CONF-9005105--4

2

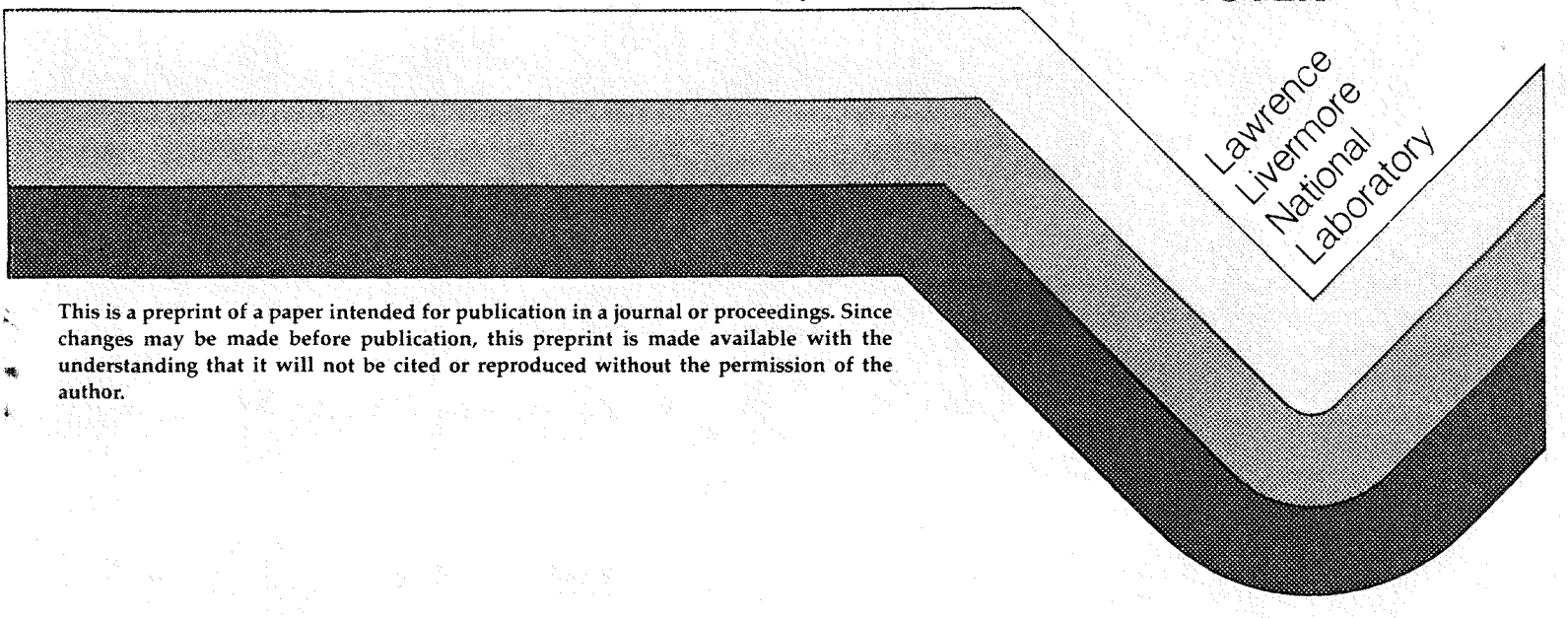
UCRL-102663
PREPRINT

The Livermore Distributed Storage System: Implementation and Experiences

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

This paper was prepared for the
Tenth IEEE Symposium on Mass Storage Systems
Monterey California
May 1990

January 1990



ps

DISCLAIMER

THE LIVERMORE DISTRIBUTED STORAGE SYSTEM: IMPLEMENTATION AND EXPERIENCES

UCRL--102663

DE90 007257

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

Lawrence Livermore National Laboratory
Livermore, California

ABSTRACT

The LINCS Storage System (LSS) has been in production at Lawrence Livermore National Laboratory since January, 1988. In the development of the LSS, the designers made key architectural decisions which included the separation of data and control messages, a network-wide locking mechanism, and the separation of the naming mechanism from other object managers. Other important issues of the system deal with space management, descriptor management, and system administration. This paper outlines the LSS software system and then focuses on these key design decisions and issues with the intent of providing some insight into the types of difficulties designers face in developing a distributed storage system.

