# DEcorum File System Architectural Overview

*Michael L. Kazar*
*Bruce W. Leverett*
*Owen T. Anderson*
*Vasilis Apostolides*
*Beth A. Bottos*
*Sailesh Chutani*
*Craig F. Everhart*
*W. Anthony Mason*
*Shu-Tsui Tu*
*Edward R. Zayas*

Transarc Corp.
The Gulf Tower
707 Grant St.
Pittsburgh, PA 15219
kazar@transarc.com, bwl@transarc.com

*ABSTRACT*

We describe the DEcorum file system, a distributed file system designed for high performance, low network load, easy operation and administration, and interoperability with other file systems. The DEcorum file system has three components: the DEcorum protocol exporter (or file server); the *Episode* physical file system; and the DEcorum client (or cache manager). Episode is a module that implements the Vnode/VFS interface, using transaction logging to allow fast recovery from crashes. To be exact, it implements a *VFS+ interface*: extensions to the standard Vnode and VFS interfaces, and two new modules, *aggregates* and *volumes*, which give flexibility beyond what is provided by Unix partitions to support administration and operation of networks of thousands of workstations. The DEcorum protocol exporter provides remote access to the Episode physical file system via remote procedure calls (RPC's). It can export access to other physical file systems, such as the Berkeley fast file system, using extensions of the physical file systems to support the VFS+ interface. The DEcorum client exports a Vnode interface, but obtains its data by making RPC's to a DEcorum protocol exporter. It caches data from the file server. To synchronize accesses to files, preserving single-system UNIX semantics, it relies on typed *tokens* obtained with the data: guarantees provided by the server that various operations can be performed remotely. Tokens can be revoked by the file server using separate RPC's. A locking hierarchy is used to avoid deadlock between clients accessing files and servers revoking tokens on the same files; we explain the hierarchy and informally sketch a proof of its correctness.

## History and Motivation

The DEcorum file system is a component of a larger system, the DEcorum Distributed Computing Environment (DCE). The DEcorum architecture was jointly developed by Hewlett-Packard, IBM, Locus Computing, and Transarc. The DEcorum file system component is a follow-on of AFS (formerly the Andrew file system).[1] The DCE is built as a layered architecture from modular components. The file system utilizes a collection of modular components, including Hewlett-Packard's NCS 2.0[2] remote procedure call facility, Hewlett-Packard's PasswdEtc

[1] Alfred Z. Spector and Michael L. Kazar, ''Uniting File Systems,'' *Unix Review*, March 1989.

[2] NCS 2.0 adds support for pipes (streaming), long-haul operation, authentication, and connection-oriented transport to NCS 1.5

authorization component, MIT's Kerberos authentication system, and many others.

AFS is a distributed file system originally developed at Carnegie Mellon University's Information Technology Center (ITC); its development has subsequently been taken over by Transarc Corporation. It offers local-remote file access transparency, and ease of operation and administration. It uses caching to achieve high performance and low network load.

The goal of the design of the DEcorum file system is to carry over the benefits of the existing AFS design, but to improve on it in its known areas of weakness. Specifically,

- Interoperability with existing file systems is improved, so that if a file server is installed on a host running UNIX, the server can export file systems that were already in use on that host.
- File server availability is increased (mean time to recover is decreased), through design of a physical file system that does not require a lengthy file system salvage process after a crash.
- Improved caching algorithms support strict UNIX single-system semantics in sharing of files. As a side benefit, network traffic is lessened, and client performance improved.

This list is not meant to be exhaustive. Some other ways in which improvements are made to AFS are mentioned in the following section.

This paper describes the design of the DEcorum file system. We emphasize the aspects that improve upon the older AFS design. To provide context, we also describe some aspects that were carried over from AFS.

## Overview

The three principal improvements described in the previous section are achieved as follows:

- Interoperability with existing file systems is improved by a clean separation between the higher level of the file server, called the *protocol exporter*, and the lower level, which is the *physical file system*. The separation is at the level of the virtual file system (VFS).[3] We define a physical file system as a module that implements the VFS interface, and stores file data on a disk (as opposed to using file data retrieved from other nodes in a network). Though a physical file system has been designed specifically for use with the DEcorum protocol exporter, other physical file systems can be used. For instance, the file systems, descended from the Berkeley Fast File System,[4] supplied by vendors of various platforms can be exported. Section 1 describes the structure of the DEcorum client and file server, and their interfaces with non-DEcorum file systems and other code. In Sections 2, 3, and 4 we expand upon the structural descriptions of Section 1: Section 2 describes the Episode file system, Section 3 describes the DEcorum protocol exporter and other server components, and Section 4 describes the DEcorum client.
- Cache consistency is achieved by the use of typed *tokens*, representing guarantees made by the file server to the client about what operations the client can perform locally. The corresponding ''callback'' system in AFS, because it was untyped, was not capable of representing the full variety of guarantees that might be useful to the client. Section 5 describes the token system. Section 6 discusses the potential deadlocks that arise from caching, and how they are avoided in the context of the token system.
- The need for a file system salvage (the notorious *fsck*) is obviated by the use of *logging* in the server's physical file system. Section 2.2 describes the logging and recovery system and its position in the structure of Episode. We show that logging, far from causing excess disk activity during routine operation, actually *improves* performance, by reducing the need for synchronous writing of meta-data to disk.

