How to Upgrade 1500 Workstations on Saturday, and Still Have Time to Mow the Yard on Sunday

Michael E. Shaddock, Michael C. Mitchell, and Helen E. Harrison
SAS Institute Inc.

How to Upgrade 1500 Workstations on Saturday, and Still Have Time to Mow the Yard on Sunday

Michael E. Shaddock, Michael C. Mitchell, and Helen E. Harrison – SAS Institute Inc.

ABSTRACT

Imagine: It's Saturday afternoon. You run a script, watch it for a while, then go home. When you come back the next day, 1500 workstations and fileservers have new operating systems installed, complete with all your local customizations, with the user data on each one undisturbed and without leaving your office. On December 17, 1994, we did just that.

This paper will describe the infrastructure that allows us to perform completely automated updates of a large distributed network of HP UNIX computers. First, we will describe the policies we designed for distributed systems administration. Next, we will describe the tools which we developed or collected to implement these policies, and we will describe how to put it all together to do an upgrade. Throughout we will explain the philosophy behind it all and how our particular implementation could be generalized to other sites. Finally, we will describe some of the lessons learned along the way.

Support Philosophy and Design Goals

In order to support a large number of workstations and fileservers with a small number of system administrators, we decided very early in the design phase of our network to do everything possible to make all of the machines look the same, but still allow for per-host tailoring. This goal was helped considerably by the fact that our network consists of only Hewlett Packard 9000/700 series workstations and fileservers.

The second design goal was that we try not to modify any more system files than necessary. This would allow us to move from one operating system version to another without having to track down a large number of system files that we changed in each version.

Our basic philosophy is that network services should be centrally administered, and should be replicated and distributed. AFS replicated volumes and BIND, the Berkeley Internet Name Daemon, are excellent examples of how we wanted to do things. Each uses a master copy, which is then replicated to multiple distributed service providers. If any one service provider becomes unavailable, the service requesters would automatically shift to another service provider. We wanted both our day-to-day systems and our support infrastructure to follow this paradigm.

In addition to using replicated distributed services, we also try to cause each workstation to use the service provider which is "closest" to it on the network. Our network is heavily subnetted, and we want to reduce inter-subnet traffic and network latency.

System Design

Our standard system configuration is an HP700 with two internal disks. We named the two internal disks / and /local. The root disk, /, is virtually identical on every workstation and fileserver (except for licensed software differences which are managed by software distribution tools), and is where the operating system binaries, such as /bin and /usr/bin, reside. The /local disk is used for data that is machine specific. This machine specific data includes user home directories, backup tables, the workstation's AFS cache, and other local configuration files. As a result of this design we were able to set up a disk "cloning" system. If a workstation loses its system disk, we are able to replace it quickly and easily. If a /local disk breaks, we have everything that we need on the system disk to bring the system up to the point of being able to restore the necessary data on /local.

Since almost all of the machine specific data is located in /local, and since all of the system disks are virtually identical, the amount of data that we actually need to back up is greatly reduced. There are still a few files in / that need to be backed up, but this number is very small. If a system has more than the standard two internal disk drives, the additional drives are typically mounted as subdirectories of /local. Since our backup software traverses directory trees in a manner similar to *tar*(1), mounting a new disk under /local automatically adds it to our list of things to be backed up. HP-UX for HP 700 series machines does not support disk partitioning, so we were limited to using multiple disks in order to segregate system data from other data, specific to that machine. One could, however, achieve similar results by using physical disk partitioning on other systems which do support it.

Tools

Hostclasses

Hostclasses are a way of using a symbolic name to define a set of machines, and to use set operations upon those sets. They were initially designed and implemented at MCNC [1]. This initial implementation read the hostclass information directly from files in a known location. We have modified hostclasses to use a client/server approach, which includes multiple replicated servers, in keeping with our overall philosophy. Hostclasses can be incorporated into applications either through a user level program or a set of C library functions.

Hostclasses can be used for myriad applications. For example, we have one hostclass called loc.DC. This defines all of our machines in our main Data Center. We also have a hostclass called appl.AFS, which lists all of our machines which are AFS fileservers. The intersection of these two hostclasses,

```
=loc.DC % =appl.AFS
```

lists all of the machines in our main Data Center which are AFS fileservers. One of our most common uses for hostclasses is to list which machines have which extra products installed, such as the ANSI C compiler or Japanese NLIO support. Hostclasses allow for centralized list management independent of any individual application. A hostclass is similar to a *netgroups* (4), except that hostclasses are designed to be used in a more general way.

