Software Maintenance in a Campus Environment: The Xhier Approach

Math Faculty Computing Facility*
University of Waterloo
Waterloo, Ontario
N2L 3G1 Canada

July 10, 1991

Abstract

Xhier is a system for software maintenance and distribution, currently in use at the University of Waterloo. It allows easy, automatic software installation on a variety of UNIX systems. This paper describes some of the design goals of **xhier**, its structure and operation, as well as some of the problems we have encountered, and some future goals.

1 The Motivation for Xhier

The Math Faculty Computing Facility (MFCF) is a major UNIX support organization at the University of Waterloo, and, in addition to other responsibilities specific to the Math Faculty, provides software support for UNIX machines all over campus using the **xhier**[1] software maintenance and distribution system.

MFCF started doing UNIX support before networks and workstations were so widespread. It was not too many years ago when the MFCF UNIX environment was a single DEC VAX 11/780 running 4.2BSD UNIX. In that environment, adding and updating software was relatively easy, since operating system releases were infrequent and the amount of third party software was small. It was easy to just add some source to `/usr/src` and install some commands in `/usr/bin`. If there were bugs in the OS that needed fixing, or enhancements that were desired, it was a (relatively) simple matter to modify the BSD source, and install the change. A few more BSD VAXes were added without too much pain, and careful use of tools like *rdist*(1) and *rcp*(1) made it possible to keep things up to date.

Once UNIX machines started being more common on campus, and different models and brands of machines started appearing, it quickly become obvious that a different approach was required. With operating systems being updated as often as every few months, different operating systems using different file system

*The work described in this paper is due to the efforts of all members of the Math Faculty Computing Facility software group, although the paper itself was compiled by John Sellens, currently with the Department of Computing Services at the University of Waterloo. Any and all errors, confusion and inaccuracies should be attributed to him. Contact `jmsellens@watserv1.uwaterloo.ca`.

[1] The name **xhier** is derived from the BSD *hier*(7) "file system hierarchy" man page, where the "x" prefix indicates eXperimental, though eXtended, eXtravagant, and eXhausting have all been offered as alternatives.

organizations, and with useful third party software proliferating, a more structured and automated approach was needed.

Development of the **xhier** system was started in early 1989 and has been ongoing ever since, with many enhancements and fixes still planned. MFCF currently provides software support for 11 different machine "architectures", using the automated tools provided by **xhier**. **Xhier** is currently used by over 250 machines, with over 1200 commands in more than 200 software "packages" available, with over 14,000 package installations on the various machines.

2 The Problem to be Solved

MFCF was faced with the task of providing software support for multiple architectures, serving multiple machine administrations, and different user groups and purposes. A research machine in the Computer Science department might have different needs than a student workstation cluster in Electrical Engineering, though both would want some form of software support, and some selection of locally installed commands. Additionally, most users want a consistent environment, regardless of the architecture and operating system of the machine they currently happen to be using.

Accordingly, it was necessary to find some way to distribute software to, and make it available on, large numbers of different machines. This had to be done with a large amount of configuration flexibility (so it could be tailored to the needs of different machines and administrations), and it had to be as automatic as possible, in order to make the most efficient use of the software support staff.

Faced with this environment, MFCF's work on **xhier** has proceeded with these primary goals:

- to simplify the distribution of updates
- to simplify the installation (on multiple architectures)
- to facilitate sharing software via remote file systems, and
- to minimize changes and additions to the file structure distributed with operating systems.

An important underlying goal is that of automation of as many aspects of the software maintenance and distribution process as possible.

Xhier takes a different and more automated approach than systems like the **depot**[1]. Appendix B discusses some of the differences between **xhier** and the **depot**.

3 The Overall Structure

In order to minimize the changes to stock file organizations, it was decided to locate **xhier**'d software in a separate (hopefully) unique location – under `/software`[2].

