Constructing An Application Server

In a dynamic networked environment in which there are several hundred workstations and servers there is a constant demand for new versions of software. In this environment software installation procedures must be quick, flexible, and tolerant of change.

by Jill E. Swenson

With the rapid advancements and constant changes associated with computer technology, new and better hardware and software capabilities can always be expected. Hardware comes and goes as needs fluctuate, and software popularity tends to bloom and fade as fast as software features evolve. All this change makes managing software installation and updates across more than 600 UNIX® systems a very challenging task. This paper describes a project to simplify software administration among a group of workstations at HP's Integrated Circuits Business Division.

At one time our division software was purchased by system owners and installed on individual workstations as requested. Users shared common applications by connecting their workstations in HP-UX* clusters in which one machine's disks were used by many diskless workstations (cnodes).[1] These clusters grew large, and access to software became a more important component in deciding system configuration than performance or networking issues. Some software was shared between cluster servers through NFS mounts, and this led to what we call "spaghetti mounts," which is many machines mounting portions of each others' disks to access shared software (Fig. 1). Additionally, software installations were not tracked, software versions were not synchronized, and network licensing was not effectively implemented.

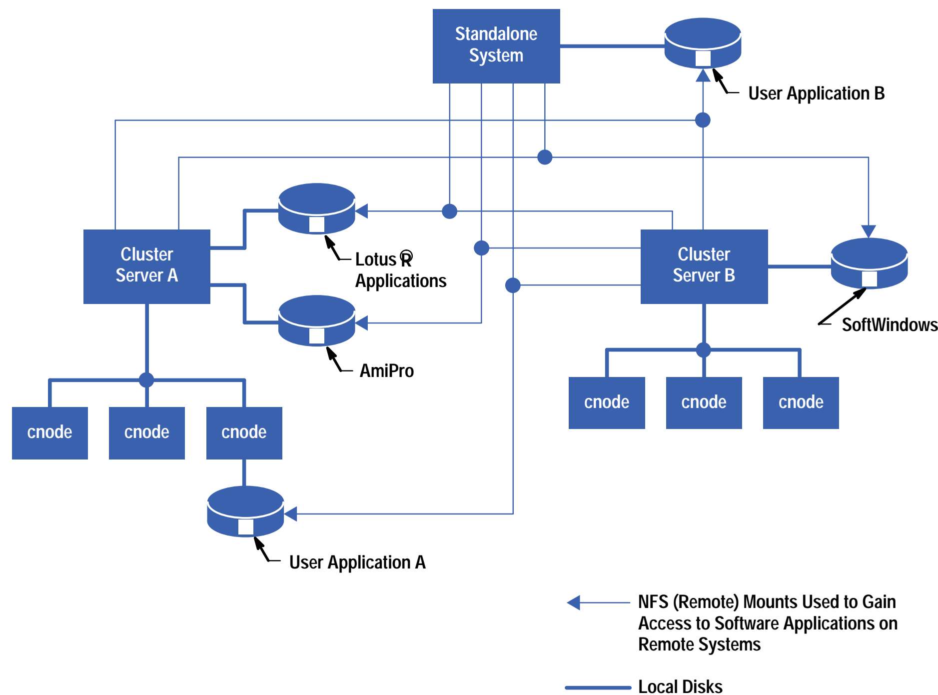

***Fig. 1.** An example of "spaghetti mounts," which is many machines mounting portions of each others' disks to access shared software.*

My goal for this project was to find a way to reduce users' dependencies on cluster configurations, to untangle the mounts, and to improve the way we installed, updated, removed, and tracked software. To do this, I moved the application code files to a central server and connected all the user workstations to this server (see Fig. 2). File servers are not an unusual concept, but application servers create unique challenges, and this one needed to be as easy to use as possible.

