CITI Technical Report 94-2

Using DFS Without DCE/LFS

Charles Lever
Chuck.Lever@umich.edu

ABSTRACT

The Open Software Foundation's Distributed Computing Environment includes a networked file system called DFS. Many independent system vendors wish to use their own physical file systems with DFS protocol exporters provided by Transarc instead of using DCE/LFS, Transarc's high performance Local File System (also known as Episode). It is possible to use a non-DCE/LFS file system with DFS, but the road to implementation is paved with compromise.

January 19, 1994

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

Using DFS Without DCE/LFS

Charles E. Lever

January 19, 1994

The Open Software Foundation's Distributed Computing Environment (DCE) enables applications to take advantage of disparate resources on a network of heterogeneous hosts. Because the hosts may be running different operating systems on different hardware, applications access DCE services via standardized interfaces. A standard access method for file system data is one of the services provided by DCE.

Data stored in DCE's Distributed File System (DFS) can reside on any machine participating in DCE and running a DFS **protocol exporter**. DFS Client applications see a file system interface that is compliant with the IEEE 1003.1 POSIX standard [1], providing byte-stream files, hierarchical directories, etc. The client workstation has a DFS **cache manager**, which off-loads many data requests from file servers by maintaining a consistent cache of data which applications on the client may be using. OSF provides a POSIX-compliant, high performance file system designed by Transarc for use with DFS protocol exporters. This file system is called DCE/LFS, or Episode [2].

The IFS project at the U-M's Center for Information Technology Integration (CITI) has implemented a version of DFS that can serve data stored on IBM S/390 mainframes. We are interested in providing DFS service from the mainframe in order to take advantage of centralized resource management and the high bandwidth, reliability, and total storage capacity of the mainframe I/O subsystem.

One claim made by DFS developers has many people excited about DFS: DFS can use native file systems as easily as it can use DCE/LFS because the DFS protocol exporter uses extensions to the ad hoc industry-standard interface to physical file systems known as VFS [3, 4]. Most mainframe operating systems, however, do not have a native POSIX-compliant, byte-stream file system. To provide DFS service from a mainframe, a physical file system in which DFS is to store its byte-stream files must be chosen. An immediate alternative is to use a ported version of the DCE/LFS file system. Other alternatives include a ported version of UFS or to use a native non-byte-stream file system with some amount of extra functionality.

What features are necessary in a physical file system in order to provide partial or complete DFS service? Towards answering this question, this paper reviews the use DFS makes of evolving interface standards; its system security, administration, and recovery models; and especially client performance requirements. It is possible to use a non-LFS file system with DFS, but the road to implementation is paved with compromise.

DFS System Characteristics

A DFS client has several fundamental expectations of DFS file servers, involving function, performance, and security. As more data and users are added to systems, DFS system administrators and users expect that DFS will scale well and be easy to administer.

The following sections outline the DFS file server characteristics and design requirements that depend on characteristics of the physical file system supporting DFS protocol exporters.

Performance

Applications that access data via a DFS client expect performance as good as a file system that stores data locally. DFS employs a sophisticated local disk caching mechanism to ensure that DFS performance comes as close as possible to a local file system's performance [5].

Because more users can access data stored in DFS, performance expectations are more stringent for DFS than they are for a traditional local file system. The designers of DFS believe that, in a given cell, there may be tens of thousands of users. The IFS project at the University of Michigan provides a distributed file service that stores data not only for naive users reading their mail and storing small amounts of data, but also for applications requiring large amounts of data to be stored and delivered quickly, such as applications run by economists, population statisticians, or X-ray crystallographers. Future users will need high bandwidth capability and guaranteed low request latency to deliver audio and motion video data to multimedia applications.

Logical Volume Managers and RAID technology are fast becoming methods by which modern file systems can gain orders of magnitude in storage capacity and access speed [6]. The Veritas VxFS and the AIX JFS file systems have companion logical volume managers. Because DCE/LFS uses a standard kernel device driver interface, a logical volume manager can be easily designed and added.

Scaling

DFS clients maintain a local cache of data also to reduce server load [5], allowing a high client-to-server ratio.

Whereas a server's network interface may bottleneck client requests, our experience shows that as this bottleneck is eased, a server's disks and its physical file system will be the next performance constraint. This constraint depends directly on how much data the physical file system contains and how many users access that data concurrently.

