CMU-ITC-86-053
Published in:

EUUG Conference Proceedings
Manchester, UK
Autumn 1986

Volumes: The Andrew File System Data Structuring Primitive

Bob Sidebotham

The Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA. 15213 (USA)
bob@andrew.cmu.edu.ARPA

ABSTRACT

Volumes are the basic organizing mechanism for data in the Andrew file system, a large distributed system designed to support thousands of workstations. Volumes provide the basis for addressing, storing, and accessing the information within the system. They are independent of the physical configuration of the system, and may be freely moved about as the needs of the system dictate without disrupting the ongoing work of users. Various other features of volumes combine to make them a powerful mechanism supporting the administration of this file system.

Introduction

The Information Technology Center was established in cooperation with the IBM corporation to enhance the level of computing technology at Carnegie-Mellon University[1]. One result of this collaboration is the Andrew file system, otherwise known as Vice[2]. In Vice, dedicated servers cooperate to provide user workstations and other computers access to a huge common file system.

There have been two implementations of this file system, Vice1 and Vice2. The current implementation, Vice2, was designed and rebuilt from scratch in response to experience with our first system. A major feature of the new implementation is the concept of a *volume*, a collection of files forming a partial subtree of the file system hierarchy. This paper will discuss the rationale for adopting volumes, the architectural issues, and details of the implementation.

The Redesign Effort

Vice1 and Vice2 are architecturally very closely related. Both systems distinguish between clients and servers; client workstations are *not* file servers. The servers each store some subset of the file system space, with possible overlap between the servers for availability and performance reasons. Whole file transfer is used in both systems to transfer files to and from workstations, and the workstations are responsible for caching entire files on their disks. The systems attempt to emulate Unix semantics at the workstation by intercepting file system calls (such as open and close) and forwarding them to a local process. This process, in turn, is responsible for fetching files, or information about files, from the appropriate servers, storing them back as necessary, and maintaining the local cache. Read and write system calls are *not* trapped--they go directly to the local, cached file.

A critical difference between Vice1 and Vice2 is in the low-level naming of files. In Vice1, the entire pathname of a file was presented to the server on every request, and the files were cached by the workstations using the pathname as the key. This had a number of consequences, the most significant of which was that directory rename was, for practical purposes, impossible to implement: the rename of a directory would invalidate all the keys for entries below that directory. In Vice2 therefore, we decided to use low-level

file identifiers, *fids*, in the interface between the file servers and clients. This had a number of immediate, useful consequences: a general rename became possible, in fact easy, to implement; symbolic links became easy to implement (since the pathname resolution now had to be done at the workstation); the server load was reduced (for the same reason); and cache management, in general, became easier because of the fixed length nature of fids, as opposed to pathnames.

Here is a description of the calls in the Vice2 client/server interface which directly relate to file storage. Ancillary parameters, such as access keys, are left out of this description. A *dfid* is a fid which happens to refer to a directory:

fetch(fid)
: Returns the file status and, optionally, the data for *fid*.

store(fid, file, status)
: Replaces the existing file associated with *fid* with the new file and status.

create(dfid, name, status)
: Creates an entry *name* in directory *dfid* with initial status *status*.

remove(dfid, name)
: Removes the entry *name* from directory *dfid*, releasing the underlying file from storage if this is the last referencing entry.

symlink(dfid, name, linkcontents)
: Creates a symbolic link *name* in directory *dfid*.

makedir(dfid, dname, status)
: Make a new, empty directory *dname* in directory *dfid* with initial status *status*.

removedir(dfid, dname)
: Remove the directory *name* from directory *dfid*.

rename(olddfid, oldname, newdfid, newname)
: Rename the file, directory or symbolic link in directory *olddfid* called *oldname* to *newname* in directory *newdfid*.

Each of these calls is directed to a single server. At this point in our design discussions we had not yet understood how the workstation would decide which server to contact with any given request. We had to devise an efficient mechanism to allow a workstation to discover which server was responsible for any given fid.

At the same time that we were redesigning the interface between the client and server, we were also acutely aware of the need for easy administration of the file system. We had to a large extent ignored this problem in Vice1, and now was the time to do something about it.

These two forces together resulted in the invention of *volumes*.

Volumes

We decided to divide the fid into two parts, a volume number and a file number within the volume. All of the files in a volume are located at the same server. The workstation discovers the location of a volume by consulting the *volume location data base*, which it does by making a *GetVolumeInfo* call to a randomly selected server. The volume location data base is replicated to all of the file servers.

