Depot-Lite: A Mechanism for Managing Software

John P. Rouillard and Richard B. Martin – University of Massachusetts at Boston

ABSTRACT

Previous versions of depot type schemes for maintaining software, including the NIST depot[5], Xheir[8], CMU depot[2] and its extensions[10], address the needs of a large site with knowledgeable software specialists. At the University of Massachusetts at Boston, software support is provided by a number of student operators, and faculty members. A new lighter weight depot scheme was needed to reduce the amount of knowledge needed by these less experienced software installers. In addition, we needed a depot scheme that would allow the concurrent operation of multiple versions of the same software package on a given host. This paper presents a refinement of the depot scheme that works well for a smaller site, that does not have dedicated software specialists, as well as a larger site that is interested in getting the efficiency and modularity advantages that the depot scheme offers.

Although we use the *modules* package and the *amd* automounter to ease the administration of depot-lite, neither tool is mandatory for its operation.

Background

The Department of Mathematics and Computer Science at the University of Massachusetts at Boston provides the computing facilities for 700 students and staff on 4 sparc/10's, one sparc 4/490, and one sun 3/180. The few additional computers (approx. 10 sun3's/386i's and DECstations) are used along with 10 X-terminals and 40-60 z29/vt100 type terminals to access the 6 primary computers.

Because of the diversity of people who use any given computing node, it is very important to be able to provide multiple versions of a software product on each node. It is also very important that all of these versions of software coexist. Previous versions of the depot scheme [2, 5, 8] provided for only one installed version of a software package per host. In our environment, as in many commercial software houses, having access to only a single version of a software package is unworkable. Access to multiple versions of software makes it easy to identify and track bugs between different software versions using ''side by side'' comparisons of the different software revisions.

The support staff for the Computer Science Department consists of one senior system administrator, one system manager, six student operators at the level of a novice/junior system administrator, and two lab directors. In addition to the paid staff are two volunteers who are responsible for the software setup. One deals with installation and maintenance of utilities from the Free Software Foundation, and TeX. The other does many of the tasks of a senior system administrator, but acts primarily as a toolsmith for the senior system administrator, and the lab directors. In addition to these two volunteer software installers, some of the student operators, as well as some of the faculty contribute to the installed software base.

Depot-Lite Introduction

Certain features were needed in depot-lite that we didn't feel were sufficiently well developed and embodied in any of the other depot schemes. Table 1 provides a comparison of what we considered the most important features in choosing a depot scheme. In our environment, the following factors needed to be considered:

- Due to the varying schedules of many classes and research projects, it is necessary to wean people off of older versions of software, while providing the newest version for others. Thus, multiple versions of the same software package must be able to coexist on a given machine.
- In the past changing versions of software occasionally required large amounts of system administration time to track down and resolve problems when were introduced. We needed the depot-lite scheme to support full testing of the software in place. Once the testing phase is over, promoting a software package from the testing to production phase must be simple, and not introduce problems that will prevent proper operation in the production role. Also this promotion should require no knowledge about the particular application that is being promoted.
  As a corollary to the promotion ability, demoting broken or obsolete software must be an easy task that can be assigned to even a novice student operator, who may be the only person on duty and able to respond to a software malfunction.

- Some of the people who are interested in adding to the available installed software do not want, and in the interests of accountability/security should not have root access, therefore it must be possible to install software without requiring root access.
- Because of the large number of software packages that are available, multiple installers with varying levels of knowledge[1] we needed to minimize the amount of time spent customizing software for the depot system. This leads to two types of depot-lite installations: the collection[2] and the module installations. These two types will be discussed in section ''Software Installation''.
- Because of the lack of sophistication on the part of some software installers, the directions for depotizing software must be easy to provide and use.
- The exact arrangement of the software (e.g., locations of configuration files vs. sharable data files vs. non-sharable executables vs. sharable executables) must be choosable after installation. Thus the different types of data must all be handed in the context of depot-lite. This allows flexibility in sharing and arranging software by the more experienced software installers, without having to reconfigure and re-install the software package[3]. Once this reorganization has been done, the installed product can be used as a template by novice/junior software installers when they install the next iteration of the software.
- Although we support mainly computer science students, who should be able to maintain and manage their own environments, we also have a number of people who are interested in using the computer as a tool rather than as a learning exercise. Because of this, there must be easy access to the new tools.