It is rumored that in the next release of DCE, the DFS network protocol will allow file sizes of up to 2^{64} bytes. The current POSIX standard file system interface does not allow access to bytes beyond 2^{32}-1. However, other non-POSIX interfaces accessing files through the DFS cache manager can store and retrieve bytes beyond byte 2^{32} in a file.

Reliability

In DFS, the physical file system performs significant crash-recovery processing. When a file server stores the data of thousands of users, file server outages can have widespread effects, especially if crash recovery is time- and resource-intensive.

It has long been recognized that `fsck` [7] is not an optimal recovery solution because the amount of time for it to recover a single partition depends on how much data the partition contains [5]. Modern file system designs, such as JFS [8] or Sprite LFS [9], reduce recovery time to mere seconds. The methods they use to reduce recovery time also have benefits for metadata write perfor-

mance, a long-time bottleneck for UFS-like file systems.

One major performance compromise for file systems is that they must write metadata to disk serially. File system metadata is information that describes the structure of an on-disk file system. It is important that metadata always remain consistent so that, in the event of a crash, a file system can find all user files again. UFS, for example, mitigates this performance problem by allowing some lazy updates of on-disk metadata.

DCE/LFS implements a database-style metadata change log similar to that described in [10]. Applications can force user data and metadata changes onto disk, regardless of how many files are open and being written to. DCE/LFS designers streamlined the logging mechanism to allow highly concurrent updates of user data and metadata to occur, while at the same time greatly reducing the recovery time in the event of a system crash. Usually, the file system replays only the active portion of the log to recover a consistent on-disk file system state.

Thus, the physical file system supporting DFS protocol exporters must be capable of storing large amounts of data. It must also have low access latency and fast recovery.

File Space Administration

Increasing the amount of data stored in a file system increases the amount of administration required. DFS provides a unique administrative model that greatly eases the task of administering data stored for thousands of users on tens of file servers. The key to the DFS administrative model is the fileset [11].

A **fileset** is an independent subset of the namespace that looks like a single, whole UFS-style file system. (In fact, a fileset is mountable by a server's kernel.) Usually, administrators allocate one or more filesets where each DFS user will store her own files. Administrators also typically join the filesets to the whole DFS hierarchical namespace (users can also do this). Many filesets can reside on a single disk partition, and administrators can add filesets to, or remove them from, a disk partition without reformatting that partition. Administrators can limit the number of bytes in a fileset dynamically, but filesets do not have a static file-count limit.

Administrators can move filesets among servers in a cell to keep the server load balanced or to recover from server problems without interrupting the structure of, or access to, the file system as a whole. Each fileset is stored and moved independently so that administrative operations briefly affect only one fileset at a time. Fileset location information is stored separately from the filesets themselves.

Some file systems, such as UFS, do not provide the fileset abstraction. A UFS file system, when exported by the DFS protocol exporter, appears as a single fileset, administered locally, which DFS tools are not able to move. This is desirable when DFS is simply providing network access to a host's locally administered file system. However, a fully functional DFS implementation requires that a fileset-like abstraction be present in the physical file system to allow remote administration.

The DFS protocol exporter relies on the physical file system to provide **cloning**, a function related to filesets. A cloned fileset is a read-only snapshot of an online, read-write fileset. Generating a clone should be a low-overhead operation. Several important administrative operations make use of clones. For instance, a fileset is cloned first, then backups are created from the clone, ensuring a consistent version of the files and fileset on backing store and keeping the availability of data in the read-write fileset very high. During a fileset move operation, a clone is created so that most of the data in a fileset can be moved (potentially a long-running operation for large filesets) without making the read-write fileset unavailable to users. Fileset replication, which allows sev-

eral copies of a fileset to exist on different servers, also uses the clone operation.

This administrative model has consequences for the physical file system a DFS file server uses to store data. DCE/LFS uses a block copy-on-write mechanism to implement fileset cloning [2]. This allows clones to use little disk space, without taking the read-write fileset offline for long periods of time while creating the clone. Whole-file copy-on-write, as used in AFS 3, can be inefficient when updating large files. DCE/LFS developers admit that probably the most complex parts of DCE/LFS are those which manage block copy-on-writes [12].