OVERVIEW OF THE SOFTWARE SYSTEM

The LINCS Storage System (LSS) is based on the IEEE Mass Storage Reference Model[1] and integrates host, central, and archival storage systems into one transparent, logical system (see Figure 1). The host system storage media consists of solid state and rotating disks on local machines. The central system medium is a collection of magnetic disks on the storage machine. The archival system medium is cartridge tape kept either in on-line robotic devices or in off-line vaults.

The software components of the LSS were designed to operate as servers on a distributed network.[2] Their design extends the classic client-sever model through the use of multitasking within servers and clients to support concurrent access to objects. The two main types of LSS servers are name servers, which translate human-oriented names to machine-oriented object identifiers, and bitfile servers, which manage access to bitfiles on disk or on archival media (currently model 3480 tape cartridges). Host machines and the storage computer have name servers that manage name/object identifier pairs, bitfile servers that manage bitfiles on the various storage media, and bitfile movers that move bitfiles between archival tape and central disk and between host bitfile servers and the central server.**

In the remainder of this section we briefly outline the abstract objects managed by the storage system and the major components of the system. The two main abstract objects of the LSS are the bitfile, implemented by the bitfile servers, and the directory, implemented by the name servers. A bitfile is viewed as a descriptor and a body. The descriptor contains attributes of the bitfile, such as the time the bit segment was last written and the bitfile size. The body is a sequence of uninterpreted, logically contiguous bits. The operations that can be performed on the body include functions such as *create*, *destroy*, *read*, and *write*. Fields of the bitfile descriptor can be read with the *interrogate* operation and, when allowed, written with the *change* operation. A directory is viewed as a descriptor and a list of entries. The descriptor contains attributes of the directory such as the time an entry was last inserted or deleted and the number of entries contained in the directory. An entry is a human-name/object-identifier pair. Directories may be *created*, *destroyed*, or *listed* and

** In Figure 1, the bitfile movers are represented by the arrows between the servers.

entries may be *created*, *deleted*, *fetched* or *renamed*. As with the bitfile descriptor, fields in the directory descriptor may be read with the *interrogate* operation and written with the *change* operation.

The name servers provide a transparent naming service based on the use of unique, network-wide object identifiers for all resources. Object identifiers in the system have the same structure regardless of the type of object they identify. This allows objects of varying types to be cataloged in the same directory, providing a uniform naming mechanism across all objects. Since the directory object identifiers are also globally unique and can be stored in directories, a logically single, hierarchical, directed-graph directory structure can be built. This structure supports name transparency; translation of a pathname initiated from any site in the network always references the same object. Furthermore, there is no necessary relationship between the directory location and the location of the objects cataloged in the directory. To help reduce network delay when accessing a directory, it is planned that the host name servers will cooperate with the central name server to cache and migrate directories.

The bitfile servers provide network-wide random access to bitfiles. The design model for the integrated network bitfile system is to have fully integrated host, central, and archival bitfile servers and movers which support automatic migration throughout the entire storage hierarchy. The connection between the central and host bitfile servers is not yet fully implemented. However, the current bitfile system hierarchy is integrated from the central bitfile server through the archive's off-line vault volumes. When the bitfile system is fully integrated, a client on a host will request a bitfile access from his local server. If the local server has no knowledge of the bitfile being requested it will contact the central server. The central server will find the bitfile, which may reside on another host machine or on central media or in the archive, and send a copy to the requesting host bitfile server. Until the hierarchy is fully connected, a client on a host machine may access a bitfile managed either by the central or by the archival bitfile server through direct communication with the central server or by invoking a utility that copies the bitfile to a local bitfile server.

The central and archival levels of the storage hierarchy are integrated, using a 200-gigabyte disk cache as a front-end to tape. As new bitfiles accumulate, they are automatically written to tape. The descriptors for all bitfiles on tape are maintained on disk for easy access. On a read request, a bitfile is automatically cached from tape to the disk cache. Bitfiles are purged from the disk cache using a bitfile-size–time-of-last-access algorithm.

Using this overview as background, we now focuses on several key design decisions and issues.

SEPARATION OF CONTROL AND DATA

The separation of control and data messages in the LSS improves transfer rates and allows third-party copies during which data does not pass through bitfile server buffers. Data associated with reads or writes is received or sent on a data communication association (source-destination address pair) separate from the control communication associations; see Figure 2a. The third-party copy model is illustrated in Figure 2b. This design can create a pipe-line of services as shown in Figure 2c.