An immediate consequence of this decision is that the rename function, which is directed to a single server, is applicable only *within* a volume. Ordinary Unix has a similar restriction (files cannot be renamed across file systems), so we decided that we could live with this decision. This fact, however, dictates something about the nature of the files in the volume: they need to be closely related, in order for rename to be useful, and there should be enough of them so that the restriction on rename is not perceived by users as a problem.

Each volume is, in fact, a piece of the entire file system hierarchy; in this respect it is like a Unix "file system". Like Unix file systems, the volumes are glued together at *mount points* to form the entire system hierarchy. Unlike Unix file systems, however, we did not want volumes to conform one-to-one with disk partitions. We decided that volumes should normally be considerably *smaller* than disk partitions, so that many of them can reside on a single partition, sharing the available disk storage. By using this approach, the problem of disk space allocation is minimized: if a partition becomes full, the system administrator merely has to move one or more volumes to another partition on the same server or on another server. This operation can be made with little or no disr-

uption to users of the system, as will be discussed later in this paper. For now, however, note that the volume location data base gives the current location of any volume in the system--the physical location of the volume is not related to its place in the file system hierarchy.

Given the desire to make *rename* a usable operation, and the desire to maximize flexibility by keeping the sizes of volumes relatively small, we decided that a typical volume would contain the files belonging to a single user. As of this writing, CMU has about 1200 users and a little more than 1200 volumes, one for each of those users, plus a small number of system volumes.

A number of other applications of volumes were discovered during the course of the design. In the present system, they are used as the unit of quota enforcement (each user has a quota assigned to his or her volume), and as the unit of accounting. An automated backup system backs-up each volume every day; when a user needs files restored, the *entire* volume is restored and the user picks and chooses files from this copy of his or her old volume. Volumes are also *cloned* every day; the backups are actually made from the cloned volume and the user also has access through the cloned volume to his or her previous day's work. Finally, we decided that replication of frequently used system files could be accomplished simply, by freezing the state of selected *read/write volumes* to produce *read-only volumes*. Identical copies of the read-only volumes could be made available at multiple sites.

Client-Initiated Volume Operations

There are two types of operations that can be performed on volumes: client-initiated operations which do something *within* a volume, and administrative operations which operate on the volume as a whole. The client-initiated operations, *fetch, store, create, symlink, link, remove, makedir, removedir* and *rename,* are defined by the client/server interface. Note that none of these operations *ever* cause the file server to traverse a file system pathname: the directory operations always specify the directory involved and the name of the component within the directory. This considerably simplifies the file server and, of course, reduces overhead. Two additional operations have been provided: *getvolinfo* and *setvolinfo* allow the parameters associated with a volume (disk quota, etc.) to be queried or changed from a workstation. The *setvolinfo* call is, of course, restricted to system administrators.

Administrative Operations

Administrative operations are not part of the client/server interface definition; instead they are initiated by the servers, by a *server control machine*, or by a *staging machine*. The servers initiate nightly volume cloning operations; the server control machine coordinates most other operations, such as moving volumes; and the staging machines control the automated volume backup and retrieval mechanisms. The administrative operations include *create volume, purge volume, make backup volume, dump volume, restore volume, move volume,* and *release volume*. *Create* and *purge volume* do the obvious things, *create* ensures that the volume number assigned is unique. *Make backup volume* is invoked nightly by the file servers themselves to create new read-only backup volumes for each user. Copies of these volumes suitable for storage on tape are created by *dump volume* at the request of the staging machines. Volume restore requests are honored by the staging machines and delivered to the file servers using the *restore volume* function. *Move volume* does the obvious thing, and *release volume* is used to create a read-only copy of a volume and propagate further copies to designated replication sites. All of these administrative operations can be performed without disrupting usage of the system.

Other functions can be implemented as combinations of these operations. Functions that are currently planned include: a load balancing operation to distribute volumes over the file servers in an optimal way based on the sizes of the volumes and traffic patterns; an operation to distribute all of the volumes on a partition to other sites in the system, to be used when a disk is suspect, or is due for reformatting; and an operation to automatically restore any missing volumes in the system, to recover, for example, from the total failure of a disk.

One unexpected side effect of the administration of the file servers at the granularity of a volume has been that the privacy of the owners of the information is safeguarded. For example, to honour a restore-file request the entire volume is restored. No one but the original owner of the files needs to have direct access to the files within the volume, and the operations staff does not need to be told precisely which files to restore, as is the case in many systems. In fact, no administrative operations are performed on individual files. We avoid scanning directories to remove "core" files and the like, and instead use our ability to set quotas and use the available resources efficiently (by locating volumes appropriately) to control disk usage. Finally, when a user's account expires, his or her volume is purged using the *volume purge* operation; again, no directories need to be scanned.