The DFS protocol exporter and client depend on the server's physical file system to implement filesets and fileset cloning so DFS protocol exporters can provide the most useful administrative tools. The success of an alternative file system in providing the full suite of DFS features depends on its ability to provide the same abstractions.

Security—Authentication and Authorization

DFS operates in an environment where the MIT-developed security service, Kerberos [13], authenticates network entities such as users, client applications, and servers. **Authentication** is a way of guaranteeing the identity of a user or application on an unsecured network. A DFS server must guarantee the identity of a service requester so that only those who have proper authorization for data can access it.

The DFS implementors have recently studied various methods [12] of allowing an unauthenticated network file system, Sun's NFS [14], to gain access to files stored on a DFS server. If an NFS server, for example, has direct access to a physical file system where DFS file data is stored and the NFS server does not authenticate its clients, any user claiming to be authorized can access data on that file server. That is, the NFS protocol allows clients to forge their identity and gain access to data for which they may not be authorized.

Some implementors have chosen to concurrently export multiple file system protocols with varying degrees of security from the same shared physical file system. If any one of these protocol exporters does not authenticate remote requests, then all data stored in that physical file system is exposed. For example, allowing a DFS protocol exporter, an NFS server, and a server message block (SMB) file server to share the same physical data repository would be a security disaster. In short, data stored in a shared repository is only as safe as the least secure way of accessing that data.

In order to protect access to data stored by DFS, DFS implementors have suggested that all alternate protocols should access files stored on a DFS server through a DFS client, ensuring that Kerberos has authenticated all non-DFS users. While some believe this is inefficient [15], the DFS client itself can be designed so that there is minimal or no overhead for caching data; i.e., the intermediate DFS client serves only as a pass-through mechanism for requests. In this way, both authenticated and unauthenticated clients can access the DFS data repository quickly without compromising the security of data stored in DFS.

A server must determine if a user has authorization to make requests against particular files. The DFS protocol exporter itself does not check whether a request against a file is authorized; rather, it lets the physical file system do this, since this is where the file's **access control list** (ACL) is stored. Such a physical file system must have an interface to store and retrieve authorization information and be able to evaluate authenticated identities with the authorization information it has stored for a file.

Traditional UFS uses **mode bits** stored in a file's metadata to record authorization information. Many system designers have decided that mode bits are simply not flexible

enough for distributed file systems, given the increased number of files and the number of users accessing these files in such an environment. The POSIX standard 1003.6, Draft 13 [16] contains a standard description of ACLs. Transarc and OSF have adopted this draft to provide access control to files and directories stored within DFS.

While mode bits can be mapped to an ACL, ACLs cannot always be mapped back, since an ACL can contain a richer set of authorization information. There is no "`ACCESS`" RPC routine in the NFS network protocol, so the current NFS release uses an ad hoc method to determine whether access is authorized. An NFS client accessing DFS files simply examines a file's mode bits, which may not accurately reflect the full authorization status of the file. The next release of NFS is destined to contain an "`ACCESS`" RPC routine to allow NFS to coexist with file systems that require ACL support.

Multiple protocol exporters can share a physical file system, but administrators of such a system must be aware that "back doors" exist. Authentication and ACLs are necessary to provide the kind of resource authorization flexibility that heterogeneous and highly distributed environments require. Old security methods do not easily substitute for them.

Interface Issues

The DFS protocol exporter expects to interface with a physical file system that exports files through a VFS interface, as described by [4] and amplified by Transarc [3]. The physical file system is anticipated to handle many interesting functional features of DFS, such as fileset cloning and access control lists, which are available through the extended VFS+ interface.

Interface semantics

Adapting DFS to a pre-existing file system solves the user- and application-migration problem, since DFS can then export legacy data while the data remains available to native applications. In the case of mainframes, however, native applications generally manipulate records, not bytes. Converting byte stream data to record data, and vice versa, can be done on an application-by-application basis. It is not appropriate, however, to provide in the file system a single general solution for the conversion problem. Having to build a physical file system that supports DFS with a POSIX file system is a win; having to build with a non-POSIX file system that doesn't have a VFS interface requires much more design and implementation work.

The physical file system writes directories, but allows applications to read them directly through the `vn_readdir()` function. This function provides a byte-stream directory read operation so that applications can acquire the names of files and subdirectories contained in any directory. This interface exports a well-known, but not standardized, directory format that may or may not be the same as the internal directory format of the file system. This format contains the file and directory names themselves in ASCII. These names can contain almost any character, even blanks, which can be a problem for mainframe file systems, for example.