Under `/software` are the package directories. In **xhier**, software is organized into "packages" of related software. For example, there is a **tex** package, containing TeX related software, and a **gnu** package containing GNU software. Separating software into separate packages has two main advantages: it allows a

[2] `/software` is typically a symbolic link to some appropriate location, but references to **xhier**'d software use the standardized name `/software`, rather than the actual location of the directory on a particular machine.

```
% lc /software/x11
Directories:
.admin          bin             data            doc             export
include         lib             logs            man
```

Figure 1: A Typical Package Directory

system administrator to pick and choose what software to make available on a machine (which may depend on licensing or size considerations), and it allows the grouping of related software for maintenance and similar purposes. Additionally, it allows multiple versions of the same software to be installed and available at the same time.

The other goals are approached using the details of the organization of software packages, and by the use of support programs.

Xhier recoginizes that there are three main steps in making software available to users:

- The files must be placed in an appropriate location, and be compiled if necessary.
- The software must be configured and initialized appropriately.
- The software must be made available to the users (typically through search rules).

The structure and tools of **xhier** are intended to make these steps as easy as possible.

4 Software Package Organization

Each software package contains all the files associated with that software, along with other files required for the distribution and installation of the package. A package typically consists of a number of sub-directories, with the common ones being:

- `bin`, `maintenance` and `servers` contain commands for users, system and software maintainers, and other programs (respectively) to use
- `lib` for libraries
- `man` for man pages
- `data` for data and configuration files

The structure and contents of a particular package are determined by the needs of the package, but directories like `bin` and `man` are used by other programs to make package components available to users (more on this below). Figure 1 shows a typical package directory.

Two sub-directories are used by **xhier** for distribution and installation. The optional `export` directory (figure 2) contains information to be used outside the package. It contains such things as `services` and `inetd.conf` file entries required by the package (e.g. if the package contains a network daemon), commands to be run by *cron*(8), commands to be run on system startup, as well as information on userids and

```
% lc /software/x11/export
Files:
boottime       crontab        inetd.conf     services
```

Figure 2: A Typical Package `export` Directory

```
% lc /software/x11/.admin
Files:
Dependencies   Install        Maintainer     Options        Targets
file-types
```

Figure 3: A Typical Package `.admin` Directory

groups required by the package, and patches to standard system files like `/etc/rc`. These file fragments and patches are applied automatically to the system configuration files when a package is installed.

Each package is required to have a `.admin` directory (figure 3), which contains information used in the installation and distribution of the package. It typically contains 4 files:

- `Maintainer`, which gives the mail address of the person responsible for installing and maintaining the software
- `Dependencies`, which lists other packages required for proper operation of the package (e.g. a package might make use of commands provided by other packages)
- `Options`, which sets various options that determine how the package is distributed and installed
- `Install`, a program (typically a shell script) that performs any package specific initialization to prepare the package for use on a particular machine (e.g. initialize configuration files, check for errors, replace stock files if necessary, etc.)

A couple of other files (`Targets` and `file-types`) are sometimes used to influence the distribution of a package[3].

The package structure makes each package a separate entity, containing all the knowledge required to install and use the software it contains. It also provides a consistent structure so that packages tend to look very similar, which makes maintenance easier.

See appendix A for an example of how packages are created and installed.

5 Complicating the Structure

Now for the complication. Recent versions of UNIX have reorganized parts of the file system to make file sharing easier. For example, SunOS has `/usr/share`, which contains those files that can be shared among all machines, regardless of CPU type. **Xhier** takes this idea a step (or several steps) further.

[3]These files are somewhat of a kludge, and may disappear in the future.

Separating files based on their type has two main advantages, both of which help to satisfy the goals of **xhier**. It makes it easier to share software via remote file systems (e.g. NFS), because you can share as much of a package as possible, and minimize duplication across machines of the same or different hardware architectures. And it makes software distribution easier because it makes it easier to use a tool like *rdist*(1) to distribute the appropriate file types to the appropriate machines e.g. shared files like man pages can be distributed everywhere, while executable binaries can be distributed to just those machines with the same architecture.

