MULTI-RESIDENT AFS:
AN ADVENTURE IN MASS STORAGE

Christopher Kirby, Christopher Maher, Bill Zumach
Pittsburgh Supercomputing Center

1. Abstract

The Pittsburgh Supercomputing Center has been working to integrate distributed file system technology with hierarchical mass storage. We produced a system utilizing the Andrew File System that can be interfaced to many mass storage systems. We retained the semantics of AFS and compatibility with standard clients and servers. The architecture has a logical separation between the facility that provides the user interface and access semantics and the management of the storage systems that contain user data. Support for file level replication is provided for high availability to data in a fashion that is transparent to users. This system is called Multi-Resident AFS.

2. Introduction

There has been a great deal of work in the last ten years in making distributed file systems faster, more functional, and easier to use. This has lead to their widespread use from the desktop to the supercomputer. The explosive growth of data being stored in these systems, combined with the exponential growth in client capacity, emphasizes a necessary capability, support for hierarchical mass storage. The Pittsburgh Supercomputing Center has been working for several years to address this problem. Our experiences with our system based on the Andrew File System (AFS) from Transarc will be described. While the paper will concentrate on AFS, the lessons learned apply to most, if not all, distributed file systems.

3. The Problem

In 1990, the Pittsburgh Supercomputing Center was facing a serious problem in providing access to the large volume of data produced by our users. We had thousands of users spread out across the United States and needed a way to provide them with easy access to the data they produce on the center's supercomputers and as a vehicle for collaboration with other scientists. While we were already running the Los Alamos Common File System (CFS) [1] to service our mass storage needs, it had only a simple get/put file interface and no distributed access.

We identified distributed file systems technology as a way to provide our users with both a friendly interface and the ability to access their data from their home sites. However, there were few products available that offered both distributed access and support for high latency, high capacity media like tape. The products we did find offered FTP and NFS interfaces but suffered from several significant problems. When tape mounts weren't completed within some minimum time, users would see NFS time-outs. All of the software had to be run on a single, high capacity server machine; there was no support for distributing the load across several file servers without duplication all of the equipment. This inability to spread the load over multiple servers was made more painful by the fact that the NFS interface offered by these products had to query the servers to check the consistency of any data they might have cached. This was a major performance problem considering that our clients were much faster than any server available. There was also very little in the way of security in these systems. They were designed with the model that every machine and network between the data and users was under a centralized control. Such an environment is becoming increasingly rare and certainly didn't include our site.

Some other, more long range, problems we observed in the available products included the lack of options for server platforms. These systems were generally designed for a specific machine and there was little choice in hardware. This was made even worse by the observation that there were also no standards to allow us to move data from one vendor's system to another, thereby locking us into a single product line.

While the amount of data we store is much larger than most sites, our problem of not being able to afford as much magnetic disk as our users want is universal. The need for secure, distributed access to data from many clients is not limited to supercomputing centers.

4. Existing Solutions

There are a number of ways to approach this problem. Some systems extend the UNIX™ file system at the kernel level to intercept file I/O and do the necessary work to bring the data onto a disk. Examples of this method include NAStore from NASA Ames [2], InfiniteStorage from Epoch Systems [3], and CRAY Data Migration Facility from CRAY Research. The file system is periodically scanned for unused files which are moved to slower, cheaper media. Distributed access to the data is generally provided by NFS exporting the local file system with the resulting potential for NFS time-outs mentioned earlier. For those systems not provided by the vendor of the operating system, there is the even bigger problem of maintaining the kernel modifications across versions of the operating system and across hardware platforms.

Another approach to solving this problem is to let the mass storage system export an NFS server interface without an explicit UNIX file system underneath it. The file data is spread out among disks and tapes but appears as a single NFS mounted partition to the users. The file name space is kept in some sort of database. UniTree [4], which is provided by several vendors, uses this approach. This has the advantage of a standardized, user-mode interface running on the server. There are none of the problems typically associated with changing a kernel to provide the necessary interface. The down side of this approach is that the mass storage vendor has to reproduce the UNIX file systems technology in user mode that vendors provide in the kernel. The caching strategies and use of kernel memory and buffers are not available to them. This causes a general performance problem when doing operations that don't involve much actual movement of data.

More recently, the Center for Information Technology Integration at the University of Michigan proposed the mass store itself be considered the file system, not just a bag on the side of a disk-based file system [5]. This is based on the approach taken by the Plan 9 [6] file system where high-speed RAM acts as a cache for magnetic disk, which in turn acts as a cache for WORM optical disk. Disks and memory would be used as high speed caches in front of the relatively slow tape or optical mass storage system. The idea is to create a transparent byte-level interface to files located in the mass storage system. A major problem with this approach becomes apparent when file system meta-data is considered. File systems generally do many small, random access operations on meta-data files, such as directories. If a directory file is not in the disk or memory cache, performance is very poor when referencing any file below it in the name space. Given that tape systems are rarely random access and have generally poor start/stop behavior, meta-data operations couldn't be handled until the file had first been moved to the disk or memory cache. The implications of this are more clear when one considers what it would be like to fsck such a file system. That operation performs random access reads to every meta-data file in the system to check its integrity.

5. Why AFS?

We chose AFS [7][8] for our distributed file system for a number of reasons. From a user's perspective it has several desirable features including a global name space where the path to a file is the same from any client, Kerberos V4 security, access control lists on each directory, semantics reasonably close to UNIX file system semantics, and good performance using client-side caching. Some of the important management features include the volume abstraction, location transparency for volumes, read-only volumes, replication of volumes across multiple servers, and volume quotas. We also observed that AFS scaled well; its servers could support a much larger number of clients per machine than NFS. This is due to its use of callbacks to inform clients when data they had cached became invalid. This eliminates the need for clients to make RPCs to the servers whenever they want to validate some object they have cached.