[1] Some don't even know about make(1).

[2] The use of the term collection should not be confused with the CMU use of the term collection. In the CMU depot paper, our collection is termed an environment.

[3] Note that this may require the use of symbolic links, however configuration files are usually read once at software startup, and as such don't generate much traffic.

The depot-lite that developed produced the following attributes:

- Supporting multiple versions of the same software is trivial. Full application testing can be done in place. Promoting software to production level, or demoting old software can be easily done with a one line change in an automount map.
- Tracking down all of the parts of the application is easy. Since all of the files needed to run the application are separated into a tree hierarchy with well known names, moving, replicating or sharing or removing the entire installation is greatly simplified.
- Software installation responsibility can be delegated to people without root access, or without general access to the /usr/local tree. As a matter of policy we try to reduce the need to use the superuser account as much as possible. Using depot-lite, the only steps that need to be done as root are:
  - Create the root directory for the tools, and change the owner of the directory to be the person building/installing the tool.
  - Add an entry to the /tools automount map to point to the newly created directory.

Feature	Depot Lite	NIST Depot	CMU	CMU Ext	Ericsson	Xheir
Supports multiple versions	Y	N	N	N	Y	N
Software testable in place	Y	Y†	Y†	Y†	Y	Y†
Easy transition from testing to production	Y	N	Y‡	Y‡	nd	Y‡
Large learning curve for system	N	Y	Y	Y	na	Y
Requires custom tool installation	N	Y	Y	Y	na	Y
Sofware installable by non-root user	Y	Y	N	N	na	Y
Software transferable between collections	Y	na	N	N	na	na
Supports per domain customization	Y	Y	N	Y	na	Y
Easy access to tools	Y	N	Y	Y	nd	Y
Heavy reliance on symbolic links	N	N	N	N	nd	Y
Has self-contained trees	Y	Y	Y	Y	Y	N

† Testing requires an entire host dedicated to testing
‡ Automated tools are required
na/nd Not Applicable/Not Discussed

Table 1: Comparison of main features found in various depot schemes.

- Optionally a module file can be created that is owned by the software installer.

  Once these steps are done, the person working on the tool is free to do the installation as a regular user. No special group or user permissions are required.
- Very low learning curve. The installation of the tool under depot-lite does not require large amounts of expertise for a particular tool. The stock installation procedures for the tool can be used to a large extent thus reducing the lead time needed to provide new tools.
- Reduces the chances of inadvertently modifying the software installation. Once newly installed software tree has been unchanged for 3 weeks, an automatic script changes the owner of the files and directories, and removes write permissions from the files and directories in the tree. Once the software is locked, there is no way for the original installer to change it without root access.
- Saving disk space is automated. We have a utility program scans the installed software tree and links together files that have identical permissions, owner, group and contents. In addition putting shareable files into a particular location allows an automated procedure to replace real files with links to this special area.
- Categorizing files into shareable, administrative etc. can be deferred until later with a minimal penalty.
- Depot-lite attempts to minimize the number of symbolic links that are dereferenced over NFS in a software installation. This is particularly important since NFS doesn't cache files as does AFS.

Server Filesystem Layout

The server view of depot-lite differs from the ''classic'' NIST Depot [5] in that files are grouped by architecture and OS before they are grouped by software package. Figure 1 displays this difference in detail.

Since the architecture split occurs first, it is possible, with a single mount, to provide a fully functioning `/tools` hierarchy to a workstation. Tracking the software revisions available on the various platforms is also made easier by splitting on the architecture first. The ability to see every version of software available for a given platform, in a tools hierarchy by performing an `ls` is a large win when developing automated software auditing tools.

