A Caching File System For a Programmer's Workstation

by

Michael D. Schroeder, David K. Gifford and Roger M. Needham

SRC Research Report 6

The work reported here was done by the authors at the Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304. The writing and publication of this paper was supported by the DEC Systems Research Center, where Michael D. Schroeder now works. David K. Gifford is at the Laboratory for Computer Science, 545 Technology Sq., Cambridge, MA 02139. Roger M. Needham is at the Computer Laboratory, Corn Exchange St., Cambridge CB2 3QG, UK.

This paper has been accepted at the 10th ACM SIGOPS Symposium on Operating Systems Principles, whose proceedings will appear as the December 1985 issue of *Operating Systems Review*.

Authors' Abstract

This paper describes a workstation file system that supports a group of cooperating programmers by allowing them both to manage local naming environments and to share consistent versions of collections of software. The file system has access to the workstation's local disk and to remote file servers, and provides a hierarchical name space that includes the files on both. Local names can refer to local files or be attached to remote files. Remote files, which also may be referred to directly, are immutable and cached on the local disk. The file system is part of the Cedar experimental programming environment at Xerox PARC and has been in use since late 1983.

Capsule Review

A personal workstation with a local disk generally uses the disk to hold a local file system. In a large collection of workstations connected by a network, the workstation file systems contribute to the good responsiveness and high availability of the individual workstations. But if users of different workstations are working together on a project, the separate file systems get in the way. Typically, the users will cope with the problem by manually copying files through a shared directory on a file server, but this process is slow and error-prone.

The Cedar programming environment project used a simple technique based on *DF files* to manage file sharing in a systematic way. This paper describes CFS, a workstation file system designed to support the use of DF files. CFS is unusual in that it only allows the sharing of immutable files; rather than changing an existing shared file, you make a new version. Workstations with CFS share files using a file server that provides file version numbers and whole-file transfers. CFS does not require other file server features like random file access or transactions.

CFS manages part of the local disk as a cache of immutable shared files. The DF file methodology allows CFS and the file servers to largely ignore the cache consistency problem.

Experience with Cedar shows that DF files plus CFS form a simple and workable solution to the problem of multi-person file sharing in a workstation based programming environment. For some other applications of workstations, it seems clear that a database server will be a useful addition, giving data sharing at a finer grain than the immutable files of CFS. An interesting open question is whether or not there are applications that demand something in between a database server and CFS.

— Mark Brown

Introduction

A configuration of personal workstations, each with a local disk, connected to shared file servers by a local area network can provide a responsive base for software development by a team of programmers. The workstations provide each programmer with dedicated hardware resources that respond quickly to interactive demands. The file servers provide a way for the group of programmers to share information. This paper describes a distributed file system, called *CFS*, designed to support group programming in this hardware context. CFS was developed as part of the Cedar experimental programming environment [8, 18, 19] at the Xerox Palo Alto Research Center.

A file system that supports a group of cooperating programmers has two important jobs to do. First, it must help each programmer manage a private file naming environment in which to work. Second, it must help the group share consistent versions of the software subsystems being developed in parallel. CFS addresses these requirements by providing each workstation with a hierarchical name space that includes the files on the local disk and on all file servers. The local files are private to the workstation. The remote files are sharable among all workstations. CFS supports a simple copying model of file sharing. A client of CFS creates a file on the local disk. To make that file available for sharing, the client transfers it to a file server, giving it a remote name. A client on another workstation can then access the file by its remote name and transfer it to that workstation's local disk. The basis for consistency in sharing is atomic creation of each remote file.

A distinctive feature of CFS is that only immutable files may be shared. An immutable file has two important properties: its name may not be reused and its contents may not be altered. Thus, the name of an immutable file signifies the fixed contents of the file, not the file as a container for variable information. All remote files in CFS are immutable and only remote files are shared. As we will see, sharing only immutable files makes it easy to support consistent sharing and makes it easy to implement a distributed file system.

Two other key features of CFS are the ability to attach local names to remote files and the caching of remote files on the local disk. These two features work together to decouple the management of the local naming environment on a workstation from the management of space on the local disk.

CFS was designed to be used by software management tools like Cedar's *DF package* [16]. The tools in the DF package provide a way to define and share a static snapshot of a software subsystem. The definition is a list of component file names recorded in a *DF file*. The components may be source files, object files, documentation files, and other DF files. A DF file is the value of a subsystem, not a reference to it. A programmer using a particular DF file to identify the components of a subsystem can be certain to find a set of file contents that represents a consistent version of the subsystem. The immutable files provided by CFS directly support this snapshot view of subsystems. A particular version of a subsystem is shared via an immutable version of a DF file that names immutable versions of the component files.

The tools in the DF package work by establishing a correspondence between remote files named in DF files and local file names on a workstation. The programmer then works in this local naming environment. The facilities in CFS for attaching local names to remote files allow setting up the local naming environment without actually copying the corresponding files from the file servers. The presence of the actual files on the local disk is managed independently by the local cache for remote files.

