Technique for redundancy control in a distributed hierarchical filestore

K Lunn and K H Bennett*

A means of controlling multiple-copy redundancy of files through a hierarchical directory system is described. Both the access paths to and the individual copies of files are replicated in such a way that a file can be located if and only if there exists a volume online which contains a copy of that file. A simple mechanism based on time-stamps is used to resolve inconsistencies between files before access. The algorithm is suitable for a distributed filestore on a local area network, or even for a multiple-volume filestore on a stand-alone machine. It is not proposed as a sensible solution to wide area network file storage on a large scale. The hierarchy is UNIX-like (UNIX is a trademark of Bell Laboratories), but without links.

Keywords: file organization, redundancy control, distributed filestorage

INTRODUCTION

Distributed filestorage is a topical subject that has been investigated in different ways. One approach is effectively to ignore user-naming and concentrate on low-level details concerning location, access and update of files, and the control of replication[1]. Files are referred to by system-interpreted names, called unique identifiers, and the mapping of user names to unique identifiers is implemented by a higher layer of software. Another approach is to combine stand-alone filestores to provide a distributed filestore by linking together the name spaces to give a larger name space[2].

This article describes an approach that concentrates on the user-oriented file naming scheme. There is a general consenus that hierarchical organization of directories and files provides a useful naming scheme, e.g., UNIX. Hierarchies are typically spread across a number of volumes. There are several ways to construct a name space for multiple-volume filestores. This article presents an alternative method that provides extra reliability and is suitable for a multiple-volume filestore on either a single machine or a network. The name of a file is independent of the location of copies of the file, and the location can be changed without changing the name. The naming scheme and the associated algorithms have been described by Lunn[3,4].

Essentially, the naming scheme provides multiple copy redundancy of both files and directories. The higher echelons of the hierarchy have the highest levels of redundancy. It is possible to trace any file on a volume, using its full path name, if that volume is accessible. Consistency between copies of a file is ensured before access to any of those copies is allowed.

The naming scheme has been implemented as a prototype on a local area network. Two LSI11/02 microcomputers each ran a single volume file server, and the directory system. The filestore was accessed by a personal workstation over a Cambridge Ring.

CONSTRUCTING MULTIPLE-VOLUME FILESTORES

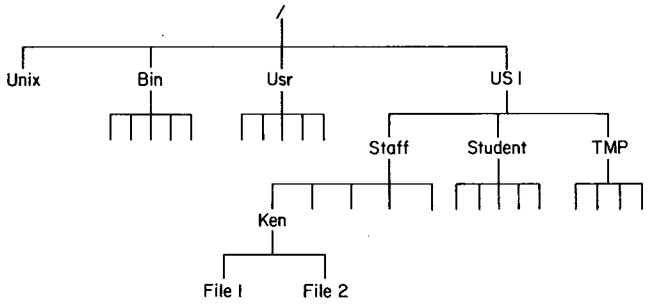
UNIX

Each volume in a UNIX filestore is organized as a hierarchical file system. One system is designated as the root file system. Other volumes can be mounted on the root file system, so that the root of the mounted file system replaces a directory in the root file system. Further

University of Bradford, West Yorkshire BD7 1DP, UK
*University of Keele, Staffs ST5 5BG, UK

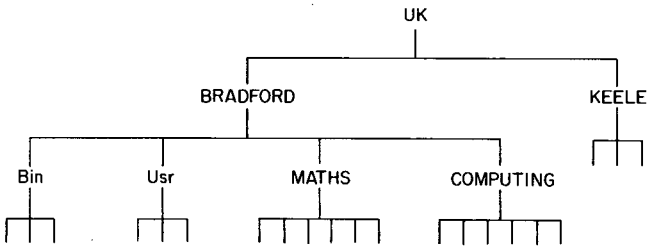
 0144–817X/84/030157–05$03.00

volumes can be mounted on mounted file systems, and so on. Thus a typical file system might look like



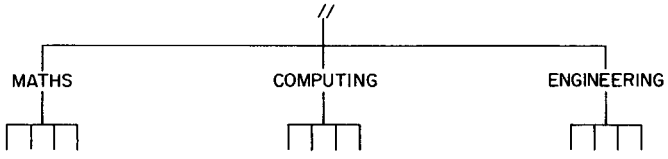
Here a volume has been mounted at US1, and the file system for that volume becomes a subtree of the total online filestore. Similarly a volume has been mounted at TMP. Thus the filestore naming scheme is infinitely extensible.