Sasify

Our primary software configuration management tool is called `sasify` (formerly called `doit`). It uses a central database called an *action file*, which contains a list of actions to apply to a host, and applies them. These actions can include downloading and installing a new kernel, deleting files, installing patches, etc. There is a file stored on each machine that defines the current `sasify` *level*. When a system reboots, it performs its normal startup procedures, then downloads a copy of `sasify` from one of a set of known servers, and runs the downloaded `sasify`. `Sasify` checks to see what the current `sasify` level is, downloads a copy of the action file that pertains to the local host, and performs any actions necessary to update the system to a new level. There are also ways to specify actions that should always be run before and/or after any level-specific actions.

`Sasify` uses hostclasses to determine which actions at a specific `sasify` level are to be run on which hosts. For example, your first level-specific action might be "for all machines that do not run X.25, download version 9.77 of /hp-ux." Your second action would probably be "for all machines that run X.25, download version 9.77_x25 of /hp-ux." In the first instance, we would use the hostclass expression

```
=sasify.HP700 - =appl.X25
```

whereas in the second instance we would only need to specify

```
=appl.X25
```

Following our support paradigm, `sasify` keeps a single centralized "database" of all of the actions necessary for all of our HP 700 series machines. After a new version of the action file is installed, it is replicated to our `sasify` database servers. When `sasify` downloads a new copy of the action file, it actually gets it from one of the five database servers.

In addition to the action file being replicated from a central location, all of the data that `sasify` downloads is also replicated from a central location. `Sasify` then picks the "closest" of the data replicas. If it cannot reach the data replica that it prefers, it will randomly pick another of the replicas.

`Sasify` can be used to maintain multiple configurations. It uses hostclasses to determine which configuration a machine should use, so it is not limited to supporting just one type of machine architecture. When we were designing `sasify`, we knew that we would eventually want to use it to maintain systems other than our main HP network, and designed it accordingly. We currently support 4 distinct HP configuration models and are working on extending `sasify` database to include the Suns that we support as well.

Since `hostclasses` and `sasify` were presented at LISA VI in 1992 [2], we will not go into more detail about their internal workings.

There are a number of additional software maintenance and distribution solutions that have been developed at other sites. These include package [3], depot [4, 5], and config [6], each with a different feature set, which are designed to solve similar problems. Most of these other packages specialized in tracking local software updates. We designed our package so that it would not only enable us to download new software versions to our workstations, but would also provide methods for adding and deleting files, and for running arbitrary programs. If you have not already adopted a formal software distribution system there is a good chance that you will find one already written which will meet your needs.

Getticket

Experience has shown that even though we have replicated most of our services, there are still times when we want to control exactly how many systems are accessing a service simultaneously. In addition there are occasionally some services which are inherently difficult to replicate, perhaps because of licensing restrictions, or which generally receive only incidental use, but may be accessed more frequently during an upgrade. For example, all of our extra HP products, such as the ANSI C compiler, are loaded from a single location and are accessed only when a system disk is replaced or a product is installed on a new machine. During a full upgrade, however, we reload this software on each machine which is licensed for it. To provide this controlled access to these services, we wrote `getticket` and `ticketd`. The client program, `getticket`, queries `ticketd` for a *ticket* to a particular service. `Ticketd` knows which hosts provide which service, and how many tickets are available for that service on each host. It then hands out tickets to this service in a round-robin fashion to distribute the load between the service providers. The ticket that is given out has the name of the service provider embedded in it, so any scripts that we write do not have to know anything about who the service providers are, or how many of them there are. `Getticket` is also used to return tickets to `ticketd` once they are no longer needed.

The `ticketd` program handles the tickets for multiple services simultaneously. Each service is defined in a `tickettab` file. The `tickettab` file lists the service name, the ticket lifetime, and the names of the service provider and a count for that provider. Each service provider can have a different number of `tickets` which it contributes to the pool of tickets for that service. The `tickettab` file for the `hpux_patches` service might look like:

```
service  hp-ux_patches 14400
milton    8
hasbro    4
tonka    16
```

This specifies that for the service `hp-ux_patches`, all tickets will timeout after 14400 seconds (four hours). There are a total of 28 tickets in the service pool: 8 from `milton`, 4 from `hasbro`, and 16 from `tonka`.