Simplicity and good performance were primary goals in the CFS design. Forcing all sharing to be through file servers eliminates workstation code that responds to file requests from other workstations and from servers. Sharing only immutable files means that the workstation cache machinery can ignore the possibility of remote files changing. Using simple atomic updates to server directories to support consistent sharing eliminates the need for transactions and long-term locks on the file servers. In addition, to reduce the load on the shared file servers and reduce the complexity of the workstation cache machinery, CFS transfers and caches whole files rather than individual file blocks.

This paper documents the CFS design. After listing the facilities required in the file servers used by CFS, the paper presents the key features of the design and shows by example how these features are used by the DF package to support group programming. Then some detailed points about naming, binding and caching are considered, and the implementation structure is sketched. The final section bounds the design by discussing potential goals not addressed and directions for future exploration. The paper concludes that, when used with software management tools like the DF package, CFS effectively supports the development of large programs by groups of programmers. An appendix defines the semantics of the key operations in the CFS interface.

Related Work

Much work has been done on distributed file systems and many of the recent efforts are surveyed in Svobodova's article [17]. Most designs start by distributing a traditional time-sharing file system over multiple computers attached to a network. The clients on all computers see the same set of shared, mutable files. This traditional model of file system semantics is easy for clients to understand, but an efficient distributed implementation is quite complex. The simplest implementations, such as the Newcastle Connection [5], provide direct access to the blocks of files from a named collection of file system instances. Performance is improved in the Apollo Domain file system [11] and Sun Microsystems' NFS [22] by adding local caching of file blocks. The ITC distributed file system [15] adds location transparency for files and replication of read-only files. It has adopted the transferring and caching of whole files used in CFS, but still maintains the traditional client model of shared, mutable files. The performance implications of this combination are not understood yet. In all these cases, the file system provides no assistance in organizing the consistent sharing of sets of files. The LOCUS file system [21] addresses consistent sharing with sophisticated locking and transaction mechanisms for shared, mutable files. It also provides for replication of such files. This combination of functions produces interface semantics and an implementation that are quite complex.

CFS differs from these systems by changing the semantics of the traditional file systems interface, as described earlier, to reflect the intended use. These semantics are carefully selected to provide the functionality required to support group programming efforts while enabling a simple, efficient distributed implementation.

File Servers

CFS integrates a private, local file system for a workstation with the shared, remote file systems on network file servers. The client interface to CFS is in the workstation. All shared mechanism is in the file servers. The network interface of the file servers is considered to be internal to the CFS implementation, to be used only by the CFS code in a workstation, but this restriction is not enforced. CFS uses the IFS file servers that are common in the Xerox research and development community. Before describing the key features available at the CFS client interface, we outline the services provided by these file servers.

Each file server provides a shared hierarchical directory. Access control mechanisms define which authenticated users are able to access and manipulate each file. Using a file transfer protocol [3], new files can be stored and existing files can be read, renamed and deleted. These operations are on whole files. File names include version numbers and when a new version of a file is stored the file server automatically generates a new version number for its name. The file servers also allow directories to be enumerated and information about existing files to be retrieved.

Updates to file server directories are indivisible and serialized. Thus, transferring a new file to the server, assigning it a new version number, and entering its name in the file server directory appear to be a single act. If any step fails then no trace of the attempt remains visible. This atomicity is implemented with simple mutual exclusion in each file server.

CFS does not require file servers to provide locks that can be held between file operations. No transaction facilities covering multiple operations are needed. CFS does not need to read or write file server directories as files; it can use remote directory operations.

Key Features of CFS

We now describe in more detail the features of CFS, as viewed from the client interface in a workstation, that support consistent sharing of collections of software and management of the local name space. The appendix contains detailed descriptions of the CFS operations that embody these features.

CFS provides a uniform hierarchical naming structure for local and remote files. A complete file name consists of a server, a root directory, zero or more subdirectories, a simple name, and a version. The server part names the file server that stores the file. For example, */ivy/Cedar/CFS/CFSNames.mesa!5* might be the name of version 5 of a program source file as stored in the file server *ivy*. An empty server part means a file on the local workstation. For example, *//Cedar/CFSNames.mesa!1* might be the name of a copy of the same file on the workstation.

CFS generates the version part for all new file names. The new version is the successor of the highest existing version, or 1 if no version exists. The version part of a file name argument to a CFS operation on an existing file may be a variable or be omitted. The variables allowed are !L, meaning the lowest existing version, and !H, meaning the highest existing version. When omitted the version part defaults to !L or !H, depending on the operation being invoked, *e.g.*, !L for *Delete* and !H for *Open*. This sort of version naming first appeared in Tenex [2].

CFS encourages the view that all files are immutable. It enforces the immutability of remote files — they may not be altered once created, except to be deleted. Existing local files may be modified, but this feature is used only for special purposes such as updating local log files. Tools such as the editor and compiler treat local files as immutable too, by always creating new file versions when writing results to disk. The Swallow file system design [13] first explored the benefits of immutable versions.

A local working directory provides the naming environment in which a programmer works. CFS prepends the current local working directory name to any file name argument that does not start with the character

"/". There is no search rule mechanism, however, as the use of search rules is in conflict with the philosophy of precise specification of subsystem components embodied in the software management tools.