6. Mass Storage Extensions to AFS

The most fundamental goal of our extensions is to preserve the semantics of AFS and not attempt to make it an expert in mass storage systems. By leveraging each type of technology that is available, we provide a framework to allow each component to do what it does best. The file system handles the user interface and access semantics, and the mass storage system handles the bytes. The logical separation is that the file system handles the meta-data, and the mass storage system handles the user data. The system we created to implement this model is named Multi-Resident AFS.

Many changes are required to make AFS function as part of a true mass storage system. The main design goals are hierarchical storage management, fault tolerance, and access to existing technology, while still retaining compatibility with standard AFS clients and servers. We achieved the former goals through extensive generalizations of the AFS server code, and the latter by working within the standard AFS data structures.

7. Multiply-Resident Data

Our extensions allow AFS to move from a system capable only of serving UNIX disk partitions to a true hierarchical storage system. Where files were once restricted to exist in only one place, they are now able to exist in up to 32 separate locations. These new copies of data, known as residencies, give AFS many advantages. These additional residencies not only protect AFS against media failure, but also provide high availability of user data. If a storage system is unavailable for preventive maintenance or any other reason, AFS will automatically obtain the data from the highest priority storage system that is available. This is all completely transparent to users. In this manner, we can take preventive maintenance time with minimal impact on our users.

This functionality is similar, but orthogonal, to AFS volume replication. Here we are talking about a per-file replication, where each file is treated independently. In AFS volume replication, all of the files within a volume are copied at the same time and to the same destination.

The ability to have multiple copies of file data becomes more important as support for mass storage is added. A single 8 mm tape can hold 50,000 files with an average size of 100 KBytes. The loss of this single tape for any reason would have an enormous impact on users. As file servers gain the ability to utilize ever larger capacity media, this danger grows. This applies to disks as well as tape, given how fast the storage densities of disks are growing.

Figure 1 shows how an AFS volume, the management unit for an aggregate of files, changes with this additional support. Notice that files can be put on different storage systems based on their sizes.

8. Consistency

Allowing a file to exist in more than one storage system brings up the problem of insuring that each copy contains the same data. When a multi-resident file is modified, only the new, modified copy remains. All of the references to copies of the previous version of the file are removed. This new version of the file is then treated the same as any new file in terms of data migration. The file server also guarantees that no partially created copy of a file will ever be referenced.

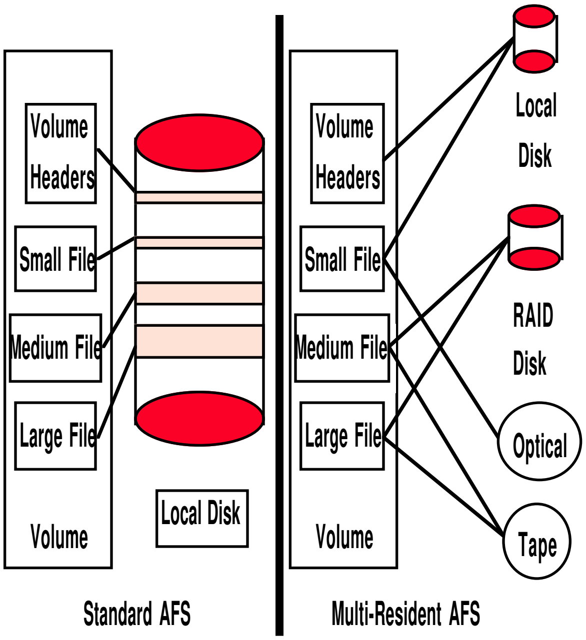

Figure 1. *Volume layout*

9. Data Migration

With the addition of multi-resident support, it is necessary to have a data migration system to add and remove these additional residencies. There is a small replicated residency database which contains information about the known storage systems, but not volume or file level information. For each storage system, the information includes a relative priority, a desired file size distribution, the list of machines that offered the storage system, and various data migration policy variables. The goal of the data migrator is to optimize the use of storage by transparently moving data between members of the storage hierarchy. This includes adding redundant copies of data to provide fault tolerance, removing copies of data from storage systems whose free space fell too low, moving hot files to faster media, and moving cold files to slower media. Another important task of the data migrator is to insure that each storage system has the desired file size distribution. This way we can tune each storage system to a specific target range of file sizes, making it much more efficient. The policies governing these actions can be changed easily, either by changing the contents of the residency database or by changing a configuration file. All policy variables in the system are dynamic and can be changed without interruption of service.

The data migrator also has the ability to move files or whole volumes to off-line storage. These file residencies are unavailable to users but can be migrated back by the site manager. This feature is intended to handle the case where a collection of data is no longer in use, but may be needed at a later date.

It should be noted that while this system moves data between storage systems, it has no knowledge of what those storage systems do with the data. For example, AFS might create a residency in an NFS archival system in which the data initially lands on a disk and is later moved to an optical platter. AFS has no need to know that the file has been moved within the NFS storage system.

Another important responsibility of the data migration system is to create random access residencies of files on demand. AFS clients access files from the AFS server in a random access fashion. If the file does not exist on a random access storage system when a fetch request is received, the data migration system must create an additional residency. We call this process "spooling a file", and we call storage systems that do not offer random access "archival" storage systems. This ability to spool a file allows AFS to make use of storage systems for which random access is either not available, or too slow to be practical.