Xhier currently recognizes six file types:

- **share**: files like man pages, shell scripts, and (typically) the `.admin` and `export` directories
- **arch**: files that are different on different hardware/operating system combinations, most commonly executable binaries
- **spool**: files of a transient nature, like print requests
- **local**: files specific to a particular machine, such as local configuration or log files
- **regional**: files shared between clients and their password and home directory server, such as nntp server names
- **admin**: files shared among all machines in a particular administration, such as printer permission files

The **regional** and **admin** types are variations on the **local** type that seem to be useful. In practice, it seems that there are actually more file types (e.g. **man** might be used to describe man pages and documentation, because you might not want those locally on *every* machine, in order to save space), but these six types are recognized explicitly in the **xhier** file structure.

This is accomplished through the use of the `/.software` directory. `/.software` contains one sub-directory (or, more typically, a symbolic link to, or an NFS mount of, an appropriately located directory) for each of the six basic file types, with each sub-directory containing package directories. For example, the man pages for the **tex** package are in `/.software/share/tex/man`, while the command binaries are in `/.software/arch/tex/bin`. Note that `/software` and `/.software/share` are the same place[4].

Every package has a shared component; in particular, the `.admin` directory must exist, and is always a shared directory. And since `/software` and `/.software/share` are the same place, this ensures that, for a package called **pkg**, `/software/pkg` will always exist.

It is desirable to hide the file type structure of a package from the outside world, so that a user or another program does not need to be concerned with the internal structure of a package. So, even though every file in a package can be referenced through `/.software` type directories, references from outside the package itself are always made through `/software/pkg`. (A package is allowed to know its own structure, and refer through `/.software` as appropriate.) This is accomplished by the (liberal) use of symbolic links in the `/software/package` directory. For example, `/software/tex/bin` is actually a symbolic link to `/.software/arch/tex/bin`. It is also possible to have a shared directory contain real files along with symbolic links to files in directories of other types. For example, a package might have commands that

[4] This means that one of `/software` and `/.software/share` is redundant, but `/software` was the original name, and provides a nice name for people to use, while `/.software/share` was included primarily for completeness.

```
#include "../PackageName"
Program(callpat,c,user)
```

Figure 4: Typical `xh-imakefile` file for use with *xh-make*

are primarily shareable shell scripts, with the few compiled commands being referenced through symbolic links in the shared `bin` directory[5]. Note that these links, while pointing to different absolute locations on different machines, are shareable, since they link through `/.software`.

The result is that the file type directories under `/.software` provide separate hierarchies for different types, which makes file sharing and distribution much easier. For example, diskless workstations will usually share and mount all file types from their server, with the exception of the `local` type, which each client will have its own version of. The `/software` link provides a naming convention for access to the components of a package. Note that a given machine will have the correct type hierarchies for itself only, e.g. `/.software/arch` on a VAX contains only VAX binaries, and on a Sun will contain only Sun binaries.

6 Automating Everything

Xhier is built on the idea that automation is good. Accordingly, there is an **xhier** program to do just about anything to do with software installation and distribution.

There are currently about 80 commands in the **xhier** package, most of them shell scripts. Commands are used for such things as package creation, maintenance and installation, and distribution, along with some utilities that are useful for day-to-day operation (log file rollers, etc.). The following is a summary of some of the more prominent commands.

The *xh-mkpkg* command creates a package skeleton in the `/software` directory, with command and support directories, and some template files, that can be filled in by the package creator. This makes the initial creation of a package simpler, because a package creator doesn't have to remember all the details. *xh-make* is used as a cover for *make*(1) and *imake*(1) and uses a standard *imake* template to make it easy to compile and install a program (see figure 4). *xh-make*, when used with an `xh-imakefile`, ensures that any architecture-specific libraries and commands are used.

Once a package has been compiled and installed into the package directory, the *xh-install* command makes it available for use. It does this by running the package's `Install` script, to do any package-specific tasks (such as package configuration, or replacement of any stock files), and then, if that is successful, invoking other programs to patch system files (like `/etc/services`) from the fragments in the package's `export` directory.

