protect a filesystem, allow additions only, or allow all changes. The protections *relink* and *redirect* are specific to the needs of SLINK in incrementally maintaining repositories.

The *relink* protection allows one to protect file and directory structure while changing links. This allows one to build a filesystem consisting of files and directories, and then augment it with links that cannot overwrite those files and directories, so that the filesystem becomes a combination of an unchanging core of files and a relatively fluid set of links. We use this mechanism to protect very important parts of */local* that should never be overwritten, such as */local/bin/perl* and */local/bin/tcsh* (without which no users can work), by making them actual files.

SLINK updates images by promoting symbolic links pointing to directories into directories of links to the contents of those directories, where needed. Over time, *unlink* requests can result in directories of symbolic links that are indeed equivalent with single links. The *redirect* protection tells SLINK that it can change these directories of links back into single links on command. One does not always want SLINK to do this; certain directories must exist whether or not they are equivalent with single links or empty, e.g., lock directories.

Protections for all filesystems are specified in a single file for each machine, which is kept separate from SLINK's configuration file, though the configuration file can also contain protection directives mixed with regular SLINK commands. The file is read by a Perl5 library function that then communicates policy to SLINK.

A typical machine's protections might be:

```
freeze /
protect /usr
redirect /local
freeze /local/man/cat*
```

The protection of a path is the protection of the longest prefix of the path with an explicitly defined protection. In the above scheme, */local/man* has the protection '*redirect*' because its longest prefix with an explicit protection is */local*, while */etc* has the protection '*freeze*' (because its longest explicitly protected prefix is /). The star convention works as in *sh* and is checked against pathnames dynamically as SLINK executes, so that paths SLINK creates are protected once created. For example, if */local/man/cat5* does not exist, when it is created, it will be treated as frozen. These conventions were adopted to make virtual protection files as brief as possible.

Virtual protections keep SLINK's brute power from destroying the system, even for root users, by stopping dangerous acts. For example, the protections

```
freeze /
relink /local
freeze /local/man/cat*
```

protect systems from student mistakes by only allowing linking in */local* and its subtrees, except for the formatted manual page directories */local/man/cat1*, etc. This keeps SLINK from placing links into those directories that might conflict with the proper function of the *man* and *catman* commands. So, if naive administrators exclusively utilize SLINK, they can affect nothing but the user's environment, no matter what their privilege or what SLINK commands they issue.

A virtual protection failure is not an error, however, because very often one wishes to structure a filesystem by copying everything from another parallel one, except for specific things. For example, the commands:

```
freeze /
protect /usr
freeze /usr/spool
link /lusr /usr
```

add links from */usr* to any file absent from */usr* that is in */lusr*, ignoring */usr/spool* for obvious reasons. This allows us to 'fill out' a small */usr* partition with files from a remotely mounted full copy */lusr*, a hack we use quite frequently when pressed for space on small or old workstations.

As another example, the commands

```
freeze /
relink /local
destroy /local
```

will destroy every symbolic link in */local* while leaving files and directories alone, because the '*relink*' protection prohibits SLINK from removing them. Using this sequence, one can restart SLINK from scratch in building a repository, in the same manner as Depot.

SLINK's *copy* command must be used with discretion. Its results are not as easily documentable or reversible as those for the *link* command, and we discourage its use except for special purposes, like embedding changes within an image of a read-only filesystem. Suppose, for example, that we have a CD-ROM containing UNIX manual pages that we'd like to augment with our own manual pages. We can mount the CD-ROM as */cdrom*, put our manual page hierarchy into */myman*, make */usr/man* an empty directory, and then instruct SLINK to:

```
freeze /
replace /usr/man
link /cdrom /usr/man
copy /myman /usr/man
```

The copy operation will override the contents of */cdrom* with those of */myman*, creating a union of the two hierarchies in */usr/man*. Of course, we could do this with manual paths, but it serves as a demonstration of SLINK's power. One can, for example, create a modified copy of a read-only */usr* in the same way, whether mounted from a CD-ROM or via NFS.

SLINK limitations

There are also several things that SLINK does not do that other tools support. Because it is an incremental strategy, SLINK will function even in the presence of conflicts. After several conflicting commands, the source of a file with conflicts is the one specified most recently. SLINK will inform the user of any conflicts it finds, but one usually has to execute a set of configuration commands twice in order to determine whether there is a persistent conflict in instructions or just a transient conflict between older and newer software versions. SLINK is completely tolerant of errors in configuration and will happily ignore all configuration lines it cannot parse, implementing the ones it can act upon. In general, the tool operates in a much less constrained (or paranoid) manner than Depot, Cicero, and their relatives.