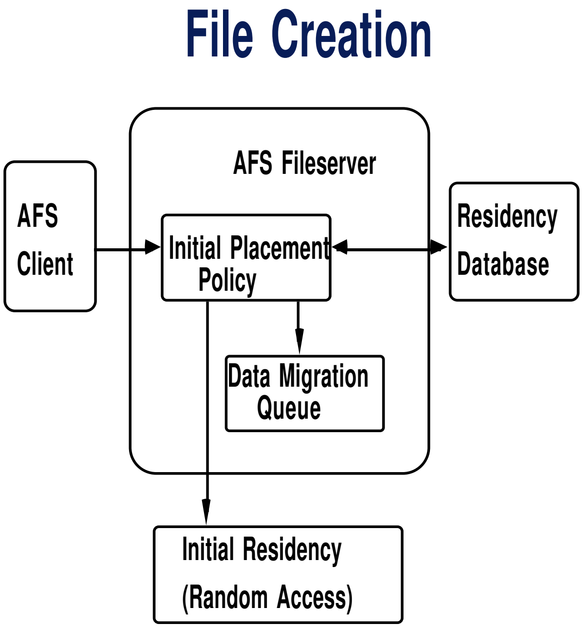

Figure 2.

10. Data Flow

At this point we will describe how data is moved through the storage systems. Figure 2 shows how a file is created. A storage system is chosen based on the file's size and the free space on the random access storage systems. The residency database is consulted to determine which storage system is the best choice for this new file. This newly created file is also added to a data migration queue for possible creation of additional residencies on archival storage.

Figure 3 shows how this data migration queue is processed into requests to copy the file to one or more archival storage systems. The residency database has a policy variable for each archival storage system that indicates how old a file must be before it gets an additional copy made there. Setting the minimum age to a small value reduces the window of vulnerability to media failures, but this also increases the likelihood that a temporary file will be moved to archival storage unnecessarily. For our environment we have found a minimum age of six hours to be a reasonable value.

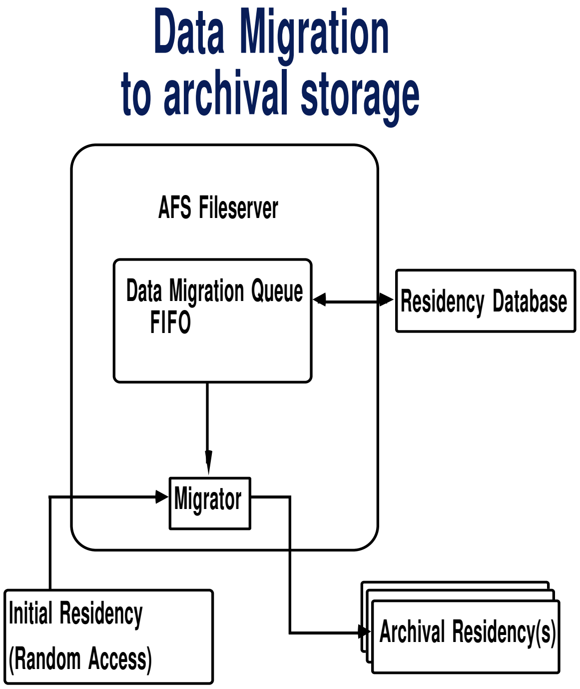

Figure 3.

Figure 4 shows how space is made available when there is a shortage. There is a new process that runs on each file server machine which checks storage systems for insufficient free space every five minutes. It processes the volume meta-data and bins up files that are candidates for removal from each storage system. A primary weighting function is used to determine into which bin a file should be placed and a secondary weighting function is used to sort the files within a bin. These weighting functions are configurable, we use seconds since the last user access as the primary weight and file length in bytes as the secondary. The deletion policy also includes the ability to provide a list of free space thresholds to be applied to different primary weight ranges.

Once the bins of candidate files are created, the free space reported by the residency database is compared to the threshold that should be applied to the highest weight bin. If there is insufficient free space on a particular storage system, requests to remove candidate files from that storage system are sent to the file server until we reach a stopping threshold. These requests must go through the file server to preserve the AFS locking semantics.

One of the major advantages of using last access time as the primary weight is that it eliminates the need to process the volume meta-data every time the checker wakes up. Any bins of candidate files that remain from the last run are still valid. The highest weight bin from the last run is still the highest weight bin because all files age at the same rate. This would not be true if the primary weight was file length.

Space Reclamation

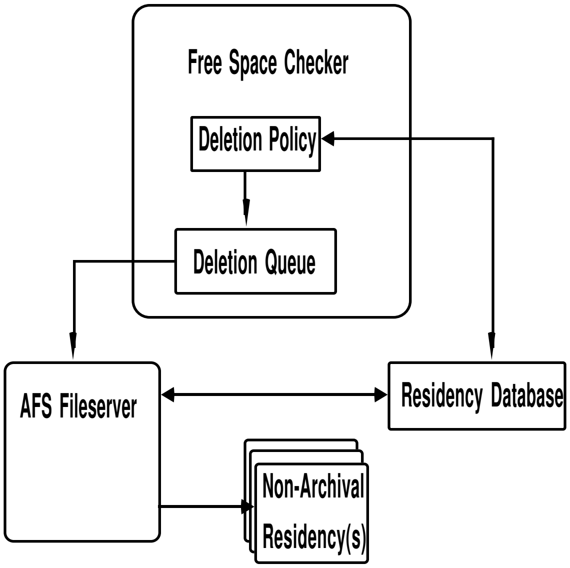

Figure 4.