Other differences between DEcorum and AFS are described incidentally to various sections. Particularly worthy of note are POSIX-compliant access control lists (Section 2.3), provision for diskless clients (Section 4.2), and provision for lazy replication of files (Section 3.8).

(Lisa Zahn *et al.*, *Network Computing Architecture*, Prentice Hall, 1990).

[3] S.R. Kleiman, ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX,'' *USENIX Conference Proceedings* (Atlanta, Summer 1986), pages 238-247.

[4] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, ''A Fast File System for UNIX,'' *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pages 181-197.

Throughout the paper we compare the DEcorum design with those of other distributed file systems, notably AFS and NFS,[5] and with non-distributed file systems. In the last section we summarize these comparisons and summarize the contribution of the DEcorum design.

## System structure

### 1. Overview of system structure

We speak of a node in a network as either a *file server*, maintaining file systems on disk and *exporting* them to other nodes; or a *client*, running applications that access files exported by servers. A node can simultaneously act as both server and client. The DEcorum file system includes software for both servers and clients. Figure 1 shows the structure of the server side of a DEcorum file system, and figure 2 shows the structure of the client side. In both diagrams, an asterisk indicates a component that is not part of the DEcorum software, that is, a component taken from whatever existing UNIX kernel runs on the node.

Client and server communicate via remote procedure calls (RPC's); thus RPC interfaces are shown in both diagrams. On the client side, the community of server file systems appears as a single file system, presenting a Virtual File System (VFS) interface to the Unix kernel.[6] The *cache manager* is the component that implements the remote file system, including subcomponents to manage a cache (using a native file system), and to fetch and store files from and to remote servers.

On the server side, we again rely on the VFS interface. A file system that is to be exported must present a VFS interface, to which we add (to the extent that it is practicable) extensions, called the *VFS+ interface*, to make possible the volume-level operations discussed in later sections. A *glue layer* implements synchronization, ensuring that each VFS+ operation acts on files on which sufficient locks (*tokens*) are held to guarantee serializable results. The glue layer is transparent from the point of view of the programmer, in that it presents the same VFS+ interface to the layers above it as is presented to it by the file systems below. The generic system calls sit atop the VFS+ and glue layers, making the file systems available to local users of the file server node; but in addition, the *protocol exporter* makes the file systems available to clients. The diagram also shows a component labeled ''various servers;'' these components run on some file servers, and maintain various (global, replicated) databases for use by client cache managers.

The client structure of Figure 2 is carried over with few changes from AFS. The server structure of Figure 1, however, has changed. In an AFS server, the protocol exporter and underlying file system were not separate components. Thus the protocol exporter could not make use of other file systems, such as the native file system of the node. In turn, the node's own kernel could not make use of the server's file system. (Only if the node were a client as well as a server could local users access the server's file system.) The ability to export native physical file systems to client nodes is the principal difference in function between DEcorum and AFS.

### 2. Episode physical file system

Episode is a fast-restarting UNIX file system. It is intended to provide the capabilities needed for a large-scale distributed system with performance as good as the best existing physical file systems. There are several capabilities not generally available in vendor-supplied file systems. Most important are support for logical volumes, support for access control lists, and the ability to recover quickly from crashes.

We assume that Episode is based on a standard UNIX disk partition using the facilities of the kernel device driver. It supports both local and remote use: the former as a local file system when individual volumes are mounted; the latter as exported by the DEcorum protocol exporter. To provide a uniform level of access, it implements an upward compatible superset of the standard VFS functions.

The code is designed to take advantage of a multi-threaded environment and asynchronous I/O. Thus it would normally run in the UNIX kernel, although a non-kernel environment with those facilities would not be precluded. Running in the kernel also eliminates the system call overhead paid by mixed kernel and user implementations, as

[5] E. Walsh *et al.*, ''Overview of the Sun Network File System,'' *USENIX Conference Proceedings* (Dallas, Winter 1985).

[6] S.R. Kleiman, *op. cit.* The interface between generic system calls and specific file systems varies between different vendors' UNIX kernels; in this paper, for simplicity, we use the name VFS for all of them.

observed in the AFS file server.

Because Episode is intended to support high levels of concurrent access, it is designed with finely grained locking, and as few points of global contention as possible.

In the next sections we describe the capabilities of Episode that represent advances over older physical file systems: the volume/aggregate concept; access control lists; and logging. We also discuss some implementation issues, including the *anode* abstraction, which is comparable but not equivalent to the inode structure in other systems.

## 2.1. Volumes and Aggregates

In many UNIX environments, a file system (a mountable subtree of the directory hierarchy) is identified with a partition (a disk, or a quantity of disk storage, managed as a unit). In AFS, and in DEcorum, these two concepts are distinguished. To avoid conflict with standard UNIX terminology, a mountable subtree is called a *volume*, and a unit of disk storage is called an *aggregate*.