When an application opens a file, it specifies the ASCII pathname of the file. The kernel, in cooperation with the DFS cache manager, looks up the file by starting at the current directory or the top of the file system hierarchy and walks down the directory tree specified in the path. This is called a **pathname lookup**. At each directory in the path being looked up, the process continues only if the next directory exists and the application is authorized to do a lookup in that directory. If there is a directory anywhere in the specified path that has an ACL that does not authorize that application to do a lookup operation, the application is not allowed to open the file. Some file systems, such as the CMS Shared File System, allow access to a file based only on the authorization of that file,

not on its containing directories. DFS ACLs control directory lookup operations as well as read or write access to files.

The DFS protocol exporter and the fileset location database also get in the act during a pathname lookup. The DFS cache manager checks each directory in any pathname lookup with a separate RPC to the DFS protocol exporter, the results being cached by the cache manager to improve performance. If the cache manager encounters a junction between two filesets, it will consult the fileset location database to discover which protocol exporter contains the new fileset. Thus, the server's physical file system only has to store and retrieve directories, symbolic links, and junctions—not handle the mechanics of a pathname lookup.

A DFS FID is a numeric value, shared between a sever and client, that represents a file or directory. DFS FIDs have a number of important characteristics that must be preserved if a fileset is moved between DCE/LFS-based DFS servers and non-DCE/LFS-based servers. A DCE/LFS FID, which is part of a DFS FID, is the real index in the on-disk fileset table of a file's anode. Some implementors of other file systems have considered generating FIDs from other sources so that FIDs would always be numerically very large, even if a fileset only contains a few objects. The fileset move operation does not change the numeric value of any FIDs; it is unclear to us how DCE/LFS might react to a fileset that is very large and sparsely populated.

A DFS FID also contains an anode's **uniquifier**. Every fileset has its own 32-bit uniquifier that the physical file system increments every time the fileset changes in some way. When an anode slot is reused (that is, when a file is deleted, and its anode is used for a new file), the fileset uniquifier value is copied into the reused anode's uniquifier. DFS cache managers cache these uniquifiers to determine when a FID has been reused.

The VFS Interface

To run DFS without modification requires the existence of at least a VFS interface. Without a fully functional VFS+ file system, the DFS servers can only provide limited functionality. Transarc provides some code that allows a limited-function DFS to run on a VFS file system.

Vendors can reimplement DFS to use their own native file system, but there are disadvantages to this approach if the native physical file system does not already have a VFS or VFS+ interface. Such a reimplementation of DFS will always be behind the current release.

The DFS protocol exporter, the replication server, and the fileset server access the server's physical file system directly through VFS+. Reimplementing DFS on a file system without VFS+ means changing not only the protocol exporter, but also the fileset server and the replication servers. These three servers account for a significant body of code, both in quantity and complexity. In addition, managing quality control in two or more versions of DFS might be costly.

Access Synchronization

A significant design feature of DFS is its mechanism for synchronizing access to data in files to create the illusion that all network applications are running on a single system [3]. Clients ask a server to notify them of any changes made to data in a file so they can maintain the latest copy of the data in their caches, while also allowing other clients to make changes to the data. DFS protocol exporters grant **tokens**, which represent the right to perform a particular operation on files. Tokens have types, depending on how the client intends to access the file (read, write, whether the file should be locked, etc.). The DFS protocol exporter grants and revokes tokens as several clients read and update data in the same file so that all clients can maintain a consistent image of the file's data in their caches.

Part of the VFS+ interface is a layer of protection, the "glue layer," through which the server's kernel accesses the physical file system, acquiring tokens so that it will not cause the protocol exporter to break unknowingly any guarantees it may have made to remote clients. The architecture of the VFS+ interface assumes that all applications, local and remote, will access data through this interface, and not some other proprietary interface behind it.

For example, a VFS+ layer can be implemented on a mainframe file system to appear to that file system like just another local application. In such an implementation, the semantics of vnodes and DFS token management change because local accessors can change data in files without the knowledge of the protocol exporter. This subverts any DFS token guarantees about data in the file.