Figure 5 shows how a file is spooled from a non-random access storage system to a random access storage system when a client tries to read it. As mentioned previously, the FetchData RPC requires a random access copy of the file. When the RPC is processed, the file server looks for a random access copy of the file. When one isn't found, a new copy is made on a storage system chosen by the residency database, the decision being based on file size and storage system priority. Once the new copy has been created, the FetchData RPC continues. The new copy is not considered valid, and is not referenced, until it has been made completely. This is intended to prevent a file from having multiple residencies with different data.

Data Migration from archival storage

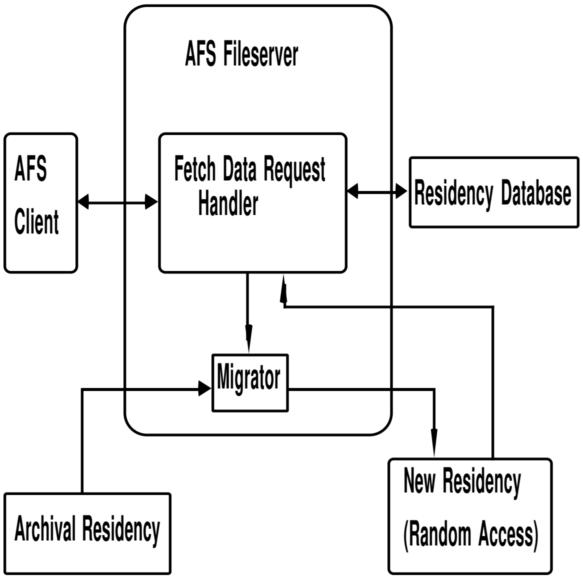

Figure 5.

One of the other interesting features of this data migration architecture is that by adding additional file residencies as soon as a file has reached a minimum age, we can free up space on our random access storage systems at the speed of a delete operation. Many systems don't begin to create additional residencies until the random access storage system starts to get full. That can often lead to a situation where free space can only be made available at the speed at which one can write data to the generally slower archival storage system. Also, unlike most other architectures, when one modifies a file there is no requirement that it be on random access storage before the modification can begin. Most products spool a file to a disk first, which is wasted effort if the modification involves a complete overwrite of the file. Our system treats the new version of a file as a separate copy of data and will only access the previous version in the event that some part of the old file was not overwritten. This also avoids getting into the situation where a file whose old version was small and whose new version is large ends up residing on a storage system reserved for small files.

Considering the exponential growth of data being stored, it becomes very difficult, as well as expensive, to do backups. The ability to create additional residencies for each file helps a great deal with this backup problem by offering a new level of fault tolerance. Our extended AFS system essentially provides a continuous, per-file, backup system. This has enormous performance advantages over the usual snapshot backup systems which operate on an aggregate of files. Since every file is being treated independently, there is no need for the system to be quiescent to insure self-consistent meta-data. Additionally, we can control which files are backed-up, and to where, based on characteristics such as size and access history. However, this does not handle the case where a user deletes a file by accident. The backup volume ability in AFS provides some limited recovery from this failure, but backup volumes are often remade each night. If a user does not realize the mistake in time, the backup volume will not help. More will be said on this point in the lessons learned section.

Another important advantage of multiply-resident files is that we are able to bring new storage systems into service and take old ones out of service in a user transparent fashion. It is often the case that when replacing a major storage system, there is no simple way of moving the data from the old system to the new one. With only two directives, our AFS system will do the migration automatically.

11. Generic I/O Interface

Many of the storage systems we wanted to integrate with AFS did not offer the UNIX file system interface required by the AFS open-by-inode I/O calls. To solve this, we generalized the I/O system of AFS. By doing all I/O through a generic interface, the UNIX file system dependence of AFS is removed. This allows AFS to take advantage of the available technology without having to reinvent it. In order to interface to a given storage system, only a small I/O library is required. AFS can then access a wide variety of storage systems, including UniTree, CRAY Data Migration Facility, Epoch, and a Maximum Strategy RAID disk system.

There are three types of routines that must be provided when creating an I/O interface. The first group of routines does file management. Open, Read, Write, Seek, Truncate, CloseIfOpen, Stat, and FsyncFile are generally well understood routines so we won't describe them here, except to say that an interface that doesn't support random access would not need to actually implement a seek routine. The other file management routines include IncDec and BulkIncDec, which are used to change the volume reference count on file data, and GenerateLookupTags, which is used to create a 64-bit identifier for a new file on a storage system. The first two routines are similar to link and unlink, and the last routine is similar to create.

The next group of routines are storage system management routines. They include GetFileSystems, GetFsAdvisoryLock, ReleaseFsAdvisoryLock, StatFs, VerifyResidency, ListFiles, ListVolumeHeaders, and Configure. These routines can be thought of as operating at the UNIX partition level. GetFileSystems is similar to returning a mount table. GetFsAdvisoryLock and ReleaseFsAdvisoryLock are used to control access to a storage system during execution of a salvage operation, the AFS equivalent of fsck. StatFs returns the free and total space available on a storage system. VerifyResidency reports whether a storage system is available for use. ListFiles is used to dump the list of files that exist on a storage system, this is equivalent to dumping the inode table on a UNIX file system. ListVolumeHeaders returns the list of AFS volumes that exist on a given local disk storage system. Finally, Configure is used to set up a storage system when it is first brought into service.