The optimization of third-party copies is supported by *receive-any* and *send-any* mechanisms in the LINCS message-system interface. These mechanisms use association capabilities called stream numbers to make communications secure; associations are defined by the triple (source address, destination address, stream number). The receive-any and send-any mechanisms allow a process to declare that it is willing to receive or send messages with one or more members of the triple unspecified—i.e., any source, or any destination, or any stream number. For example, the third-party copy protocol uses the receive-any by indicating a specific destination address and a specific stream number but an unspecified source address; a rendezvous occurs when a message arrives matching the receive-any's specific destination address and specific stream number. The third-party copy protocol uses the send-any by indicating a specific source address and a specific stream number but an unspecified des-

tination address. The message is flow-blocked at the source machine until an *ack* with the specific source address and stream number is received by the source machine, completing the rendezvous by matching the acknowledging destination address with the any-destination address.

The third-party copy utilizes these mechanisms in the following protocol. When a controlling process requires data to be moved between second and third processes, it first sends a message to the second, over a control association, that includes: an indication that the process is to use the send- or receive-any mechanism in the data transfer, an indication whether the process will send or receive data, and a stream number. In response, the second process will: 1) invoke a receive-any, if receiving data, or a send-any, if sending data, using the stream number received from the controlling process and 2) send a message back to the controlling process containing its local data address used in invoking the receive- or send-any. The controlling process then sends a message to the third process that includes: an indication that the process is to use a send- or receive-specific in the data transfer, an indication whether the process will send or receive data, the data address received from the second process, and the stream number. In response, the third process invokes a receive-specific, if receiving data, or a send-specific, if sending data, using the data address and stream number received from the controlling process. The *any* message of the second process will rendezvous with the specific data message or specific *ack* of the third process and the data will flow directly between these processes. When the data transfer is complete both source and sink processes send a message to the controlling process acknowledging completion of the transfer (see Figure 3).

The LSS utilizes bitfile movers to actually transfer the data. A bitfile mover performs data transfer directly between channels such as memory or networks or devices such as disk or tape. In our UNIX implementations, the bitfile movers have been implemented as either normal user processes or as kernel entities, the former for portability and the latter for performance. All bitfile movers use the same communication and lightweight tasking libraries, whether in kernel or user space. On a single machine, data is transferred by the mover between a device and application memory; over a network, data is actually moved directly between bitfile movers. For a full discussion of the bitfile movers, see reference 3.

Because the LSS separates data and control messages and utilizes the mover processes, the bitfile servers never touch the sequence of bits in the bitfile body. This not only avoids data copying when only one machine is involved in a transfer, but it is also important when a program on machine A invokes a transfer of all or part of a bitfile from machine B to machine C. The program on machine A controls the movement of the data but the data flows directly from machine B to machine C. This architecture gives designers the flexibility to place data movement control on lower-performance machines without impacting the performance of the system.

The LSS experience with separation of data and control messages has shown this mechanism to be valuable for both performance and modularity.

ARCHIVAL SPACE MANAGEMENT

Storage technology is failing to keep pace with the rapid growth of supercomputer memory capacities. It has been the experience at Lawrence Livermore National Laboratory (LLNL) that a small number of large bitfiles can occupy a large part of the system's storage resources. These bitfiles are the results of scientific simulations and their sizes are proportional to supercomputer memory size. The capacity of new robotic libraries, such as the STC 4400, once thought to be adequate, will soon be insufficient to handle the data produced at a large scientific laboratory. To fully utilize available on-line robotic devices, the LSS completely fills each tape cartridge by writing data until an end-of-file error is received. A tape may hold many files or a file may span many tapes. The LSS also manages the on-line archive as a level in the storage hierarchy. Inactive bitfiles migrate to lower-level, off-line vault volumes when on-line archival space is needed; bitfiles in the

vault are moved up to the on-line robotic devices as they are accessed.

Algorithms used to manage space on magnetic disks cannot be used to manage archival media that do not provide random write access. To reuse free space on a magnetic tape the active data on the tape must first be copied to a new volume, an operation called *repacking*. Since copying is an expensive operation, only volumes with a considerable amount of free space should be repacked.

The necessity of repacking to reclaim unreferenced space affects the criteria by which LSS bitfiles are chosen to migrate to the vault. While it is desirable to use a space-time product algorithm that allows smaller bitfiles to remain higher in the hierarchy longer than larger bitfiles, it is not desirable to migrate a few short bitfiles from many robotic tape cartridges to vault cartridges if the space released by these bitfiles does not make any of the robotic tapes candidates for repacking. Only bitfiles that together account for enough free space on their volumes should be migrated to the vault.