Because SLINK is an incremental environment synthesizer and does not have the luxury of creating a whole new environment each time it is invoked, it must also support cleanup functions not supported in other tools. There are functions that scan user environments for dangling links, files, and other exceptional conditions, and functions that clean up after installation mistakes. There are functions that optimize link structure to utilize a minimum of symlinks. No matter how easy SLINK is to use for naive administrators, a relatively skilled administrator is still needed to perform these complex functions.

SLINK is capable of synthesizing very complex user environments. The virtual protection system, while certainly useful, can lead to unpredictable results in the hands of the uninitiated. To aid in debugging, we provide a separate script *slinkls* whose purpose is to show the structure of what SLINK created in an easily understandable form. This program takes a large amount of computer time to look at an environment and express it in terms of the SLINK commands needed to create it. Currently *slinkls* only describes linked structures, because it does not have enough information to describe *copy* operations. SLINK's incremental strategy can backfire if used carelessly, and *slinkls* does not fully address the problems that can arise.

SLINK's assertions work on filesystem trees as easily as upon individual files, so that the number of SLINK commands needed to install a package varies directly with the lack of parallel structure between package and image. If an installer forms a package tree that is exactly parallel in structure with the image tree, the whole package can be made available with a single SLINK command. Lack of parallelism means that several commands may be required. In extreme cases, especially when installing commercial software, every file may need to be linked with an individual command. Nonetheless, it is always possible to link any package into the image, though well-structured packages are always easier to link than badly structured ones. Disciplined administrators are 'rewarded' with a painless incorporation process, while undisciplined ones are 'punished' by having to write more SLINK commands.

Our experience with less flexible tools has led us to believe that repository administration is easier to perform and to teach when reinforced than when enforced. In practice, doing something 'wrong' has very little impact; novices' mistakes can be easily repaired by a few simple SLINK directives. This means, however, that every administrator must refer to a policy document as well as tool documentation. We find such documents to be shorter, easier to write, and easier to understand than docs for a complex tool.

Our Policy

Our policy requires administrators to install software in disjoint package trees, which are then combined to form an image filesystem that the user sees on a particular workstation, utilizing a combination of symbolic links and file copying. This image contains all commands, libraries, and documentation the user might want, with one directory per type of information, similar to the structure of */usr/local*, e.g.

Our policy for package installation is fairly simple:

1. To the extent possible, install software in disjoint subtrees, one per software distribution.
2. To the extent possible, mimic the structure of */usr/local* in those subtrees.
3. To the extent possible, programs and libraries in packages should refer to files in their own package by their full path names in the package tree.
4. To the extent possible, programs and libraries should refer to files in other packages as if they are installed in */usr/local* or an equivalent public space, except when a program or library depends upon a specific revision of a file, in which case that file should be referred to by its full pathname in the package hierarchy.
5. To the extent possible, keep files not needed during package operation separate from needed files so unneeded files can be deleted.

An ideal package tree contains a *bin* for commands, a *lib* for libraries, a *man* for manual pages, a *src* for source code, etc. It may also contain an *etc*, an *sbin*, etc, as appropriate.

In this policy, all requirements are voluntary, and designed to be taught to aspiring administrators in one day of instruction, along with instructions on using SLINK. The policy above is the first level of several levels of detail, written to be easily understood by novices. Advanced administrators can take the time to delve deeper into its subtleties.

Delegation

Our policy and SLINK simply suggest the appropriate course without enforcing it. A particular administrator's powers and privilege are determined instead by normal UNIX filesystem protections. We maintain order in the community by carefully delegating both increasing privilege and trust relative to experience and prior performance.

There are five levels of delegation: beginning, novice, apprentice, root, and netadmin. Beginning administrators are asked to install packages in their home directories or in a practice tree until they are proficient enough to work in the real package tree. Novices are given ownership of particular package directories in the true package tree and asked to install particular packages in each. When they prove competent, they become apprentices and are given membership in a group with privileges to modify the package tree and user image tree. When apprentices prove competent, they are given root privileges and additional instruction on modifying files in places other than the package and image trees. When root users become proficient, they are trained in network administration and control of global network configuration.

SLINK is not a set-user-id program. It always operates with the privileges of the invoker. Without appropriate group privileges, SLINK will do nothing for novices. Apprentices can execute SLINK or modify the user environment directly in emergencies, but cannot modify system files outside the trees made available to normal users. Virtual protections are not limits, but reminders of one's responsibility to the community. For this reason, any administrator is allowed to override virtual protections in SLINK's configuration file (though the master file of virtual protections cannot be changed). Anyone who has the physical ability to make such changes might as well be trusted to do so through SLINK rather than manually.

Security

System integrity and security are serious problems in any system maintained by extensive delegation. Our strategy has always been to evaluate people carefully before increasing their privilege or responsibility. To avoid inadvertent system corruption, we typically only allow one undergraduate to hold root privileges at a time, so that conflicting actions between two students are less likely. While this makes root privileges a much sought trophy, we continually stress that these privileges are a responsibility, not a power.