Once a package has been installed, and is working on one machine[6], it can be distributed to, and built on other machines. This is a two-step process. The *xh-distribute* command is used to distribute the package contents and structure to other machines. It creates a `Distfile` for use with *rdist* that sends the appropriate

[5]This latter structure, while within the rules, can sometimes be confusing, so it is usually better to make the directory the symbolic link, and its contents be real files, rather than a mixture of the two.

[6]Currently, this must be a machine higher up the tree, usually the one central machine, but this will change in the future, so that any machine can be the initial "maintenance" machine for a package.

files to the appropriate machines lower on the tree. For example, it will send the shared files everywhere, but architecture specific files will only be sent to machines of the same architecture (though it will complain if files on a machine of a different architecture are out of date, or missing). *xh-distribute* then invokes *rdist* and summarizes the output for easier human consumption.

The second step is *xh-sdist*, which distributes the source for the package to machines of other hardware/software architectures (the "architecture master" machines)[7], and runs *xh-make* on those machines in the background, collecting the output and mailing it to the invoker. *xh-sdist* has a number of options and configuration files to control exactly what gets distributed to the remote machines (e.g. `.o` files are not distributed). *xh-sdist* has turned out to be an incredibly useful tool – it makes updating software and installing fixes almost trivial[8].

To help ensure that software is kept up to date, working, and consistent on all machines, *xh-maintenance* is run weekly from the *cron* (on every **xhier**'d machine). *xh-maintenance* does some housekeeping and consistency inspection, and runs *xh-install* and *xh-distribute* on all packages, arranging to send the appropriate output to the appropriate package and system maintainers.

Since *xh-maintenance* is run weekly, starting at the same time on every machine, it can take several weeks for package updates to reach the bottom of the distribution tree[9]. And so the other important distribution-related tool is *xh-dist-recurse*, which attempts to distribute a package recursively down the distribution tree. This can be very useful in distributing fixes quickly, in case you've inadvertently broken something and don't want to wait for the usual propagation delays to distribute the fix.

7 Making it all Available to the Users

In a conventional software installation, software is installed in one location only (commonly `/usr/local/bin`), or in other OS-specific directories (such as Sun's `/usr/openwin/bin`), which the user is expected to add to `PATH`. Expecting a user to know where software is installed is not always reasonable. For many years, MFCF has had a command, *showpath*, to help users set their `PATH` without having to know the specifics of a particular machine.

With **xhier**, once the packages are installed, there are many different package directories, each containing command directories (e.g. `bin`), `man` directories, `include` directories, and/or `lib` directories. The *showpath* command could simply produce a list of all the `bin` directories in all the installed packages. This, however, quickly becomes too cumbersome to handle – a 3,000 character `PATH` containing 200 directories is a little large. It also becomes inefficient since some shells hash the `PATH` contents, but often only the first few elements (csh apparently hashes the first 8 elements of `PATH`).

Xhier approaches this problem by creating a directory that contains a link to each of the installed, packaged commands, using the *xh-make-links* command. These links have to be symbolic links, since the commands are not always going to be on the same filesystem (or even the same machine). There are actually three of these directories, one for user commands, one for maintenance commands, and one for commands used by other programs (servers). These directories are system-wide and system-specific in that they contain links to the commands for only those packages that the system administrator wishes to be

[7] *xh-sdist* sends the source to a temporary directory on the remote machines, and it is automatically removed after a few days.

[8] It should be pointed out that *xh-sdist* is somewhat of a kludge, because it introduces an arbitrary distinction to some machines, that of being an architecture master, when it should be possible to distribute source to and build a package on just about any machine.