This approach has been extended to cover a distributed filestore by the Newcastle Connection, which implements a network operating system using standard UNIX operating systems. As well as mounting the file system of a volume it is possible to mount the filestore of another UNIX system. Thus, we might see a filestore with a naming scheme like



UK, BRADFORD, KEELE, MATHS and COMPUTING are the roots of separate UNIX systems. The files of these separate UNIX systems can be accessed as if they were part of the local filestore. Further UNIX systems may be mounted on, for example, the BRADFORD, KEELE or COMPUTING systems. Similarly, the UK system could be mounted in another UNIX system. Thus, it is possible to build an infinitely extensible filestore (both upwards and downwards) out of a number of separate UNIX filestores.

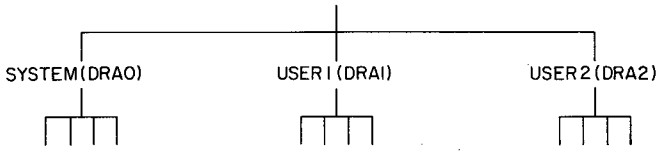
Another technique for uniting multiple UNIX-like filestores is popular. A global root is constructed from unique names given to all the file stores on the network. A user accesses local file names in the normal way, and specifies a full path name from the global root to access remote files. Thus a system might look like



Here MATHS, COMPUTING and ENGINEERING represent roots of separate filestores on separate machines. A user on the COMPUTING machine might refer to a file by the path name /usr/ken/file1. A user on another machine would refer to it relative to the global root (denoted //) as //COMPUTING/usr/ken/file1.

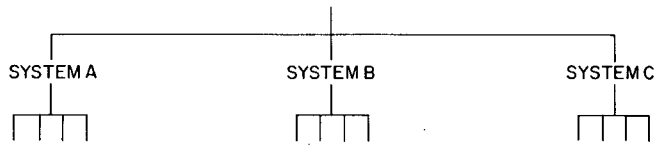
VAX/VMS

A common way of uniting file systems on volumes is typified by the VAX/VMS operating system. All online volumes are uniquely named. The name may be that of the drive on which it is mounted, or a user-provided name. A typical configuration might look like



There are three discs mounted on drives DRA0, DRA1 and DRA2, with names SYSTEM, USER1 and USER2 respectively. A user may name a file in two ways, using the drive name or the volume name as the name in the apparent root directory. Thus USER1 : [KEN.FILE1] and DRA1 : [KEN.FILE1] refer to the same file. The name between "[" and "]" is a path name delimited by ".", and a full hierarchy can be constructed.

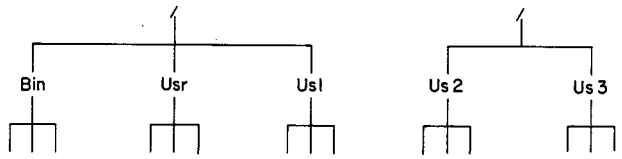
The network version of this uses names of systems. A configuration might look like



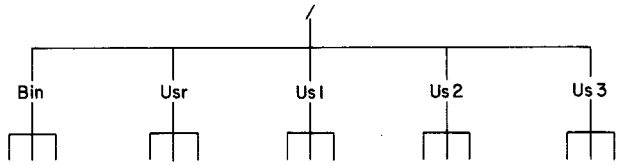
Here SYSTEMA, SYSTEMB and SYSTEMC are names of computers on a network. A file on SYSTEMA might have the name USER1 : [KEN.FILE1]. A user on SYSTEMB would refer to it as SYSTEMA : : USER1 : [KEN.FILE1].