In six years and for about 20 students, we have guessed wrong only once. One student administrator did not take his responsibility seriously, and violated the privacy of other students. He was suspended from the University for his actions.

Of course, there have been many times when a new software package did not function, or when a student's actions disrupted service for a short time. Fortunately, SLINK allows one to easily reinstall older versions by changing the configuration file, so we have experienced minimal downtime due to these errors. In cases where the operation of a new package revision is questionable, we install it first in the experimental tree */local/new*. When it is verified as working, we install the new version in */local* and the old version in the archive tree */local/old*, so that users who still need the old version can access it. Users are told that new packages always appear first in */local/new* and old packages persist in */local/old*, so there is minimal disruption of service even if a new package fails.

Advanced Usage

Of course, the description of our policy and SLINK above is quite oversimplified and suitable mainly for novices. In practice, however, few repository maintainers must deal with more complex issues, and these issues are almost always issues of policy rather than tool use.

One pressing problem in installing advanced software packages is to insure that users have proper settings for environment variables. Solving this problem is simple provided that policy clearly indicates a course of action. Our policy is that */local/env* contains files of the form *package.cshrc*, *package.profile*, etc., for each installed package and relevant shell. When the user invokes a shell, all startup files matching that shell in */local/env* are sourced by the system shell startup file. These startup files are general shell scripts that can perform a variety of functions, including defining aliases and setting environment variables.

Admittedly, this is an imperfect solution. Mistakes in a startup file can disable a shell, and any student with access to */local/env* can execute an arbitrary command as any user. So trust between students and administrators is again essential, and */local/env* must be constantly monitored for malicious changes.

A second problem that plagues all repository management tools is that inter-package dependencies make it difficult to separate interdependent software tools into distinct packages. A classic example of this is the emacs info tree, to which all gnu applications contribute, but which has a unique index in a separate package from any of the applications. Our policy, where possible, is to separate such trees into their own packages so that they will outlive any contributing package. In the case of info, we utilize the *image*

/local/info as the actual info tree, and create the index as a file in that tree. This is terrible style, but no object-oriented approach can do better.

Libraries for Developers

Our tools are not only provided in a Perl5 script 'slink', but are also included in a set of Perl5 library modules that can be utilized by other tool implementors. Creating this library was much more difficult than writing SLINK itself, because of the amount of specification required by library functions and the complex couplings that occur between them. The library has several general-purpose modules for distinct purposes, of which the most important are:

- Duper.pm – implement high-level filesystem assertions.
- Logger.pm – enable writing of log and error files to disk.
- Mapper.pm – remember a map of things created by SLINK.
- Protector.pm – read and interpret virtual protections.

To utilize the library within a Perl5 program, one must create an object instance of *Slink::Duper* by a somewhat involved procedure:

```
use Slink::Logger;
use Slink::Mapper;
use Slink::Protector;
use Slink::Duper;

$logger = new Slink::Logger ({ });

$mapper = new Slink::Mapper ({
    'logger'    => $logger
});

$protector = new Slink::Protector({
    'logger'    => $logger
});

$duper = new Slink::Duper ({
    'logger'    => $logger,
    'mapper'    => $mapper,
    'protector' => $protector,
});
```

After this quite complex initialization, with many other options not listed due to space requirements, SLINK's functions are available as the following library functions, where *$source* is a source pathname and *@images* is a Perl5 array of images to create:

```
$duper->link($source,@images);
$duper->unlink($source,@images);
$duper->copy($source,@images);
$duper->uncopy($source,@images);
$duper->destroy(@images);
```

The Future

Several ongoing enhancements are under development for SLINK.

SLINK's major deficiency is that it can not check the consistency of a configuration before implementing it. Each SLINK command is incremental, so that its effect is dependent upon existing conditions that may be created by preceding commands. Thus, the only way to reliably check a sequence of commands for consistency is to implement them. A safe way of checking a configuration is to create a memory representation of the machine's filesystem and have SLINK work upon that image rather than the real filesystem in order to check consistency. This would provide a dry-run mode for SLINK that would inform users of what SLINK would do if invoked. This is difficult, however, because of the lack of truly portable mechanisms for manipulating UNIX filesystems in Perl, and because of subtle differences in the function of particular filesystem commands in different varieties of UNIX. NFS semantics are particularly difficult to infer without trying to modify an NFS-mounted filesystem. We currently think it unlikely that a perfect, portable simulation will ever be achieved.

Root access is too powerful and needs to be controlled by informing root users of the consequences of their acts, or even by prohibiting dangerous acts. In solving this major problem, SLINK's virtual protection scheme is only a rough beginning. As systems become more complex and file sources more varied, we need a mechanism whereby vendors and administrators alike can document the disposition of files so that administrators and tools will not make mistakes in modifying those files. SLINK's protections are the bare minimum needed for our maintenance policy to work; I foresee many more, for uses beyond our site and needs:

- *rdist* – this file is distributed via rdist.
- *vendor* – this file should only be modified by the vendor.
- *from <host>* – this file was generated on the given host.
- *created-by <command>* – this file can be generated by running the given command.
- *depends-upon <path>* – this file depends upon the given one for its contents.