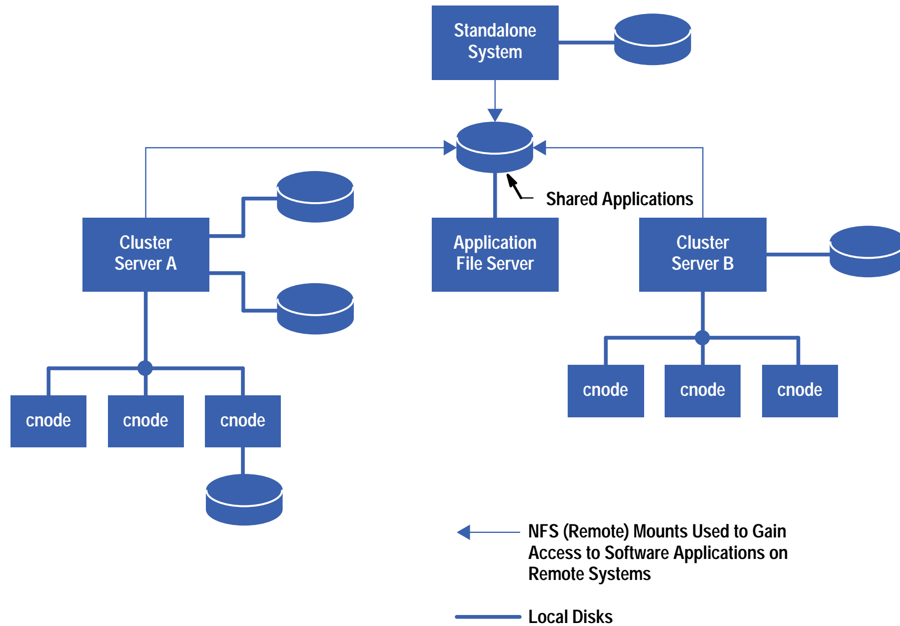

Fig. 2. A network configuration in which many machines are mounted to an application server to gain access to shared software.

Developing the Structure

Since we had some spare equipment and disk space, a central software server was a convenient choice for the application server. The basic concept was in place—add disks to the application server, install software on those disks, mount those disks to client machines, and enable the client machines to run the software. Two issues that had to be dealt with were how to mount disks so that symbolic links would be followed correctly on the application server and on the clients, and how to isolate each application and allow it to reside in its desired location on each system. Finally, since saving administrative effort was equally important, the entire process needed to be automated.

A search of the literature produced a helpful article describing how to manage local software on a network.[2] Even more conveniently, the article included the author's installation script. Although not entirely suitable for our environment, this article provided valuable ideas for establishing mount points, isolating applications, and creating configuration files for each application.

Mount Points

To identify mount points, I looked at documentation for the HP-internal UNIX Common Operating Environment (UX-COE) and the HP-UX 10.0 operating system. Both documents recommended using different locations for mounting local and remote (NFS-mounted) disks. This meant that disks physically connected to the application server would be mounted in one location on the server (e.g., /mnt/disk1) and in a different location on the server's clients (e.g., /nfs/servername/ disk1) (see Fig. 3). This approach makes logical sense, but it is limited because there is a route to a file on the server that the client doesn't know about. In other words, the server could refer to a file by the name /mnt/disk1/filename, but that name would not be valid on a client. The client must call that file /nfs/servername/disk1/filename.

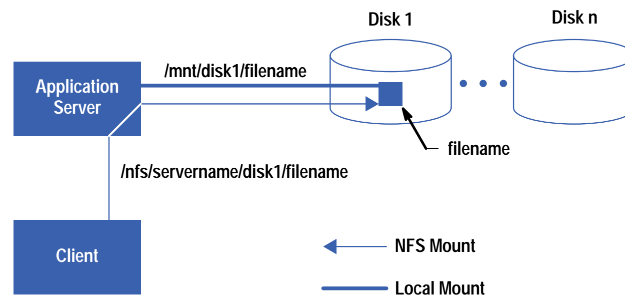

Fig. 3. An example of using different locations to mount local and remote disks.

Why Is this a problem?

Calling a file by multiple names typically does not cause problems. However, most application installation programs have a problem with multiple names because they act under the assumption that code is installed on the machine where it will be used and not on an application server. Many installation programs detect the directory into which code is installed and then plug that directory name into scripts, use it to create symbolic links, and so on. The resulting scripts and links work correctly on the application server, but fail on client systems.

To handle this, disks were mounted in the recommended locations and symbolic links were created on the application server to allow it to refer to its own disks under the names used by the clients. Thus, when software is installed it is always redirected, if possible, to the name used by clients.

In retrospect, this was not the best solution. Some particularly difficult installation processes resolve all symbolic links to their absolute paths and use these paths to create scripts and symbolic links. Thus, no matter what name was used for the installation directory, I would find some reference to /mnt/disk1 tucked away in a software configuration file or script. The only way to fool these programs is to either ignore recommended standards and mount the disks to /nfs/servername/disk1 on both the client and the server, or create "fake" symbolic links to the mount points on both machines.

After considering both options, I determined that the first option had the least number of disadvantages and was the simplest to implement.