The `ticketd` program builds a ticket out of the service provider name, an underscore, and an internally generated number (the seek offset into a file). A ticket from the `hp-ux_patches` service might be "milton_080". The entire string is returned to the `ticketd` program so that the service provider section can be pulled out of the ticket with the Korn shell syntax "${ticket%_*}", assuming the ticket is in the variable "$ticket" (and the hostnames do not contain underscores). The command

```
echo $ticket | awk -F_ '{ print $1 }'
```

is another way to get the service provider part of the ticket.

While `getticket` was designed originally to manage access to services on specific hosts, it can be used more generally. The service provider field can be any string. It does not have to be a hostname. This feature allows the `getticket` program to be used as a simple licensing agent. For instance, suppose you have a 20 user license for the image viewer `xv`. The `tickettab` file might looks like this:

```
service xv 1209600
xv 20
```

You could then use a simple wrapper program that uses the `getticket` library routines to get an `xv` ticket, runs the real `xv` program which has been hidden away, then returns the ticket. This is a handy way of complying with licensing restrictions on programs which do not support license management.

Netdistd, Update, and Filesetload

HP provides two programs with HP-UX that are used for distributing software across a network. The first of these, `netdistd`, is the distribution server. A `netdist` area includes software subsets, which HP refers to as *filesets*, and patches. The other program is `update`, which communicates with a `netdistd` to download filesets to the local machine. `Update` connects either to a single default `netdist` server or an alternate server specified on the command line. In order to make HP's update system fit our support paradigm we wrote a wrapper program called `filesetload`. `Filesetload` is told which service to use, and which filesets to load. `Filesetload` then checks to see if any of the specified filesets need to be installed, and if so, it uses `getticket` to get a ticket to the specified service, runs `update` to download and install the filesets, uses `getticket` to return the ticket, and then sends the log from the update to a mailing list of system administrators. Since `filesetload` is run every time a machine reboots, an unexpected benefit is that whenever a system disk is replaced any additional licensed products will be automatically reinstalled.

Sortaddrs

During an operating system upgrade, there are many machines rebooting simultaneously, each downloading a large amount of data. To prevent network bottlenecks it is helpful to balance the network load. `Sortaddrs` was written to address this problem. `Sortaddrs` takes a list of hostnames, sorts them by subnet address, and prints out the sorted list. The list is sorted so that the first entry is from subnet A, the second is from subnet B, and so on, until we cycle back to subnet A.

Putting It All Together

HP-UX Recovery System

HP, like most UNIX vendors, provides tools to build a memory-resident operating system and filesystem. They refer to this as a *recovery system*. The typical use of a recovery system is to create a tape to be booted when a system suffers from catastrophic failure of its system drives. We have used these tools to build a custom version of the recovery system with our support tools installed, which is only used during an operating system upgrade. During an upgrade of this type `sasify` installs some support files in /local, downloads the recovery system as `/hp-ux`, and then reboots. When the system boots, it is then running our recovery system, which does not access the internal disks. We then mount these disks under temporary names, copy any "precious" data from / to /local, unmount /, `newfs` the / disk, download a *dump*(8) image of the standard system disk, and restore it as /. Next we copy the previously saved precious data from /local back to /, and reboot again. At this point, `sasify` picks up where it left off, and continues with any remaining updates.

Since the recovery system has a limited size, we had to write smaller versions of some of the standard system utilities. For example, *mount*(8) takes up 180 KB of disk space. Our "expert friendly" version of *mount*(8), which does no significant error checking, but which is sufficient for our use while running the recovery system, only uses 12 KB. We were able to realize similar savings with several other programs that we needed on our recovery system.

Since the HP recovery system uses a memory resident file system, all of the programs necessary for the recovery operation (`shell, mount, restore, cpio,`...) are contained in the data segment of the kernel image. We had to choose very carefully what would go into the recovery image, because the HP boot ROM would not load a kernel bigger than about 6 MB. We also wanted the recovery image to be as useful as possible, so we included a few things we could have copied to the /local disk instead of leaving memory resident. We

