A Simple Caching File System for Application Serving

John D. Bell
Ford Motor Co.

A Simple Caching File System for Application Serving

John D. Bell – Ford Motor Co.

ABSTRACT

In large installations, it can become very difficult to install many hundreds of applications on every workstation. To make every package available requires that the users' command search path be very long, or that there be 'wrapper' scripts written for every application which are maintained in a single directory. This policy also requires that each workstation have enough local disk to contain all files for every application. One traditional solution is to make applications available to workstations by serving them via NFS. However, this method can cause severe network loads, and it makes every user dependent upon the uninterrupted operation of the network.

This paper describes a set of programs designed to eliminate these problems. They provide for the automatic configuration of the wrappers needed for each application (including setting environment variables, etc.), in an architecture-independent fashion. They also implement caching from NFS-mounted "lockers" to a local file system, based upon file access patterns. This gives the advantages of single point of installation and minimization of local disk requirements, while providing the user with reduced dependence upon the network and increased speed of access. These programs require no kernel modifications nor other special facilities, and have proven to be portable across many versions of UNIX.

Introduction – Rationale for our Solution

Maintaining the set of available end-user applications is one of the most important tasks of a systems administrator. Indeed, one could argue that the applications are **the** justification for the computers to be used at all, and hence for the systems administrators to have any work! The evolution of the methods used to maintain these applications bears examination.

Local Files

In installations which consist of just a few dozen or less workstations, with just a few applications, it is feasible to locally install everything on every machine. However, this can quickly get out of hand. Ten applications, each with an installed footprint of 100 megabytes, would require an extra gigabyte of disk on every workstation. With only 50 workstations, if each of 10 applications is updated twice a year at one hour per upgrade (locally) per system, this would consume 25 person-weeks, or almost half of one system administrator's time. Although tools like *rcp* and *rdist* can help, configuring and maintaining the system lists and command scripts for these is time-consuming and filled with opportunity for errors. Clearly, as the installation grow in size, heterogeneity of platform types, and richness of applications, this becomes unsupportable.

NFS Files

The next obvious method for distributing applications is NFS serving of the files. This has the distinct advantage of giving one point of installation and configuration. However, files delivered over a 10-Mbit per second Ethernet network are accessed almost one order of magnitude slower than on a local disk. Also, this solution mandates that the network be fully reliable and not over-crowded. In addition, simply distributing the files via NFS does nothing to simplify the length of the command search path; in fact, if NFS-served directories are in the $PATH, the user's shell will be slowed down tremendously (or possibly completely stalled) when it hashes its lookup of executable commands.

Cached Files

This lead to the solution developed at Ford. Like the NFS solution, there is a single point of installation. However, (after migration), files are accessed from a local disk, which makes it both faster and more robust in the face of transient network failures or overloads. On the down side, the system is somewhat more complex than a simple NFS server scheme. Also, since there is no callback mechanism, files are not immediately consistent everywhere. (In practice, since differences usually only occur when a new package is installed or an existing one is updates, the propagation delay is acceptable.)

How It Works

The system we use can be divided into two almost disjoint components. The first is an applications dispatcher. This takes the place of individually created startup scripts, and centralizes the configuration necessary to get the various applications to run. It also provides just one directory where all these entry points can be found. The second component is the set of programs which maintain the caching file system, and migrate file in and out based upon access times.

'runprog'

The applications dispatcher is called *runprog*. It is a relatively small, compiled (for speed) program, which takes its configuration from a table of structures, one for each entry point supported. *runprog*, when invoked as "runprog update", automatically creates symbolic links to itself named according to the various programs configured into it. Invoking *runprog* by one of these other names causes it to dispatch to that program.