[9] This propagation delay can be a blessing, because it can keep broken packages from infecting the entire distribution tree.

```
setenv PATH    `showpath usedby=user $HOME/bin standard`
setenv MANPATH `showpath class=man standard`
setenv EDITOR  `showpath findfirst=vi`

% echo $PATH
/u/jmsellens/bin:/.software/local/.admin/bins/bin:/usr/ucb:/bin:/usr/bin
% echo $MANPATH
/software/.admin/man/:/usr/man/
% echo $EDITOR
/.software/local/.admin/bins/bin/vi
```

Figure 5: Typical extract from a user's `.cshrc` file, and resultant variables

"default" packages[10]. A user's `PATH` consists of the directory of links, the appropriate stock directories (for that architecture), and any other directories the user wishes to include. These directories of symbolic links might be expected to cause a performance penalty on command invocation, but, in practice, it seems not to be a problem.

This "searchrules" approach to commands works well, but it is harder to apply it to the other parts of a package. Many *man*(1) commands understand a `MANPATH` variable, but most compilers and loaders don't have an easy way (i.e. an environment variable) to extend the searchrules for include files and libraries. Include files and libraries are linked (again using *xh-make-links*) into the standard locations honoured by the OS, usually `/usr/include` and `/usr/local/lib`.

One more minor complication. MFCF had been using the *rman* remote man command[11] to provide man pages to smaller machines, where there is often not enough disk space to keep man pages locally. Local modifications to *rman* allow the remote man page server to deal with packaged man pages by special interpretation of the `MANPATH` variable.

Figure 5 shows typical commands that would be included in a `.cshrc` file, and common values of key environment variables.

8 Some Lessons Learned from Xhier

Xhier tends to point out some of the more obscure points of software installation. One prime lesson is that doing software distribution can turn out to be a complicated process, with a certain amount of overhead.

Automation is a necessity, once more than a few machines are involved, but it turns out that some things can be hard to automate. For example, it's not too hard to add an entry to the `/etc/services` file, but it's a little harder to notice conflicts or replacements, and some packages might want a particular userid to exist (e.g. "games"), but it's hard to add an appropriate account to a `passwd` file automatically. The current **xhier** utilities handle *inetd*, *cron*, and `/etc/rc` entries fairly well (by automatically editing the appropriate system files, and disabling conflicting stock entries), but `passwd` and `group` file entries still

[10] In practice, the set of "default" packages is virtually the same everywhere.
[11] By Jonathan C. Broome, posted to `net.sources` in 1985.

have to be applied by hand (local administrators are prompted to do it when a package is installed). And while automation allows easy installation and distribution, it also makes it easy to distribute broken software and automatically destroy dozens of machines.

Some software comes with particular pathnames built-in, often hard-wired pathnames are sprinkled throughout the code (especially files destined for `/etc`), and it's often hard to change these pathnames to refer to locations under `/software` instead. In other cases, they are easy to change, but the same symbolic name is used for different kinds of files. For example, the *make* variable "USRLIB" might be set to `/usr/lib`, and be the location for both object libraries and data files, or "ETCDIR" might be used to contain local configuration files and system maintenance commands. Files are often named for their location, rather than for their function. **Xhier**-ing a package forces an installer to interpret the given pathnames, and choose appropriate **xhier** pathnames instead.

In addition, certain pathnames, like `/usr/lib/sendmail` and `/usr/ucb/lpr`, are known to many programs, and so any replacement versions of these programs must also replace the well-known stock pathnames (with links to **xhier** file locations).

It's hard to organize NFS (or, presumably, other remote file system) mounts to make software sharing foolproof. This problem is especially evident with symbolic links. A symlink to an absolute pathname will start looking for the file at `/` on the local machine, even though the link is on the remote machine. And conversely, a relative symlink will tend to stay on the remote machine, unless it passes up through a mount point. Some uses seem to call for absolute symlinks, and some seem to call for relative symlinks, but so far we have been unable to come up with a rule (or even a guideline) that works in all cases.

Choosing what goes in what package is hard. It's not always easy to group software into appropriate packages, and different administrators might want different parts of a package, instead of the whole thing. Large packages are easier to deal with, but small packages give greater flexibility.

It's easy to generate lots of junk mail, so efficient ways of summarizing and distributing it are required.