We set up the names of the architecture and OS directories to correspond to the identifiers that AMD[6] uses for its architecture and OS version. This allows us to easily create a generic map entry for any given software product. Appendix A shows a sample generic entry for our tools map. In addition to each of our standard architecture directories (e.g., sun4-sos4, mips-u4_2 (Ultrix 4.2)) we have two special directories: share and domain. The structure of the share and domain directories is the same as the structure of the architecture directories. Each software installation has its own directory tree hierarchy with a name like `share/xv-3.00a-l1`. The items in the share tree are shareable across multiple architectures. There are two types of sharable trees: installation and skeleton.

Installation trees are created for tools whose installation hierarchy supports multiple architectures and platforms by default (e.g., Interleaf). Other tools that can be made shareable because the tool is primarily script based (e.g., modules, majordomo) also form installation trees. Since the installation trees have all of the necessary files that implement the software tool, these trees can be copied and will provide the complete software package.

Skeleton trees are used to minimize the disk space needed when replicating installation tree hierarchies. A skeleton tree is created by copying an

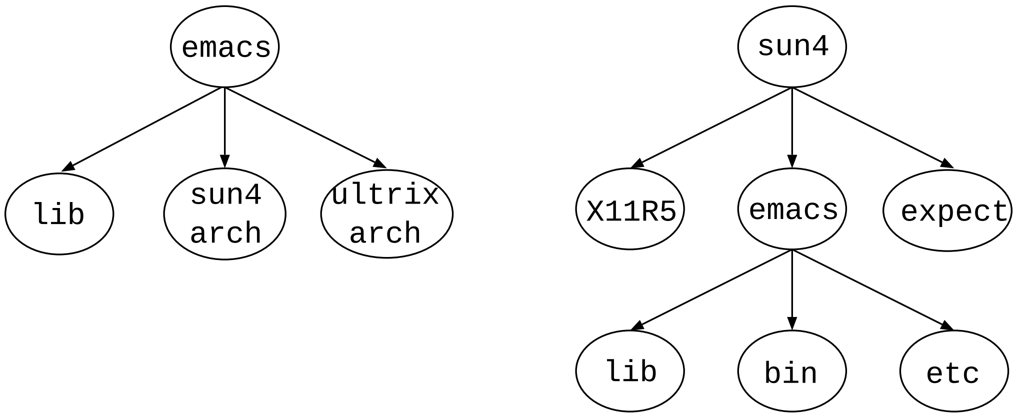

Figure 1: A view of the NIST and depot-lite arch splits. The figure on the left is the NIST server layout, while the view on the right is the depot-lite server layout.

installation tree from one of the architecture specific directories, and then removing all files that are architecture specific. This leaves a skeleton tree of files that are shareable between platforms. Manual pages and documentation are ideal candidates for the share tree since applications are rarely dependent on them for proper operation. Files that are shareable between architectures, but are required for proper operation can also be placed into the share tree. This share hierarchy can then be replicated on another server to provide a measure of fault tolerance. Unlike an installation tree, a skeleton tree is not a viable or usable tree. Some of the files necessary for the software packages operation are missing. The sole purpose of the share tree is to reduce the size of the installation tree so that the installation tree can be replicated utilizing less disk space.

The domain tree is similar to a skeleton share tree. However, it contains only configuration files that are likely to change from domain to domain. This permits different domains to maintain their own directories that are referenced via symbolic links from the installation tree. Using this mechanism each domain or department can customize the installed software. This mechanism could be used on a per machine basis if necessary.

Operations on Software in Depot-Lite

Software in depot-lite goes through the usual stages: installation, test, lockdown (or freeze), production, and deletion. There are two methods for installation depending on what method will be used to allow users to access the software. Testing is done ''in place'' making it much less likely that errors due to changing environments will go undiscovered. Locking or freezing the installed distribution attempts to provide a stable version that can not be modified without leaving some record of the change. Promotion to production status is easily done by even the most inexperienced operator, and may occur before or after the software is locked, although locking before promotion is the preferred order. Deletion of software is a multi-step procedure that provides for a test interval to make sure that the software is indeed removable.