Several operating systems already have a token-like system to guarantee data consistency. These are, more often than not, lock managers. Excluding access to data is possible with the DFS token manager; however, tokens have a limited life span that can expire asynchronously, and clients can choose to ignore tokens. When comparing tokens to a pre-existing lock management implementation, it is important to completely understand the semantics of token management.

Architectural Considerations

The DFS server suite expects primitive file system operations to be asynchronous and reentrant, allowing multiple threads to execute inside the DFS. Other kernel routines, such as paging support, can call a kernel-resident VFS interface [17]. Kernel and VM integration are not necessary on systems such as mainframes, where DFS will be exporting file systems that are not part of the operating system nucleus or directly involved with virtual memory support. However, studies by DCE/LFS implementors [10] have demonstrated that running the file system and exporter in the kernel together with networking support enhances performance.

Some file systems keep an in-memory control block for each file within a disk partition. The amount of virtual storage that can be used to maintain these control blocks limits the number of files that can be stored in such a physical file system. On VM/ESA, for instance, a virtual machine can own very large address spaces (up to 2 gigabytes), but page faults for such large address spaces cause resource serialization, which can affect performance in the virtual machine that owns the address space.

LFS and other vnode-based file systems have a fixed number of control blocks that hold the status of the most recently active files in the physical file system. The file system determines which files these control blocks represent, and therefore can control, when data in the control blocks is moved between memory and permanent storage without causing a blocking page fault or other serialization.

Accessors of vnode file systems generally do not use their open or close operation. A VFS file system creates a vnode when a file is first referenced and generally destroys it if the file it represents hasn't been used recently and a fresh vnode is needed. A vnode file system controls the existence of vnodes in much the same way an operating system manages real storage frames for virtual memory. Files may be in an opened or closed state at a higher level, but there is no requirement to call `vn_open()` or `vn_close()` for a file represented by a vnode, and in fact, none of the DFS servers open a file in the physical file system before they use it. In traditional UFS, `vn_open()` and `vn_close()` do nothing. A physical file system must handle implicit open and close operations.

On-disk structure

Extending the file system to implement ACLs, PLists, and filesets is difficult. Access control information is part of the file system's metadata and thus is subject to the same integrity guarantees [2]. Unlike other file system metadata, ACLs are byte streams

that can be as long as 8 kilobytes. Like metadata, however, ACLs do not appear in the hierarchical namespace, nor are they accessible via a DFS FID. Adding ACLs to a file system is not trivial and usually requires incompatible changes to on-disk file system data structures.

System PLists have the same properties as ACLs. Like ACLs, they are metadata and have a maximum length of 8 kilobytes. Neither do they appear in the file system directory hierarchy. User PLists have unlimited length, and though considered a part of the file system metadata, they are not subject to the same integrity guarantees. DCE/LFS currently has no VFS+ functional interface that supports PLists, although the internal function is already extant.

ACLs and PLists are not the only extra items of metadata that may be foreign to a non-DFS file system. DFS file-level metadata includes information about the last file access and modification times, as well as mode bits and other DFS client status and security information (as described by the POSIX standard). In addition, fileset metadata (e.g., the fileset header) must be stored somewhere in the physical file system.

A well-known deficiency of AFS 3 is that it has some difficulty maintaining files that have a zero-link count during system instability. Often an application opens a temporary file to acquire a file descriptor, then unlinks the file so that it will be garbage-collected if the application crashes. If the file server should crash, the unlinked file and its contents will be lost, even though the application is still running. DFS resolves this problem by maintaining references to zero-link-count files in a special area in the fileset header. Part of DFS server recovery reviews this list and makes these zero-link-count files available for clients during token state recovery processing. This processing is built in to DCE/LFS and should be available in any physical file system supporting DFS.

Concluding Remarks

It is clear from examining the DFS/LFS architecture that the server's physical file system provides many important features in DFS. These features include providing fileset abstraction, cloning, quick recovery from system crashes, and high performance. Creating a substitute for DCE/LFS requires a substantial design and implementation effort. The DCE/LFS designers started from scratch because the functionality they wanted did not exist in any file system available to them, and they believed that adding it would be at least as much work as starting over [10].