1	/etc/disktab	table of disk-drive geometries, used by 'newfs'
131	/etc/fsck	make sure the filesystems are OK
164	/etc/init	runs the /etc/rc actions
1	/etc/inittab	tells /etc/init what to do
16	/etc/newfs	initializes the filesystem
119	/etc/mkboot	installs the bootstrap program
57	/etc/mkfs	makes a new filesystem, called by 'newfs'
20	/etc/mknod	makes a 'special' device
12	/etc/mount	mounts a file system
1	/etc/rc	startup script
12	/etc/reboot	reboots the system
65	/etc/restore	restores a dump image
20	/etc/umount	unmounts a file system
25	/bin/chmod	change the protection modes of a file
20	/bin/chown	change the owner of a file
49	/bin/cpio	file archiver
16	/bin/date	set/display the time
90	/bin/dd	changes blocking factor of data
98	/bin/gzip	compress/decompress program
16	/bin/ln	links two files together
172	/bin/ls	get a directory listing
86	/bin/mkdir	make a directory
94	/bin/rm	removes a file
32	/bin/sed	stream editor
262	/bin/sh	command interpreter
29	/bin/stty	set terminal characteristics
12	/bin/sync	updates super-block
70	/lib/dld.sl	dynamic loader
856	/lib/libc.sl	shared C library
317	/usr/lib/uxboot.700.gz	compressed bootstrap program, used by 'mkboot'

Figure 1: Included Programs

pared down the size of executables wherever possible, by writing our own simplified versions of `reboot` and `mount`, by using `gzip` to compress the boot strap loader (`uxbootlf.700`) installed on the system disk by `mkboot`, and by stripping the symbol table off anything that had a symbol table. Figure 1 shows a list of all the programs we included and their sizes, in KB. We could save some space by including the `/etc/unlink` program instead of `rm`, but we would also have to write our own `rmdir` program. The `ls` program is an extravagance; we would like to find something smaller, but `echo *` is not as easy to use. We would also like to include `tar`, but it weighed in at 200 KB. For our purposes, `cpio` at 49 KB does just as well.

The `/etc/rc` script used by the recovery system first tries to mount the /local disk. If that succeeds, it then executes a shell script placed on the /local disk (/local/recover/recover) by sasify. If the mount fails or the shell script is not found, it prints a message on the console and the shell is started. This lets us use the same recovery system for system updates and for emergencies.

We use `sasify` to load into the /local/recover directory any programs needed to finish the update. Currently that includes `find`, `hostname`, `ifconfig`, `sum`, `telnet`, and a few others. `Find` is used to generate a list of filenames to save before the update. `ifconfig` and `hostname` are used to set up the networking so that the dump image and check sum file can be pulled across the network. `Sum` is used to verify that no errors occurred in the transfer of the dump image. `telnet` is used to send a last-gasp error message when a failure occurs from which we cannot recover. We use `telnet` to connect to the SMTP port of our mail gateway and send it a hand-crafted SMTP message.

How Many, How Quickly

As previously mentioned, during our testing phase we can determine how long the update process takes. We also time how long certain phases of the update process take. We can use these timings, along with our knowledge of how many service providers we have and how many simultaneous connections the service providers can support, to calculate how quickly we can reboot the machines. We also know the total time we want the update to take. If our reboot rate cannot get all of the machines updated in the time frame that we want, then we know we will need to adjust the number of service providers where possible. For example, lets say that a workstation takes 30 minutes for a complete update, we do not want any more than 5 machines talking to a single update server at the same time, and we have 10 update servers. Since we have 1500 workstations, that means it will take at least (1500 * 30) / (5 * 10) minutes or 15 hours to complete the update. The workstations should be rebooted (15 * 60 * 60) / 1500 or 36 seconds apart. This gives an upper bound for the reboot interval.

In most circumstances we can reboot the machines much faster. Only a portion of the 30 minutes it takes to update the machine needs to be rate-limited to 50 machines at once. A fair amount of time is taken by checking disk consistency, mounting disks, enabling the network, and other local processing. The only part that has to be rate-limited is the section that downloads data from the `sasify` and `netdist` servers. If the average machine spends only 15 minutes of the 30 minutes downloading data, we can reboot a machine every 18 seconds instead of only every 36 seconds. The entire update would take about 8 hours instead of 15 hours.

There is a danger in updating machines this quickly. Since it takes 30 minutes for one machine to finish, and machines are updating every 18 seconds, if there is a mistake in the update procedure (30 * 60) / 18 or 100 machines could be affected before anyone notices the failure of the first workstation.

Doing the Upgrade

Since all our workstations run `sasify` as part of their normal boot-up processing and since `sasify` is doing the updating, all we have to do to trigger an update is reboot a machine. To facilitate rebooting 1500 machines we wrote a Korn shell script that reads a list of hostnames and a time delay value. It reboots each machine in the list and waits the specified number of seconds before rebooting the next machine in the list.