9 Future Plans and Dreams

The current distribution mechanism uses a tree structure, with each member in the tree "pushing" packages to lower nodes in the tree. This is less than optimal, but was necessary since the only distribution tool available at the time of implementation was *rdist*. Some of the problems with this approach are

- Distribution is scheduled by the upstream host(s). A subsidiary node has (essentially) no control over when its software gets updated. This is annoying when a guaranteed stable environment is desired, such as when one is trying to finish a thesis, or a machine is used for student classwork. In practice, this hasn't turned out to be a problem, since a request to an upstream administrator is usually enough to modify the distribution list.

- It is inconvenient to have two (or more) distinct machines be the "masters" of two different packages and exchange packages with each other, because the use of *rdist* forces essentially "wide-open" access for the upstream machine on the downstream machine (via `/.rhosts`).

- This style of distribution seems to require wide open access to a machine by its distribution machine.

- Its hard for a given machine to dictate exactly what software it gets, because that is decided by the upstream host(s).

For these reasons, distribution control will pass to the receiving machines, and the tree structure will be replaced with a more flexible one, where any two cooperating machines can exchange whatever software they wish to.

This, however, requires a program like *track*(1) that can do most of the same things that (the locally enhanced) *rdist*(1) can do. This work is currently underway. It also requires fairly major changes to the mechanisms that determine which parts of which packages get sent to which machines. These changes are almost in place, to be used with the current distribution mechanism until something "track-like" is ready.

Most machines don't have enough local disk space to contain all packages, or even just the ones they want to use. Current space usage on the primary maintenance machine[12] is

```
% du -s /.software/*
105     /.software/admin
272164  /.software/arch
11823   /.software/local
727     /.software/regional
231340  /.software/share
```

or a total of over 500 megabytes, in addition to the space required by the OS[13]. As a result, remote file system mounts (currently only via NFS) are required to make the necessary software available on many machines. This is usually done in bulk, where a machine might mount all of `/.software/share` and `/.software/arch` from a willing server. Things get complicated when you wish to have some packages locally and some remote mounted, since you either have to mount each package individually, or arrange that `/.software/*/package` are either real directories or symbolic links through the appropriate mount points.

At present, *all* remote mounting is done manually, which is getting less and less practical. Some method needs to be devised to automatically make the appropriate mounts and links, which will allow a local administrator to easily add a software package to a machine. It's not at all clear what the best approach to this problem is.

Xhiering a software package forces a software maintainer to look beyond the local machine, and consider how a package would be installed and used on different machines, with different administrators, perhaps wishing different defaults or environments. However, the current outlook tends to stop at the edges of the campus. A "shared" file tends to mean one that applies to the campus as a whole, rather than to the "world". In fact, there is currently no way in **xhier** to indicate a file with type "campus" i.e. there is no `/.software/campus`. It would be nice if it was possible to distribute **xhier** and **xhier**'d packages to other organizations, but that brings other problems to light, such as the necessity to trust a remote software provider.

10 Conclusions and Observations

With **Xhier**, it is now almost trivial to make a new software package available to hundreds of machines. It's easy to make and distribute fixes or enhancements.

[12] "watmath" a MIPS RC6280

[13] This excludes the spool files (especially news articles) under `/.software/spool`.

Xhier allows MFCF to provide software support, and a consistent user environment, on many more machines than would be otherwise possible.

Xhier does have a number of faults

- It is large, and still changing, and it is not always easy to understand how all the parts fit together. However, most software maintainers do not need to know how it works, just how to create a package and make it available.
- It tends to create a maze of symbolic links.
- Requires a fair amount of machine overhead to keep it shuffling software around.
- It relies on having a fairly BSD-ish set of system services.

but overall it tends to meet the needs. For MFCF and the University of Waterloo at least, **xhier** is a success.

A Creating and Installing a Package

Most software packages are very easy to create, though large pieces of software, like *gcc* or the X11 software, are more complicated. Typical package creation and installation goes something like the following example installation of package **foo**.