Software Organization

The next question was how to organize the software on the shared disks. Opinions varied. One approach was to put all code into a common bin directory and set it up so that the administrator owned the bin directory on all machines. Although this would ensure consistency, it wasn't very flexible. Since all users don't have identical software needs (some may be testing a new version of software while others might be installing software packages that are not shared with others), it did not make sense to force all systems to have an identical bin directory.

The approach used in **reference 2** suggested isolating each application in its own directory tree and organizing the components of the application in explicit directories (documentation files under the doc subdirectory, binaries under bin, etc.). This idea of isolation was good, but it required spending time unnecessarily reorganizing the code components.

The software packages we were dealing with came from multiple developers, each of whom had a different idea about where an application's files should reside and the relative location of different components such as library, binary, and configuration files. There was some concern that many of these software packages would not react well to being moved around or being forced to run out of an unexpected directory. The best approach was to install each application into a separate directory on the shared disks, and to use symbolic links to allow the application's files to be logically located wherever they would normally expect to be installed.

To keep this neat, each software package is installed into its own directory as if that directory were the root directory on the system. In other words, if an application called editor came with a file called editor.dict, which it expected to reside in /usr/local/lib, a directory for the new application would be created:

```
/nfs/servername/disk1/editor
```

and the editor.dict(ionary) file would be installed in

```
/nfs/servername/disk1/editor/usr/local/lib
/editor.dict
```

When this software is finally hooked up to client systems, they will each have a symbolic link:

```
/usr/local/lib/editor.dict ->
/nfs/servername/disk1/editor/usr/local/lib
/editor.dict
```

The software can find its data file where it expects, and all the files associated with the editor program can be isolated in one directory tree which can be easily removed, replaced, or modified as needed.

Each application's directory tree mimics the root file system, making it easy to see where the software components will ultimately reside on client systems. Developers like the design because they can easily see the relative relationship of all their files in isolation.

There are two significant issues with this approach. First, it requires that a large number of symbolic links be maintained on multiple systems. Second, some applications might require configuration changes on the client or insist that certain files physically reside on the client's hard disk.

The solution to both of these problems is a configuration file and a process (script) to use it.

Configuration File

Every application on the application server has an associated configuration file. This is an ASCII file, created manually, that contains all the instructions necessary for installing the application on a client. For most applications, the configuration file simply lists a number of symbolic links to create. In the example above, one line in the editor program's configuration file would look like:

```
LINK:/nfs/servername/disk1/editor/usr/local/lib
/editor.dict:/usr/local/lib/editor.dict
```

Each line in the configuration file contains a tag field followed by several colon-separated parameters. The tag field is a word that has meaning to the configuration script. In this example, the LINK tag tells the script to create a symbolic link using the next two fields as arguments. The syntax for the LINK command is:

```
LINK:/link/to/directory:/link/from/directory
```

Additional tags cause the configuration script to copy files (COPY), unlink directories (UNLINK), or execute programs.

Creating a configuration file is the most time-consuming action required when installing an application. It requires a good knowledge of the directory structure on client machines and an understanding of the run-time environment expected by the application. The overall goal is to minimize the number of links while allowing multiple applications to be installed where they wish. However, with a little creativity, nearly any application can be dealt with, and once the configuration file is constructed, it remains unchanged and can be used on every client.

The configuration file is also used to uninstall software. An option on the configuration script causes it to deal with certain commands in reverse, removing links and files that had been copied to the client.

In accordance with HP-UX 10.0 file structure recommendations, the configuration file is created in /etc/opt/appname/config.fs. To make the configuration files readable from client machines, they are placed under each application's directory tree. Thus, the /nfs/servername/disk1/editor/etc/opt/editor/config.fs file would be the configuration file for the editor application mentioned above.

The Script

The configuration script is used to install an application located on a file server on a workstation (i.e., it sets up links to applications on the server). Because the configuration (and application) files reside on a remote server, a new client must mount the remote server's disks before installing software. The installation script establishes mounts (-m option) to the file server and creates the required links (-c option) using information from the configuration file. The installation script has an option (-u) to undo a mount and uninstall software. The command line to the configuration script to establish the mount points and set up the links for the editor application would be:

```
scriptname -m servername:/mnt/disk1:/nfs/
servername/disk1 -f -c /nfs/servername/disk1/editor/etc/opt/editor/config.fs
```

The –m option introduces the server and client involved in establishing the mount points, and the –f and –c options introduce the configuration file.

The configuration script is written in Perl (Practical Extraction Report Language) because this is the language that provides the easiest and most efficient techniques for reading through the configuration file once, storing the information, and acting on it later. Shell scripts require more coding to perform the same steps, and a C program would have to be compiled for every platform we support. Thus, Perl was a good choice. Fig. 4 shows the portions of the installation script responsible for mounting disks and reading a configuration file.

