THE LIVERMORE DISTRIBUTED STORAGE SYSTEM: REQUIREMENTS AND OVERVIEW

UCRL--102664

DE90 007255

Carole Hogan, Loellyn Cassell, Joy Foglesong, John Kordas, Michael Nemanic, George Richmond

Lawrence Livermore National Laboratory
Livermore, California

ABSTRACT

This paper outlines the requirements for a large-scale, distributed storage system, and describes how a system being developed at the Lawrence Livermore National Laboratory, called the LINCS Storage System (LSS), meets those requirements. The LSS provides distributed storage in an environment that includes supercomputer host machines and a large-capacity, hierarchical central repository. The paper defines the key terms and concepts of the LSS and gives a brief architectural overview of the system. Major system components are described in more detail. The contributions of the LSS to the design of large-scale, distributed storage systems are identified and discussed throughout the paper. Finally, the current status of the development of the LSS is described, including a brief summary of hardware and software components.

INTRODUCTION

The computing environment at the Lawrence Livermore National Laboratory (LLNL) includes supercomputer host machines that weekly generate 50 gigabytes of data for long-term storage. These large-capacity storage needs, constrained by budgetary limits, have led to the integration of host, central, and archival storage systems into one transparent, logical system called the LINCS Storage System (LSS) (see Figure 1).** The significant contributions of the LSS to the design of large-scale, distributed storage systems, explained in detail in later sections of this paper, include:

- System components modeled on an extended client-server model, for reasons of modularity, extensibility, portability, and transparency.
- A pipelined, message-passing communication protocol between system components, to permit efficient asynchronous communication and provide part of the flexibility necessary to move components to any machine in the system, as needed.
- Separation of control messages from data messages in communication between system components, to improve performance and support movement of third-party and pipelined data.
- Automatic migration of bitfiles and directories between host and central and between central and archival storage, to improve performance and storage-space management, and to aid in presenting a single, integrated view of all levels of the storage system.
- Caching the entire bitfile when an access to it is first made, to improve performance by saving the overhead of multiple requests for repeated access to the same resource.
- Portability of system components as a separate goal, to ensure the flexibility that permits moving components to any machine, regard-

** Significant terms and concepts used throughout this paper are defined in the GLOSSARY Section at the conclusion of the paper.

less of the operating system running on that machine.

- Separation of the naming service that maps human-oriented names into machine-oriented identifiers in a single system component, rather than replicating it in all components, for reasons of flexibility, extensibility, simplicity, and performance.

Other contemporary storage systems have succeeded in integrating multiple storage levels, including the SUN Network File System (NFS),[1] Carnegie-Mellon University's Andrew File System (Andrew),[2] the Los Alamos National Laboratory Common File System (CFS),[3, 4] the NASA-Ames Research Center Mass Storage System (MSS-II),[5] the National Center for Atmospheric Research Mass Storage System (NCAR),[3, 4] and the University of Michigan's Institutional File System (IFS).[6] Each of these systems has made significant contributions to certain aspects of storage system design and to certain types of storage environments. NFS integrates workstation file systems by permitting file access to several systems from a single machine. Andrew extends the workstation environment by integrating thousands of workstations into one storage system. IFS extends the Andrew effort by providing a file system that supports tens of thousands of workstations. It also provides an archival storage system as a backup for the workstation file systems. CFS performs migration of files from central to archival storage in a supercomputing environment. MSS-II provides high-performance transfers between central storage and host supercomputers. NCAR provides a fast data path from device controllers in central storage directly onto a high-speed network to the supercomputer host machines.

The LSS contribution is to provide full, transparent integration of large-scale storage across multiple supercomputer host systems and across all levels of the storage hierarchy depicted in Figure 1. For system users and designers, there are several advantages to a fully integrated, distributed storage system. For system users, including those who write and those who utilize application programs, the primary advantage is simplicity. They see the same directory structure from any vantage point. They need to know only one interface for accessing bitfiles and directories, whether local or remote, because location is transparent. All bitfiles and directories are accessed as if they were local. Thus, the LSS provides a user view of a single storage system.

By contrast, in systems like NFS and CFS, users must be aware of the location of their resources. In NFS, the location of the particular file system where the desired files reside must be known and mounted before the files can be accessed. (However, once mounted, file access is transparent.) In CFS, the user must be aware of whether his files are local or on CFS. A user must explicitly request files to be moved between local storage and CFS.

Having one transparent, distributed storage system also means that LSS users need not be familiar with different storage system characteristics such as bitfile lifetime and maximum bitfile size. They need not be concerned with tasks that are typically required when bitfiles must be explicitly stored. These tasks include breaking large bitfiles up into smaller pieces to meet size limitations on a particular medium and copying bitfiles to permanent storage so they won't be destroyed when space on that medium becomes scarce. Also, unlike the case of UNIX, users are not faced with the prospect of being unable to run if space becomes scarce on a particular medium.

For system designers, the primary advantage of a fully integrated, transparent, distributed storage system is resource management flexibility. The freedom to migrate bitfiles allows the system to optimize performance, maximize media utilization, control network traffic, incorporate new technologies, and minimize operational costs. High performance at low cost is achieved by keeping active bitfiles on expensive, fast-access storage and less active bitfiles on cheaper, slower-access storage. Also, with only the inactive bitfiles on off-line storage, there are fewer fetches to that medium, reducing the amount of storage space and manpower required to deal with it. Moreover, the load on the network is significantly reduced, since there are fewer fetches to lower levels of storage.

The remainder of this paper outlines the requirements of a large-scale, distributed storage system, presents an architectural overview of the LSS, and discusses how the LSS components, defined below, meet these requirements.