The last type of routine is the optional Import function. It is necessary to address the problem of how to merge an existing mass storage system with AFS. The Import routine allows one to create a file in the AFS name space and map it to a pre-existing piece of user data. It is intended that this routine will just alter the lookup handle of the user data to conform with the AFS naming scheme. No actual movement of the user data should be involved. This can be thought of as a rename operation, but not a copy. This feature also allows one to store data into a storage system shared with AFS, using some other mechanism, and later add it into the AFS name space.

12. AFS Servers Share Storage Systems

Up to this point we have only discussed a single AFS server environment. But in practice we have several servers, each with its own set of storage systems. Because of the expense involved in providing every server with direct access to every storage system, we provide a mechanism that allows the servers to share their storage systems. This mechanism, known as the remote I/O server, allows one AFS server to execute I/O calls on a storage system that is not directly connected to it. The remote I/O server is a small RPC service that allows a machine to offer to AFS servers all of the storage systems to which it has direct access. This has the added advantage of allowing us to bring more machines into the server hierarchy without having to run full AFS file servers on them. As long as a machine is capable of running an AFS RPC service, it can offer storage systems to AFS file servers. It should be noted that each file server has a set of storage systems defined to be its "local disk" which only it may use. This "local disk" system corresponds directly with the standard AFS UNIX disk partitions in that data that is private to a server is kept here.

13. Salvaging

Salvaging consists of comparing the meta-data files, including directories, with what actually exists out on the various storage systems. Our system has a rewritten salvager to deal with the new problems that arise in this system. Now that files and directories can be spread out across multiple storage systems and multiple servers, the salvage process becomes much more complex. It is quite possible that when a salvage is being executed, not every machine or storage system is available. The new salvager is smart enough to handle this case without removing references to files whose existence can't be verified.

Another major issue is salvaging directories that are not on a random access storage system. The salvager must read the contents of every directory so that files can be claimed. Any files that have not been claimed need to be added to a lost+found directory. Many hierarchical storage products do not allow directories to be migrated off of the local disk because of the need to have them available for salvaging in a random access fashion. The new salvager gets around that limitation by guaranteeing that directory check operations do no seeks and therefore do not require random access. This allows directories to be migrated off of the disks without the need to spool them back from tape whenever the salvager runs.

There is also support in the salvager for doing selective salvages of storage systems. As a rule, we trust our archival storage system not to lose files. Therefore we don't include it in storage systems we salvage by default. This saves a great deal of time.

14. Comparison With Standard AFS

In light of what has been described above, we will now take a look at how some fundamental AFS concepts have changed. In standard AFS, the logical unit of files, a volume, is tied to a specific UNIX partition on a specific server. This has been extended to allow the volume to span many storage systems, as shown in Figure 1. By limiting only the volume headers to the original server and partition, we can effectively utilize a hierarchy of storage systems for user data, while retaining backwards compatibility with standard AFS volume utilities. Previously, one had to worry about assigning too many volumes to a single UNIX partition and the degree to which its space was being over-allocated. In the new model, this is no longer a consideration.

Standard AFS has the concept of a read-only volume that is a replica of a read-write volume that may exist on another server and/or partition. These read-only volumes are used to distribute the load to a very active, and slowly changing, collection of files among a number of AFS servers. They also provide a limited amount of fault tolerance. If one replica is unavailable, AFS clients will automatically fail over to another replica. The improvement we have made to this system is that the read-only replica(s) and the read-write volume actually share copies of data in the shared storage systems, even if the volumes reside on different servers. In this manner we gain all of the advantages of having replicated volumes without paying the storage penalties. It is important to remember that we no longer use replicated volumes to protect against media failure; the multiple file residencies take care of that. Therefore we lose nothing by having the replicas share copies of data. The replicated volumes are used to spread out the load of AFS name space operations.

The lookup information for files in standard AFS is kept in one of two volume meta-data files in a data structure called a vice node (vnode). In order to support multiple residencies of a file, while working within existing data structures, we create chains of vnodes. The original vnode of a file is unchanged except that two previously spare fields are used to point to other vnodes. These other vnodes hold the lookup information for additional file residencies. This approach allows system utilities and dump formats to be backwards compatible.

Standard AFS keeps track of very little in the way of a file's usage history. We add a per-file access history with the addition of another vnode to the chain. This history includes details of the most recent accesses and keeps various counters that let us know how well the system is managing each file. This information is used by the data migration system to determine which files should be migrated and to where. We also provide reporting tools that generate reports based on this information.

15. Current System Usage

Our current mass storage system is shown in figure 6. We use the local disk and the Maximum Strategy RAID disk banks for random access storage and CRAY Data Migration Facility (DMF) for archival storage. We configured our residency database to put files 64 KBytes or less on the local disk residency and larger files on the RAID disks. Files that are over 4 KBytes will be migrated to DMF.

The AFS clients range from hundreds of workstations to a CRAY C90. The bulk of the files are created by the workstations and the bulk of the data is created by the CRAY. The workstations include many personal workstations and a few dozen members of workstation farms doing large scale data processing.