In CFS, all access to file servers is in units of whole files. Thus, new remote files may be created only by copying from existing files. Local files, however, are held open by clients while being read and written in smaller units. Readers/writer locking is provided within a workstation to synchronize such local access by multiple processes in a workstation. Clients can read remote files in smaller units too, but only the cached copy of the remote file is held open, the cache having been filled by a whole file transfer from the server.

CFS uses a form of symbolic links between file names, an idea introduced in CTSS [6] and developed in Multics [1], to make it inexpensive to give a local name to a remote file. CFS forms an attachment between a local name and a remote file by storing the remote file's name in the local directory entry. Forming an attachment is viewed as lazy copying and is done with a mode of the *Copy* operation. Access to the remote file is delayed until the file contents associated with the local name are needed by the client. As with symbolic links in other file systems, the target file of a CFS attachment may turn out to be inaccessible when needed. Unlike other file systems, however, the target of a CFS attachment is immutable and cannot change. Attachments are useful because they separate the management of the local name space from the transfer and storage of files. With attachments it is practical to always set up a complete local naming environment for a programming task, even when only a few of the files named will eventually get used.

CFS uses the portion of the local disk not occupied by local files as the cache for remote files. All requests to open remote files for reading are satisfied from the cache. Except for performance effects, the client cannot tell whether the requested file was already in the cache or had to be transferred from the remote server. The cache is managed automatically using an approximate LRU strategy.

Use of CFS

We now describe how these features of CFS are used with tools from the DF package to manage a local naming environment and to share consistent versions of multi-component subsystems among programmers. A tool called *BringOver* is used to incorporate a subsystem version defined by a DF file into a local naming environment. A tool called *SModel* is used to generate and share the DF file that describes a new subsystem version.

In a DF file the identities of source files, object files and other DF files that are part of a subsystem are specified by remote file names with version numbers. The BringOver tool uses CFS to copy each listed component file from the file server to the current local working directory, if the component is not already present. The local name that is the target of each copy operation is the simple name part of the remote name listed in the DF file. (Collapsing to simple names in this way can generate name conflicts, which in Cedar are avoided by careful name choice!) When BringOver is finished, each subsystem component from the DF file appears in the current working directory as the highest version of the simple name.

Attachments allow significant optimizations of BringOver. Before CFS, BringOver actually had to transfer the contents of missing files to the workstation disk — a fairly expensive proposition. Retrieving the entire Cedar system with BringOver took more than an hour and frequently would exhaust available local disk space before completing. Using the attachment mode of copying in CFS, BringOver simply associates local names with remote file names. No files other than the DF files that need to be read by BringOver are transferred. Thus BringOver is fast and does not fill up the local disk.

Figure 1a shows an example working directory in which the simple name *x.df!1* is attached to a previously created remote DF file. When the user issues the command "BringOver x.df", BringOver opens *x.df* and reads the contents of the attached, remote DF file.

Figure 1b shows that BringOver has created the attachments *a.mesa!1* and *b.mesa!1* for the components listed in the DF file. Creation of these attachments has no effect on the presence or absence of remote files in the cache. At this point only */ivy/Cedar/5.2/x.df!40* is certain to be in the cache (since BringOver had to read its contents).

After using BringOver, the programmer makes changes to subsystem components. He usually presents single-component file names without version parts as arguments to the editor, compiler and other tools. The compiler and binder refer to object files using such names. The current working directory is the naming environment in which these single-component names are bound to the collection of source and object files that define a particular subsystem. Figure 1c supposes that the programmer has modified *b.mesa*, say using the editor. The editor stored the modified source file in a new local version, *b.mesa!2*. Note that this new local file has not yet been transferred to the file server.

After a new consistent version of the subsystem under development has been created, the *SModel* tool is used to move the changed components back to their remote home on a file server. Each changed file is transferred back to the remote server and the existing local name is attached to the new remote file. In addition, a new version of the DF file is created to list

```
 Local Working              Remote File
   Directory                  System

 ---------------    ---------------------------
| x.df!1 -------|--|-> /ivy/Cedar/5.2/x.df!40  |
|  (Attachment) |  |         Contents: X       |
|_______________|  |___________________________|
```

Figure 1a: An Attachment to a DF File

```
 Local Working              Remote File
   Directory                  System

 ---------------    ---------------------------
| x.df!1 -------|--|-> /ivy/Cedar/5.2/x.df!40  |
|  (Attachment) |  |         Contents: X       |
|               |  |                           |
| a.mesa!1 -----|--|-> /ivy/Cedar/5.2/a.mesa!36|
|  (Attachment) |  |         Contents: A       |
|               |  |                           |
| b.mesa!1 -----|--|-> /ivy/Cedar/5.2/b.mesa!28|
|  (Attachment) |  |         Contents: B       |
|_______________|  |___________________________|
```