Software Installation

Because we wanted to minimize the time required to build and install a software package, we decided to try to go along with the software provider's installation/build process as much as is possible. To accomplish this, we created two different installation models: collection and modules.

A collection installation requires that the installed software conform to a standard tree format with bin, man, lib, doc, etc and include subdirectories. It can be used for many tools that expect to live in `/usr/local`. Most of the gnu utilites that use configure[4] can be installed easily using a collection installation.

The modules installation is easier to use for a large software package that has a lot of expectations about its installation environment. In addition software that is not in general use[5] is often handled as a modules installation. A module installation does not require major restructuring of the makefiles and code to impose some external structure on the authors installation tree. Users access this software by changing their execution search path as in the ''classic'' depot[5]. At UMB we have automated this operation through the use of modules[3]. Also we use modules as a discovery and documentation mechanism for these tools.

For either installation type the root of the installation tree is changed to its canonical location:

```
/tools/<software name>-
  <software version>-<local modifier>
```

where `software name` is the name of the software e.g., xv, emacs, perl etc. The `software version` is the release of the software in a dotted version form, e.g., 2.14pl11, 3.00a, 4.036 etc. The `local modifier` is used when the installed version of the software supports features that were added locally. For example, some of the initialization code of xv version 3.00a was rewritten to allow setting of the title and the resource name that would be used by xv. Since this was the first modification that was done a local modifier of `-l1` was used, and this version of the software was installed into `/tools/xv-3.00a-l1`.

Installation for Modules

Once the root of the installation tree is changed, the rest of the software can be installed exactly as the author of the software intended it to be installed.

The ability to use the author's makefile setup and installation instructions can greatly reduce the amount of work needed to depotize complex tools as compared with the approaches of the CMU depot[2], NIST depot[5], or Xheir[8].

Installation for Collections

Besides the change that implements a different root directory, using the automatic collection creation utilities requires that the software form a standard Unix tree with bin, etc, lib, man and other directories. If the software is set up to install into a directory such as `/usr/local`, then this hierarchy of directories is already provided by the author of the software, and no additional work is needed.

---

[4]The `--prefix` option makes changing the root of the install tree trivial.

[5]For example PBM utilities, Tex/LaTeX, system administration utilities

If however, this structure is not provided, extensive work may have to be done to change the software's idea of its installation environment. In cases like this, a modules installation for the software is preferable.

Software Testing

Testing of newly installed software is done by prepending the path of the newly installed software package to the users PATH environment variable. Then the user is using the new version of the software rather than the older version. Once the tool has been sufficiently tested, it can be installed as a default version by simply remapping the simple name[6] to the new version name e.g., (emacs → emacs-19.33) by changing a single line in the automount map.

Locking the Installed Software

On a daily basis a script is run across all known tools trees on each server. This script performs the following actions:

1. See if the tooldir has remained unchanged for 21 or more days.
2. If so, remove write permissions from all files and directories in the tooldir.
3. Save the name of the owner in the file `.owner` in the root of the tooldir.
4. Change the owner of the tools to root and the group to daemon making special note not to change any suid/sgid executables.
5. Print a list of the locked tooldirs that have been found for use by the script that links identical files together to save space.

Once the locking is done, the files are unwritable by the original installer. Any changes that are made after the ''stabilization period'' of 21 days need root access. If changes have to be made after lockdown, the person who wants to make the change runs a script that breaks any hard links that were created, makes the file writable, and changes the group and owner to those of the user invoking the script. In addition to changing the files, it sends mail reporting that a late change is being made[7].

Deleting Old Software

Deleting old software is a multi-step process that is carried out by a single script.