Along with the name mapping, *runprog* supports configuration options for the various programs. It can invoke the target program by the absolute (mapped) pathname, for applications that derive auxiliary files relative to the directory where they are installed. It can restrict running any configured program to just certain users or hostnames, or any except certain users or hostnames. It can set any environment variables before the target program; this can be used to set items like the PATH, the installation base directory for a package, or the shared libraries' search path. *runprog* also runs set-UID root to permit it to do usage accounting, and to permit a target program to be invoked as a particular special UID (not the user's). For example, we configure "kermit" into *runprog* as set-UID to "nuucp", so that it may manipulate the lock files for the serial communication device files it shares with UUCP. Please refer to Appendix A for a configuration file example.

The Cache

The actual file caching system consists of four components: one or more read-only NFS-served file systems (which we called "lockers"), which contain the source files; the local hierarchy (the "cache"), which contains either copies of the files in the lockers, or symbolic links to them; a program which updates the hierarchy in the cache to reflect changes in the various locker hierarchies; and a program which migrates files into and out of the cache, based upon access times.

'.rplink' Symlinks

An example should make this clearer. Here is a typical mapped directory from our system. It would be revealed by invoking *runprog* with the arguments "listd *directory_name*"; see Figure 1.

Here, the various GNU tools, served from the locker "red.pto", are mapped into the cache. Note that the final component of the locker pathname is equal to the architecture of the client machine. Listing the locker directory gives:

```
% ls -bCF /ford/red.pto/loc/GNU
aix/       irix6_r8k/   solaris/
irix5/     osf1/        sun4/
```

On each client, the appropriate version is mapped to a uniform pathname in the cache. When the cache is empty, listing the directory gives the output show in Figure 2. The symbolic link ***.**rplink* (for "runprog link") points to the corresponding directory on the locker; the file ***.**rpkeep* indicates that this is a mapped directory, and hence must always exist in the cache. The contents of the locker's directory are made to appear here by indirect references through ***.**rplink*. It is the access through these links which triggers migration.

Keeping Caches Up-To-Date

Periodically (via *cron*), a program is run on each server against each locker to generate a list of changed files (either added, deleted, or contents or *stat()* information changed). These change logs are made available to each client in a sub-directory called "/ford/*lockername*/changes". At some later time (again via *cron*), each client runs a program which examines these lists for all lockers, and updates the

```
% runprog listd GNU
Name   Client Directory -> Server Directory
GNU    /ford/server/loc/GNU -> /ford/red.pto/loc/GNU/solaris
```

Figure 1: Symbolic links

```
% ls -AF1 /ford/server/loc/GNU
total 14
-rw------- 1 root devel  0 May 7 11:04 .rpkeep
lrwxrwxrwx 1 root devel 29 May 7 11:04 .rplink -> /ford/red.pto/loc/GNU/solaris/
lrwxrwxrwx 1 root devel 11 May 7 11:04 bin -> .rplink/bin/
lrwxrwxrwx 1 root devel 15 May 7 11:04 include -> .rplink/include/
lrwxrwxrwx 1 root devel 12 May 7 11:04 info -> .rplink/info/
lrwxrwxrwx 1 root devel 11 May 7 11:04 lib -> .rplink/lib/
lrwxrwxrwx 1 root devel 11 May 7 11:04 man -> .rplink/man/
```

Figure 2: Listing with empty cache

mapped directories in /ford/server appropriately. This program can merge an arbitrary number of days of change logs, so that a client that has been down or off the network will bring itself back into synchronization as soon as it updates the first time.

Migration of Files

In the second phase of the update process, each client, after reconciling its cache hierarchy with the union of the locker hierarchies, examines the access times of the various symbolic links in the cache. Since the *readlink()* call updates the access time of the symbolic link, any file or directory linked back through *.rplink* which has been accessed more recently than the link's creation date indicates that the corresponding entry should be moved into the cache. That is, the update software destroys the symbolic link and copies the directory or file that the link pointed to. Note that this is the same mechanism used by other systems which run without kernel modifications (*lfu* and *nightly* – see below). As an example, after the listing of "/ford/server/loc/GNU" above, running the *update-client* script produces the output shown in Figure 3. The *.rplink* has moved one level deeper in the directory hierarchy, as can be seen by listing the contents of the "bin" sub-directory; see Figure 4. And now, having accessed each of these symbolic links which point to *their* corresponding *.rplink*, another cache update will bring each of these files local; see Figure 5.