REQUIREMENTS

The users of a storage system generally desire unlimited storage capacity and fast access to their stored data. These desires are frequently offset by the high cost of fast-access storage hardware. One solution, adopted in the LSS, is a hierarchy of storage media with varying access times. Good architectural design hides the different performance characteristics of the media in the hierarchy and minimizes software development and maintenance costs. Thus, the requirements of a distributed storage system result from users' needs, cost constraints, and sound architectural design principles.

While user needs and cost constraints can vary from site to site, there are several general requirements that remain constant. These requirements are tabulated below.

Performance — The time necessary to access resources must be minimized. Furthermore, to give users a single view of the system, the time required to complete an operation should be as independent of the location of the resource as possible.

Capacity — The system must be able to store ever-increasing amounts of data as computers with greater computing capacity become available. The system should also provide users with the ability to store bitfiles without size restrictions.

Transparency — The system must provide users a single, homogeneous view of storage for ease of use. There are four main dimensions of transparency:
1) Syntactic transparency—access to both local and remote resources use the same operations and parameters.
2) Semantic transparency—behavior of storage operations is independent of resource location and error types.
3) Name transparency—a resource can be accessed with the same name from any site in the network.
4) Location transparency—location of a resource should not be inferable from its human-oriented name.

Integrity — The system must preserve data integrity, software bugs and hardware crashes notwithstanding.

Synchronization — The system must provide mechanisms for sequencing access to shared resources to preserve data consistency.

Availability — The system must provide mechanisms that ensure the continuous availability of system components and resources.

Security — The system must provide mechanisms that implement both mandatory and discretionary access control policies to protect user data.

Naming — The system must provide a reliable way to name resources at various system levels, from human-oriented names to machine-oriented identifiers, so that users can store, locate, and retrieve their data and the system can locate dynamically relocatable resources.

Resource management — The system must provide mechanisms, including accounting and allocation controls, to efficiently man-

	age resources at several levels.
Error control and recovery	The system must provide controls to detect and recover from failures at many levels.
Modularity	The system software components must be written as independent modules with well-defined interfaces for ease of system construction and maintenance, and to provide the building blocks to recursively expand the system, if desired.
Portability	System modules and utilities must be written in a high-level, portable language and be built on libraries in such a way that the system can run on a variety of operating systems and hardware configurations.
Extensibility	The system must be written flexibly so that it can be easily expanded to include newer technologies as they are developed.
Random read/ write access to bitfiles	The system must provide users with random read/ write access to their bitfiles to eliminate the delay inherent in sequential access and to provide the flexibility that many applications require.
Standards	The system should adhere to standard communication protocols and provide access to and from other file systems.

The various ways that the LSS meets these requirements are discussed throughout the remainder of this paper. The next section provides an overview of the key architectural concepts of the system.

ARCHITECTURAL OVERVIEW

The key components of the LSS are taken from the IEEE Mass Storage Reference Model (Reference Model).[7] They include name servers, bitfile servers, storage servers, and bitfile movers. The servers are built on the message-based client-server model, including lightweight tasking for multiprocessing and concurrency, because of its particular suitability for a distributed storage system. The client-server model, its suitability for distributed systems, and the functionality of the LSS components are discussed in this section.

Client-Server Model

The LSS is built upon a message-based architecture using the client-server model, as shown in Figure 2. In this model, the client process prepares a message containing the identifier of the object to be accessed and the operation to be performed. The request message is sent to the server, where it is processed and a response message is generated. Each server or object manager presents to its clients an abstract object, defined by three essential parts: a logical representation or data structure, a set of operations that can be performed on the logical representation, and legal sequences of the operations. The implementation details of the physical representation are known only by the server. Thus, the abstract-object mechanism allows each server to choose the implementation best suited to its environment. Two servers manage the same type of object if their abstract-object definitions are the same, regardless of the method they choose to implement the object.[8]

An example of a familiar abstract object, the UNIX file, has a logical representation of a header and a byte stream. The set of operations includes *stat*, *open*, *close*, *read*, *write*, and *seek*. The legal sequence of operations requires, for example, that a file must be opened before it can be read. A server that implements the UNIX file abstract object may choose to store the header information on magnetic disk, which is easily accessible, and to store the body or byte stream in noncontiguous segments on magnetic tape.

The syntax of the messages sent between clients and servers is uniform. A request message contains an operation identifier and its

parameters. The reply message contains the results from the operation, including error indications. Bulk data may flow in separate communications for efficiency and flexibility, as described later in this paper.

The client-server model is particularly well suited for a distributed storage system because it fulfills the requirements of modularity, portability, syntactic and semantic transparency, and extensibility. The model supports modularity because the details of the storage devices are hidden behind the interface by the server modules that implement the abstract object. Therefore, clients of the system need not concern themselves with the differences between the various physical media. Also, all servers managing the same type of abstract object support the same well-defined interface. Message-based communication aids in achieving the requirements of modularity and portability. Message passing permits clients and servers to execute asynchronously rather than blocking on procedure calls. It also provides the flexibility to move servers and clients to other machines because they are not dependent on such things as global variables or particular databases.

Syntactic transparency is provided because access to an object, whether local or remote, occurs through the same operations and parameters. Semantic transparency is achieved because the behavior of storage operations is independent of operand location and types of errors. Extensibility is provided by the underlying message-passing system and the modularity of the client-server model. Modularity, in particular, supports extensibility because it permits more complex systems to be built recursively. It also allows for the easy replacement of older modules to incorporate new technologies.