First, the source is unpacked into `/usr/source/foo`, and the `Makefiles` are modified as appropriate, to set the right options, and installation locations. Many times this is done using an `xh-imakefile` (see figure 4 for an example) that is processed with *xh-imake* to allow easier per-architecture customizing.

The command

```
% xh-mkpkg foo
```

is used to create a skeleton structure in `/software/foo`. This structure is then adjusted by hand as required. Usually this just means removing the parts of the skeleton not required by this package.

Then the files in the `.admin` directory are edited as appropriate. The appropriate options are set from the defaults in the `Options` file, and the `Install` script is modified to do the appropriate things on installation and uninstallation. Often this means just ending up with the script

```
#!/bin/csh -f
exit 0
```

The **foo** package name is then "registered" by adding it to the *software*(i) man page[14].

The software is installed via

```
% xh-make install
```

in the source directory. Once that completes, the commands

```
% xh-install foo
% xh-distribute foo
% xh-sdist -m install
```

[14] This step needs a little refinement.

install the package locally, and then distribute the package structure and source to the other architecture master machines, and compile and install the package there.

Once this has been done, the **foo** package is available to any **xhier**'d machine on campus, with very little effort.

B Comparison to the Depot

The **depot**, as described in [1], has some similarities to **xhier**, but is less automated, and is based on a client-server distributed file system environment, rather than **xhier**'s greater emphasis on the use of local file space to hold packages.

Both systems separate software into packages, and define a structure for the individual packages. The **depot** requires human intervention for each installation, in order to execute an optional installation script, and to modify system files (such as `inetd.conf`) if necessary; **xhier** uses the *xh-install* program and the package's `Install` script. The **depot** uses redundant servers for the packages, but distributing a package to a redundant server is done manually; **xhier** has automatic distribution of the packages. The **depot** is geared primarily to the use of server machines that contain all the files for all architectures. This makes it less useful for standalone machines (such as a home workstation), since such a machine would have to have the binaries for every architecture, requiring more disk space (unless manual pruning was used).

Maintenance activities are less automated in the **depot** as well. In order to recompile or update a package, a maintainer has to go to the **depot** administration machine of the appropriate architecture, and compile and install the package there. **Xhier**, on the other hand, uses the *xh-sdist* command to rebuild and install a package on all subsidiary architectures, which helps ensure that packages are kept up to date on all architectures.

Since the **depot** uses *automount* mount points, as listed in an NIS database, to make packages available to client machines, it is harder for a system administrator to tailor the set of packages available on a particular machine or set of machines. **Xhier** makes it easy for an administrator to pick and choose what packages are available on his or her machine(s). Admittedly, in a primarily client-server, workstation environment, this is less of a problem.

With **xhier**, a system administrator can control when packages get updated on his or her machine, which is useful in a student lab environment, or on a personal workstation when trying to complete a project. Since the **depot** is based on the client-server model, it is much more difficult for a client to "refuse" an update.

Xhier provides a set of tools for modifying, or replacing vendor provided files, which makes it possible to correct flaws in stock systems, or customize them to local requirements. **Xhier** provides simple interfaces to system services such as `/etc/rc` and *cron*. This means that a package maintainer does not usually have to worry about system-specific details.

Xhier makes extensive use of search paths in order to hide the details of an implementation from the users. With the **depot**, a user must know where programs and man pages are located in the hierarchy in order to make use of them.

C Xhier Availability

Xhier is still under development, and still contains many transitional provisions to correct past mistakes, and it is currently difficult to bootstrap **xhier** into a new environment (such as another campus). It is also a large collection of source, which relies in part on modifications to licensed software.

As such, **xhier** is not currently available for distribution, though we hope to be able to make all or part of it and its tools available at some point in the future.

References

[1] Manheimer, K., B. A. Warsaw, S. N. Clark, and W. Rowe, “The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries”, *LISA IV Proceedings*, October 1990, pp 37-46.