The cache-updating software examines how much space is available in the destination file system, sorts the files destined for local migration by access time, and moves in the most recently used ones that will fit (allowing the file system to fill up to 95% of capacity). However, this is not a one-way process; files that are currently in the cache, but have not been accessed in a "long time" (45 days or more), have the local copy removed, and a symbolic link through *.rplink* recreated in its place.

```
# /ford/lib/runprog/bin/update-client
+[dir] /ford/server/loc/GNU/bin <- /ford/red.pto/loc/GNU/solaris/bin
+[dir] /ford/server/loc/GNU/info <- /ford/red.pto/loc/GNU/solaris/info
+[dir] /ford/server/loc/GNU/include <- /ford/red.pto/loc/GNU/solaris/include
+[dir] /ford/server/loc/GNU/lib <- /ford/red.pto/loc/GNU/solaris/lib
+[lnk] /ford/server/loc/GNU/man <- /ford/red.pto/loc/GNU/solaris/man
```

Figure 3: Listing after running *update-client* script

```
% ls -AF1 /ford/server/loc/GNU/bin
total 270
lrwxrwxrwx 1 root devel 33 May 7 16:41 .rplink -> /ford/red.pto/loc/GNU/solaris/bin/
lrwxrwxrwx 1 root devel 17 May 7 16:41 addftinfo -> .rplink/addftinfo*
lrwxrwxrwx 1 root devel 16 May 7 16:41 afmtodit -> .rplink/afmtodit*
lrwxrwxrwx 1 root devel 11 May 7 16:41 b2m -> .rplink/b2m*
lrwxrwxrwx 1 root devel 15 May 7 16:41 bdftops -> .rplink/bdftops*
lrwxrwxrwx 1 root devel 13 May 7 16:41 bison -> .rplink/bison*
    125 lines deleted
lrwxrwxrwx 1 root devel 14 May 7 16:41 zforce -> .rplink/zforce*
lrwxrwxrwx 1 root devel 13 May 7 16:41 zgrep -> .rplink/zgrep*
lrwxrwxrwx 1 root devel 13 May 7 16:41 zmore -> .rplink/zmore*
lrwxrwxrwx 1 root devel 12 May 7 16:41 znew -> .rplink/znew*
```

Figure 4: Deepening directory levels

```
# /ford/lib/runprog/bin/update-client
+[reg] /ford/server/loc/GNU/bin/addftinfo <- /ford/red.pto/loc/GNU/solaris/bin/addftinfo
+[reg] /ford/server/loc/GNU/bin/afmtodit <- /ford/red.pto/loc/GNU/solaris/bin/afmtodit
+[reg] /ford/server/loc/GNU/bin/b2m <- /ford/red.pto/loc/GNU/solaris/bin/b2m
+[lnk] /ford/server/loc/GNU/bin/bdftops <- /ford/red.pto/loc/GNU/solaris/bin/bdftops
+[reg] /ford/server/loc/GNU/bin/bison <- /ford/red.pto/loc/GNU/solaris/bin/bison
    125 lines deleted)
+[reg] /ford/server/loc/GNU/bin/zforce <- /ford/red.pto/loc/GNU/solaris/bin/zforce
+[reg] /ford/server/loc/GNU/bin/zgrep <- /ford/red.pto/loc/GNU/solaris/bin/zgrep
+[reg] /ford/server/loc/GNU/bin/zmore <- /ford/red.pto/loc/GNU/solaris/bin/zmore
+[reg] /ford/server/loc/GNU/bin/znew <- /ford/red.pto/loc/GNU/solaris/bin/znew
```

Figure 5: Files local after cache update

Similar Systems

There are several other software systems which locally cache files from remote file systems. Commercially available packages includes *cachefs* from Sun Microsystems [1], and DCE DFS from Transarc (and other vendors) [2]. Other packages developed by the Usenet community include *lfu* by Paul Anderson [3,4], and *nightly* by Hal Pomeranz [5]. Each of these will be considered in turn.

cachefs

This is available on Sun's Solaris 2.*x* and Silicon Graphics' Irix operating systems, and is distributed ''free'' with them. It is (superficially) the simplest of the packages, and has the additional advantage of keeping files on the client continuously up-to-date with respect to the server (by a callback scheme). However, it is not available on all the architectures required at Ford, and it cannot map the contents of more than one NFS mount point into the cache. It is also not clear how may clients can be supported by one server, nor how big the cached area can be.