In the LSS, the archival server maintains a table of the current number of referenced blocks on each archival volume. The server maintains a minimum number of free volumes by repacking volumes on which less than half of the data blocks are referenced. If there are not enough free blocks on candidate volumes, it migrates bitfiles to the vault according to a space-time algorithm, except that bitfiles are migrated only if they result in volumes becoming candidates for repacking. Migrating bitfiles rather than volumes to the vault minimizes the number of active bitfiles in the vault and minimizes the number of cartridges handled manually (particularly convenient since LLNL's vault and on-line facilities are located in different buildings).

ARCHIVAL FILE DESCRIPTOR MANAGEMENT

In the LSS, bitfile descriptors for bitfiles managed by the archival server are kept on dedicated, redundant disk for fast queries and updates and to add flexibility in assigning the physical location of the bitfile body. Because the bitfile descriptors are vital to the system, great care is taken to ensure their integrity and recovery.

To ensure the integrity of the bitfile descriptors, atomic transactions are used to perform updates to the duplicated disks. To protect against multiple disk failures, a log of all descriptor insertions, deletions, and modifications is maintained. Sixteen descriptor updates (contents of one physical disk sector) are collected in an internal buffer before being atomically written to disk. The buffer is also flushed to disk every two minutes if not enough updates have been collected. The log is also written to tape periodically to guard against the loss of the log disks. Since recovery after many months of updates from an incremental log would be slow, a complete tape backup of all the archival server disks is performed weekly. To recover from a catastrophic failure, the incremental log from the last week is applied to the latest full backup.

File descriptor management in the LSS ensures fast access and data integrity at a moderate cost. In twenty-five years of operation of the LSS and the predecessor system, no failures occurred that required accessing the backup tapes.

FILE SYNCHRONIZATION

To preserve a bitfile's consistency when it is accessed concurrently by multiple applications, the LSS uses a distributed locking mechanism with notification instead of lifetime timeout; an application maintains a lock until it is finished accessing the bitfile or it is notified that another application is interested in accessing the bitfile. When an application receives a notification it may release the lock and allow access to the bitfile by the other application or it may refuse to give up the lock. An application might reasonably keep bitfiles locked for months if no other application wishes to access them, so locks do not timeout. If an application holding a lock does not respond to a request to release the lock, a lock-breaking mechanism can be used by the requesting application to recover the lock.

The LSS locking mechanism is based on classic read/write locks,[4] with extensions for notification and lock breaking. A lock held by a user application may span many read and write accesses, whereas the bitfile servers lock objects only for individual accesses. There are three

types of locks: *no-lock*, *read-lock* and *write-lock*. A write-lock allows both reading and writing by the client holding the lock. There may be multiple concurrent read-locks on a bitfile allowing multiple readers, but a write-lock excludes all other readers and writers. Locks may be categorized into levels, with the highest level being a write-lock and the lowest level being a no-lock. Each bitfile server manages locks for its own clients.

The lock operations on a bitfile include *set-lock*, *reduce-lock* and *break-lock*. Set-lock is used by clients either to set a lock or to lower the level of a lock on a bitfile. Reduce-lock is used to ask a client to lower or release its lock on a bitfile. A client sends a break-lock request to a bitfile server to break a lock held by a client that is down. The granularity of locking is a bitfile. A lock request includes a bitfile identifier, a lock type, and a notification address, which is the requesting client's address in the case of a set-lock, and the lock holder's address as well in the case of a reduce- or break-lock. The notification address in a lock is used to identify the lock holder.

Crash recovery considerations for locking include: 1) how to proceed when a bitfile server is unavailable, and 2) how to restore lock state when a bitfile server reinitializes after a crash. If a bitfile server fails to respond to a reduce lock request, causing the requesting bitfile server to refuse an application access to a bitfile, then the application may send a break lock request. In the LSS, breaking a lock invalidates changes that were made while the lock was in effect. A client would choose this action only when losing updates is preferable to waiting for the bitfile server to return to service.

ADMINISTRATIVE REQUIREMENTS OF A DISTRIBUTED FILE SYSTEM

We have identified six administrative requirements for the LSS: 1) to ensure fair use of storage, 2) to ensure efficient use of storage, 3) to achieve high performance, 4) to minimize cost, 5) to provide for accountability of storage resources, and 6) to permit cost recovery. The proposed method for meeting these requirements is a combination of charging and allocation. Charging refers to a mechanism by which users are charged for the storage resources (space and transfers) they consume. Allocation refers to a mechanism for ensuring that each user or group of users has an appropriate share of the available storage resources.[5] In this section, we will focus on the difficulty of designing mechanisms that are consistent with the administrative requirements and with the design goal of location transparency and with user expectations.