1. Notify all users that a tool will be going away via mail to msgs, and an announcement in the appropriate newsgroups.
2. Deny access to the the top level directory for the tool by changing the directory mode to 000. This will cause any attempts to access the files under the tree to fail, and makes backing out a deletion trivial. Simply changing the permissions on the directory back to 555 will make it available again, and will also cause the automatic deletion process to abort.
3. Schedule an at(1) job three weeks in the future that will perform the actual deletion using `rm -rf`. Also scheduled at the same time are jobs that mail reminders to the system administrators that the tool is going away.

[6]The simple name is merely the name of the package with no version information.

[7]Note that the unlocking script has not yet been written.

Depot-Lite Operations

A number of scripts and techniques have been developed to support maintainance operations in depot-lite.

Creating Collections

This is the one part of depot-lite that requires a bit more hand operation. In previous attempts to solve this problem, solutions were based on cp and autoexecuted shell scripts and lfu[8] [1]. However the complexity of specifying the files to be included was getting prohibitive, and was going away from the ''lite'' philosophy. We now create a master collection by hand using some tools that automate the process of linking the bin and man subdirectories for a tool. The collection is automatically propagated to various machines.

Since the master tree is hand crafted, modules type installations can have their binaries placed into collections, which is an advantage. Also, we have discovered that the collections are not often modified, the contents of the files that comprise the collection change as newer version of software are developed, but the tools that comprise the collection don't change often. The number of links that have to be created are small since only the man and bin directories[9] have to be linked into the collection.

The propagation of the master directory is done by a modified version of the nightly program written by Hal Pomeranz[7]. Nightly was originally written to mirror one tree hierarchy, using symbolic links, into an area such as `/usr/local` on a local workstation. It then would keep track of file accesses in the `/usr/local` tree, and replace frequently accessed files with real copies of the files rather then just symbolic links. Using nightly we get local file caching that reduces the NFS traffic across our network, and it also provides a measure of fault tolerance since heavily used programs are cached on local disk and are thus not subject to the whims of NFS service.

Before putting nightly into production however, we had to ''cache'' the symbolic links in the master file system. Given the following setup:

[8]The version of lfu was modified to allow forming a collection from discrete trees.

[9]The other directories in the collection: lib, etc, doc and so forth are not used by the software, and users prefer to run the /tools tree to locate library and other files if they have a need to do so.

- The master filesystem is located in `/tools/local`
- The file `/tools/local/bin/expect` is a symbolic link to `/tools/expect/bin/expect`

nightly would make the local image of the file, `/usr/local/bin/expect`, a symbolic link to `/tools/local/bin/expect`. When executing the file this extra link causes an unnecessary NFS readlink. So nightly was modified to cache symbolic links. When nightly runs in link caching mode, the local image of the file, `/usr/local/bin/expect`, would be a symbolic link to `/tools/expect/bin/expect`. This eliminates an NFS readlink, but still allows the local caching ability of nightly to function.

Providing a Default or Collection Specific Version

We have found it useful to provide an entry in our automounter maps that refers to the default version of the software that is installed. This entry has the form `/tools/<software_name>`. This is used by scripts (e.g., expect, tk, and perl) so that magic numbers such as `#! /tools/tk/bin/tk` keep working even if the version of the software changes.

Keeping all of the collections pointing to the proper versions of the tools may seem to be a problem especially when multiple collections need to be supported (e.g., `/usr/old`, `/usr/new`). We handle this by adding symbolic links based on the name of the software (e.g., `/tools/emacs.new`, `/tools/emacs.old`). The software in the collection is then accessed through these links. Thus changing the software that is part of the old and new collections is made much easier.

Because we support multiple versions of some programs that have recently changed major version numbers (e.g., expect, tcl/tk, emacs) we have found it useful to provide ''base version'' entries such as `emacs-19` and `perl-4`. This way scripts that work under a previous major revision have a simple way of getting access to the default installed version of the earlier revision of the tool.

Conserving Disk Space

The primary mechanism for conserving disk space is an automated script that finds files with identical permissions, owner, groups[10] and contents (the discovery step) and hard links them (the linking step) together. If your server exports tools repositories for multiple architectures, the savings can be considerable.