Several compromises are necessary to use an existing file system. Designers should review at least the following issues:

- Security requirements, if you want to share the data among different protocol exporters or local accessors.
- Administrative requirements, for controlling how locally accessed file systems will be administered and whether DFS administrative mechanisms will be supported.
- Performance and scaling, especially of backup tools and recovery mechanisms. How will the file system store large files (greater than 1 gigabyte), and how it will deliver high-request bandwidth?
- Record structure and file name conversion issues.
- Authorization information conversion, interpretation, storage, and retrieval.
- Storage and retrieval of PLists and other metadata new with DFS.

As a simple example, consider exporting the DOS FAT file system (FFS) via DFS. There already exists a VFS interface for DOS FFS [8]. However, there is no fileset abstraction; if we desire remote administration via DFS fileset tools, we must add some form of filesets and cloning.

We must consider how FFS will store ACLs, PLists, and other metadata foreign metadata or consider compromise solutions with limited functionality. If we require decent security, we will have to add some provision for authorization and authentication checking to the VFS layer. We should also consider maximum file size, file-name length and character limitations.

Will the DFS protocol exporter run on a UNIX system or a DOS system? If from a DOS system, how will local applications access the file system being exported? Does DOS FFS support threads and asynchronous I/O? Will it perform adequately? What recovery mechanisms are already in place?

Clearly, the whole of DFS relies upon many special features of the new DCE/LFS physical file system. If there is any desire for functional equivalence with DCE/LFS implementations, we will have to add most of DCE/LFS to any pre-existing file system we might wish to export with DFS.

Acknowledgments

I'd like to thank Bob Riddle for his support of this work. Thanks to Mike Graham, Ron Zeilinger and Charles Antonelli for their constructive comments. Finally, thanks go to Andy Adamson for asking me to assemble an informal talk that helped me to organize the ideas in this paper. This work was partially supported by IBM.

References

1. IEEE, "POSIX: Part I, System Application Program Interface (API) [C Language]," *IEEE Standard 1003.1-1990*, January, 1990.
2. Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham; "The Episode File System," *Proceedings of the USENIX Conference*, Jan. 1992, pp. 43–60.
3. Michael L.Kazar and Edward R. Zayas, *DCE DFS VFS+ Interface Specification*, Transarc Corp., 1991.
4. S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Summer Conference Proceedings*, Atlanta 1986, USENIX Association, 1986.
5. J. H. Howard, M. L. Kazar, S. G. Nicholes, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988.
6. David A. Patterson, Garth A. Gibson, and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the 1988 ACM SIGMOD Conference on Management of Data*, Chicago IL, June 1988, pp. 109–116.
7. Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Gabry, "Fsck: The UNIX File System Check Program," *UNIX System Manager's Manual*, BSD Virtual VAX-11 Version 4.3, USENIX, April 1986.
8. A. Chang, M.F. Mergen, R.K. Rader, and S.L. Porter, "Evolution of Storage Facilities in AIX Version 3 for RISC System/6000 Processors," pp. 105–110, *IBM Journal of Research and Development*, Vol. 34, No. 1 (Jan., 1990).
9. Mendel Rosenblum and John K. Ousterhaut, "The Design and Implementation of a Log-Structured File System," *Thirteenth ACM Symposium on Operating System Principles*, October 1991.
10. C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," pp. 94–162, *ACM Transactions on Database Systems*, Vol. 17, No. 1, March 1992.
11. Robert N. Sidebotham, "Volumes: The Andrew File System Data Structuring Primitive," *European UNIX User Group Conference Proceedings*, Aug., 1986.
12. Class materials and presentation of Transarc's DFS Implementor's Class.

13. J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," pp. 191–202 in *Usenix Conference Proceedings*, Dallas, Texas, February 1988.

14. D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss, "Overview of the Sun Network Filesystem," *Winter Usenix Conference Proceedings*, Dallas, 1985.

15. D. Muntz, P. Honeyman, "Multi-level Caching in Distributed File Systems," CITI Technical Report 91-3, August 1991.

16. *IEEE Standard: Portable Operating System Interface for Computer Environments*, IEEE Standard 1003.6, Draft 13, The Institute of Electrical and Electronic Engineers, 1992.

17. Robert A. Gingell, Joseph P. Moran, William A. Shannon, *Virtual Memory Architecture in SunOS*, Sun Microsystems, 1987.

The Open System Foundation, OSF, and DCE are trademarks of the Open System Foundation.

DFS, LFS, and Episode are trademarks of the Transarc Corporation.