Components of the LSS Architecture

The key components of the LSS are the bitfile servers, the storage servers, the bitfile movers and the name servers (see Figure 3).* The bitfile servers provide access to bitfiles. The storage servers allocate and access physical storage. The bitfile movers transfer bitfile data between channels, such as networks, or devices, such as disk or tape. The name servers map human-oriented object names to machine-oriented object identifiers. There are several architectural features of these servers that make them particularly well suited as components in a distributed storage system. First, they conform to the server portion of the client-server model defined above, thereby meeting the requirements of modularity, portability, syntactic and semantic transparency, and extensibility. Second, they also extend the model through the use of lightweight tasking to provide multiprocessing and concurrent access (see Figure 4). Light weight tasks also have less state than traditional processes such as UNIX processes. Because the cost to create, destroy, and perform a context switch with lightweight tasks is less expensive, performance is increased and memory overhead is reduced. Third, the model-supported, asynchronous message passing permits pipelining multiple requests of unlimited size for performance efficiency. Fourth, the particular format used in the message passing between clients and LSS servers provides flexibility and extensibility. The parameters are not position dependent in the LSS syntax, so various operations can be easily extended. Also, there is one standard network format for each parameterized datatype, allowing machines of different architectures to communicate easily.[9]

The LSS servers are organized in a hierarchy that corresponds to the three levels of storage in the system: host, central, and archival (see Figure 3). Objects are stored at each level of the hierarchy, according to size and frequency of access. Servers cooperate to provide transparent access to the objects. They do this by moving the objects between host, central, and archival storage, as appropriate. Host bitfile servers interact with the central bitfile server to move bitfiles between host and central storage, or between host bitfile servers. The central and archival bitfile servers interact to move bitfiles between their levels. Similarly, host name servers cooperate with the central name server to move directories between the host and central directory systems, or between host directory systems. These interactions are invisible to the client.

* The storage servers have been incorporated into the bitfile servers and, hence, are not shown separately in Figure 3. The bitfile movers are represented by the arrows between the servers.

Figure 5 presents a brief scenario to describe how a file is accessed in the LSS. Suppose a user on host A desires to read a bitfile named /a/b/c. Suppose further that the file is physically located in the archive bitfile server's database and that the directory entry containing the bitfile identifier is located in host B's directory system. To the user, the read is accomplished as follows. A process running on the user's behalf on host A contacts the name server running on A to fetch the bitfile identifier for the bitfile (message 1). Once the process obtains the identifier from A's name server, the read request is completed by the process sending the request to the bitfile server running on host A, along with the identifier obtained from the name server (message 7).

To users, the local name and bitfile servers represent the entire storage system. Access to any object, wherever located, is accomplished through requests made only to the local servers. In the current example, the fact that the desired directory entry and bitfile have been obtained from remote servers is invisible to the user on host A. To actually accomplish the read, the system took the following steps. First, the name server running on host A determined that the directory entry for /a/b/c was not local, so it requested its only supporting server, the central name server, to fetch the entry (message 2). The central name server, through tables kept for this purpose, determined that the directory where the entry was cataloged was located on host B. The central name server then sent a message to B's name server requesting the identifier (message 3). B's name server returned the directory containing the identifier to the central name server (message 4) which, in turn, returned it to the name server on A (message 5). The central name server also updated its tables to show that a read-only copy of the directory was on host A. The name server on A then returned the identifier to the user's process from the directory provided by B (message 6).

Next, the user's process on host A sent the read request with the bitfile identifier to the bitfile server running on host A (message 7). The server determined that the bitfile was not local, so it requested its only supporting server, the central bitfile server, to read the file (message 8). The central bitfile server determined, through tables kept for this purpose, that the bitfile was located in the archive bitfile server's database, so it sent the read request to that server (message 9). The archive bitfile server located the bitfile in its database and returned it to the central bitfile server (message 10) which, in turn, returned it to the bitfile server running on host A (message 11). The central file server also updated its tables to indicate that a read-only copy of the bitfile was on host A. The server on A then provided the bitfile to the user's process (message 12).

Should a user on host B now make a read request of its bitfile server for the same bitfile, the same sequence of steps will take place, with the following two exceptions. First, the name server running on B will be able to fetch the entry directly out of its database. Second, if it no longer has a copy of the bitfile, the central bitfile server will have the choice of reading it from either host A or the archive. The archive will be chosen since it is on the same machine as the central bitfile server, making communication with it faster than with the bitfile server on host A.

To extend the scenario further, if a user on host C now desires to write file /a/b/c rather than to read it, the system will take the same steps it took for the user on host A when the original read request was made, with the following exception. Since writing to a file modifies it, the central bitfile server must inform the bitfile servers on hosts A and B that their copies of the file are no longer current. Therefore, further read requests by the users on A and B will result in the same sequence of steps taken as described in the original scenario, except that the bitfile will be read from host C rather than from the archive.

An important caching design consideration is how much of a resource to cache at once. It is a key LSS design feature that an entire bitfile is cached when an access to it is first made. This is unlike NFS, which fetches only small portions of a file at a time, so that several requests must be made to obtain the entire file. The LSS designers chose to cache entire bitfiles at once because (1) host storage space can accommodate entire bitfiles, (2) experience indicates that the entire bitfile will probably be accessed,[10] and (3) since the entire bitfile will be accessed, it is a performance improvement to cache it all at once and save the overhead of

multiple requests to obtain a single resource.* Similarly, an entire directory is cached when an access to it is first made. However, experience with caching directories is meager. Access patterns for directories may prove to be sufficiently different to justify caching only a portion of a directory at a time. Should this be the case, the directory caching algorithms can be modified to take advantage of the actual access patterns.

The following sections describe the key components of the LSS in more detail, particularly discussing how these components meet the requirements given above.

DESIGN

In this section, requirements issues concerning the two major LSS components, the name server and the bitfile server, are covered first. Requirements issues common to several or all of the LSS components are then discussed.