If however, you split each architecture onto different partitions, or onto different hosts, another mechanism for saving disk space is needed. At this time the discovery mechanism has not been automated, although the ''linking'' step has been.

In order to use the linking script, the directory `/tools/share/<tooldir>` is populated with the tools tree stripped of all architecture dependent files. After populating the share directory, a script is run on each server that compares the files in `/tools/share/<tooldir>` against the equivalent directory in the architecture dependent tree. All files in the architecture dependent tree that are identical to files in the share tree and are not multiply linked are replaced with symbolic links to their shared counterparts. Again in normal operation, only locked software trees are candidates for this operation.

The reason that the link count is looked at is to eliminate unnecessary symbolic links. It is assumed that each disk that has a skeleton share tree, can also have a corresponding full install tree. Since there is no space to be saved by replacing a hard link between the shared and install tree with an equivalent symbolic link, files that have more than one hard link to them are not candidates for replacement by symbolic links. Excepting files of this sort from being symbolically linked saves some disk space, but more importantly unnecessary NFS readlink calls are avoided.

Providing per Division/Domain Configuration Files

A technique similar to the `/tools/shared` tree is utilized to maintain configuration files that may be different between different departments that utilize a shared tools repository. The directory `/tools/domain/<tooldir>` is searched, and symbolic links to the files in this directory are put into the architecture dependent tree.

To use this all departments that utilize the shared software tree must have their own versions of the configuration files. Some automatic discovery can be provided by scanning the automount tree looking for unsatisfied symbolic links into the `/tools/domain` tree, but an automatic procedure to distribute the configuration files is lacking. This makes it very important to lock down the exported software tree so that you don't inadvertently break another department's installation.

It would be very nice to have an automatic mechanism for distributing configuration files to other departments when the need to split off a configuration file develops. One scheme that has been proposed is to keep a copy of the configuration file in the shared tree so that the remote department can pick up the default configuration file from the `/tools/share/<tooldir>/domain` hierarchy and install it into their own domain hierarchy.

[10]This forces the conservation step to take place after the software is stable and has been locked down.

Finding Software Interdependencies

One problem with many tools, especially library packages such as tcl, or perl 5 is that the library may be removed but tools using that package aren't removed.

To discover such problems, a script scans all tool area files on a server for the string `/tools/`. If the string is found, it is truncated to the name of the root tool directory. For example if the string is `/tools/tk-3.3/lib/tk`, the path that is used is `/tools/tk-3.3`. The script checks for the existence of the file `.owner` in the tool's root directory (e.g., `/tools/tk-3.3`). If that file is not found, the missing tool directory has its permissions changed so that it is again available. In addition mail is sent reporting that the tool was ''recovered''. If the tool can't be recovered (because its tree was really deleted), mail is sent warning that the tool has been removed, but it should still be active.

There is one complication with this setup. As mentioned in section ''Deleting Old Software'', the first step in deleting old software is to change the directory permissions to 000. Some operating systems allow a program running as root, to find the file .owner even though access is technically denied. To combat this problem if you are so afflicted, the script can be run as a non-privileged user. Sadly the need to run as a non-privileged user also prevents the script from automatically reactivating the directory by changing it to mode 555. Consequently a warning is sent to the appropriate parties.

Scanning Automount Tables

One problem we haven't dealt with yet is the duplication of information concerning what tools are available. This is stored in two locations: the filesystem and the automount map. Tools get deleted or moved and as we all know storing information in two places provides the opportunity for the data to get out of sync.

To combat this problem, all entries in the automounter table are scanned looking for the file `.owner` in every directory under the `/tools` automount point. If the automount point under `/tools` exists, but the file is missing, then mail is sent to warn that there may be a mismatch between the advertised tools, and the available tools.

Using Depot-Lite Without an Automounter

At UMB we use the amd automounter to maintain our maps. A similar setup can be done with a Sun derived automounter using the ability to create a map with variables that are expanded at runtime to select the architecture and operating system of the mount. However a properly arranged depot-lite can work well without the automounter.