The predecessor system to the LSS, which controlled only central and archival storage, lacked adequate administrative measures. It allowed unrestricted, free use of storage, and provided no incentive for users to restrict the amount of data they generated, or to delete unwanted data. On the host disks, storage was also free, but an attempt was made to restrict the amount of data on disks by purging idle bitfiles. This measure proved to be ineffective. To prevent their bitfiles from being purged, users simply ran applications that periodically touched their bitfiles to keep them active.

To dispel the notion of free storage, a new charging policy was implemented in the early days of the LSS. The motivation behind this initial policy was to recover the cost of the system and to discourage users from maintaining all of their bitfiles on expensive fast-access storage. The algorithm for computing charges was [bitfile-length times bitfile-age times charge-rate], where the charge rate varied from one storage device to another. The bitfile age was the smaller of the time since last charged or the time since creation. The information needed to compute the charge was collected weekly as each bitfile server traversed its descriptors. A separate utility computed charges based on the reports from each LSS server, decremented user bank accounts accordingly, and produced billing reports.

The problem with this scheme is that the same bitfile is multiply charged if it happens to reside on more than one bitfile server. This is the case, for example, when a central bitfile is cached on a host disk. The appearance of multiple charges for the same bitfile, and of charges that reflect the location of bitfiles in the system, clearly defeats the system design goal of transparency. Mechanisms to remedy this problem are being considered. One such mechanism allows users to classify bitfiles as active, archival, etc., to

help them control their costs and to improve the effectiveness of migration algorithms. We are also considering a fixed rate that is independent of the storage medium. As an interim fix, the system has been changed to charge only for bitfiles stored on archival volumes.

Charging alone does not meet all of the administrative requirements. Because there are physical limits to a storage system and because purely economic factors do not seem to be adequate, we are considering allocation limits to ensure fair sharing of storage resources and to ensure good performance of the system. The proposal we have adopted, though not yet implemented, includes two types of allocation, global and disk. Global allocation applies to the total space acquired at all levels of storage, including host disk, central disk, and archival tape. Each user has a global allocation, which is an upper limit on the total amount of data a user may store. Once this limit is reached, the user may not create more bitfiles, or extend existing bitfiles, until he generates free space by destroying bitfiles.

Disk allocations apply only to host and central disk. Users have maximum disk allocations but are allowed to exceed their allocations if space allocated to other users is not in use. When space is needed, bitfiles belonging to users who have exceeded their allocations are migrated first. For host disk, users also have minimum disk allocations. Users' bitfiles will not migrate if their space utilization falls below this amount.

Designing adequate charging and allocation schemes for a transparent system poses a difficult challenge. Charging schemes that strive to do accurate cost recovery are inherently inconsistent with the goal of transparency. There is much to learn about selecting and implementing effective administrative policies in these areas.

NETWORK-WIDE NAMING MECHANISM AND USER EXPECTATIONS

Before we implemented a network-wide naming mechanism, the LSS was a distributed system but not a transparent one. Users were aware of object location and had to explicitly invoke transfer routines to move bitfiles between host machines and the central server. Inactive host bitfiles were automatically destroyed, and access to archival bitfiles was relatively slow. The central name server and each host maintained separate, unconnected directory structures which did not migrate between machines. As a result of this environment, users wrote programs to keep bitfiles on host disks by periodically accessing bitfiles stored on the hosts. Users resisted the idea of location transparency because it meant they would lose control of access time.

Location transparency was achieved in two phases. The first included creation of a network-wide directory structure, for name transparency; and the second implemented caching and migration of directories, for performance. In the first phase, we connected the directory structures stored on the host machines with the directory structure in central. Central and archival bitfiles and central directories could then be cataloged in directories on the host machines and vice versa, creating a transparent, cross-machine naming structure. Because bitfiles and directories have globally unique identifiers, the storage system could locate the resources regardless of the directory in which they were cataloged and regardless of which server managed the resource.

In this strange, new storage world, users became frustrated when they unknowingly crossed machine boundaries in perusing through their directory structures because performance suddenly declined. Yet, many resisted the idea of caching and migration of bitfiles and directories to improve performance; these users desired to control location of their bitfiles and directories to ensure fast access. We believe, however, that the system can best manage object location, much as demand paging systems manage virtual memory.