Name Server

In the Reference Model, the basic purpose of a name server is to meet the naming requirement: to map a human-oriented object name ("string") to a machine-oriented object identifier ("identifier") which can then be used to access an object.[11, 12] In the LSS, that purpose is fulfilled by cooperating directory servers. Directory servers manage abstract objects called directories. Each directory consists of a descriptor and a body. The descriptor contains administrative information such as the number of entries in the body and the last time the body was modified. The body consists of string/identifier pairs, or entries. The pairing of strings and identifiers in directory bodies constitutes the mapping from human names to object identifiers that is central to a name server. Since directories can store entries for any type of resource, identifiers to directories can themselves be entry components in a directory. This allows the creation of arbitrary directed graphs. As a result, strings become links in pathnames, and mapping becomes pathname resolution. Pathname resolution can span directory servers (see Figure 6). Operations on directories include functions such as *create*, *insert*, *delete*, *list*, *fetch-identifier*, *interrogate-descriptor*, and *change-descriptor*.

Network-Wide Naming Mechanism. A key design feature of the LSS directory servers is that they provide a network-wide naming mechanism for the objects they catalog. This feature fulfills the requirements of name and location transparency. Specifically, name transparency requires that the same name resolve to the same object from all sites in the network, so that the user need not be concerned with the site he logged onto in naming his objects.[13] The directory server supports name transparency by providing a logically single, directed-graph directory structure. Location transparency means that the name of an object need not change when the object is moved. Embedding the current location of an object in the object's name to facilitate finding it violates location transparency.[13] The directory server supports location transparency by mapping the human name to a globally unique object identifier.

Performance. An obvious concern with a network-wide naming service is performance. Without optimization, network accesses to a single directory structure can be unacceptably slow. In the LSS directory server, performance is optimized through the caching and migration of directories. Specifically, access to the directory structure is synchronized by the directory server running on the central storage system. Cooperating directory servers run on each of the host machines. When a directory structure is accessed from one of the host machines, the specific directory involved is cached from the central directory server to the server running on the accessing host. Thereafter, until the directory is purged or is migrated back to the central directory server, all accesses to it from that host will occur locally. Migration will occur if the directory has been modified and either it is no longer in use on the host or write access to it is requested by a different host. In all cases,

---

* Bitfile sizes are increasing as supercomputer technology advances. As bitfile sizes increase, the advantages of caching entire bitfiles at once diminish. If the advantages diminish enough, the LTSS bitfile servers can be modified to cache significant portions of bitfiles at a time rather than entire bitfiles.

accesses to directories are synchronized to protect the consistency of the data.

Separation of Human Naming from Other Object Servers. Another key design feature of the LSS directory servers is that they are entirely separate entities from the other LSS servers.

Advantages. Separating human naming from object servers other than the directory servers is not unique to the LSS.[14] This design choice has several advantages:

- The directory servers can serve as a common mechanism for naming different types of objects, thus facilitating total system extensibility.
- The other servers can function in a variety of user environments, since they are independent of human-oriented naming conventions.[11]
- The other servers can be optimized to manage their own objects without the need to deal with human-oriented names.[12]
- New objects can be named in the same way as existing objects.[15]
- In addition to the directory servers, it allows for several forms of application-dependent and general-purpose higher-level name services that can also catalog identifiers.[12]
- Applications can create, access, and destroy objects without ever storing the identifiers in any name server.[15]

Disadvantages. Depending on the implementation techniques chosen, separating the naming functionality can also have disadvantages. These can include performance degradation, a more complicated resource-management scheme, and more complicated security mechanisms. For a complete discussion of these issues, see reference 16.

Bitfile Server

In the Reference Model, the purpose of the bitfile server is to provide access to bitfiles. The LSS bitfile server does this. The abstract object it manages is a bitfile, which consists of a descriptor and a body. The descriptor contains named fields with various restrictions on access. These fields specify attributes of the bitfile, such as time of creation, length, and body location. The body is an unstructured stream of bits available for random reading or writing by the client. Operations on bitfiles include functions such as *create*, *read*, *write*, *change-descriptor*, and *interrogate-descriptor*.

Performance. A very important concern when analyzing a storage system that spans machines is performance. The LSS cooperating bitfile servers use several techniques to enhance system performance, both on a single machine and across several machines, to meet the performance requirement. These techniques include: caching and migration, separation of control messages from data messages, and certain optimization mechanisms.

Caching and Migration. A major technique used to enhance system performance is the caching and migration of bitfiles. The particular storage medium for a given bitfile depends on the current demand for the bitfile, its size, and the time it was last accessed. When a client on a host accesses a bitfile located in central storage, the local bitfile server cooperates with those on the central storage machine to cache the bitfile locally for fast access. Although the time to complete the first access will vary depending upon the current location of the bitfile, future accesses will be fast because the bitfile is now local.*

Access to the same bitfile from several hosts is coordinated by means of locking. In general, a bitfile can be local to several hosts for simultaneous read access. However, to preserve data consistency, before a bitfile is modified, all read-only versions on other hosts are first invalidated. When the modification is complete,

---

* A further optimization utilized by MSS-II allows access to the bitfile as soon as part of it is received by the host. Currently in the LSS, access is not allowed until the entire bitfile is transferred to the host. However, plans for the future do not preclude this optimization.

the updated version moves to the hosts when new access is requested. For a complete discussion of the locking mechanism, see reference 16.

Infrequently accessed bitfiles migrate down to the archival bitfile server, where they are placed on cartridge tape. Initially, they reside in an on-line repository. Eventually, dormant bitfiles migrate from the on-line repository to cartridges stored in an off-line tape vault. When a bitfile in the vault is accessed, it first moves to the central bitfile server and then to the appropriate host bitfile server. Each server manages its scarce resources by moving bitfiles up or down the hierarchy, as shown in Figure 7.