Both methods for using depot-lite without the automounter require symbolic links to map the real mount location for the various tools to their logical location under `/tools`. The first scheme introduces a performance penalty since the symbolic links are being accessed by NFS.

Create a directory that has symbolic links for every package pointing to a mount area contained in the `.mount` subdirectory. Mount this directory on the client system as `/tools`. Then mount all other tool areas[11] into the `/tools/.mount` subdirectory.

The second mechanism is the same as the first with the exception that `/tools` and the symbolic links are all on the local disk of the workstation thus eliminating the initial symbolic link lookup via NFS.

In the second case, it is easy to create a tool that scans all the mounted directories and updates/creates new symbolic links in `/tools` as needed. It is possible to create a shell script that automatically repoints the symbolic links in response to a downed server thus providing a minimal level of redundancy.

Conclusion

In response to some specific circumstances at our site, we have developed a depot like mechanism that embodies a number of features that we found critical to smooth operation:

- Depot-lite allows the concurrent operation of multiple version of the same software.
- Depot-lite achieves all of the regular benefits of software modularization without requiring an intricate and extensive series of supporting programs.
- Depotizing a software package under Depot-lite does not usually require a redesign of the installation procedure.
- Depot-lite supports multi-domain sharing of software packages through a flexible yet deterministic file tree layout.

All of our work is not yet done however. Better tools to assist in creating collections need to be developed.

The current interdependency tool calls strings(1) directly from within a perl script, however, strings(1) sometimes misses valid strings. It would be much better to write a strings replacement in perl and use that rather than the strings(1) program.

Also the timing interdependencies of the various checking tools need work. We receive a number of spurious warnings about tools not being present when they are in their three week waiting period to be deleted.

Despite these shortcomings in our automated tool suite, we consider depot-lite useful.

[11] Obviously, the fewer the tool areas (not tools) that are present, the easier this scheme is to implement.

Availability

The tools and slides will be available via anonymous ftp from `ftp.cs.umb.edu` in:
/pub/bblisa/talks/depot-lite-tools.tar.Z
/pub/bblisa/talks/depot-lite-slides.tar.Z.

Author Information

John Rouillard graduated from the University of Massachusetts at Boston with a B.S. in Physics in 1990. Since then he has been a contractor specializing in tool building and automation of various system administration tasks. In January of 1994 he took over release engineering and development from Brent Chapman for the Majordomo mailing list management tool. At the same time, he became a Senior Systems Consultant for the Mathematics and Computer Sciences Department's Software Engineering and Research Labs at the University of Massachusetts at Boston where he continues his system administration automation tasks for various clients of the Lab.

Richard Martin attended Merrimack College and UMass-Boston. He has been System Programmer with the department of Math and Computer Science since 1984, maintaining hardware and software and training sysadmins.

Bibliography

[1] Anderson, Paul, ''Managing Program Binaries in a Heterogeneous Unix Network'', LISA V proceedings, September 1991, pp 1-9.
[2] Colyer, Wallace and Walter Wong, ''Depot: A Tool for Managing Software Environments'', LISA VI proceedings, October 1992, pp 153-159.
[3] Furlani, John L., ''Modules: Providing a Flexible User Environment'', LISA V proceedings, September 1991, pp 141-152.
[4] Lundgren, Thomas, ''A File Server Directory Tree Hierarchy'', Based on Ericsson Telecom Document ETX/TX/DD-91:110 Uen Rev. B, January 16, 1992.
[5] Manheimer, Kenneth, et al., ''The Depot: A Framework for Sharing Software Installation Across Organizational and Unix Platform Boundaries'', LISA IV proceedings, October 1990, pp 37-46.
[6] Pendry, Jan-Simon and Nick Williams, ''Amd The 4.4 BSD Automounter Reference Manual'', Supplied with the amd software, March 1991 for version 5.3 alpha.
[7] Pomeranz, Hal, ''A Simple Caching Strategy for Third-Party Applications'', SANS III Proceedings, USENIX Association, 1994, pp. 117-122.
[8] Sellens, John, ''Software Maintainance in a Campus Environment: The Xheir Approach'', LISA V proceedings, September 1991, pp 21-28.
[9] Smallwood, Kevin C., ''Guidelines and Tools for Software Maintaince in a Production Environment'', LISA IV proceedings, October 1990, pp 47-70.
[10] Wong, Walter C., ''Local Disk Depot – Customizing the Software Environment'', LISA VII proceedings, November 1993, pp 51-55.