Several times I have learned the need for the *rdist* protection, by changing files that were later overwritten by *rdist* automatically! It would have helped greatly if I was informed, at the time, that I was doing something foolish.

Conclusions

While we are quite happy with our community of apprentices and admins and the educational experience the community provides, there are many limits to using a community for system and network maintenance. A community is a rapidly changing constituency, where students become involved, become uninterested, and are hired by industry to run networks (even before graduating!) without warning. As the primary goal of an advocate is to control her or his own environment, documentation is nonexistent. This in turn means that an ideal task for an advocate is an installation that will be obsolete soon, so that another advocate can start over and redo the work each time a release becomes available. It is thus essential to have a full-time system administrator for the purpose of providing a sense of continuity in this rapid-turnover environment.

Also, we have admittedly made a conscious decision to favor productive output over system consistency. This decision is not for everyone, and would not work in production environments, though we have fared well. Serious concerns over security, integrity, and trust of student workers invalidate our whole management philosophy when sensitive information is at risk. In these cases a volunteer is nothing less than a security risk.

Rather than being diminished in importance, the full-time administrator must perform more roles: mentor, manager, and auditor. The result, however, is that much more can be accomplished than one full-time administrator could possibly do without a backup of a community of helpers. Another very positive side-effect is that users who need something very often stop complaining and start taking responsibility for everyone's environment. This leads to a general appreciation for the job of administrator and the difficulties and skills it encompasses. And this, in my view, advances the profession from skill to art.

Acknowledgements

I wish to thank the many people whose thoughtful input and tolerance improved this software and policy. David Krumme was a constant supporter and advocate of the software and endured many brainstorming sessions to hammer out the truth from rough ore. Greg Owen was instrumental in implementing the initial algorithms. Grant Taylor forced me to put my policy in writing, and his thoughtful arguments over several months convinced both of us that we were moving in the correct direction. George Preble, Chris Leduc, Jonathan Rozes, Allan Stratton, and many others endured the initial bugs in trying out new features.

Availability

SLINK is freely available from ftp://ftp.cs.tufts.edu/pub/slink. The version described herein is 5.0.2.

Author Information

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M.I.T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Electrical Engineering and Computer Science at Tufts. In 1996 he received the Leibner Award for excellence in teaching and advising from Tufts. He has assisted in maintaining the Tufts computer systems for Computer Science teaching and research since 1985, when he was a Ph.D. student, and is currently responsible for maintaining the largest independent departmental computer network at Tufts. He can be reached by surface mail at the Department of Electrical Engineering and Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@cs.tufts.edu. His work phone is (617)627-3674.

References

[1] Jonathan Abbey, "*opt_depot* web site", *http://www.arlut.utexas.edu/csd/opt_depot/opt_depot.html*.

[2] David Bianco, Travis Priest, and David Cordner, "Cicero: a Package Installation System for an Integrated Computing Environment" *http://ice-www.larc.nasa.gov/ICE/doc/Cicero/cicero.html*.

[3] Wallace Colyer and Walter Wong, "Depot: a Tool for Managing Software Environments", *Proc. LISA-VI*, 1992.

[4] Michael Cooper, "Overhauling Rdist for the '90's", *Proc. LISA-VI*, 1992.

[5] Alva Couch and Greg Owen, "Managing Large Software Repositories with SLINK", *Proc. SANS-95*, 1995.

[6] Alva Couch, *SLINK Manual*, 1996. *http://www.cs.tufts.edu/ couch/slink.html*

[7] Tim Hunter and Scott Watanabe, "Guerrilla System Administration: Scaling Small Group Systems Administration To a Larger Installed Base" *Proc. LISA-VII*, 1993.

[8] Kenneth Manheimer, Barry Warsaw, Stephen Clark, and Walter Rowe, "The Depot: a Framework for Sharing Software Installation Across Organizational and UNIX platform boundaries", *Proc. LISA-IV*, 1990.

[9] Arch Mott, "Link Globally, Act Locally: A Centrally Maintained Database of Symlinks" *Proc. LISA-V*, 1991.

[10] John P. Rouillard and Richard B. Martin, "Depot-Lite: A Mechanism for Managing Software" *Proc. LISA-VIII*, 1994.

[11] John Sellens, "Software Maintenance in a Campus Environment: the Xhier Approach", *Proc. LISA-V*, 1991.

[12] Walter C. Wong, "Local Disk Depot - Customizing the Software Environment" *Proc. LISA-VII*, 1993.