Separation of Control and Data. Another performance-enhancing technique is the separation of control and data (see Figure 8). Bitfile access requests are sent as control messages to the bitfile server. Requests to read or write the bitfile establish separate data connections over which the data is moved. The bitfile server itself never handles the data. It performs the necessary space allocation actions through the incorporated storage server and informs a subordinate server, the bitfile mover, of the data movement parameters. The bitfile mover performs the actual data movement from device to device or between device and network connection, avoiding expensive memory-to-memory data copies. Separation of control and data messages also supports third-party control of data transfers and permits pipelining of the transfers.[17]

Optimizations. The central bitfile server is designed to deliver higher performance than the archival bitfile server. The central bitfile server manages an active subset of the bitfiles in the entire system on fast-access disks. Because it manages only active bitfiles, it can keep in memory the look-up table that maps the bitfile identifier into the location of the descriptor. The archival bitfile server manages bitfiles on slower-access tape cartridges. However, its look-up table is kept on disk for faster access. Also, access to the bitfiles stored on cartridge tape is optimized by locating the descriptors on disk, providing fast queries and updates to descriptors as well as flexible relocation of data fragments.

Resource Management. The LSS bitfile servers use two mechanisms to manage their bitfiles in meeting the resource management requirement. These mechanisms are bitfile segmentation and speedy migration to tape.

Segmentation. Bitfile space is managed by maintaining a list of segments in each bitfile descriptor. Each segment contains a portion of a bitfile, on either disk or tape. As a bitfile grows, new segments are allocated. If a bitfile has too many segments, the segments are consolidated by compacting data on disk. The benefit of segmentation is easy media space management and the ability to transfer large, contiguous blocks of data. Its alternative, storing an entire bitfile contiguously, requires expensive media-compaction techniques when the space remaining is not large enough to hold the next bitfile. Further, the LSS segmentation design is less expensive than UNIX data blocks. The UNIX blocks require more overhead to manage and more disk head movements because they are small, fixed-size structures.

Migration.[*] Bitfiles remain in the central bitfile server's disk cache until disk space becomes scarce. At this point, any bitfile that has not been updated since it migrated to the archival bitfile server can simply be removed from the central bitfile server's disk. Migration is not required because the bitfile has not been modified. Bitfiles that have been modified are candidates for migration. Once bitfiles have been migrated, they can be removed from disk. Removal is based on a formula that weighs size and time of last access. Large or old bitfiles are removed first. The central bitfile server migrates bitfiles to the archival bitfile server as soon as one tape cartridge of data has been accumulated. A tape cartridge may contain several bitfiles or only a portion of a large one. The goal of this process is to keep a certain fraction of the disk cache space free for new data and to maintain a good hit rate to increase the overall performance of the system.

---

[*] This migration discussion specifically concerns central storage; however, the same procedures are also utilized between host machines and central storage.

Similarly, the archival bitfile server manages space in the robotic tape-cartridge system to meet performance requirements by identifying the oldest bitfiles on cartridges in the robotic system and migrating them to off-line tape volumes. The active data left on a volume can be consolidated with other active bitfiles to fill tapes that stay in the robotic system. This migration process serves the same purpose as in the central bitfile server case. Active bitfiles stay easily accessible in the robotic system while inactive bitfiles are moved to the tape vault.

Capacity. The LSS bitfile servers meet both facets of the capacity requirement. Hardware is the limiting factor in meeting the increasing storage demands made by each new generation of supercomputers. Specifically, network and device bandwidths and cpu power may need to be upgraded as supercomputer output outstrips the hardware's capacity to handle it. The bitfile servers provide virtually unlimited size with their flexibility to expand descriptors to add as many segments to the list as is necessary for each bitfile. If necessary, these segments can span disk or tape volumes.

Random Access. Archival media such as cartridge tape do not provide random write access. Therefore, the random read/write access requirement is met in the LSS by allowing clients access to bitfiles only through the host or central bitfile servers, both of which manage magnetic disk, a medium that does support random read and write access. The bitfile is cached from tape cartridge to disk, where the modifications are made. When the updated copy of the bitfile migrates to the archival bitfile server, the body is written onto a new volume, and the descriptor is updated to point to the new body.

Other Features of the LSS Components

There are several features common to most or all components of the LSS that are designed to satisfy the requirements of a large-scale, distributed storage system not fully addressed to this point in this paper. These requirements include: integrity, synchronization, availability, security, resource management, error control and recovery, portability, and standards. These remaining requirements are discussed in this section.

Integrity. All LSS servers have two principal mechanisms to preserve data integrity across machine and software failures. These are internal redundancy and data backup to more reliable media. The bitfile and directory servers achieve internal redundancy by dual atomic writes to separate disks of bitfile descriptors and directories. All data is backed up to archival media as quickly as practical to protect against disk failures. Bitfiles are copied to tape when one full tape cartridge of data has been accumulated. Directories are copied to tape once a day. To date, the LSS has experienced only one complete disk failure. This failure caused the loss of only a few bitfiles that had not yet been backed up to tape by migration.

Synchronization. Synchronization, which protects the consistency of the data, is employed in the caching and migration locking algorithms. Briefly, locks are used to synchronize access to distributed objects. Servers can place read locks on objects to delay modification during a series of reads, or write locks can be used to delay both reads and other modifications during writes. As a server write-locks objects on one machine, servers on other machines invalidate their copies of the locked objects. For a more complete discussion of the locking algorithms, see reference 16.

Availability. Three mechanisms are employed to increase system availability in the LSS: quick restarts, persistent clients and servers, and redundant systems. Typical restart times vary from seconds for some servers to five minutes for the central bitfile server, which must scan all of its bitfile descriptors to build search tables. Persistence through retries allows the failure of an individual component of the LSS to be overcome with only modest delays in service. Furthermore, the persistence of servers and clients relieves users from the burden of repeatedly transferring their data when a system component is unavailable. The LSS has a second central storage machine available to it which can be used in place of the primary machine if necessary.