An alternative approach

Lunn's[3] approach to naming differs from the above. Each volume contains a complete set of directories enabling all files on that volume to be traced without reference to other volumes. Multiple volume filestores are constructed by an overlay of all the file systems of individual volumes. Thus, two volumes may be combined like this

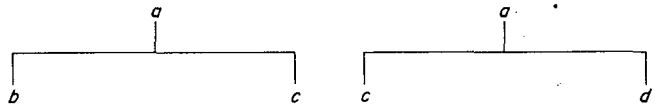


to give

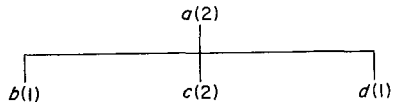


Further volumes may be overlaid on this newly created structure. The file naming is the same as in standard UNIX.

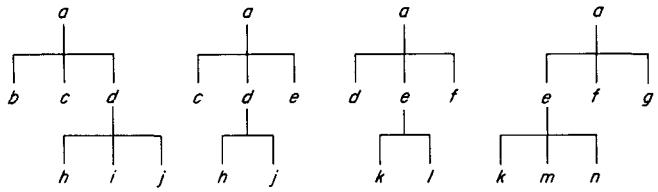
With the overlay technique it is possible to have multiple copies of files and directories. For example, we might overlay the following two structures



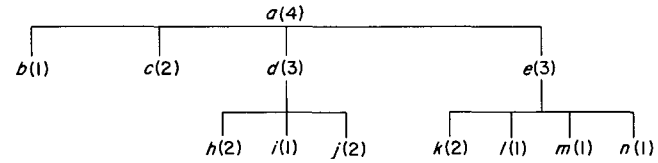
to give



The number in brackets indicates the number of objects with that name. It is possible to overlay arbitrarily complex trees. For example:



to give



Thus an arbitrary level of replication can be introduced.

Mounting a file is simply the addition of an overlay, and dismounting the removal of an overlay. To access a file it is merely necessary to trace the path from root to the given file; if two or more copies exist, an arbitrary choice can be made.

The overlay scheme has some quite distinct properties from the previously described naming schemes.

- It can cope with multiple-copy redundancy within the naming scheme.
- The path name of a file remains the same, irrespective of the location of the volume.

In UNIX, the full path name of a file on a non-root system depends on where that file system is mounted. In the other schemes, it may well depend on which system it is located. The ability to name an object uniquely irrespective of its location is highly important where names are bound into other objects or programs. On the other hand, there is no possibility of constructing an 'infinite network' like the Newcastle Connection.

Extra controls are required for a dynamic system. For example, how are updates to replicated files controlled? What happens if a volume is offline for a period? On what volumes should a file be replicated? The solutions described by Lunn[3] are presented here. Alternative solutions, based on the overlay technique are possible, and perhaps preferable in many instances, and some of these are discussed.

IMPLEMENTATION OF THE NAMING SCHEME

The techiques used to implement a prototype of the above naming scheme are described here. However, they are not ideal, and possible alternative means will be discussed.

Associated volumes

On creation, a directory is given a set of volumes, called the associated volumes for that directory, which must be a subset of the associated volumes of its parent, i.e.,

(1) assocvols $(P/d) \subset$ assocvols (P)

where P is the path name of a directory and d is a subdirectory of P. Clearly, the set of volumes available are precisely the associated volumes of the root directory.

All files referenced by a directory are stored on all the associated volumes of the directory. A directory is stored on all its associated volumes, In this way, redundancy of both files and access paths is controlled. The access path to a file can be traced if a volume containing it is online.