DFS

This product is commercially supported on a variety of architectures (however, not all of the ones found at Ford), but is an additional cost item. It can be configured to architecturally map directories (so that heterogeneous clients can refer to analogous files by the same pathname), but still cannot map multiple lockers into the cache. It is the most complex of all the schemes considered, requiring setting up a complete DCE security service (based on Kerberos) as well as multiple daemon processes on each machine. On the plus side, it was designed to support very large numbers of servers and clients, as well as a very large cached file system.

lfu

lfu has several advantages over the two packages considered above. It is available for free, and has been ported to several variations of UNIX on different hardware platforms. It appears to be able to be configured to map an architecturally-dependent locker pathname into the cache, as well as being able to serve files from more than one locker into the cache. It is considerably more complex than *cachefs*, but quite a bit simpler than DCE. (In fact, much of the design of *runprog* was directly inspired by *lfu*, although none of the code was taken from it.)

Nightly

nightly shares many of the advantages of *lfu*; it is also available for free from ''the Net'', and is extremely portable (being written in Perl). It is much less complex than DFS or *lfu*. However, it does not map the client's architecture, nor can it map from multiple lockers into one cache. It is also a ''smaller'' solution than any of the others – in the described implementation, there are only two servers and 40 clients, the locker is only about 450 Mbytes, and the cache is only about 50 Mbytes on each client.

Description of Our Environment

The software described here has been in use at Ford Motor Co.'s Manufacturing Systems Office's Powertrain Operations for about four years. We support nearly 800 workstations in seven buildings located in three different cities in southeastern Michigan.

These workstations are running eight different versions of UNIX: Sun Microsystem's SunOS 4.1.3, Solaris 2.3, 2.4, and 2.5 on SPARC processors; Silicon Graphics' IRIX 5.3 on R4000 processors, and IRIX 6.1 on R8000 processors; and IBM's AIX 3.6 and 4.1 on RS/6000 processors. We have previously supported this system on SunOS 3, IRIX 4, and Digital Equipment Corp.'s Ultrix, and have mostly ported the system to DEC's OSF/1.

Today *runprog* supports 290 entry points, which map approximately 32 gigabytes of applications code. (Not all architectures have every application loaded.) Individual client machines have caches ranging from 400 Mbytes to 1 Gbyte.

Performance Measurements

runprog compromises the access speed of an application (from NFS disks instead of local disks) with the amount of local disk storage necessary. Although the access speed of an NFS-served disk varies tremendously with the performance of the underlying network, a few statistics may be of interest.

Four large applications were tested on a SPARCstation 10 connected over 10-Mbit Ethernet to the NFS servers. These are CAD, FEA, and other modeling applications typical of those used by design engineers in our community. In each case, the application was started from an ''empty'' cache (that is, one that

Application	''empty cache'' time	final time	amount size	in cache
SDRC	1:05	:33.9	1588900	101670
CV	1:40	1:10	575001	134895
Mech	1:00	:43.2	127291	27367
HM	:38	:18.4	44651	15949

Table 1: Test results

contained nothing but the minimum mapped directory structure locally, with symbolic links back to the NFS locker for all sub-directories and files), and timed from the (command line) invocation until the first window was complete and not showing any "busy" indicator. This performance is typical of what would be observed when running the application from a pure NFS mount point. Then the cache was updated, and the application started again. This cycle was repeated until no further files migrated locally with subsequent cache updates. Note that these applications were not really "used", but just started – for a real user, more files would migrate locally with greater performance speedup. All sizes indicated here are in kbytes; all times are in minutes and seconds.