The first version of the script used a passed-in value as the time delay parameter. We soon realized that we needed a way to change the delay parameter while the updates were in progress. If you miscalculate how quickly to reboot machines the servers could become overloaded with update requests. In the calculations above we guessed that a server could handle five connections simultaneously. What happens if the servers can only handle four connections? We wanted a knob we could turn to speed up or slow down the reboots.

We added this knob by having the reboot script read a file containing the delay time every time it was going to delay. We now pass in a file name instead of a delay time. When we start upgrading the workstations we try to use a larger delay than is really necessary. We then watch the server load as the workstations start updating. As the workstations complete their upgrade successfully we decrease the time delay in steps, thus rebooting machines more quickly, until the calculated frequency is reached.

Upgrading Servers

One tricky problem in an automated procedure of this type is upgrading your upgrade servers. Special care is needed while upgrading these servers, since some of them also run the various servers needed for doing the upgrades. We have managed to segregate the various servers well enough that we now do our global reboots in waves. First, we reboot our main `netdist` server. Then, we reboot the "database" servers one at a time. The database servers run `named`, the various AFS database processes, the hostclass servers, and the librarian services for `sasify`. Then we reboot half of the replica servers. These servers act as AFS replica servers, as `netdist` servers for HP patches, and as the download point for `sasify` data. Next we reboot the other half of the replica servers. At this point, all remaining workstations and fileservers can be rebooted.

Lessons Learned

Replicate, Replicate, Replicate

We discovered that it is important to replicate as many of your services as possible. This improves reliability, provides for load balancing, and allows for improved throughput for large-scale operations such as software updates.

Cleanly Segregate Functions

Try to segregate different functions on appropriate servers as much as possible. At one point, we were running AFS database servers on one set of machines, `named` on another set, and `sasify` librarian and data servers on a third set. In this case, trying to determine the reboot order was a nightmare. We eventually realized that named, the AFS database servers, and the librarian services all depended on each other. We relocated them onto the same set of machines, reducing the complexity of the problem significantly. Additionally, once we instituted the AFS replica servers, the reboot sequence became obvious.

Updating an Update Server

After a reboot, an average system will start its standard processes, run `sasify,` and then start its individual local processes. This means that on the `netdist` server machines we need to start any `netdist` servers before we start `sasify`. In addition, we modified `ticketd` and `sasify` to be smart about picking servers. If machine A asks for a ticket to a service that machine A provides, `ticketd` returns a ticket for machine A instead of whatever ticket was next available in the round robin queue. We made similar changes to `sasify`.

Centrally Administer Replicated Services

Central administration of our replicated services has made it much easier for us to maintain all of the necessary configurations. While this is not a luxury that some sites, particularly universities, have, we should still acknowledge its benefits.

Do Less, More Often

We discovered that it is easier and less risky to apply a few changes once a month than to apply a large number of changes a few times a year. This should be self-evident, but it did take us some time to realize this. This also allows us to track patches from HP more closely than we previously could.

Since we run an operation that is expected to be available 24 hours a day, 7 days a week, our upper-level management had to be convinced each time we need to do an upgrade that, overall, it was worth the downtime. Scheduled downtime only occurred 2-3 times per year, with many changes occurring at each of these outages. We were able to convince our management that the chances of a major error being made while only making a few changes was much smaller than if many changes were being made. We now schedule our downtime on a highly predictable monthly schedule with specific dates announced months in advance, This allows our product developers to schedule releases, regression tests, etc., without being surprised by scheduled downtime. In addition, we consolidate hardware changes to coincide with these scheduled maintenance times which has reduced the need for incidental downtime at other times. We are currently updating 1800 workstations each month.

Testing

When upgrading over 1500 workstations, a widespread failure can be particularly catastrophic and take a long time to fix. In this environment carefully controlled testing is extremely important before a major upgrade. We have a set of test machines where we do our initial testing. Once we are satisfied with these tests, we install the changes on the workstations of the UNIX support group. This gives us a chance to "test drive" the changes. During these tests we also time how long it typically takes to do an update, and use those numbers to help us determine how quickly machines can be rebooted.