Security. To comply with the Department of Energy (DOE) security guidelines, the LSS includes mechanisms that implement both mandatory and discretionary security policies. Two mechanisms are used to implement

mandatory policy. The first mechanism is encryption, which is used to protect the object identifier from forgery. The second mechanism implements seven DOE security levels. A policy based on these levels is enforced by each server and by the underlying communications systems. For example, objects of a higher level cannot be accessed or conveyed through a lower-level communications medium.[18] In the LSS, discretionary policy is implemented through access bits located in the object identifiers. These bits control the manner in which an object is accessed and include such permissions as *modify* access, *add* access and *read* access. (In other systems, these access bits are located in the object descriptors. The relative merits of locating access bits in object identifiers versus object descriptors is beyond the scope of this paper.)

Resource Management. All of the LSS servers implement accounting and allocation mechanisms. Generally, the accounting mechanism permits charging for storage to recover operational costs. An indirect benefit of charging is better utilization of storage space, as users tend to keep only data that is truly useful to them. The allocation mechanism is provided to ensure an equitable distribution of storage space among all users. For a detailed description of the accounting and allocation mechanisms, see reference 16.

Error Control and Recovery. Error control and recovery is achieved through several mechanisms, including log files, restart messages, persistence, and reconstruction of descriptor tables. By referring to log files kept for this purpose, several servers can restart after crashes at the point where they left off before crashing. These files record the activity of the servers at crucial moments and can be referred to when the servers begin running again. Further, some servers send messages to their principal clients, informing them that the servers are running again. Persistent clients also aid in error recovery by repeatedly sending the same request message until either a response is received from the server or it is apparent that the server is down for an extended period. As another aid to recovery, the central bitfile server is able to quickly rebuild its in-memory table of descriptors because it does not have to follow indirections to find the actual data blocks, unlike the UNIX *fsck* routine. The integrity mechanisms discussed above also provide error control and recovery.

Portability. Portability of the LSS components is achieved through the use of a high-level programming language and machine-independent interfaces to libraries. All LSS servers and utilities have been written in the C programming language. Machine dependencies are hidden in a small set of macros and library routines. The LSS runs on machines with different word sizes and bit orders.

Two libraries of routines in particular provide machine and operating-system independence. The first supports lightweight tasking, and the second provides interprocess communication. These libraries are small and can be easily ported to other operating systems including those running on most of today's supercomputers.

Standards. Standards have been a high priority during the design and development of the LSS. LLNL personnel are active in the development of the Reference Model, which is becoming the basis for a mass storage system standard. The LSS follows the design principles stated in the Reference Model.

The LSS also provides interfaces to several standard protocols. These protocols include TCP/IP, a standard network communication protocol; FTP, a standard file transfer protocol; and NFS, a de facto standard file access protocol.

THE LSS TODAY

Hardware

Currently, the host computers are Cray supercomputers. The central storage computer is an Amdahl 5868 configured as a pair of 5850s running UTS, which has a 200-gigabyte online disk cache. The archive includes five online STC 4400 robotic tape cartridge systems, housing 5 terabytes of data on 30,000 tape cartridges. The archive also includes an

additional 6000 cartridges stored in an off-line vault. The Crays and the Amdahl machine communicate via a Network Systems Corporation HYPERchannel.**

Software

Portable C language production versions of the bitfile and directory servers are complete. The host and central directory systems have been connected into one logical directory structure. However, the current directory structure still reflects that the host and central directory systems are physically separate. Currently, there is no automatic migration of directories between the host and central directory systems, but work on this mechanism is in progress.

The integration of the central and archival bitfile servers is complete, with automatic migration of bitfiles between these levels. Implementation to connect the host and central bitfile servers by adding automatic migration between these levels is in progress.

To date, unlike the NFS, Andrew, and IFS environments, there are no plans to extend the LSS to workstations. Although the system architecture is extensible to that environment, the resources required for software support are currently unavailable.

In the LSS, users can write their own code to directly access host and central bitfiles and directories. However, the system also offers an interim mechanism to bridge the gap between the host and central bitfile and directory systems. Bitfiles can be transferred between the host and central systems or between host systems by an external, persistent utility and server. Users invoke bitfile transfers explicitly, by executing the utility that translates user requests into requests to the persistent server. Bitfiles can also be transferred using standard NFS and FTP clients on host machines.

The maximum bitfile size is currently limited to the smaller of either the size of one disk or two tape cartridges. These capacity limitations will be relieved in the near future.

CONCLUSION

The requirements of a large-scale, distributed storage system have been met in the design of the LSS. Its main features include:

- System components built on an extended client-server model
- An asynchronous message passing protocol
- Separation of control and data messages
- Automatic caching and migration of data
- Movement of entire bitfiles when an access is made
- Portability as a specific design goal
- Isolation of the naming service in a separate system component

These features will result in a fully integrated, transparent, distributed storage system in a supercomputer environment.

ACKNOWLEDGEMENTS

We gratefully acknowledge the contributions of Samuel S. Coleman and Richard W. Watson in the design of the LSS and in helping us bring this paper to fruition. We also thank Mark R. Gary and Richard P. Ruef for their invaluable work in the development of the LSS. This work was performed by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48 under auspices of the U.S. Department of Energy.