Figure 1b: Attachments Created by BringOver

```
 Local Working              Remote File
   Directory                  System

 ---------------    ---------------------------
| x.df!1 -------|--|-> /ivy/Cedar/5.2/x.df!40  |
|  (Attachment) |  |         Contents: X       |
|               |  |                           |
| a.mesa!1 -----|--|-> /ivy/Cedar/5.2/a.mesa!36|
|  (Attachment) |  |         Contents: A       |
|               |  |                           |
| b.mesa!1 -----|--|-> /ivy/Cedar/5.2/b.mesa!28|
|  (Attachment) |  |         Contents: B       |
|               |  |                           |
| b.mesa!2      |  |                           |
|  Contents B'  |  |                           |
|_______________|  |___________________________|
```

Figure 1c: New Version of a Source File

```
  Local Working              Remote File
    Directory                   System

 ---------------     ---------------------------
| x.df!1 -------|--|-> /ivy/Cedar/5.2/x.df!40   |
|  (Attachment) |  |         Contents: X        |
|               |  |                            |
| a.mesa!1 -----|--|-> /ivy/Cedar/5.2/a.mesa!36|
|  (Attachment) |  |         Contents: A        |
|               |  |                            |
| b.mesa!1 -----|--|-> /ivy/Cedar/5.2/b.mesa!28|
|  (Attachment) |  |         Contents: B        |
|               |  |                            |
| b.mesa!2 -----|--|-> /ivy/Cedar/5.2/b.mesa!29|
|  (Attachment) |  |         Contents: B'       |
|               |  |                            |
| x.df!2 -------|--|-> /ivy/Cedar/5.2/x.df!41   |
|  (Attachment) |  |         Contents: X'       |
|_______________|  |____________________________|
```

Figure 1d: Attachments Created by SModel

the components of the new subsystem version and then is copied to the remote server. Figure 1d shows the state of the file system after SModel has been run. SModel created *x.df!2* as a new local file, then copied it to the file server and attached the local name *x.df!2* to the new remote file.

SModel maintains the consistency of multi-component subsystems as viewed by clients. The last action of SModel is to copy the updated DF file to the server. Since file creations are atomic on the file servers, and since all subsystem clients retrieve the components via the DF file, a client doing a BringOver while the SModel is in progress will get either the old subsystem or the new one, but not a mixture. Knowledge that a new version of a subsystem is available can be communicated implicitly via higher-level DF files or outside the system via word-of-mouth, a computer mail system, etc. Programmers who wish may continue to use the old version of the subsystem, via the old version of the DF file, until it is deleted from the file server.

The example in this section shows the overall pattern of how CFS works with the system modelling tools to support group programming. Not all use of CFS to access remote files, however, is via DF files. For example, document display programs accept remote names and use CFS to retrieve and cache the files to be displayed. Users often use this facility to poke around the remote file servers directly, without the intervention of DF files.

More About Naming

It is acceptable for multiple names to be bound to the same immutable file contents and for some or all of these name bindings to be broken later. Thus, copying and deletion are reasonable operations on immutable files. Strictly speaking, however, names for immutable files should never be reused. The version naming mechanism in CFS does not eliminate the possibility of name reuse. If all the versions of a file are deleted then the record of the highest version that has existed is lost and version numbering for that name will start over at 1. If the highest existing version is deleted then that version number will be reused. With version naming it is hard to eliminate these flaws. Permanent memory of the highest version issued for each name would be required. In practice, using version numbers to approximate non-reused names for immutable files has proved adequate. People do not delete the highest version of a remote file unless the name is to become dormant.

As a safeguard against reused version numbers causing confusion, CFS allows a file's creation time to be included with file name arguments to CFS operations. The creation time, defined as the local clock reading when the contents of a file were first generated, is a file property that CFS propagates when a file is copied or renamed. If a creation time is specified with a file name argument then CFS searches for the file version with that creation time. Any version part in the name argument is treated as a hint about which version to check first. The creation time of a remote file may be recorded in an attachment.

DF files frequently specify the creation times along with the complete names for component remote files. BringOver includes these creation times in the attachments it makes. This extra information provides assurance that incorrect component versions will not be found, even if version numbers in the DF files are incorrect or if version numbers on the file servers have gotten scrambled. Object files produced by the compiler contain the simple names and creation times of other object files read during compilation. The debugger presents these names with creation times to CFS when opening object files in the local working directory to read symbol tables.

More About Versions

The version variables allowed in file name arguments are used mainly when referring to local files. Most remote files are referred to through DF files by specific version. During periods of system development, however, the DF file for one subsystem may refer to the !H version of the DF file for another subsystem. The !H reference provides automatic access to the most recent version of the latter. As part of the system release process, the !H reference is replaced by a specific version number and creation time.

For an operation on a remote file, correctly binding a version variable in a file name argument to a particular version requires checking with the server. If the server is inaccessible then the binding cannot be performed and the operation will fail, even if versions of the file happen to be in the workstation cache. To allow the operation to succeed in this case, CFS lets the client specify that remote checking should not be used to bind a version variable. Without remote checking CFS binds the version variable relative to the (possibly incomplete) set of versions in the cache; only if no cached version is present is the remote server interrogated. Clients turn off remote checking when the consequences of retrieving an out-of-date version are small and the consequences of retrieving nothing are unacceptable. For example, when starting up Cedar the display font file is opened for reading using a !H version variable. If opening the font file with remote checking fails then an attempt is made to open it with no remote checking, because without a display font Cedar cannot tell the user what happened.