The Time It Almost Didn't Work

When we first started updating the servers we had some network hardware problems. The symptoms were that the `sasify` program would hang without completing the transfer of the system disk image. We were worried that more network hardware failures could cause many workstation updates to fail in such a way that we would have to walk to them and restart them. We quickly wrote a program that would `fork()` and `exec()` its argument list and wait a specified number of seconds before killing its child process. This program was used to limit the time `sasify` could be hung. If `sasify` failed our script would restart it at the beginning (up to ten times). This change to fix the hung session problem was put into place after some testing.

We started the workstations while we analyzed the failures from the hung machines, still trying to find the original problem even though we had worked around it. Analysis of the hung machines suggested some additional changes could help prevent some of the failures. The new changes were added while the workstation update was in progress. There was little or no testing done to the new changes. (Big Mistake!)

Half an hour latter we started noticing that workstations were not finishing their update. We started to look for a reason and found a syntax error in the new changes. About 100 machines were trying to load the wrong things. We stopped the update process for the rest of the machines, then tried to figure out how to fix the 100 'broken' machines. We came up with a solution that required us to just restart `sasify`, tested it on a few machines, then started to walk to the 100 'broken' machines. Our use of `sortaddr` insured that they would be spread out all over our campus.

When we got to the first machine, we found it had already restarted `sasify`. We watched it until we knew it was running correctly, then we moved on to the next machine. It too was re-running `sasify`. It was then that we realised that the code to limit the execution time of `sasify` had kicked in, causing each machine to start over, this time executing the corrected code. It saved us from having to walk to 100 machines!

Conclusions

While the implementation that we describe was done on an HP platform, we believe that many of the concepts are generalizable to any UNIX platform. The specific tools that we use to manage this network are only part of the whole picture. You also need an overall policy which will guide your support structure development as your network grows. We have found that a comprehensive strategy, consistently applied across all support solutions, not just those designed specifically for software updates, makes performing major software updates highly efficient, even in a large network.

Availability

For information on the availability of any of the tools mentioned in the paper, please send email to heh@unx.sas.com.

Author Information

Helen E. Harrison is the UNIX Support Manager at SAS Institute Inc., where her group provides hardware and software support for a network of over 1800 UNIX workstations and servers. She has been involved in UNIX systems administration for over 12 years and holds a B.S. in Computer Science from Duke University. Reach Helen at SAS Institute Inc, SAS Campus Drive, Cary, NC 27513; or by e-mail at heh@unx.sas.com.

Michael Mitchell is a Systems Programmer in the UNIX Support Group at SAS Institute Inc. He has been involved in Distributed Computing for over 8 years and UNIX systems for 15 years. He holds a B.S. in Computer Science and a B.S. in Electrical Engineering, both from North Carolina State University. Reach Mike at SAS Institute Inc., SAS Campus Drive, Cary, NC 27513; or by e-mail at mcm@unx.sas.com.

Michael Shaddock is a Systems Programmer in the UNIX Support Group at SAS Institute Inc. He has been involved in UNIX systems administration for over 8 years and holds a M.S. in Computer Science from the University of North Carolina at Chapel Hill. Reach Mike at SAS Institute Inc., SAS Campus Drive, Cary, NC 27513; or by e-mail at shaddock@unx.sas.com.

References

1. Helen E. Harrison, Stephen P. Schaefer, and Terry S. Yoo, ''Rtools: Tools for Software Management in a Distributed Computing Environment,'' *Proceedings of the Summer USENIX Conference*, pp. 85-93, San Francisco, CA, June, 1988..
2. Mark Fletcher, ''doit: A Network Software Management Tool,'' *Proceedings of the USENIX Systems Administration (LISA VI) Conference*, pp. 189-196, October 19-23, 1992., Long Beach, CA.
3. Transarc Corporation, ''AFS System Administrator's Guide,'' FS-D200-00.10.4, pp. 14-1 - 14-26, Pittsburgh, PA.
4. Walter C. Wong, ''Local Disk Depot - Customizing the Software Environment,'' *Proceedings of the USENIX Systems Administration (LISA VII) Conference*, pp. 51-55, Monterey, California, November 1-5, 1993.
5. Wallace Colyer and Walter Wong, ''Depot: A Tool for Managing Software Environments,'' *Proceedings of the USENIX Systems Administration (LISA VI) Conference*, pp. 151-159, October 19-23, 1992., Long Beach, CA.
6. John P. Rouillard and Richard B. Martin, ''Config: A Mechanism for Installing and Tracking System Configurations,'' *Proceedings of the USENIX Systems Administration (LISA VIII) Conference*, pp. 9-17, September 19-23, 1994., San Diego, CA.