GLOSSARY

archival storage	Cartridge tape on the storage machine.
backup	A copy of data kept for redundancy.
bitfile	An object consisting of a bit string of arbitrary length and a set of attributes.
bitfile mover	A manager that moves the bits in bitfiles between

	channels or designated address spaces.
bitfile server	An object manager that creates and provides access to bitfiles.
caching	The act of automatically moving an object such as a bitfile from a slower-access storage medium to a faster-access medium.
central storage	A collection of magnetic disks on the storage machine.
directory	A cataloging structure consisting of pairs of human-oriented names and machine-oriented object identifiers.
host storage	Solid state disks and rotating disks on local machines.
IEEE Mass Storage Reference Model	The document developed by the IEEE Mass Storage Committee proposing a common design for archival storage systems.
migration	The act of automatically moving an object such as a bitfile from a faster-access storage medium to a slower-access medium.
name or directory server	A manager that provides a mapping between human-oriented names and machine-oriented object identifiers; these terms are used interchangeably through-out this paper.
off-line	A storage medium requiring human intervention for access.
on-line	A storage medium not requiring human intervention for access.
persistence	The act of persisting through periods of system unavailability to accomplish an operation.
storage server	A manager that allocates and accesses physical storage.

REFERENCES

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

2. J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, Vol. 29, No. 3, March, 1986.

3. William Collins, Marjorie Devaney, and David Kitts, "Profiles in Mass Storage: A Tale of Two Systems," DIGEST OF PAPERS, Ninth IEEE Symposium on Mass Storage Systems, October, 1988, pp. 61-67.

4. Samuel S. Coleman, "Storage Architecture and Issues," *Proceedings*, Cray User Group, April, 1989, pp. 89 -93.

5. Robert L. Henderson, "MSS-II and RASH A Mainframe UNIX Based Mass Storage System with a Rapid Access Storage Hierarchy File Management System," *Conference Proceedings*, Winter USENIX Technical Conference, San Diego, 1989, pp. 65-84.

6. Private telephone communication with Elaine M. Wolfe of Transarc Corporation, The Gulf Tower, 707 Grant Street, Pittsburgh, Pennsylvania 15219, (412) 338-4400.

7. Stephen W. Miller, "A Reference Model for Mass Storage Systems", *Advances In Computers*, Vol. 27, 1988, pp. 157 - 209.

8. Anita K. Jones, "The Object Model: A Conceptual Tool For Structuring Software," in *Operating Systems - An Advanced Course*, Springer-Verlag, Berlin Heidelberg, 1978, pp. 8 - 16.

9. R. W. Watson, "Requirements and Overview of the LINCS Distributed Operating System Architecture," *Proceedings*, Cray User Group, April, 1984.

10. M. Satyanarayanan, "A Survey of Distributed File Systems," *Annual Review of Computer Science*, Vol. 4, 1989.

11. Samuel S. Coleman, "Storage in Supercomputer Environments," *Proceedings*, Cray User Group, June, 1988, pp. 423-428.

12. R. W. Watson, "Identifiers (naming) in distributed systems," in B. W. Lampson, M. Paul, and H. J. Siegert (eds.), *Distributed Systems - Architecture and Implementation*, Springer-Verlag, New York, 1981, pp. 191 - 210.

13. Peter Lawrence Reiher, *Naming Issues In Large Scale Distributed Systems*, Ph.D. Dissertation, Department of Computer Science, University of California, Los Angeles, 1987.

14. Liba Svobodova, "File Servers for Network-Based Distributed Systems," *Computing Surveys*, Vol. 16, No. 4, Dec. 1984, pp. 353-398.

15. Samuel S. Coleman, and Richard W. Watson, "Designing Archival Storage Systems for Distributed Supercomputer Environments," submitted for *Computer*, May, 1990.

16. Joy Foglesong, George Richmond, Loellyn Cassell, Carole Hogan, John Kordas, and Michael Nemanic, "The Livermore Distributed Storage System: Implementation and Experiences," DIGEST OF PAPERS, Tenth IEEE Symposium on Mass Storage Systems, May, 1990.

17. Mark Gary, "Overcoming UNIX Kernel Deficiencies In A Portable, Distributed Storage System," DIGEST OF PAPERS, Tenth IEEE Symposium on Mass Storage Systems, May, 1990.