Appendix A: Generic Tools Automount Entry

Figure 2 shows a typical map entry and the default options for the tools map. The `/defaults`

```
/defaults       type:=nfs;opts:="ro,intr,nodev,grpid"; \
                      sublink:="${/key}"

emacs  type:=link;fs:=/tools/emacs-19.22;sublink:=.
emacs-19  type:=link;fs:=/tools/emacs-19.22;sublink:=.
emacs-19.22  -type:=link \
        host==terminus;fs:=/disk/sd1f/tools/${arch}-${os} \
        host==lab;fs:=/usr/local/tools/${arch}-${os} \
        || \
        -type:=nfs;opts:="rw,intr,nodev,grpid" \
        host==cs;rhost:=ra;rfs:=/disk/id000h/tools/${arch}-${os} \
        host==apollo;rhost:=ra;rfs:=/disk/id000h/tools/${arch}-${os} \
        || \
        -type:=nfs \
        arch==sun4;rhost:=terminus;rfs:=/usr/local/tools/${arch}-${os} \
        arch==sun4;rhost:=ra;rfs:=/usr/local/tools/${arch}-${os} \
        arch==sun3;rhost:=lab;rfs:=/usr/local/tools/${arch}-${os} \
        os==u4_2;rhost:=ra;rfs:=/disk/id000h/tools/${arch}-${os} \
        arch==sun386;rhost:=ra;rfs:=/disk/id000h/tools/${arch}-${os}
```

Figure 2: The standard amd automounter map entry for a tool.

line specifies the default options that apply to the automounted file system unless overridden. We assume that all mounts are of type nfs, and the nfs mount uses the options: readonly, interruptable, no devices, and bsd group semantics. The sublink option specifies that the automount directory link (e.g., `/tools/emacs-19.22`) should point to a subdirectory of the mount directory specified by the key (i.e. emacs-19.22). This is needed since the mount is done at the architecture/os level rather than at the tool level.

Immediately before the actual emacs-19.22 tool entry are some additional entries that make life easier. The simple entry `emacs` and the base version entry `emacs-19` are both entered as links to the highest stable installation version of emacs v19.

The standard entry is divided into three blocks. Each block is separated from the others by `||`. The first two blocks are used by architecture masters. These masters are the hosts that are used to build and install the tool. The third block is used by clients.

The first block consists of entries to provide the tool to a host that can serve it from local disk. Since it comes from local disk, the type of entry is a link entry. The component ‘‘host==terminus’’ is a boolean test that is true only on the host terminus. The ‘‘fs’’ entry is the parent directory that the link should point to. The actual directory that is pointed to for `/tools/emacs-19.22` on terminus is `/disk/sd1f/tools/sun4-sos4/emacs-19.22`. Note that `$arch` and `$os` are replaced by the machine architecture and operating system identifiers that are built into amd.

The second block is used to provide the tool to an architecture master that does not store the tool on local disk. The mount options in the second block provide read write access to the mount point. Without write access, performing an installation would be difficult.

The third block provides client access to the installed tools. The NFS mounts are performed with the default options specified by the `/defaults` entry. Note that there are two entries that match if the host is a sun4. This provides a replication server since AMD will attempt to mount the corresponding directories at the same time. The machine that answers first will be used. If that machine should crash, amd will attempt to mount `/tools/emacs-19.22` from the alternate server.