Summary

This paper has described a relatively simple solution to the problems associated with distributing and maintaining a large base of applications programs in a very large, heterogeneous workstation environment. The solution has many of the advantages of simply NFS-serving the files, yet optimizes its use of network resources.

Future directions for enhancement include the ability to distribute the administration further (via 'subscription list' of various packages without regard to the actual locker containing them). Also, the mechanism used to configure a particular package into *runprog* must be simplified.

Please contact the author by email at the above address for availability.

The author wishes to acknowledge the essential work of Ken Fox of Ford Motor Co. in the initial design and implementation of *runprog*, and the work of Gary Ross of Ford and Clinton Pierce of Decision Consultants Inc. in the maintenance and enhancement of the software and documentation. He thanks each of them for all their invaluable help.

Author Information

John D. Bell is a senior consulting software engineer with ASG Renaissance, where he has been on several assignments over the last nine years at a major automotive manufacturer in the Detroit, Michigan area. He studied at The American University, Case Western Reserve University, and Ohio State University, and subsequently has been programming and administrating UNIX systems for the last 13 years. He may be reached by U.S. mail at ASG Renaissance, 3000 Town Center, Suite 2237, Southfield, MI 48075, or by electronic mail at "jbell4@ford.com".

References

[1] *cachefs* by Sun Microsystems, Inc. A white paper is available at http://www.sun.com/sunworldonline/swol-08-1996/swol-08-sysadmin.html .

[2] DCE DFS by Transarc. A white paper is available at http://www.transarc.com/afs/transarc.com/public/www/Public/ProdServ/Product/Whitepapers/OSF_Whitepapers/dfs.ps .

[3] Anderson, Paul. "Effective Use of Local Workstation Disks in an NFS Network", *USENIX LISA VI Conference Proceedings*, 1992.

[4] Anderson, Paul. "Managing Program Binaries in a Heterogenous UNIX Network", *USENIX LISA V Conference Proceedings*, 1992.

[5] Pomeranz, Hal. "A New Network for the Cost of One SCSI Cable: A Simple Caching Strategy for Third-Party Applications", *USENIX SANS III Conference Proceedings*, 1994.