Associated volumes allow tailoring of the hierarchy to meet specific requirements. For example, a user with data which must be online with a higher expectancy might be allocated a directory with three or more associated volumes. A user with multiple associated volumes in his/her home directory may create subdirectories with fewer associated volumes for less critical applications. Suitable accounting should prevent profligate misuse of available redundancy. Clearly the higher echelons of the hierarchy should not generally be used for storing files.

Primitives may be required to increase or decrease the set of associated volumes of a directory. This is necessary to add new volumes to the system, or to remove volumes from the system. Clearly these must not violate the constraint (1) above. Thus if a directory gains an associated volume, that volume must be associated with the parent director. If a directory loses an associated volume, then that volume must be disassociated with all subdirectories.

The active filestore

In a large filestore, only a part is likely to be in use at any one time. The algorithms[3] take advantage of this fact. All files which are open, and the directories on paths to those files are said to constitute the active filestore. For each directory in the active filestore, a single process is created to administer activities in that directory. In the implemented prototype, a single site was elected to run all the directory processes. Clearly, these processes could be distributed according to the location of data or according to other criteria such as load balancing.

The choice of a single directory process for each active directory solves synchronization problems. However, it does imply an unfortunate overhead in process creation. It may also cause problems if a directory process fails, breaking the path to an open file. Other solutions to synchronization may be worthy of consideration. In a sense, the process-per-directory is a move toward a centralized solution, but it is one way of enforcing consistency constraints on a directory.

The notion of an active filestore is largely introduced for efficiency. Ideally, a process for each directory should be running or ready to run all the time. It would be very difficult to implement this on existing hardware and operating systems.

Consistency resolution

For high accessibility, it must be possible to access files even if a number of volumes are offline. This can lead to situations where copies of files are out of date. On creation a directory process resolves whatever consistencies it can detect before allowing access to the contents of the directory.

Each file on a volume is timestamped, the timestamp being stored in the copy of the parent directory on that volume. This implements a function

time: volume, pathname → integer

If P is the path name of a directory on $v1$ and $v2$, and $v1$ and $v2$ are online, the directory process enforces the constraint

(2) $\text{time}(v1, P/n) = \text{time}(v2, P/n)$

for all files named n in directory P. The directory uses a careful replacement strategy for update, and on update of a file it copies the replacement file to all (online) volumes on which it should be stored, and allocates a new time to the copies. If constraint (2) is violated, the directory process selects the online volume $v1$ for which

$\text{time}(v1, P/n) \geq \text{time}(v2, P/n)$

for all online $v2$ in assocvols(P). Whenever $v2$ violates the constraint (2), the directory process replaces the copy of the file on $v2$ with the version on $v1$, and updates the timestamp on $v2$ so that

$\text{time}'(v2, P/n) = \text{time}(v1, P/n)$

where time' is the time function after resolving inconsistencies. In this way, files can be updated when some copies are offline, and the offline copies can catch up.

Another problem arises if a file is created when some of the associated volumes of the directory in which it is created are offline. The directory process enforces the constraint

(3) $P/n \in \text{names}(v1)$ AND $v2 \in \text{assocvols}(P)$
$\Rightarrow P/n \in \text{names}(v2)$

where $\text{names}(v)$ is the set of path names of objects on volume v, and where $v1$ and $v2$ are online. That is, if a file exists in a directory on a volume, then it must exist on all the associated volumes. This is simply done by copying a version of a file onto all volumes which should but which do not contain it. Files created in this way are given the timestamp of the file which is copied.

Finally, it is possible to delete a file where not all copies are online. This is done by leaving an assassin for the file in the online directories so that future activations of the directory can remove any remaining copies of a file. The assassins are given timestamps, so that a recreated file cannot be deleted because of an old assassin. Thus if

$\text{time}(P/n, v1) < \text{time}(P/n, v2)$

and P/n is the name of an assassin on $v2$ and a file on $v1$, the file on $v1$ is replaced by an assassin. If P/n is the name of an assassin on $v1$ and a file on $v2$, then the assassin on $v1$ is replaced by a copy of the file on $v2$. When all associated volumes of a directory are online it is safe to remove any assassins in that directory.