Two potential problems with always creating a new version are increased use of disk space and increased disk allocation activity. For local files in CFS these problems are mitigated by automatically limiting the number of versions that are kept. Each local name has a property called its *keep*, a numeric value that specifies the number of versions of the local name to keep around. Automatically processed keeps first appeared in the Alto operating system [10], although the feature got little use. In CFS, whenever a local name is created its keep is inherited from the highest existing version or set from an argument to the operation doing the creation.

Keep processing occurs when creating a new version of a local name. In this case CFS will enumerate existing versions in decreasing order. After *keep-1* versions are encountered in this enumeration, additional versions will be deleted if not open. The disk file of a deleted version will be reused for the new version being created. For example, if the only existing version of a file is named *Example.bcd!4*, if it has a keep of 1, and if no client has it open, then creating *Example.bcd* will cause *Example.bcd!4* to be deleted and its disk file to be reused for the new file *Example.bcd!5*. Keeps typically are set to two for source files and one for derived files. Because most files on a particular workstation are only read, however, the average number of versions per file on a workstation is close to one.

CFS provides no automatic mechanisms for deleting unneeded versions of remote files. Client tools exist that will delete all files from a remote directory that are not named in a specified set of DF files.

Caching Immutable Files

Caching immutable files is easy. Because remote files are immutable, changes that occur on file servers need not be reflected into workstation caches. Clearly, the properties and contents of existing remote files cannot change and creation of new remote files need not be reported. The case of deletion, however, may be less clear.

With immutable files, deletion does not change the abstract state of the file system. Deletion does not cause the file to cease to exist, it just frees some space on a file server. Leaving a deleted remote file in a workstation cache is like keeping an out-of-print book on your bookshelf. To avoid confusion, however, a remote file should be deleted only when it is no longer being used. Then the deleted version will fall out of the workstation caches quietly from lack of use. While one can construct scenarios where continued use of a cached, deleted version could cause confusion, in practice these cases do not occur — programmers need not use file deletion as a mechanism for passing messages! To help users retain their sanity, CFS does remove a deleted remote file from the cache on the workstation that caused the deletion.

Implementation and Performance

With the exception of a performance optimization to the existing file servers, CFS was implemented entirely by workstation code. Figure 2 illustrates that this code depends upon an implementation of the file transfer protocol to access remote file servers. It also depends on a lower level file system in the workstation, called *DiskFile*, that allocates sectors on the local disk into disk files named by unique identifiers. CFS uses these disk files to implement both local files and cached, remote files. A disk file includes a property page in which CFS records the complete name, length, creation time and other properties of the corresponding CFS file.

The performance optimization to the file servers is a request/response protocol for getting information about a file. The request packet from the workstation contains a complete file name with either a version number or a version variable. The response packet from the file server will either indicate that no matching file was found, or give information about the file that matches. The information includes the correctly capitalized file name (with version number), the creation time, and the byte length. This single packet protocol is used to reduce the overhead of finding out versions and creation times from a file server. In particular, when opening

```
                clients of CFS
                      |
 ------------------------------------
|CFS                                 |
|     local directory hierarchy      |
|                &                   |
|       cache of remote files        |
|____________________________________|
        |                   |
 ---------------    -----------------
|FTP            |  |DiskFile         |
|  access to    |  |  local disk     |
|  file servers |  |  files w/ uid's |
|_______________|  |_________________|
```

Figure 2: Structure of CFS Implementation

a file specified by version variable and no creation time, CFS uses this protocol to bind the version variable before looking in the cache for a specific remote file.

CFS implements both the local file name hierarchy and the index for the cache of remote files with a B-tree keyed by complete file names. The B-tree is permanently stored in a disk file. A B-tree entry for a local file contains the unique identifier of the corresponding disk file. An entry for an attachment contains the name and possibly the creation time of the associated remote file. An entry in the B-tree for a cached remote file contains the unique identifier of the disk file that is the cached copy of the remote file.

Determining when to flush a file from the cache is left up to the DiskFile machinery under CFS. When CFS starts, it registers a procedure with DiskFile which is to be called to remove a remote file from the cache. DiskFile calls the procedure from a detached process that tries to keep 1000 pages free on the local disk. DiskFile will call the procedure synchronously with a client allocation request only when that request cannot be satisfied from the set of free pages already available on the disk. As a result, most allocation requests are satisfied without synchronously flushing the cache.

Having DiskFile trigger cache flushing helps to control disk fragmentation. DiskFile's allocator demands to find reasonably sized runs of pages and will call the cache flusher synchronously to make them available if necessary. Another virtue of this call-back scheme for cache flushing is that it allows DiskFile to share the disk dynamically among multiple clients. For example, Alpine [4] is a transactional file system that, when run on a workstation, also uses DiskFile to provide storage for its data base. When Alpine demands a bigger file for its data base, DiskFile can call CFS to flush the cache to make room.