The volume/aggregate distinction is carried over to DEcorum from AFS.[7] We review it in this paper, however, because it is so important. Administration of networks of thousands of users is not practical without this distinction.

The distinction allows volumes to be *moved* among partitions, and even moved from one server to another. This capability in turn allows the system administrator to perform load balancing. Episode supports dynamic volume motion, without taking down any servers or making any facilities unavailable except the volume itself, and that transparently to application programs (which are blocked for a short time).

Equally importantly, volumes can be *cloned*, that is, a read-only copy (snapshot) of a volume may be made within an aggregate. Using volume cloning and moving, the system administrator can improve load balancing and availability of utilities. The administrative procedure of cloning a volume and moving the clone(s) is called *read-only replication* of the volume; we discuss replication further in Section 3.8. Episode supports cloning at the lowest level: a *copy-on-write duplicate* of a file can be created, in which, instead of data blocks and indirect blocks, there are pointers to the corresponding blocks of the original. The original file becomes the read-only version, while in the copy, any writing of data causes separate copies to be made of just as many blocks as required.

The unit that is backed up is the volume, not the aggregate. Thus the system administrator can back up a volume by cloning it, and later (at leisure) writing the clone to removable media. The volume is unavailable only transparently, for the time required by the cloning; and only one volume is unavailable at a time. Moreover, the clone can continue to exist on disk indefinitely; while it exists, files can be restored from it directly, bypassing the removable media.

In addition to the standard Vnode and VFS interfaces, Episode implements a volume interface and an aggregate interface. A VFS is a mounted volume, but the volume interface is separate from the VFS interface, because operations such as moving and cloning can be performed on volumes that are not mounted.

## 2.2. Buffer package and logging system

Crash recovery in the DEcorum file system is based on using log-based recovery techniques to either undo the operations that were begun but not finished, or to complete operations that finished, but had not been completely written to the disk.

The goal of using logging techniques for this purpose is to enable the DEcorum file system to restart operation as quickly as possible after a crash. Certain operations, if interrupted, would leave the file system in an inconsistent state, unsafe to use. Logging and the associated recovery techniques will allow the system either to complete or to completely undo such operations before resuming normal operation.

It should not be expected that logging would entirely eliminate the need for disk salvaging. Media failure will normally necessitate salvaging.

One might expect that logging would reduce overall file system performance due to the fact that certain information must be written both to the file system itself and to the log. Instead we expect the performance of the DEcorum physical file system to exceed that of the Berkeley fast file system (FFS). The FFS schedules large numbers of writes to file system meta-data as soon as the meta-data are modified. Examples of such meta-data include inodes, indirect

[7] John H. Howard *et al.*, ‘‘Scale and Performance in a Distributed File System,’’ *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pages 51-81.

blocks and directories. While these updates are generally performed asynchronously, they generate so much disk traffic that overall system performance is affected. In addition, a second update to a disk block already queued for one of these asynchronous writes will generally wait until the write operation completes, also hurting performance. The FFS performs all of these extra writes in order to ensure that certain information is written before other information, to simplify the job of *fsck*.

In comparison, a log-based file system need not force modified meta-data to the disk in order to guarantee file system consistency in the event of a crash. Rather, the file system will be consistent, if a bit old, after the log has been replayed. The file system may periodically batch-commit all pending transactions, but fidelity to the spirit of the UNIX file system only requires batching commits every 30 seconds, or when a *sync* or *fsync* system call is executed. In addition, these batch commits only require writing data sequentially to the end of the log; disks are especially efficient at performing these types of writes.

Thus, the DEcorum file system should actually generate considerably fewer disk updates, especially when performing operations that primarily change file system meta-data, such as file creation, deletion, and truncation.

Each aggregate has a log. This is an area of disk, not necessarily contiguous, whose size is fixed at aggregate initialization. The log is not necessarily located on the aggregate that it is logging.

Changes to meta-data are logged; changes to user data are not. Changes are grouped into *transactions*, defined in the usual way: each transaction must be atomic, that is, no individual change will take effect unless all the transaction's changes do. The log entry for a change gives the old and new values for all data bytes in the change, and the identity of the transaction of which the change is part. A separate log entry notes when a transaction *commits*, that is, when all the changes comprising it have been logged. The recovery procedure replays the log, completing transactions that had committed, and undoing transactions that did not commit. The time spent in recovery is proportional to the size of the active portion of the log, not (as with *fsck*) to the size of the file system.

Transactions may not span across calls to the VFS interface, but one such call may be comprised of more than one transaction. In particular, long operations are broken into sequences of short-lived transactions, each of which leaves the file system in a consistent state. For example, truncation of a file may be broken up to truncate only one block or a few blocks at a time. Breaking up operations enables us to guarantee that transactions will be short-lived, which in turn enables us to keep the log to a small, fixed size without requiring complex algorithms for log truncation.

Transactions must be made *serializable*. Suppose, for instance, that transaction A is ready to commit, after using data modified by transaction B, which is not yet ready to commit. It must be ensured that A will not commit unless B commits also. Our implementation of serialization is outside the scope of this paper.