Weak consistency

By allowing access to any file which is online, the above algorithms provide weak consistency, in the sense that they will ensure consistency between currently online volumes, but may result in problems over a period of time. For example, suppose $v1$ and $v2$ are the associated volumes of P, and that they are online at different times. If a user creates a file P/n on $v1$, and later cannot find it because $v1$ is offline, so he recreates it on $v2$, when $v1$ and $v2$ are online together, which one does the user really want to keep (it may not be the most recently created). More complex examples can be created with more than two volumes.

If the likelihood of a volume being accessible is high, the above algorithm may suffice, and the occasional problems may be infrequent enough to be ignored. However, on a complex system, where volumes may be dismounted frequently, or where network partitioning may occur, an extra constraint may be necessary.

Strong consistency

The extra constraint of requiring a majority of associated volumes to be online for activation of a directory process removes a number of problems. Clearly, under this constraint, whenever a file is created, updated or deleted, the current majority will intersect with any future majority, so that the most recent change to a file will always be reflected in an online directory. This constraint must be taken into account when increasing or decreasing the associated volumes of a directory.

Timestamping

Whilst a timestamp is used to detect and resolve inconsistencies between copies of a file, any guaranteed monotonically increasing value might be used. If the majority online constraint is used, then an integer stored in each copy of a directory will suffice, the integer being increased every time a file is created, updated or deleted. An integer per file (a version number effectively) would also suffice.

Recovery

Careful replacement was used as a means of updating files. This simplified the prototype implementation, but it is recognized that such a policy would generally be unsuitable. Alternative means of roll-forward recovery for files which fall behind may be considered. For example, an audit trail might be stored on all online associated volumes of a directory when one or more associated volumes are offline. Such an audit trail would clearly depend on the semantics of the operations on the files involved.

CONCLUSION

This article has attempted to distil some of the ideas described by Lunn[3]. A large amount of detail has been omitted, especially concerning performance, operations on files, and protection. However, the overlay technique should be applicable independently of the file access methods used. In fact, the technique should be applicable to the naming of objects other than files where multiple

copy techniques are appropriate. The technique does not depend upon the implementation loosely described above; alternative methods of error detection, consistency resolution, error recovery and directory administration could readily be devised. Work is continuing at Keele to adapt the overlay technique using the Newcastle Connection.

ACKNOWLEDGEMENTS

The above work was undertaken as part of a project investigating distributed file storage at the University of Keele, funded by the Science and Engineering Research Council; we are duly grateful for their support. Special thanks are due to Pearl Brereton and Paul Singleton for their contribution of constructive criticism and expertise in software and hardware.

REFERENCES

1 **Sturgis, H, Mitchel, J and Isreal, J** 'Issues in the design and use of a distributed filestore' *ACM Operating Syst. Rev.* Vol 14 No 3 (July 1980)

2 **Brownbridge, D R, Marshall, L F and Randell, B R** 'The Newcastle Connection — or UNIXes of the world unite!' *Software Pract. & Exper. (GB)* Vol 12 (December 1982) pp 1147–1162

3 **Lunn, K** *Reliable file storage in a distributed computing system* PhD Thesis, University of Keele (1983)

4 **Lunn, K and Bennett, K H** 'A highly reliable distributed filestore directory system' *Proc. 2nd Int. Conf. Distrib. Comput. Syst.* IEEE catalogue 81CH1591–7 (April 1981) pp 299–307

BIBLIOGRAPHY

Lamport, L 'Time, clocks and the ordering of events in a distributed system' *Commun. ACM* Vol 21 No 7 (July 1978) pp 558–565

Randell, B, Lee, P A and Treleaven, P C 'Reliability issues in computing system design' *Comput. Surv.* Vol 10 No 2 (June 1978)

Salter, J H *Naming and binding objects, operating systems — an advanced course* Springer-Verlag, FRG (1979) pp 99–208