Figure 3 shows the response time distribution for *Open* operations as observed during a compilation of a large software subsystem. The workstation computer was a Dorado [9]. The file server computer was an Alto [20] with 512K bytes of memory and multiple 300M byte disks. The server-to-workstation transfer was over a 3M bps experimental Ethernet [12]. This file server shared all Cedar system files for approximately 30 workstations. The file server load during the measurement is not known precisely, but the times recorded are representative of daily use. Note that the distribution is bimodal. Most times are less than 0.25 second. These times correspond either to remote files that already are cached or to local files. Starting at 0.75 second are remote files that had to be retrieved. The response time distribution for these files is centered around approximately 2 seconds. Almost all the time of an *Open* is spent waiting for the disk and/or the file server.

Discussion

A potential goal of a file system like CFS might be workstation operation when file servers are unavailable. This goal was not seriously addressed by the CFS design. Realizing the goal would require predicting future needs in order to specify which remote files to keep resident in the cache. A better approach is to develop highly reliable file servers using replication. Immutable remote files make replication easy to manage.

Another potential goal not addressed by CFS was eliminating the use of workstation disks for long-term private file storage. Such private files can cause our users to become dependent on a particular workstation. We considered but did not implement a scheme where an entire private workstation environment could be copied to a private directory on a file server. This saved environment would allow the user to move to another workstation, and also would allow the user to recover from the failure of a workstation disk. After partially developing the design for such a mechanism, we concluded that the software management tools reduced the need for such automatic backup. BringOver and SModel can be used instead to backup working files in remote private directories.

An important function of CFS is to provide a complete, consistent local naming environment in which to do development work on a software subsystem. In retrospect, the local locking mechanism provided by CFS

```
    Count

    0         10        20        30        40
    |         |         |         |         |
0.00 *****************************************->331
0.25 *
0.50
0.75 **
1.00 ****************
1.25 ***************************
1.50 **********************
1.75 **************************************
2.00 ****************************
2.25 ******************
2.50 ***********
2.75 ****************
3.00 ********
3.25 ******
3.50 ***
3.75 *****
4.00 *
4.25 *
4.50
4.75 *
>5   *****

Seconds for an Open operation
```

Figure 3: Histogram of File Opening Times

works against this purpose, and should be changed. The problem is that names and contents of files are locked together. As a result, a name cannot be deleted from the local naming environment if the corresponding file is open. Since some applications depend on the Cedar garbage collection mechanism [14] to close files, files often stay open after they are needed. Thus, tidying up the local naming environment by deleting unneeded names is sometimes thwarted. It would be better to allow name deletion to occur ahead of content deletion, the latter happening automatically when no more clients had the file open. For this scheme it is necessary to lock the name and the content of a file separately.

DF files look a lot like directories and provide another way to name files. It is tempting to consider integrating the DF files with the file system directories to provide a single naming mechanism. One approach to this consolidation would be starting with file servers that named files with unique identifiers. DF files would then provide a mapping between simple names and these uid's, and would become the directories of the workstation file system. In such a design it would be necessary to retain the immutability of DF file versions to support consistent sharing. If all file system directories were immutable, then any change would require new versions of all directories in a path back to the root of the name space. Thus, a practical system probably would require both immutable and variable directories. Such a design requires further exploration. The Cambridge File Server [7], with uid-named files, multiple file name indexes and automatic deletion of unreferenced files would provide an ideal base for such an exploration.

The cache makes it possible to operate a Cedar programmer's workstation effectively with $\sim$ 20M bytes of local disk storage. This number matches well the size of hard disk available at fairly low price today. This size cache also lowers significantly the load on the file servers. In our experience, a single file server running on an Alto can support 20 or more Cedar programmers using Dorado workstations, which are about eight times faster than an Alto. It appears that the system will scale to configurations with more servers and more workstations without suffering serious loss of performance or reliability. The system also works well when file servers and workstations are separated by gateways and slower long-distance internetwork links, rather than all being connected to the same local area network.

CFS started as a conservative design intended to meet the specific set of needs presented by program development activities in Cedar. Features from previous file systems (such as versions, keeps and symbolic links) were selected and combined with a few unproven features (such as creation time naming, sharing only immutable remote files and caching whole files) to meet the requirements of a well-understood, specific application. In retrospect, the combination of CFS's semantics with the higher-level tools for maintaining consistent versions of shared software subsystems has worked extremely well. Given sufficient local storage, we now believe it is unnecessary in this application to have shared file servers that provide mutable files, page-at-a-time access to files, long-term locks, or transactions. We do not understand yet the benefits that come from adding these features.

Acknowledgments

The Cedar Interim File System, a precursor to CFS developed by Dave Gifford with help from Larry Stewart, first explored the use of an automatically managed cache of remote files on the local workstation disk. The design and implementation of CFS was done primarily by Michael Schroeder, with advice from Andrew Birrell, Mark Brown, Butler Lampson, Roy Levin, Roger Needham, Eric Schmidt, Larry Stewart, Paul Rovner and Ed Taft. Comments from Andrew Birrell, Mark Brown, John Guttag, Cynthia Hibbard, Ed Lazowska, Roy Levin, Paul McJones and Greg Nelson greatly improved initial versions of the paper.