The logging system is intricately entwined with the disk buffer cache. Higher-level file system functions must not modify buffer data directly, but instead go through logging primitives. Moreover, the same higher-level functions, having modified a buffer, do not distinguish between different degrees of synchrony in writing the buffer; they simply release it, leaving the writing to the logging system. With each buffer, the logger records the position of the most recent log entry for changes to the buffer's data; the buffer must not be written to disk until the log has been flushed to disk up to that position.

## 2.3. Access Control Lists

The idea that the mode bits associated with each file in UNIX do not allow enough flexibility in authorization strategies is not a new one. Several UNIX file systems, including AFS, supplement the mode bits with *access control lists* (*ACL*'s), association lists in which users or groups are paired with specific sets of rights to be added or denied in accessing a file. A complete description of the semantics of DEcorum ACL's is outside the scope of this paper. The principal difference between DEcorum and AFS ACL's is that in DEcorum any file or directory may have an ACL, while in AFS only a directory may have one.

## 2.4. Anodes

From a user programming point of view, perhaps the most important aspect of the abstraction offered by a file is its open-endedness: one can write to the file to any length, without worrying about where the storage for it is going to come from. In many UNIX file systems, this open-endedness is implemented at the level of the inode. Other aspects of the abstraction are implemented at the same level: authorization (mode bits and ACL's), status information (such as access and update times), and, not least, the directory hierarchy (and link counts). The ''bundling'' of all these abstractions is inconvenient to the kernel programmer, for it discourages him from taking advantage of open-endedness in situations where he does not want the other abstractions. He must either abandon open-endedness in favor of fixed limits, or re-implement some form of open-ended resource usage himself. An example of the former strategy is the fixed size limit on ACL's in AFS; readers familiar with other UNIX file systems can no doubt think of other examples of both strategies.

The DEcorum *anode* abstraction provides an open-ended address space of disk storage *and nothing more*. (To use more exact terminology, the actual disk storage is called a *container*, and the anode is the small set of bytes that serves as a descriptor for it, either on disk or in memory.) Anything that uses storage on disk is implemented as an anode: files, ACL's, volumes, aggregates, transaction logs, disk allocation bitmaps, and so on. Files are implemented as anodes with additional bells and whistles: a set of status bytes, a pointer to an ACL, and a position in the directory hierarchy.

Because all data and meta-data are stored in anodes, the disk presents a uniform interface to utilities that access it, even if they access it below the file system level. For instance, the logging system and the salvager are somewhat simpler than they would be if they had to distinguish between anode and ''other'' disk areas.

## 3. DEcorum protocol exporter and related file system servers

We describe the individual components of a DEcorum protocol exporter. We also discuss some of the global database servers that play roles in the DEcorum file system, although these are not components of file servers. Some of this structure is carried over unchanged from AFS, and so we describe it only briefly.

### 3.1. Token manager

The token manager maintains, for each file in use, a set of guarantees that have been made to various clients. These guarantees have a binary *compatibility* relation telling whether two guarantees can be simultaneously granted to two separate clients.

### 3.2. Host model

The host model maintains structures describing authenticated individuals that have made RPC's to it, and the client managers from which the RPC's originated. Information about the state of the client includes, for instance, whether all of the token revocation messages issued to it have been delivered. Information about an individual might include the Kerberos[8] (or other authentication system) identity of the caller.

### 3.3. Vnode glue layer and VFS+ interface

The purpose of the glue layer is to synchronize actions on files. For each Vnode operation provided by a conventional file system, a corresponding ''wrapper'' operation is substituted that obtains tokens and then performs the original operation. This layer must avoid deadlock; in this sense it faces the same problem, in perhaps aggravated form, that physical file systems face. For some operations, directory lookups must be performed to complete the set of Vnodes for which tokens must be obtained before the physical file system operation can be called.

In addition to the usual VFS and Vnode operations, the protocol exporter allows for additional operations to provide access to such extensions as volumes and access control lists. The Episode physical file system, of course, implements all these operations. For any other physical file system, it may be possible to provide some subset of DEcorum functionality by writing the corresponding operations. For instance, some volume operations could be implemented in a conventional UNIX kernel, assuming a mapping between volumes and file systems and between

[8] J.G. Steiner, C. Neuman, and J.I. Schiller, ''Kerberos: An Authentication Service for Open Network Systems,'' *USENIX Conference Proceedings* (Dallas, Winter 1988).

aggregates and partitions, even though no more than one file system can occupy a partition.

### 3.4. Volume registry (and volume location database)

The volume registry is a simple symbol table enumerating the set of volumes residing locally on the server. It can be used within the server, whereas the *volume location database*, a global replicated database describing which volumes are on which servers, provides service to remote clients.

### 3.5. Server procedures

The server procedures implement the RPC interface in terms of calls to the previous components.

### 3.6. Volume server

The volume server implements per-volume operations, such as moving a volume from one file server to another, and makes them available to administrators at remote clients.

### 3.7. Authentication server

All RPC's are authenticated. The DEcorum authentication service is based on Kerberos. A description of it is outside the scope of this paper.