Volume Representation

There are many representations of volumes that we could have chosen to implement the functionality described above. The one we chose performs adequately and has the advantage that it can be implemented easily on a Unix system. A volume is represented as a collection of Unix inodes, which are found by starting at a single file in the root directory of a Unix partition. This file lists three inodes: a header inode, containing information about the volume (name, number, quota, etc.), and the inode numbers of two index files, one for directories, and one for files and symbolic links. The index files contain fixed length entries for each directory or file or link. Directories are in a separate index simply because the entries are bigger--each entry contains an access list for that directory (files use the access list of the containing directory).

Each entry in an index contains status information for the corresponding fid, or it is empty. A fid is mapped to an entry in the index by taking the file number portion of the fid and splitting it into two parts: a *vnode number* (*vnode* meant, historically, *Vice inode*), and a *vnode uniquifier*. The vnode number, if odd, is a direct index into the *directory vnode index*, and otherwise into the *file vnode index*. Together these two indices will be referred to as the *vnode index*. The vnode uniquifier is incremented every time a new file is created; the vnode number, on the other hand, can be reused. The uniquifier guarantees that there will be no confusion between the server and client as to which file is actually being referred to.

The status information for a fid, obtained from the appropriate vnode index, contains the *type* of the vnode, directory, file or symbolic link, the *mode*, corresponding directly to the Unix mode bits, a *cloned flag*, for directories in volumes that have been cloned, used internally by the file server, the *link count* of the file, which is the number of directory references to it, the file *length*, in bytes, the *uniquifier*, the *data version number*, incremented whenever the file is stored and use by clients for cache validation, the *inode number* where the directory, file, or symbolic link is actually stored, the *modification time of the file*, as set by the client workstation, the *server modification time* of the file, the *author* and *owner* of the file, the user who last stored the file and the user who first created the file, respectively, a *lock* structure, for implementation of the flock system call, and, finally, the *parent directory vnode number* for the file, used internally by the file server. Directory vnodes also contain an access control list, used to control access to the directory and files within the directory.

The inode referenced by a vnode status block can contain a file, directory, or symbolic link. The inode is used to store a file as received from the workstation, to store a directory in a hashed form, suitable for rapid lookups by the file server, or to store the text of a symbolic link. In all cases the type of the inode is actually a regular Unix inode, *not* a directory or symbolic link. In order to support the *clone* function, described below, each inode may be referenced from multiple volumes. The link count of the inode reflects the number of volumes sharing the inode.

In this representation of a volume, only a single file is present in the server's Unix name space; all the other files within the volume are located simply by their inode numbers. This reflects a conscious decision to make use of the lower-level functionality of the Unix file system, while avoiding the costly pathname searches implicit in all current Unix implementations. This required a minor modification to the kernel to support

direct access to inodes, and has turned out to be well worth the trivial amount of work it required. The functions supported are: *icreate*, allocate an inode, *iinc* and *idec*, increment or decrement the inode link count, deallocating the inode if the link count goes to zero, *iopen*, get a descriptor for the inode, *iread*, read directly from an inode at a specified offset, and *iwrite*, write directly to an inode at a specified offset. The *icreate* call also allows the inode to be stamped with four parameters (in unused space in the inode), which are used for salvaging purposes. The first parameter is always the volume number and is used as a check on the other calls to prevent them from accidentally damaging inodes. Inodes that have not been allocated by *icreate* are not normally accessible with these system calls.

Support for the Client/Server Interface

Given this volume representation, it is easy to see how the various calls in the client/server interface are implemented within the server. The most common operations turn out to be *fetch* (status only), *fetch* (with data), and *store*. All of these operations are performed in a straightforward, efficient manner by the server, with no directory look-ups or modifications required. *Store* is careful to allocate a new inode for the incoming file and decrement the link count on the old one after the operation. This has two useful results: first, if a store is interrupted by a crash and ensuing salvage operation, the original file will remain intact, and second, if the inode was shared by another, read-only volume, then that volume will still retain a valid reference to the original inode after the store. The read-only semantics of that volume are therefore preserved at essentially no cost.