18. J. G. Fletcher, "A Security Policy for Distributed Systems," LLNL Working Document, June 26, 1985.

FIGURES

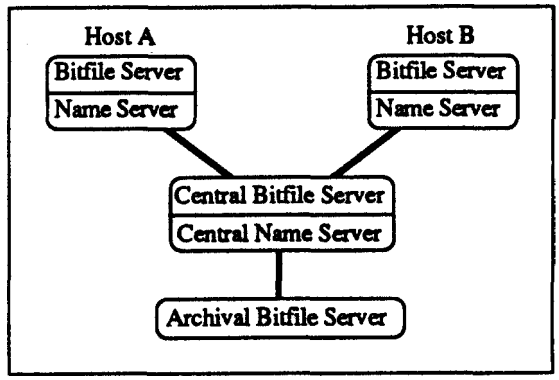

Figure 1. LSS Hierarchy

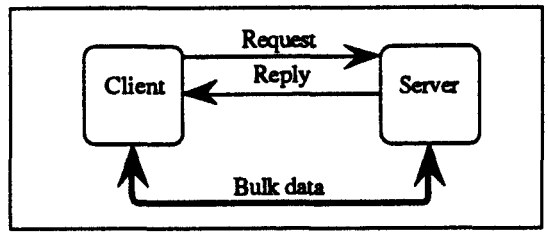

Figure 2. Client - Server Diagram

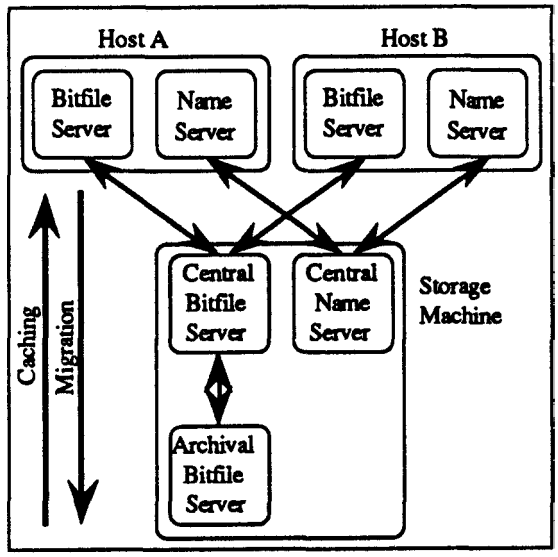

Figure 3. LSS Architecture

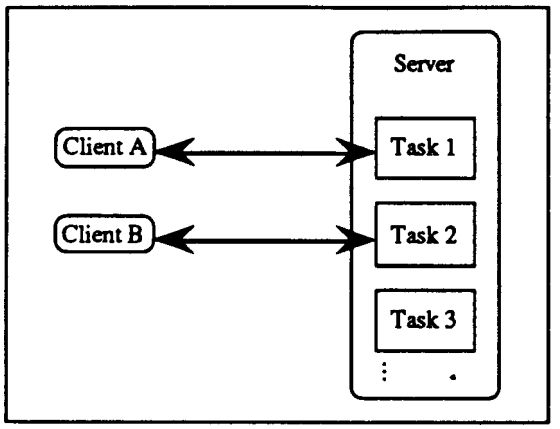

Figure 4. Client Server Tasks

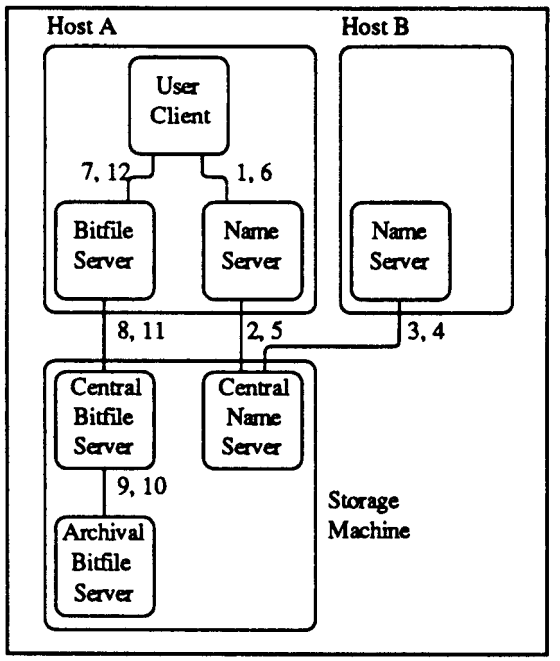

Figure 5. Message Passing Between Servers

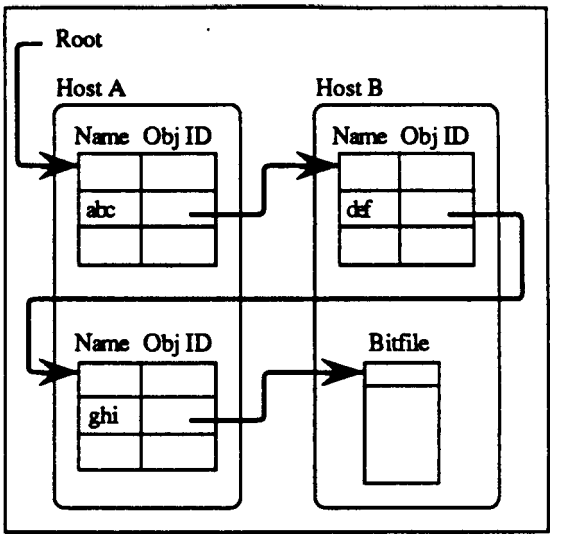

Figure 6. Pathname abc/def/ghi Resolution

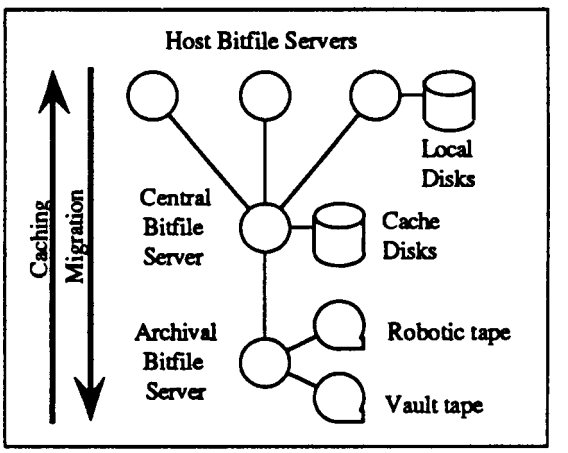

Figure 7. Bitfile Caching And Migration

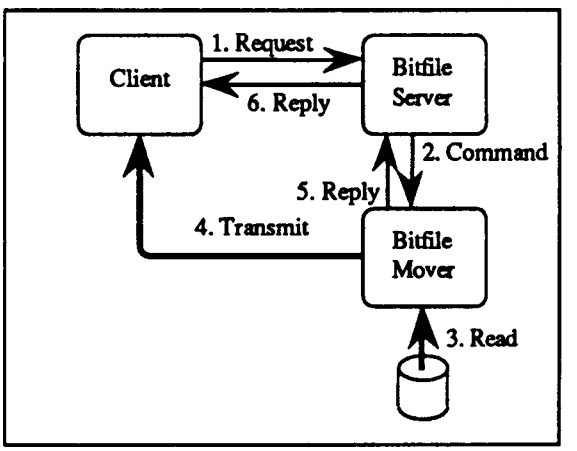

Figure 8. Separation Of Control And Data