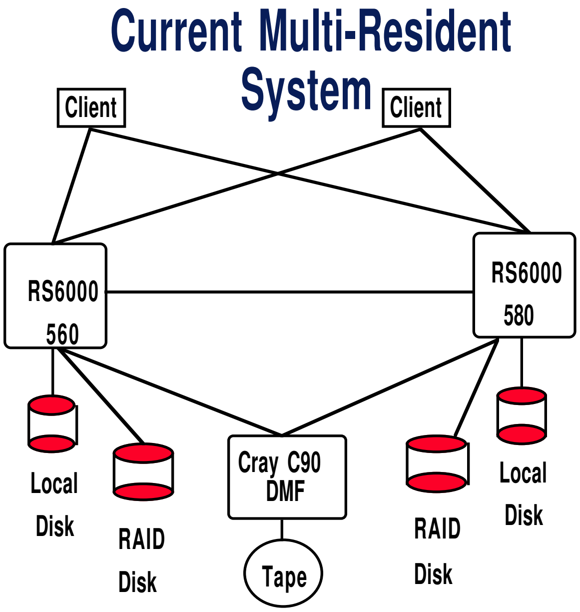

Figure 6.

Figures 7 though 10 describe how this system is being used. Figure 7 shows that our average file size is somewhat larger than the typical UNIX file [5][9].

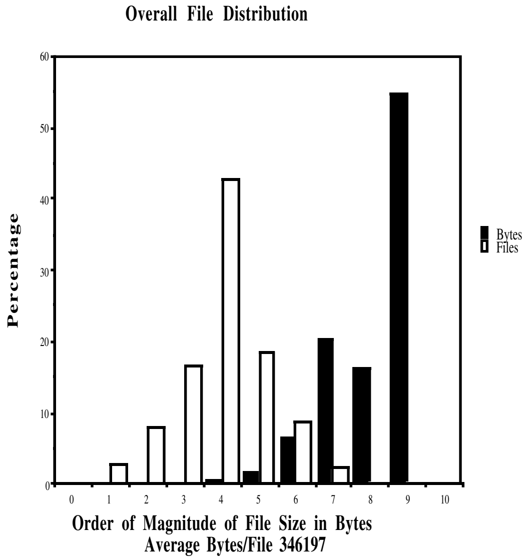

Figure 7.

Figures 8 and 9 show how the system sends small files to the local disk and larger files to the RAID disk banks.

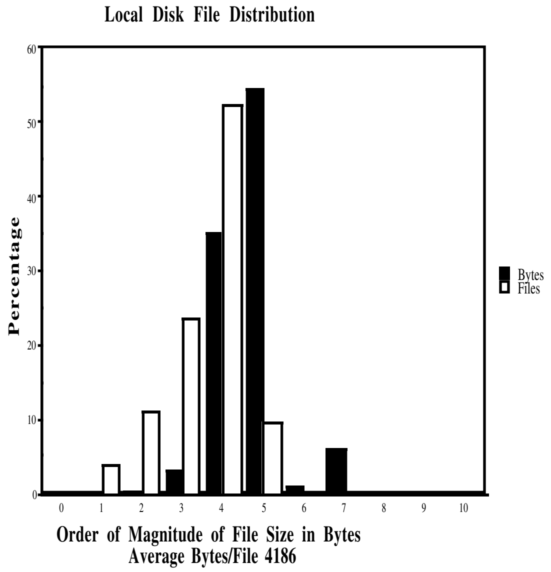

Figure 8.

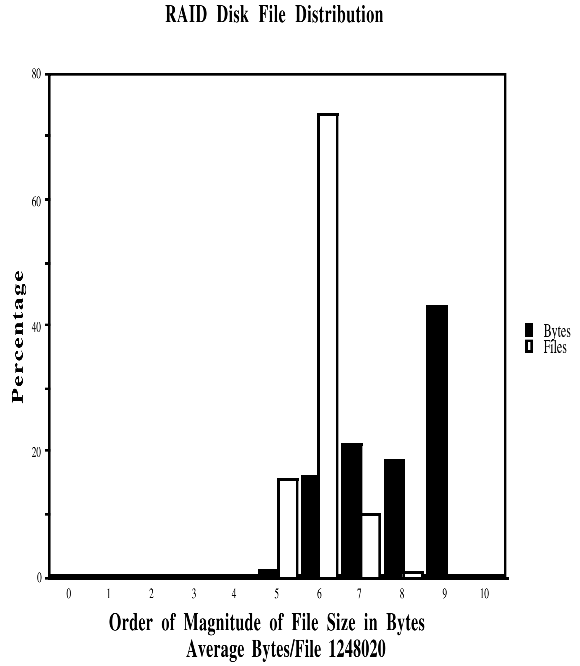

Figure 9.

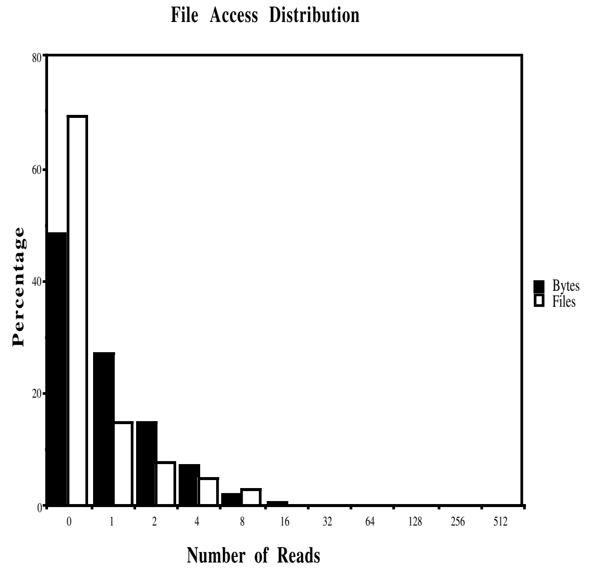

Figure 10.