The configuration script reads and acts on instructions in the configuration file. It checks the first field of each line for a tag, which is a special word (e.g., LINK) describing a command. Unknown tags are ignored.

Because the configuration files reside on the server, clients must mount the server's disk before attempting to read any configuration files. Although the configuration script can be used to mount the server's disk, it does not currently modify a client's boot procedures or its /etc/checklist file. The system administrator still needs to perform these steps manually.

The configuration script uses the same configuration file for installing and uninstalling. It does an uninstall by removing all files or directories listed as LINK or COPY in the configuration file.

Remaining Issues

With any project there are always lingering issues. This section describes some of these issues.

Software Location and Fine-Tuning. Although it's convenient to locate software centrally, not everything belongs on a remote machine. Ideally, enough software should remain on local workstations so that the users are reasonably functional when the application server is down for maintenance. This means that key files such as operating system components, screen icons, and critical applications such as electronic mail readers should reside on local systems. Also, performance is impacted by accessing remote files so finding the right mix of local and remote files can have a significant role in application performance.

```
# Options:                                      #
#  -m Mountinfo: Revive/establish mount points #
#  -c Configfile: Revive/establish links       #
#  -f:Force symlinks and mounts (removes       #
#     existing files and directories).         #
#  -u:Unmount or remove links, depending       #
#     on other options used                    #
#  -v:Verbose                                  #
#                                              #
require 'getopts.pl';
do Getopts('m:c:fuv');  (A)

# Initialize variables
$numlinks=0;
$numunlinks=0;
.
.
.
################################################
# Read mount information.                      #
################################################
sub readmount {
 print "Starting readmount subroutine \n"
 if ($opt_v);

# The mount parameter consists of the machine
# to mount from, the directory on that machine
# to mount and the directory to mount to (on
# the client machine), all separated by ":"
# characters.
# Break line into its components.
       (B)         (C)           (D)
 ($machine, $mountfrom, $mountto)=split
 (/:/,$mountpoint);
 .
 .
 .
}# End subroutine readmount

################################################
# Perform mounts, creating directories, if     #
# needed.                                      #
################################################
sub makemounts {

 $dontdo=0
 if (substr($mountto,0,1)eq '/')
    {
    print "Making" .$mountto ."\n";
    if ((system("/usr/bin/bdf$mountto|/bin/
    grep $machine:$mountfrom"))==0)
     {
     print "Correct mount point already exists\n";
     $dontdo=1;
     }
    &makedir ("$mountto");  (E)
    print "mounting $machine:$mountfrom\n";
    if ((-d $mountto)&&(!$dontdo))
     {
     system ("/etc/mount -o soft $machine:
     $mountfrom $mountto");  (F)
     }
    }else
    {
    exit 97;
    }
} # End subroutine makemounts

################################################
# Read config file, filling up variables with  #
# contents.                                    #
################################################

sub readconfig {
  print "Starting readconfig subroutine \n"
  if ($opt_v);
  while ($_=<CONFIGFILE>){  (G)

# The configuration file consists of an initial
# tag and colon-separated parameters. Break each
# line into its components

   chop;
   ($tag, @parms)=split(/:/)
   .
   .
   .
    if ($tag eq "LINK")  (H)
    {
    $linkto[$numlinks]=$parms[0];  (I)
    $linkfrom[$numlinks]=$parms[1];  (J)
    $numlinks++;
    }
   .
   .
   .
  }# while
} # End of subroutine readconfig

################################################
# Process links: Create all links specified in #
# config file (must run readconfig() first).   #
################################################
sub linkall {

 print "Starting linkall subroutine \n"
 if ($opt_v);

 for ($i=0; $i<($numlinks); $i++)
   {
   $linkfrompath=$linkfrom[$i];
   $linkfrompath=  s'[a-z_0-9.\-]*$";
     # Chop file/dir name off path

# If parent directory path doesn't exist,
# create it.
&makedir("$linkfrompath");

# Check for existing file/link (exit or
# remove it depending on -f option)
if (-e $linkfrom[$i]&&(!$opt_f))
     {
      print "Non-Link exists:" .$linkfrom[$i].
       "Terminating\n";
```