### 3.8. Replication server

In AFS, the benefits of read-only replication of a volume, as described in Section 2.1, become available only when a system administrator or operator explicitly requests creation of a replica. The DEcorum replication service implements *lazy replication* of volumes: a replica is maintained permanently, and is guaranteed to be out of date by no more than a fixed amount of time. In principle, as the amount of time approaches zero, lazy replication approaches instant (i.e. read-write) replication; but in practice, the design of the lazy replication system is not expected to perform well under those circumstances, e.g. when the amount of time is less than about 10 minutes. The client of the replica is guaranteed to always see a consistent snapshot of the volume, and is guaranteed that data in the replica are never replaced by older data. A replication server requests a *whole-volume token* to guarantee that it can use a replica of a volume; when it must update the replica, it attempts to obtain from the master copy only those files that have changed.

## 4. DEcorum client

We describe the four layers into which a DEcorum client can be divided. As with the DEcorum protocol exporter, much of the structure is carried over with little change from AFS, and so is described here only briefly.

### 4.1. Resource layer

The lowest layer is the resource layer. It maintains RPC connections and caches volume location information. When upper-level modules desire an RPC connection authenticated as a particular user, it is this module that they call.

### 4.2. Cache layer

The cache layer caches status and data information from files stored at remote file servers. It also checks that cached information is up to date. In AFS clients, vnode status information is cached in memory, while file data are cached in disk files provided by the ‘‘native’’ (generally vendor-supplied) physical file system. This structure is carried over to DEcorum, with the exception that an in-memory version of the data cache is provided as an option, enabling diskless clients to be used. The cache layer uses tokens to ensure consistency, as described in detail in Section 5.

### 4.3. Directory layer

The client has its own directory module. This enables it to perform lookups without contacting the server, assuming that it has up-to-date directory data to use. In AFS, the client copied a directory from the server on the first reference; subsequent directory operations could be performed on the client using exactly the same code used in the server. In DEcorum, this technique is not always available, since in general the client will not understand the directory format used in every server file system. Instead of caching whole directories, the client must in general cache the results of individual lookups.

### 4.4. Vnode module

The highest layer, the *vnode layer*, implements the Vnode and VFS interfaces required for the client's kernel, in terms of calls to the three lower layers described above.

## Issues

## 5. Caching and tokens

### 5.1. Tokens

The synchronization challenge in the DEcorum file system is to ensure that all users of a physical file system exported by the DEcorum protocol exporter receive single-system UNIX file access semantics, no matter whether they reference the data remotely from a client or locally via a local mounted file system. Remote clients must read and write file status and data, but ought not to incur the expense of making an RPC to the file's home machine in order to call the appropriate Vnode operation directly for every remote reference. What is required is a mechanism for allowing operations to be performed on the client, while simultaneously blocking conflicting operations at other sites. Conflicting operations, in this sense, are those that, if performed, would cause incorrect results to be obtained by other clients or local users.

The architecture chosen to solve this problem is simple. Each server includes a *token manager*, which keeps track of who is referencing files, what they are doing to the files, and what guarantees they require about what others may do to the files. For example, a protocol exporter may allow a client to read the contents of a file (from the client's own cache) until otherwise notified. The protocol exporter records that the client has received a guarantee, and it will not allow anyone to write data to the file without first revoking that guarantee (i.e. notifying the client that its cached data must no longer be used).

The token manager is invoked by *all* calls through the Vnode interface, and ensures that any guarantees incompatible with the operation being attempted are first invalidated. The token manager is invoked by the Vnode layer because non-DEcorum protocol exporters (such as NFS), as well as locally executed system calls, may perform various operations on the resident physical file systems, and these operations must be synchronized with the guarantees exported by the DEcorum protocol exporter.

We use the term *client* in this section to denote any entity, remote or local, that requests tokens on the files it desires to work with. This terminology conflicts with our usual definition of *client* as a remote user of files exported by a file server, but it is the conventional way of describing the relation of module users to a module. There are many potential clients of a token manager, including local UNIX kernels and remote file system protocol exporters. A client of a token manager registers itself with the token manager in a fairly general manner: it passes in an object of type *afs_host*, having a virtual *revoke* procedure. The *revoke* procedure is called whenever the token manager needs to revoke the token.

### 5.2. Token types

Some tokens are incompatible with other tokens. Before granting a new token, the token manager may have to revoke some already-granted tokens. The tokens currently defined provide the following guarantees:

- **Data tokens** grant the holder the right to read or write (depending on the subtype of the token) a range of bytes in a file. A read data token allows the possessor to cache and use a copy of the relevant file data without repeatedly performing RPC's to the appropriate file server, either for validating the data or for re-reading it. A write data token allows the holder to update the data in a cached copy of the file without storing the data back to the server or even notifying the server.

- **Status tokens** allow the holder to read or write (depending on the subtype of the token) the status information associated with a file. A read status token allows the holder to refer to cached copies of the status information without calling the server to check the status. A write status token allows the holder to update its cached copy of the file's status without notifying the server. The token manager blocks other VNODE functions from even looking at the file's status information before revoking any write status tokens.
- **Lock tokens** allow the holder to set read or write file locks (depending on the subtype of the token) on a particular range of bytes within a file. Without holding a lock token, a client must call the server to set a file lock; with such a token, the client is assured that the server will not attempt to set conflicting locks on the file, without first revoking the token.
- **Open tokens** grant the holder the right to open a file. There are different subtypes for different open modes: normal reading, normal writing, executing, shared reading, and exclusive writing.