Appendix: Abstracts of Selected Operations

This appendix presents abstracts of the key operations from the CFS interface. The descriptions here omit some features. In particular, the working directory mechanism is not described fully and the error reporting mechanisms are not mentioned. For all operations, any file name argument that does not start with the character "/" has the name of the current local working directory prepended before being considered further.

FileInfo [name, wantedCreationTime, remoteCheck] $\rightarrow$
[fullName, attachedToName, keep, bytes, creationTime]

The *FileInfo* procedure returns information about the file designated by *name* and *wantedCreationTime*. A missing version part in *name* defaults to !H, indicating the highest existing version. If *wantedCreationTime* is specified then the version part of *name* is treated merely as a hint; the information returned is for the file with the specified creation time, found by searching all versions of the named file as necessary. There are three cases of behavior for *FileInfo*:

Case 1: *name* is local and not attached — The complete name of the designated local file including version part is returned as *fullName*. The keep, byte count and creation time for the local file also are returned. No *attachedToName* is returned. The *remoteCheck* argument is ignored.

Case 2: *name* is local, but attached to a remote file — The complete local name is returned as *fullName*. The keep of the local name is returned. The complete name of the attached remote file is returned as *attachedToName* and its creation time is returned. If *remoteCheck* is *FALSE* then the byte count is returned as -1, thus eliminating the need to open the remote file from the cache or check with the server just to determine the byte count. If *remoteCheck* is *TRUE* then the byte count is returned. Errors such as the server being inaccessible or not finding the remote file, that are encountered when trying to determine the byte count, are suppressed and -1 is returned instead. (The client usually will want the other information anyway.) Whenever a valid byte count is returned for an attachment then the version part in the *attachedToName* is the true version number that corresponds to the creation time for the attachment; otherwise this version part is whatever hint or variable was presented to the *Copy* operation when the attachment was made.

Case 3: *name* is remote — The complete remote file name is returned as *fullName*. A keep of 0 is returned (remote files do not have keeps). The true byte count and creation time are returned. No *attachedToName* is returned. If *name* ends with a version variable and no creation time is specified then *remoteCheck* controls access to the remote server. When *remoteCheck* is *TRUE* the server is always accessed for the file information. Otherwise the version variable is bound relative to the set of versions in the cache; the remote server is interrogated only if no version appears in the cache.

Open [name, wantedCreationTime, remoteCheck, readOrWrite]
→ [openFile]

The *Open* procedure returns an object that can be used to perform read, write and other operations on the specified file. *Open* first does *FileInfo [name, wantedCreationTime, remoteCheck]*. If an *attachedToName* results then that remote file is opened; otherwise the file named by *fullName* is opened. *readOrWrite* specifies the local lock to be set. Opening a file for writing causes the creation time to be updated. When a local name that is attached to a remote file is opened for writing, the attachment is broken and the contents of the remote file are copied onto a local disk file that is given the local name. (As an optimization, the copying will be done by renaming the cached remote file when it is not currently open.) Attempting to open a remote name for writing produces an error.

Create [name, setPages, pages, setKeep, keep] → [openFile]

A new local file with the specified *name* is created and opened for writing. The creation time is set. No version part may be included in *name*. CFS will assign the version number that is the successor to the existing !H version, or !1 if no versions exist. If !1 is being created or *setKeep* is *TRUE* then the keep of the new file is set to *keep*; otherwise the keep for the new file is that of the existing !H version. Creating a file triggers keep processing for existing versions. If one or more local files are deleted as a result, then one of them will be reused for the new version. If *setPages* is *TRUE* then the number of pages in the created file is set to *pages*. If *setPages* is *FALSE* then the number of pages in the new file is the same as the reused disk file, if any; otherwise it is set to *pages*. Attempting to create a remote name produces an error.

Copy [fromName, wantedCreationTime, remoteCheck, toName,
setKeep, keep, attach] → [fullToName]

The *Copy* procedure has many cases, because it can create attachments as well as actually transfer files. The *toName* cannot contain a version part. The version of the target file created is one larger than the existing !H version. In all cases, the complete name of the target file, including version number, is returned. Note that *Copy* is the only way to write a remote file.

Case 1: *attach* is *FALSE* and *toName* is remote — CFS does an *Open [fromName, wantedCreationTime, remoteCheck, read]* and transfers the contents and properties of the opened file to the newly created file on the remote server. The file transfer occurs synchronously. If *fromName* is remote then the file is transferred via the cache.

Case 2: *attach* is *FALSE* and *toName* is local — CFS opens the source file as in case 1 and does *Create [toName, setKeep, keep]* to generate the target file. The contents and properties are transferred from the source to the target open files. If the copy is from an uncached remote file then that file is not added to the cache; the only pages allocated on the local disk are those needed to hold the target file.

Case 3: *attach* is *TRUE*, *toName* is remote and *fromName* is local — Begin as for case 1. Once the transfer is completed the local name is attached to the remote name and creation time. The source local disk file is renamed to be the cached remote file.