The remaining operations all specify one or two directories which are examined or modified appropriately. Unlike file updates, directory updates are done in place; this means that a copy of the directory has to be made if the volume has been cloned since the last update to the directory. The *cloned flag* in the directory's vnode is used to determine if this should be done. The rename operation is slightly complex: care has to be taken to ensure that a directory is not being renamed into a subdirectory of itself, which would disengage an entire subtree from the volume's hierarchy. To support this, vnodes are chained backwards using the *parent directory vnode number* field. Without having to actually read the contents of any directories, the server can quickly scan backwards from the target directory vnode to determine whether the directory being moved is above it in the hierarchy.

Implementation of Administrative Operations

The administrative operations described above are supported by combinations of five basic functions: *create, purge, clone, dump* and *restore*. *Create* and *purge* do rather obvious things, setting up or tearing down a volume appropriately. The *clone* operation makes an identical, read-only copy of a volume by copying the header and vnode index and incrementing the count on all of the inodes referenced by the index. This is a relatively inexpensive way of getting a frozen copy of a volume without having to copy any of the data. The *dump* operation converts a volume into a canonical, machine independent byte stream, and the *restore* operation reverses the operation to produce a volume. Dumps can also be *incremental*, in which case only fids which have changed since a specified date are really dumped (although directories are always dumped when making volume dumps for backup purposes, for robustness).

The administrative operations of *create volume, purge volume, make backup volume, dump volume,* and *restore volume* are implemented in a straightforward manner using these primitive functions. Dumps are made from backup volumes (which are cloned, read-only volumes), so that the volume does not need to be locked during the dump. *Move volume* and *release volume* are more complicated.

To move a volume a new backup volume is first made. The backup volume is then dumped over a stream socket to the other site which uses the restore function to recreate the volume. During this dump process, client workstations are allowed to access and update the original volume. If any updates *are* made, then the volume is locked, and *incrementally* dumped from the time the backup was made. This incremental dump is merged into the new volume at the other site.

A message exchange confirms that the volume has been successfully moved and the copy at the original site is purged along with its backup volume. A new backup volume is made at the new site. Finally, a note is left with the file server at the original site, stating where the volume is now located: clients are redirected to the new site. The note is required because the volume location data base is not updated immediately.

To release a new read-only volume to multiple sites, the volume is first cloned at the site of the original, read/write volume. If a previous release of this volume exists at a target site, then a clone is also made of that volume. The first read-only clone is then incrementally dumped from the time that the previous release at the target site was made. This incremental dump is merged into the new clone at the target site. If a target site has no previous release of the volume, then the new clone is dumped in its entirety and transmitted to the target site.

Mount Points

Mount points are the mechanism by which volumes are glued together to form the complete system hierarchy. There is no direct provision for mount points within the structure of a volume. Originally, the mount points within a volume were kept in an associated list; this was later discarded in favour of a mechanism for directly distinguishing fids as mount points. The support for this was never actually implemented at the server, and instead the implementor of the workstation software decided to use a *hack*: a symbolic link beginning with a special symbol and protected in a way which Unix does not normally support (there is no *lchmod* in Unix!), signifies a mount point. The contents of the link give the name or number of the volume. When the workstation software encounters a mount point, it substitutes the root fid of the appropriate volume.

There are presently two types of mount points, a *read/write mount point* and a *read-only mount point*. A read/write mount point signifies that the original, read/write volume corresponding to the specified name is to be mounted at that point. A read-only mount point specifies that, if it exists, the latest read-only clone of the volume should be mounted instead *unless the volume containing the mount point is itself a read/write volume,* in which case the read/write volume should be mounted. This makes it possible to build a hierarchy of read/write volumes, glued together with read-only mount points, and then to release the collection of volumes as read-only volumes without altering the mount points within the hierarchy. At CMU, for example, the directory /cmu is the read-only, replicated root directory of the Andrew file system, /cmu/rw is the read/write root of the same file system.

Read-Only Volumes

Read-only volumes are a simple way of providing cheap and reliable access to system files. They have several properties which are desirable in a large scale distributed system such as Andrew. The primary rationale is that read-only volumes can easily be replicated at multiple sites, without concern for consistency between sites. This consistency is guaranteed since read-only volumes may not be updated and because each new release of a read-only volume is assigned a unique volume number. The workstation software may freely contact any of the servers which provide a given read-only volume at any time. This serves two purposes: it improves the availability of the system significantly by ensuring that top level directories and system files are not rendered inaccessible due to a server crash, and it improves the performance of the system by reducing the load at each site.

System performance is also improved by the use of read-only volumes because no *call backs* need to be maintained by the server. Call back is the Vice2 mechanism for maintaining consistency between files cached by the workstation and files in Vice: whenever a file is changed, the server notifies those workstations which have both a copy of the file and an active call back on the file. By definition, the files within read-only volumes cannot change, so call backs are unnecessary for these files.