Tokens of any type are compatible with tokens of any other type, as they refer to separate components of files. Tokens of the same type may be incompatible with each other:

- Read and write data tokens are incompatible if their byte ranges overlap.
- Read and write status tokens are incompatible.
- Read and write lock tokens are incompatible if their byte ranges overlap.
- The compatibility matrix of open tokens is given in Figure 3.

Tokens are managed in the DEcorum file system by modifying all the Vnode functions to first call the token manager to obtain the appropriate tokens for whatever operations they will perform, then to perform the operations, and finally to call the token manager to notify it that the tokens can now be revoked at any convenient time. The modified Vnode functions constitute the *glue layer***.**

## 5.3. Token revocation

When the token manager wishes to revoke a token, it notifies the client to which the token was granted. If the token is for reading (status or data), it must simply be returned. If the token is for writing, the client must write back any status or data that it has modified, before returning the token. If the token is for locking or opening, the client may elect not to return the token at all; this is the normal action if the client has already locked or opened the file.

For remote clients, revocation notifications are sent by RPC's. Thus RPC communication between DEcorum clients and DEcorum servers is two-way: clients call servers to access files, and servers call clients to revoke tokens. This structure was not fully depicted in figures 1 and 2, in order to simplify those diagrams.

## 5.4. Comparison with other distributed file systems

Tokens implement the strongest possible consistency guarantee for users of a shared file: when one user modifies a file, other users see the modifications as soon as the *write* system call is complete. At the same time, communication between client and server is kept to the minimum, that is, shared data are transmitted only when they are actually shared. In this section we compare tokens with the spectrum of distributed file system semantic models and implementations described by Kazar.[9]

Relatively weak cache consistency guarantees are provided by the Sun Network File System (NFS). A page of cached file data is assumed to be valid for 3 seconds; if it is directory data, it is assumed to be valid for 30 seconds. The use of fixed time limits hampers the writer of a distributed application; the application must wait the specified length of time before fetching shared data, if it relies on the correctness of the data. This disadvantage of weak consistency guarantees is not accompanied by a corresponding advantage, such as low network utilization; clients must communicate with servers every 3 seconds whether or not any shared data have been modified.

AFS provides consistency guarantees at an intermediate level of strength. AFS ‘‘callbacks’’ are roughly equivalent to DEcorum status read tokens, in that the operations for which the AFS client would get a callback are those for which the DEcorum client would request a status read token. But because callbacks are the only synchronization

[9] Michael Leon Kazar, ‘‘Synchronization and Caching Issues in the Andrew File System,’’ *USENIX Conference Proceedings* (Dallas, Winter 1988), pages 31-43.

mechanism, they are overburdened. There are not separate callbacks for reading and writing, nor for status and data. Thus the AFS client cannot know when to store back to the server any data that it has modified in the cache.

In the absence of certain knowledge, the AFS client could preserve single-system semantics only by communicating with the server at every *write* system call. (The server could then break callbacks to other clients reading the data.) Instead, it stores data back to the server when the file is closed.

This system has, in perhaps diminished form, the same drawbacks as that used by NFS. The consistency guarantees are not strong enough for some distributed applications, yet communication with the server is not minimized (i.e. there is communication at every *close* system call). The use of typed tokens allows DEcorum to synchronize access to files accurately but with minimal communication.

Two further limitations of the AFS callback system should be noted:

- Callbacks cannot describe byte ranges of data. If a group of users are accessing (and modifying) the same large file, even though they may be using disjoint parts of it, the file will frequently be shipped back and forth in its entirety between nodes.
- Callbacks cannot describe open modes. Any number of AFS users may have a file open in any mode at a time. For clients implementing some versions of the UNIX file system, this is not a pressing problem, inasmuch as UNIX allows multiple users to open the same file for reading or writing simultaneously. But the exotic open modes enable clients to implement the semantics of non-standard or even non-UNIX file systems. In addition, the UNIX restriction against opening a file for writing if it has been opened for execution can be implemented; and a virtual file system can assure itself that a file about to be deleted has no remote users, by requesting an open token for exclusive writing on the file.

## 5.5. Example

We describe a short synchronization example, showing the roles of the token manager and protocol exporters in synchronizing access to a file. Consider a file stored in a BSD UNIX physical file system, being written to by both a local user, who is issuing both read and write system calls, and a remote user, who is doing the same thing to the file via a client cache manager.

We begin with the remote application issuing a *write* system call to the file. This operation is handled by the client's cache manager, which requires a guarantee that it is permitted to locally update the file. This guarantee is requested from the protocol exporter of the server node on which the master copy of the file resides. The protocol exporter registers the client with the server node's token manager as having a data write token. Once the client receives the data write token, it can handle all remote writes to the file in question without contacting the master copy.