Appendix A: A Sample Configuration File

Here is a sample of what a configuration file for “runprog” could look like:

```
!
! this file is a sample of the things
! necessary and possible in a "runprog" configuration
!
! 12 Aug 1996   JDBell  (jbell4@ford.com)
!
! lines begining with "!" are comments
! whitespace is ignored
! first, a variable assignment  (this is the usage log file)
ACCTHOST = /ford/lib/runprog/client-log
! a couple of program (class) definition
defprogram Generic {
        flags = none
        maxUsers = unlimited
        maxNetworkUsers = unlimited
}
! the "System" class inherits from the "Generic" class,
! with certain modifications
defprogram System: Generic {
        class = System
        flags = status
        acctHost = %ACCTHOST
}
! now, a directory configuration (which happens to be
! architecturally mapped)
GNUDIR = /ford/server/loc/GNU
directory GNU {
        link = %GNUDIR
        path = /ford/red.pto/loc/GNU/%arch
}
! next, several programs which are found in that
! directory
program gmake: System {
        path = %GNUDIR/bin/make
}
program gcc: System {
        environment = ( CC=gcc LIBSDIR=%GNUDIR/lib )
        path = %GNUDIR/bin/gcc
}
program gdb: System {
        path = %GNUDIR/bin/gdb
}
! finally, a single program which is architecturally mapped,
! showing off some of the configuration flags
EMULATORSDIR = /ford/server/loc/emulators
directory emulators_dir {
        link = %EMULATORSDIR
        path = /ford/black.pto/loc/emulators
}
```

```
program kermit: Generic {
        flags = SET_UID
        uid = uucp
        maxUsers = 2
        path = %EMULATORSDIR/kermit/kermit.%arch
}
!
! now, the "boilerplate" which makes the include file come out...
!
<< '
/* File: sample-configuration.h
** Template Date: June 2, 1992; August 30, 1994
** File Creation Date: %date
**
** created by %user on %host
**
*/

count := 0
witheach program
{
    if userList then
    {
        << 'char *__string_list_%(count)[] = { %userList, NULL };\n'
        count := count + 1
    }
    if machineList then
    {
        << 'char *__string_list_%(count)[] = { %machineList, NULL };\n'
        count := count + 1
    }
    if environment then
    {
        << 'char *__string_list_%(count)[] = { %environment, NULL };\n'
        count := count + 1
    }
}
if count > 0 then
{
    << '\n\n'
    count := 0
}
<< 'ProgramType programList[] =
{
total := 0
index := 0
witheach program
{
    total := total + 1
    << '  { '
    << '"%(name)", '
    << '"%(path)", '
    << '%#(name), '
    << '%(flags), '
```

```
    << '%(uid), '
    << '"%(class)", '
    << '%(maxUsers), '
    << '%(maxNetworkUsers), '
    if userList then
    {
        << '__string_list_%(count), '
        count := count + 1
    }
    else
    {
        << 'NULL, '
    }
    if machineList then
    {
        << '__string_list_%(count), '
        count := count + 1
    }
    else
    {
        << 'NULL, '
    }
    if environment then
    {
        << '__string_list_%(count), '
        count := count + 1
    }
    else
    {
        << 'NULL, '
    }
    << '%"(lockHost), '
    << '%"(acctHost), '
    << '%(index)'
    << '},\n'

        index := index + 1
}

<< '\n  ENDOFPROGRAMS
};

int totalProgramCount = %(total);

total := 0

<< '

DirectoryType directoryList[] =
{
witheach directory
{
    total := total + 1

    << '  { '
    << '"%(name)", '
    << '"%(link)", '
    << '"%(path)", '
    << '%!(link), '
    << '%!(path), '
    << '%(flags), '
    << '%"(prepare), '
    << '%"(install), '
```

```
    << '%"(cleanup) '
    << '},\n'
}
<< '\n  ENDOFDIRECTORIES
};
int totalDirectoryCount = %(total);
```

And here is the generated include file (which is compiled with the "runprog" sources):

```
/* File: sample-configuration.h
** Template Date: June 2, 1992; August 30, 1994
** File Creation Date: 6:21 pm Tuesday, August 13, 1996
**
** created by jbell4 on jdb500
**
*/
char *__string_list_0[] = { "CC=gcc","LIBSDIR=/ford/server/loc/GNU/lib", NULL };
ProgramType programList[] =
{
  { "gcc", "/ford/server/loc/GNU/bin/gcc", 0x67636300, STATUS, 0, "System", -1,
     -1, NULL, NULL, __string_list_0, NULL, "/ford/lib/runprog/client-log", 0},
  { "gdb", "/ford/server/loc/GNU/bin/gdb", 0x67646200, STATUS, 0, "System", -1,
      -1, NULL, NULL, NULL, NULL, "/ford/lib/runprog/client-log", 1},
  { "gmake", "/ford/server/loc/GNU/bin/make", 0x676d616b, STATUS, 0, "System",
      -1, -1, NULL, NULL, NULL, NULL, "/ford/lib/runprog/client-log", 2},
  { "kermit", "/ford/server/loc/emulators/kermit/kermit.solaris", 0x6b65726d,
       SET_UID, 5, "", 2, -1, NULL, NULL, NULL, NULL, NULL, 3},
  ENDOFPROGRAMS
};
int totalProgramCount = 4;
DirectoryType directoryList[] =
{
  { "GNU", "/ford/server/loc/GNU", "/ford/red.pto/loc/GNU/solaris", 20, 29,
     0x0, NULL, NULL, NULL },
  { "emulators_dir", "/ford/server/loc/emulators",
    "/ford/black.pto/loc/emulators", 26, 29, 0x0, NULL, NULL, NULL },
  ENDOFDIRECTORIES
};
int totalDirectoryCount = 2;
```