Read-only volumes have other advantages for the administration of a large system. A released read-only volume provides a frozen image of a set of system files. This guarantees to users that the environment will not be changing under them as they work, which is particularly important in a setting

where most users are distanced from the administration of the system. New releases of standard system software can be planned: the various components can be installed in the appropriate read/write volumes and only released when everything is apparently under control. This is an important consideration when you realize that thousands of users may be affected by the changes! It is, of course, possible to back out from a disasterous release by simply purging the most recent version of the offending volume. The workstation software will recover from this situtation by retreating to the previous (now the "latest") read-only volume.

Volume Location Data Base

The volume location data base lists all of the volumes in the file system. For convenience, all of the volumes are indexed both by name and by number. To support replication, the list of sites for any given volume is returned on a query, and to support read-only mount points, the latest read-only volume and/or backup volume to be cloned from the specified read/write volume is also returned.

This data base is updated relatively slowly, currently once every 1/2 hour. The update is managed by the server control machine, which reaches out to each server to get its current volume list, combines the list together into a random access file, and propogates the result back to the servers. When a volume is moved, the affected server temporarily modifies the information provided by the data base.

Salvaging

In order to support the volume structure described here, the standard file system consistency checker, *fsck*, had to be modified slightly to ignore inodes allocated by Vice when considering the connectivity of the file system. It is still used, however, to verify the low-level integrity of the file system, and the integrity of any Unix directory structures which may coexist on any of the server disk partitions. In addition, a *volume salvager* was written to verify and repair the volumes resident on any disk partition. This program is straightforward. It does rely, however, on the ability to stamp Unix inodes with enough parameters to describe their role in the file system.

Backup

An automated backup/retrieval system for Vice2 has been running since the system was first deployed. This system is considerably simplified by the use of volumes: the unit of backup is a single volume dump or incremental dump, which is stored as a single file on tape. The system does not understand the format of these dumps, and simply delivers the entire volume back to a file server whenever a restore request has to be handled. Since we always have a spare file server on hand, for emergencies, there is always disk space available for these restore operations. The restored volume is usually restored as a new, read-only volume mounted in a different place, and the user is free to peruse it at his or her leisure, copying back whatever files are needed.

The dumps are made from the users' backup volumes. These backup volumes are cloned nightly at roughly the same time, and the dumps are taken sometime later. This results in predictable logical dump times. The backup volumes are also made available directly to the users: within each user's home directory is a mount point to his or her backup volume, usually called *OldFiles*. This is in effect a window on "yesterday's" files. Since the implementation of this feature, the number of restore requests processed has dropped to virtually zero.

Future Work

As previously mentioned, the administrative operations available will be enhanced further as the system continues to grow. In addition, we are planning to experiment with the possibility of different volume types for different applications. For example, with continued growth of the system, bulletin boards for the entire campus may not fit well into either the single site read/write volume model or the multiple site read-only volume model. A bboard volume could be replicated but have less stringent consistency guarantees.

During the design stage of this project we spent a great deal of time wrestling with the idea of implementing read/write replicated volumes with full consistency guarantees. This was eventually rejected as being too

difficult and unnecessary for that stage of the project. A research project is now being set up within the Department of Computer Science at CMU to investigate this possibility.

Conclusions

Volumes are a surprisingly powerful abstraction that have made the administration of a very large distributed file system considerably easier than it might otherwise have been. They provide a basic partitioning of the system for file addressing which is independent of the partitioning of the system implied by the hardware configuration. In response to changing conditions, volumes can easily be relocated within the hardware environment, in a totally transparent manner, and without disrupting service to the clients of the system. Volumes provide the basis for quota enforcement and accounting, and provide a high degree of privacy for users by allowing administrative operations to be performed at the volume level, rather than at the file level. Volumes provide the basis for replication of frequently used files. An automated backup system is considerably simplified by the concept of a volume, and, arguably the most popular feature of Vice2, the volume cloning mechanism is an inexpensive way of allowing users direct access to their previous day's work.

References

[1] Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S.H., Smith, F.D.
Andrew: A Distributed Personal Computing Environment.
In *Communications of the ACM*, March, 1986.

[2] Satyanarayanan, M., Howard, J.H., Nichols, D.A., Sidebotham, R.N., Spector, A.Z., West, M.J.
The ITC Distributed File System: Principles and Design.
In *Proc. 10th Symposium on Operating Systems Principles*, ACM, December, 1985.