Assume that at some point a process on the server, accessing the file locally (not through a DEcorum client), decides to write some data to the master copy of the file. The process calls VOP_RDWR (the Vnode layer call for performing reads and writes) on the local Vnode, and the Vnode glue code first calls the local token manager, requesting a write data token for the file. Since there is a conflicting write data token granted to the remote client by the DEcorum protocol exporter, this incompatible token must be revoked before the local process can be granted its own token. The token manager does this: the protocol exporter is invoked again, and as part of its revocation procedure, it makes an RPC back to the client cache manager, asking it to return its token. As a side-effect of returning the token, the client will also store back the modified data pages. Once the remote client has stopped using and has returned its guarantee, the protocol exporter can return from the revocation call made by the token manager. Once the protocol exporter has returned its incompatible write token, the new write data token can be granted to the Vnode glue code, which will then perform the actual data write operation (by calling the original virtual file system's VOP_RDWR function).

The Vnode glue code need not hold onto its write data token for very long; it can return the token any time after the VOP_RDWR call has completed execution. By contrast, remote clients hold onto tokens as long as they can, to avoid unnecessary RPC's.

Should the remote user issue another call to VOP_RDWR, the client will again have to contact the protocol exporter to get another guarantee, and the protocol exporter will have to call the local token manager again to obtain another data write token.

## 6. Deadlock analysis

A significant challenge in building a system as complex as the DEcorum file system is ensuring that the overall system is deadlock-free. There are two general ways of dealing with deadlock in a distributed system: deadlock *avoidance* by partial ordering of locks, and deadlock *detection* by detecting cycles in the *waits-for* relation. We have chosen deadlock avoidance in our design. We implement this by establishing a partial order of locked resources.

The next section describes the structure of the basic calls from the clients to the server and the revocation calls from the server back to the clients. We pay particular attention to the position in the locking hierarchy of any resources that are locked. It is important that operations are serialized at the server, and that it is clients who must reconstruct the server's serialization, and not the other way around.

The following section describes how a correctly working cache manager can be built, given the constraints of locking objects in the order required by the locking hierarchy.

### 6.1. Basic Structure

The trickiest subsystem to deal with in these terms is the file server / cache manager subsystem, since the client makes calls to the file server to perform operations, which may make calls back to other clients. These return calls are the stuff of which dependency cycles are made.

The resources of interest are the vnodes on the client cache manager and the vnodes on the file server. A vnode and the token state associated with it may in some cases be components of the same locked resource; we will ignore token state in the description below.

Consider the following straightforward design: when a client cache manager makes a call to a file server, it locks its own cache vnode, and then makes the RPC, causing the server to lock its vnode and make calls back to other clients, which will lock the vnodes corresponding to the same file at their own sites only for the duration of the return call. The file server will then complete its call, and release its locks (and thread), and finally the client that initiated the call will release its locks.

This approach fails because of the difficulty in ordering the client's vnodes with respect to the file server's vnodes. Suppose that both clients A and B decide to call server S to perform some operation on some file $V$. Client A first locks its local vnode for $V$, $V_a$, while simultaneously, client B locks its vnode, $V_b$. Both A and B then make calls to S, and one of them, say A, locks $V_s$ first. If the processing of this request requires the revocation of a token from the other client (B in this example), then the request from A will require waiting for B to release the lock on $V_b$, while B is waiting for the lock on $V_s$, which results in a deadlock.

The general solution to this problem we have chosen in the DEcorum file system design is to provide two locks per client vnode. One lock serializes the high-level operations performed by the client, so that two separate threads on the same client do not attempt to fetch the same data pages for the same file simultaneously. A lower-level lock serializes operations on the client vnodes that may leave the actual vnode in an inconsistent state, such as having a file length inconsistent with the number of bytes written to the file. The higher level lock is locked by high level operations initiated by the client cache manager, and is not unlocked until the operation completes. The lower-level lock is generally held from the time a client operation begins to deal with a vnode until the time that the client makes an RPC to the server, but is released just before the server is called. As soon as the response is received, the client re-obtains the lower-level lock (never having relinquished the higher-level lock), and processes any revocation operations that executed concurrently with the RPC.

Thus, all operations lock resources in this order: first, the client initiating the operation locks the high-level lock on its local vnode. Next the client may make an RPC to a server, which will lock the vnode present at the server. The server, while holding its vnode lock on the physical file system's vnode, may make token revocation calls to other clients, which will lock the low-level locks on their respective vnodes. Thus one always locks high-level vnode locks first, then server vnodes, and then low-level vnode locks.

### 6.2. Correctness

To be convinced this locking architecture is viable, one must be convinced that the client and server can serialize operations in the same order, even though the server is really the process that determines the serialization order, by its locking of the server vnode $V_s$. One must be convinced that the cache manager's vnode state is consistent with the server's vnode state after the execution of any of these RPCs, no matter which concurrent revocation service messages were processed during the client-to-server RPC.