Fig. 4. *Portions of the configuration script for mounting disks, reading configuration files, and creating links.*

```
        exit 98:
        }

      elsif (-e $linkfrom[$i])&&($opt_f))
        {
# Force (-f) section — force the link (remove
# existing file or directory tree.
        if (-f $linkfrom[$i])
          {       ~
          print "Forcing removal of File"
          .$linkfrom[$i]. "\n";
          unlink ($linkfrom[$i];
          }
         elsif (-d $linkfrom[$i])
          {
          print "Forcing removal of directory"
          .$linkfrom[$i]. "\n";
          system ("/bin/rm -rf $linkfrom[$i]");
         }
         .
         .
         .
     }
for ($i=0; $i<($numlinks); $i++)
     {
     symlink ($linkto[$i], $linkfrom[$i]; (K)
     print "Linking from " .$linkfrom[$i].
     "to " .$linkto[$i]. "\n";
     }
}
```

(A) Input String: `-m servername:/mnt/disk1:/nfs/servername/disks -f -c /nfs/servername/disks/editor/etc/opt/editor/config.fs`

(B) Server's Name

(C) Server's Name for Disk1 `mnt/disk`

(D) Client's Mount Point `nfs/servername/disk1`

(E) Make Directory `/nfs/servername/disk1`

(F) Mount `nfs/servername/disk1 to mnt/disk1`

(G) `config.sys`

(H) Tag = LINK

(I) `/nfs/servername/disk1/editor/usr/local/lib/editor.dict`

(J) `/usr/local/lib/editor.dict`

(K) Create Symbolic Link: `/usr/local/lib/editor.dict -> nfs/servername/disk1/editor/usr/local/lib/editor.dict`

Fig. 4. (Continued)

Software Installation Programs and Headaches. Most installation programs expect the code to be installed on the machine from which it is run and don't embrace the concept of remote access. Good installation programs allow you to change the default installation location and log every step they take, and the friendliest programs place all files under a user-specified location. Bad installation programs silently tread in certain system areas by adding user accounts, modifying boot scripts, and changing the system configuration. Bypassing installation programs, tracking down all the changes they made, and rewriting processes to duplicate the installation on client machines is the most time-consuming part of installing new software.

Read-only Environment. Ideally, the application server's disks should be mounted read-only. Unfortunately, some applications require write access to shared files, so we use read-write NFS mounts. However, application directory permissions are kept tight so that users are not able to write to the server.

File Ownership Conflicts. Occasionally, two different application programs will both try to "own" the same file. I first encountered this with two Lotus® applications I was installing: AmiPro/UX and Lotus Notes/UX. Both versions came with an /opt/lotus/bin/lpconfig file and the files were not identical. Normally, the version a client receives depends on the order in which the two software packages are installed. If AmiPro is installed last, a client will run AmiPro's version of the lpconfig file. If Lotus Notes is installed last, the client will run the Notes version.

Inconsistency and Supportability. To ensure that all users, no matter which software package was installed, used the same application versions, some rearranging was necessary. I installed the Lotus applications mentioned above on the application server into their own separate directories. Next, I copied all duplicate files to a third directory, using the version of the files that came with AmiPro. Finally, I modified the configuration files for both AmiPro and Lotus Notes so that clients would link to these duplicate files in the common directory.

This fix allowed the complete AmiPro and Lotus Notes products to be installed and available. If a problem arises with any of the common files, it's easy to change the file in one place and make sure the fix reaches everyone.

Summary

Overall, the application installation process has been a great success. Users are running consistent versions of applications, sharing centrally managed licenses, and no longer choosing to be part of a cluster simply because of the software available on that cluster. Application providers are able to make modifications in one location accessed by everyone. The application server contains no user accounts, runs very few processes, and is a fairly well-behaved I/O-dedicated machine. Unlike a cluster, the server can suffer occasional downtime without bringing down client systems.

We still run clusters, and the application server works in that environment as well. But the clusters are smaller and they are being implemented for the right reasons—not because of application availability. Many users are choosing standalone workstations, where they are allowed greater flexibility to install experimental versions of code or to use as much swap and disk space as they like without impacting other users.

References

1. ***Hewlett-Packard Journal***, Vol. 39, no. 5, October 1988, pp. 6-50.
2. J. Lees, “Sharing Local Software on a Network,” ***The Journal for UNIX System Administrators***, Vol. 3, no. 4, July/August 1994, pp. 48-69.

HP-UX 9.* and 10.0 for HP 9000 Series 700 and 800 computers are X/Open Company UNIX 93 branded products.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open is a registered trademark and the X device is a trademark of X/Open Company Limited in the UK and other countries.

Lotus is a U. S. registered trademark of Lotus Development Corporation.

Go To Next Article

Go Table of Contents

Go To HP Journal Home Page