Figure 10 shows that almost seventy percent of the files, about half the data, are written but never read back from the file server. This should not be taken to mean that the file server processes mostly write requests. It demonstrates that a large amount of our files and data are not used for longer than they can remain in our AFS clients' caches. This can be attributed to both the AFS client caching mechanism and the fact that our clients have relatively large cache sizes.

As of 7/5/94 the system managed 43434 directories, with 100 MBytes, 591044 files, with 221 GBytes, and 18736 symbolic links, with 338 KBytes.

16. Lessons Learned

File servers with integrated mass storage support can service much more data per server than conventional servers. These systems not only save money by reducing the amount of disks needed, but can reduce the number of file servers needed as well. The number of disks that can be attached to each file server becomes less of a consideration. The issue of how many file servers are needed to meet the transaction load remains. Currently, we service as much data from two RS/6000 servers running this software as we can service with six DecStation 5000 servers running standard AFS.

As the space available for storing files increases, so does the average file size. Between November 1992 and July 1994 the average file size has grown from 51 KBytes to 338 KBytes. However, the median file size has not changed. This tells us that users are making use of the fact that they can now store more large files than ever before, but the typical file size hasn't changed.

File servers have more information on when a file is likely to be used again than the mass storage systems they use to store the bytes. Often the storage systems come with software which attempts to determine when files should be moved between its high and low performance media. The assumptions they use in that determination won't take into account the fact that the file server is caching the file on other storage systems that it may consider higher priority, and that AFS clients are also caching that data. For tape management systems, we find it best to issue a directive to indicate that it is relatively unlikely that it will receive any read requests for a file recently stored there. Such files are already on a random access storage system and probably in an AFS client's cache as well.

Many systems block writers when there isn't enough free space to continue. The idea is that files will be migrated to free up space and then the writers will be woken up. This does not work for AFS because there is a limited number of execution threads on the server and it ends up blocking out all client operations while it waits for space to be freed up. In addition, the AFS clients get stuck waiting for RPCs to complete to the server and AFS clients do not have the ability to interrupt Fetch and Store operations. In this architecture, if the server could keep up with the demand, it wouldn't need to block. If it can't, there is no reason to expect that blocking the writer will accomplish anything in a reasonable amount of time. Allowing servers to share storage systems does a great deal in handling bursts in load, but when space runs out, a failure should be returned to the clients. A possible improvement to this would be the ability for clients to retry store operations later. This shouldn't be difficult as the client is already caching the file, but would require modifications to AFS client software.

As with all mass storage systems where unused files are not likely to be on low latency storage, users learn not to do certain operations. A user will rarely make the mistake of typing 'file *' across a bunch of old files more than once. This learning process no doubt reduces load to mass storage system and further reduces the likelihood that old files are ever read again.

Even though the file server has longer code paths than standard AFS, the performance for most operations is at least as fast. This is true because in our experience AFS performance is limited by the RX transport layer and the ability of the server to process its UDP packets. This has remained true in the latest release of AFS, version 3.3a, when using an FDDI network. The server CPU saturates when trying to send a large number of UDP packets per second over FDDI. Therefore the extra server code does not have a noticeable effect. It is also possible to get better performance than standard AFS because of the ability to use faster media and I/O interfaces that are optimized for a certain file size range. For example, our RAID-3 disks are much faster than our SCSI disks when transfering files over two megabytes in length. These RAID-3 disks can not be accessed from standard AFS because they do not have a UNIX file system structure. When running performance tests with a CRAY C90 client, an RS/6000-580 server, and an FDDI network, there is no measurable performance difference between standard AFS and Multi-Resident AFS when reading and writing files. Writes happen at about 1.2 MBytes/Sec. and reads at about 1 MByte/Sec.

There are two operations that are noticeably slower than in standard AFS, one being file deletion and the other salvaging. Since files generally reside in more than one storage system, it takes longer to delete them. We will be addressing this problem by making file deletion asynchronous. The user will get an immediate return once the RPC has been processed, but the actual file deletion will occur at a later time. This should make file deletion faster than in standard AFS, at least from the users' perspective. The salvager is slower for several reasons. There are more storage systems for which the list of files residing there must be obtained. There is more volume meta-data than in standard AFS which must be checked for validity. For the system described earlier, we have found it to take 1.5 hours to salvage all of the volumes in the cell. The bulk of this time is spent running the ListFiles routine across the storage systems.

It has been suggested that performance could be improved if the server pre-fetched a whole directory at a time. The idea is that if a user accesses a directory, they are highly likely to read files in that directory. Analysis of our file accesses indicates that while it's true that a file in that directory is likely to be read, most of the files won't be. The net effect of this strategy is to do much more I/O on the server and to use much more high priority storage than is strictly necessary. The overall performance of the system actually goes down. A mechanism that might allow this to work effectively would be the addition of affinity groups. A file could be added to one or more groups of related files to indicate that they are referenced as a group. This ability would have another potential performance improvement when the files are out on tape. Many mass storage systems that offer media with a high latency to mount make attempts to reorder requests so that they are grouped by tape, thereby reducing the number of mounts. Since AFS clients do not send bulk requests for file fetches to the server, there is currently no ability to take advantage of this feature. With the addition of affinity groups, related files could be put on the same tape.

When configuring the system, we had to choose how old a file must be before it gets an additional residency on our tape system. The shorter this time, the faster one can free up space when needed. However, if a file is going to be temporary, adding a tape residency is a waste of resources. We found that a minimum age of six hours seems to avoid having most of the temporary files added to archival storage.