To aid in performing this serialization-after-the-fact, the file server marks every reference to a file with a time stamp, whether the reference is for a token revocation call initiated by the server, a token-returning call initiated by a client, or a call that simply temporarily locks a file but does not return any synchronization tokens to the client, but may return other information, such as the current file status. If operation $O_x$ on a file is serialized at the file server before operation $O_y$ then the per-file time stamp returned by $O_x$ for that file will be less than the the time stamp for that file returned by $O_y$. Because the time stamp is per-file, operations such as *rename*, which affect multiple files, will return multiple time stamps, one for each file of interest.

Time stamps must appear in return parameters from calls that read or write status information at the file servers, as well as in token revocation calls initiated by the file server. The time stamp is used to serialize client operations after obtaining the lower-level lock after an RPC call returns.

### 6.3. Examples

Consider the concurrent execution of a call $M_s$ to retrieve the status of a file *F*, which grants data read and status read tokens for *F*, with the execution of a token revocation message $M_{tr}$ pertaining to *F*.

$M_{tr}$ contains a time stamp indicating its serialization order with respect to *F*. It also specifies the token *T* to revoke by means of a unique token ID. Assuming that $M_s$ is the call that returned *T*, the problem is that $M_{tr}$ and the reply to $M_s$ may be processed out of order, that is, when the client processes $M_{tr}$, it may not yet recognize *T*. Thus, if $M_{tr}$ specifies a token that is not recorded as part of *F*'s token state, and vnode state indicates that a call that may return a token is in progress, the client also queues the token revocation for later processing. When the in-progress RPC completes, the client obtains the low-level lock, and then serializes the token returning operation implicit in the reply with the queued token revocation operation. This ordering is done strictly on the basis of the per-file serialization counter.

A more complex example occurs when one page is being stored back by a client, while another page's data write token is being revoked by a call from the file server. In this case, part of the execution of the data token revocation call involves a call back to the file server to store the appropriate data page. The response to this call includes the updated file's status, which must be inserted into the cached vnode when the call completes.

In essence, two concurrent store data calls are being made to the file server. One is a normal store data call, while the other is a special call issued only by token revocation code. Both *store* calls send data back to the server and receive as a response the updated file status information. Because the low-level lock is never held over normal client-initiated RPCs, the two responses may be processed in any order. Again, the file serialization counter is used to correctly serialize the responses. As soon as one response is received, the client obtains the low-level lock, and copies the updated status information back into the vnode, but only if the updated status information is labelled with a file serialization counter larger than that already labelling the vnode. After the data are copied, the vnode is labelled with the file serialization counter from the just-merged status. If the counter associated with the vnode is greater than the counter associated with the returned status information, then the returned status information is older and can be simply ignored.

### 6.4. Other considerations

After any system call completes, calls made via the cache manager will reflect the state of the file at the time the call completed, or some later time. Because old file status information is never written back over a file's vnode, once the up-to-date information is present in the cache manager's vnode, it can not be replaced with older data.

Note that when the cache manager's token revocation procedure calls back to the file server, the cache manager must ensure that some dedicated server threads are available to handle these requests. If only one pool of threads were available for all incoming requests, then it would be possible for all of the server threads to be busy when a token revocation procedure has to call back to the server, resulting in a deadlock.

In general, calls made to the file server from executing client revocation code could be eliminated in place of somewhat complex return parameters from the token revocation procedures. However, this has some aesthetic difficulties, as well as one fundamental difficulty: it is difficult for a server thread to ensure that the last few packets of its reply have been received properly by its caller; this check doesn't quite fit into the RPC paradigm. It turns out to be considerably simpler for a token revocation return call to simply call back to the file server to accomplish such things as returning unused file locks (when revoking lock tokens) or storing back modified data (when revoking write data tokens).

Deadlock avoidance for the remaining subsystems (e.g. the volume server, the volume location server) is quite simple, since in general a client will lock its own resources in some standard order, call a server, which will lock its own resources in its own standard order, and then execute without making any return calls to the client. Thus there are no tricky cycles between the clients and the servers of a particular service.

## Summary

To put the DEcorum design in perspective, we observe that the concept of a network file system is not new. Though some obvious applications of such file systems are for LAN's, AFS was specifically designed for networks of thousands of users, and we are still learning what new problems are introduced by the increase of scale. The DEcorum design focuses on three of those problems, selected for their practical and theoretical importance: interoperability; cache consistency; and server availability. It also provides very substantial POSIX compliance and does so in a highly modular fashion. DEcorum borrows heavily from the well-understood design of AFS. However, DEcorum has the potential to grow into much larger environments: users may attempt to create network file systems of national or international size and scope. We may look forward to these challenges with confidence in the robustness and flexibility of our present designs.

## Trademarks

## Acknowledgments

We are grateful to Alfred Spector, Liz Hines, and Julie Kownacki for comments and corrections. We are deeply indebted to the staff of Transarc Corporation for support and encouragement. Ideas from many people are reflected in the ultimate file system design. Some of these people are from the customers and co-developers of AFS, including Carnegie Mellon, MIT, the University of Michigan, and IBM Rochester. Other individuals include (but are not limited to) the following: Bruce Walker, Charlie Sauer, Paul Leach, Mike West, Todd Smith, Don Johnson, Joe Pato, Gary Owens, and Bill Sommerfeld.