Case 4: *attach* is *TRUE*, *toName* is local and *fromName* is remote — Like case 2 except that instead of an actual transfer of contents and properties the local name is attached to the remote name and creation time. If no *wantedCreationTime* is specified or if *remoteCheck* is *TRUE* then *FileInfo [toName, wantedCreationTime, remoteCheck = TRUE]* is performed first to determine/check the version number and creation time for the remote file. When *remoteCheck* is *FALSE* then the attachment is made to the *fromName* and *wantedCreationTime* provided, without checking either the remote server or the cache. (BringOver sets *remoteCheck* to *FALSE* to speed operation.)

Case 5: *attach* is *TRUE* and both *fromName* and *toName* are local or both are remote — This case is illegal.

Delete [name, wantedCreationTime]

A missing version part in *name* defaults to !L, meaning the lowest existing version. The *name* and *wantedCreationTime* are resolved to a complete file name using the semantics described in *FileInfo*. The named file is deleted. An error occurs if the file is currently open on this workstation. Remote deletions occur directly on the remote server. The deleted remote file is removed from the cache if present. If *name* is local but attached to a remote name, then just the local name is deleted; the remote file is unaffected.

SetKeep [name, keep]

The *name* must be local (keeps on remote servers have not been implemented) and cannot contain a version part. The keep on the !H version is set. Setting the keep causes any unopened versions that are beyond the new keep to be deleted. Setting the keep to 0 leaves the current keep but does the keep processing.

References

[1] Bensoussan, A., Clingen, C.T. and Daley, R.C., "The Multics Virtual Memory: Concepts and Design," *Comm. ACM 15*, 5 (May 1972), pp. 308–318.

[2] Bobrow, D.G. et al., "TENEX, a Paged Time Sharing System for the PDP-10," *Comm. ACM 15*, 3 (Mar 1972), pp. 135–143.

[3] Boggs, D.R. et al., "PUP: an Internetwork Architecture," *IEEE Trans. on Comm. 28*, 4 (Apr 1980), pp. 612–634.

[4] Brown, M.R., Kolling, K.N. and Taft, E.A., "The Alpine File System," to appear in *Trans. on Comp. Sys. 3*, 4 (Nov 1985).

[5] Brownbridge, D., Marshall, L. and Randell, B., "The Newcastle Connection — or UNIXes of the World Unite!," *Software Practice and Experience 12*, 12 (Dec 1982), pp. 1147–1162.

[6] Crisman, P.A., ed., *CTSS Programmer's Guide, 2nd Edition*, MIT Press, Cambridge, Mass., 1965.

[7] Dion, J., "The Cambridge File Server," *ACM SIGOPS Operating Sys. Review 14*, 4 (Oct 1980), pp. 26–35.

[8] Donahue, J., "Integration Mechanisms in Cedar," *ACM SIGPLAN Notices 20*, 7 (July 1985), pp. 245–251.

[9] Lampson, B.W. and Pier, K., "A Processor for a High-Performance Personal Computer," *Xerox Palo Alto Research Center Report CSL-81-1*, Jan 1981.

[10] Lampson, B.W. and Sproull, R.F., "An Open Operating System for a Single-User Machine," *Proc. 7th ACM SIGOPS SOSP*, Dec 1979, pp. 98–105.

[11] Leach, P. et al., "The Architecture of an Integrated Local Network," *IEEE J. on Selected Areas in Comm. SAC-1*, 5 (Nov. 1983), pp. 842–856.

[12] Metcalfe, R. and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM 19*, 7 (July 1976), pp. 395–404.

[13] Reed, D.P. and Svobodova, L., "SWALLOW: a distributed data storage system for a local network," *Local Networks for Computer Communications*, North-Holland, Amsterdam, 1981, pp. 355–373.

[14] Rovner, Paul, "On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language," *Xerox Palo Alto Research Center Report CSL-84-7*, July 1985.

[15] Satyanarayanan, M., et al., "The ITC Distributed File System: Principles and Design," to appear in *ACM SIGOPS Operating Sys. Review 19*, 5 (Dec 1985).

[16] Schmidt, E.E., "Controlling Large Software Development in a Distributed Environment," *Xerox Palo Alto Research Center Report CSL-82-7*, Dec 1982.

[17] Svobodova, L., "File Servers for Network-Based Distributed Systems," *Comp. Surveys 16*, 4 (Dec 1984), pp. 353–398.

[18] Swinehart, D.C., Zellweger, P.T. and Hagmann, R.B., "The Structure of Cedar," *ACM SIGPLAN Notices 20*, 7 (July 1985), pp. 230–244.

[19] Teitelman, W. "The Cedar Programming Environment: A Midterm Report and Examination," *Xerox Palo Alto Research Center Report CSL-83-11*, June 1984.

[20] Thacker, C. et al., "Alto: A Personal Computer," *Xerox Palo Alto Research Center Report CSL-79-11*, Aug 1979.

[21] Walker, B. et al., "The LOCUS Distributed Operating System," *ACM SIGOPS Operating Sys. Review 17*, 5 (Oct. 1983), pp. 49–70.

[22] Walsh, D., Lyon, R. and Sager, G., "Overview of the Sun Network File System," *Usenix Winter Conf. Dallas 1985 Proc.*, pp. 117–124.