It was necessary to do something to address the need to backup files in the event that a user accidentally deleted their data. Using the AFS backup volume to provide access to the previous night's file system handled most, but not all, cases. We did not have the option of doing full backups because of the scaling problem. Additionally, most of our data was out in archival storage, requiring a tape mount for almost every file just to dump its contents. There was no possibility of finishing volume dumps within 24 hours. What we chose to do was to dump the meta-data for all files, but only the data that was resident on the local disk. Since most of our files are small, they tended to land on the local disk and were included in the dump. For the larger files, we took advantage of the fact that our archival storage system supported both soft and hard deletion. When AFS files were deleted, they were still on a tape until an administrators did a hard deletion. Having the meta-data for all files in the dumps allowed us to find files that had been deleted by users.

Currently, the data must still flow through the file servers. We are adding the ability to do third party transfers to eliminate this potential bottleneck. This will allow data to flow directly between the client and the storage system without passing through the file server's memory. This feature requires AFS client modifications so it will not be available to standard AFS clients.

17. Summary

The Multi-Resident AFS architecture successfully integrates a wide variety of mass storage systems with a distributed file system. Its ability to separate the user interface and access semantics from the mechanisms by which data is stored make the architecture very flexible. We believe it addresses the issues involved in bringing mass storage systems into more main stream environments.

18. Acknowledgments

UNIX is a trademark of Unix Systems Laboratories.

19. References

[11 Collins, M. W., Mexal, C. W., "The Los Alamos Common File System," Tutorial Notes, Ninth IEEE Symposium on Mass Storage Systems, October 1988.

[2] Tweten, D., "Hiding Mass Storage Under Unix: NASA's MSS-II Architecture," Tenth IEEE Symposium on Mass Storage Systems, pp 140-145, May 1990.

[3] Foster, A., Habermehl, D., "Renaissance: Managing the Network Computer and its Storage Requirements," Eleventh IEEE Symposium on Mass Storage Systems, pp. 3-10, October 1991.

[4] McClain, F., "DataTree and UniTree: Software for File and Storage Management," Tenth IEEE Symposium on Mass Storage Systems, pp. 126-128, May 1990.

[5] Antonelli, C. J., Honeyman, P., "Integrating Mass Storage and File Systems," Twelfth IEEE Symposium on Mass Storage Systems, pp. 133-138, April 1993.

[6] Quinlan, S., "A Cached WORM File System," Software Practice and Experience, Vol 21, No. 12, pp. 1289-1299, December 1991.

[7] Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S.H., Smith, F.D., "Andrew: A Distributed Personal Computing Environment," Communications of the ACM, Vol 29, No. 3, pp. 184-201, March 1986.

[8] Satyanarayanan, M., "Scalable, Secure, and Highly Available Distributed File Access", IEEE Trans. Computers, pp. 9-21, May, 1990.

[9] Ousterhout, J., DaCosta, H.L., Harrison, D., Kunze, J., Kupfer, M., Thompson, J, "A Trace Driven Analysis of the Unix 4.2 BSD File System," Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Orcas Island, December 1985.

20. Suggested Reading

"Mass Storage System Reference Model: Version 4," edited by Sam Coleman and Steve Miller, IEEE Technical Committee on Mass Storage Systems and Technology, May 1990.

"A Joint European Mass Storage Specification Effort," edited by Dick Dixon, European Centre for Medium-Range Weather Forecasts, Thirteenth IEEE Symposium on Mass Storage Systems, pp. 110-112, June 1994.

21. Biographies

Jonathan S. Goldick received a B.S. degree in Physics and a M.S. degree in Electrical Engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 1988 and 1989, respectively. He is currently a Senior Technical Specialist at the Pittsburgh Supercomputing Center, Pittsburgh, PA, USA. His research interests include the design, development, and analysis of distributed mass storage systems.

Kathy Benninger has been a hardware systems engineer with the Scientific Support Group at the PSC for four years. Her focus is on mass storage system specification and integration and on supporting systems for scientific visualization. She received a BSEE from Carnegie Mellon University in 1984.

Christopher Kirby is a Senior Research Systems Programmer at the Pittsburgh Supercomputing Center and has worked there since 1990. He received a B.S. in Applied Math/Computer Science at Carnegie Mellon University in 1988 and a M.S. Computer Science from New York University in 1990. He was previously employed at At&T Bell Laboratories. His professional interests include design and development of mass storage and distributed file systems.

Christopher J. Maher is Director of Scientific Support at the Pittsburgh Supercomputing Center. He has been with PSC since 1988, as a Scientific Specialist, Scientific Support Manager and Scientific Support Director. At PSC he has over seen the installation of the Centers first mass storage system, supervised the development of multi-resident AFS, and is responsible for most of PSC's software development projects. Maher held postdoctoral fellowships at the Massachusetts Institute of Technology and Carnegie Mellon University prior to joining PSC's staff. He received is B.S., M.S. and Ph.D. from Carnegie Mellon University in Physics in 1980, 1982 and 1986 respectively. Maher has authored and co-authored numerous scientific publications.

Bill Zumach is a Senior Research Systems Programmer at the Pittsburgh Supercomputing Center, where he has been employed since 1992. He works on design and development of distributed files systems. He received a B.A in Mathematics from the University of Minnesota in 1987. He previously worked at the Astronomy Department of the University of Minnesota designing data collection and analysis software. His research interests include mass storage and distributed file system design and analysis.