There are two significant lessons to be learned from our experience. First, when creating a transparent storage system it is important to achieve transparency in all aspects before putting the system on-line. Second, in presenting a new storage system to users, system designers need to carefully consider what the users expect and educate them about the differences in the new system.

SEPARATION OF HUMAN NAMING FROM OTHER OBJECT SERVERS

When considering a human-oriented naming mechanism for a distributed storage system, designers must choose between one that is integral to the object managers, such as UNIX and CFS, and one that is isolated in separate name managers, such as XDFS and ALPINE.[6] We chose the latter design, recognizing that it has both advantages and disadvantages. The advantages of a separate naming mechanism include: 1) providing a uniform mechanism for naming many different types of objects, not just bitfiles; 2) providing all other servers independence from human-oriented naming conventions, allowing them to function in a variety of user environments;[7] 3) permitting the other servers to be optimized to manage their own objects without the need to deal with human-oriented naming issues;[8] 4) permitting new types of objects to be named in the same way as existing objects, facilitating extensibility;[9] 5) permitting several forms of application-dependent and general-purpose higher-level name services to be provided in addition to a particular directory service;[8] and 6) permitting applications to create, access, and destroy objects without storing the object identifiers in any name service.[9]

The disadvantages of a separate naming mechanism, discussed below, include: 1) complicating certain aspects of storage management; 2) requiring additional security mechanisms; and 3) causing performance degradation in a particular class of applications.

We believe that the advantages of a separate naming mechanism outweigh the disadvantages and that the LSS has achieved those advantages. In particular, a separate naming mechanism allows the bitfile servers of the LSS to be integrated with the naming mechanism of any existing operating system. We are now working on minimizing the effects of its disadvantages, as described below.

Storage Management

The LSS servers will perform functions given a valid object identifier containing the appropriate access rights. The general availability of certain server functions, in combination with the separation of the name servers from the other object servers, has consequences for storage management. Specifically, clients have the potential of creating lost objects, objects for which there are no extant identifiers, and of creating dangling pointers, identifiers no longer pointing to valid objects. Lost objects are created when a client deletes all references to the object without destroying the object. Dangling pointers are created when a client destroys an object but does not delete all the references to it.

The problems of lost objects and dangling pointers can be solved by correctly managing reference counts and by prohibiting explicit destroys of objects. A count of all references to it is kept with each object as part of its storage management state. When the count becomes zero, meaning the object is no longer referenced, the space it occupies can be reclaimed for further use. All applications in the LSS environment that store identifiers to objects need to increment and decrement reference counts correctly to protect against lost objects and delete object references before destroying objects to protect against dangling pointers. For example, name servers should send increment and decrement messages to the object servers when objects are inserted and deleted from directories. When the reference count goes to zero, the object can be implicitly destroyed.

However, the increment- and decrement-reference-count functions are not controlled by the current LSS access control mechanism and therefore may be invoked by any client possessing a valid machine-oriented identifier. There is no way to ensure that clients will invoke the reference-count functions correctly. Furthermore, there is no way to guarantee that all naming services or applications that store object identifiers will correctly increment and decrement the reference counts as objects are inserted and deleted from their databases. Similarly, there is no way to guarantee that all object references are deleted before the object is explicitly destroyed. In these circumstances, reference counts cannot be relied upon to reclaim storage space.

One solution to these problems is to allow only trusted naming applications to invoke the increment- and decrement-reference-count and destroy functions and to require clients to store machine-oriented identifiers only in these ap-

plications. The additional access control necessary to implement trusted applications could be obtained through several mechanisms including rights amplification capabilities, rights verification servers, or access lists. The LSS name servers would be the initial set of trusted applications, but it would be reasonable to include any naming service that could establish its correct use of reference-count and destroy functions. Of course, a requirement to use a restricted set of applications to catalog object identifiers would restrict clients' ability to use their own naming services.

Security

Security is affected in two ways by the separate naming mechanism. First, it is hampered when a client, possessing a valid object identifier, maliciously or inadvertently destroys an object still being referenced by other clients by sending repeated decrement-reference-count functions to the object server. Likewise, too many increment requests can prevent an object's implicit destruction, leaving the object available for continued access after it should have been destroyed. This problem is solved by the same trusted-naming-application mechanism suggested above that solves the storage management problems.

Second, in an environment with a separate naming mechanism, clients can obtain, through several server functions, a machine-oriented object identifier for use as parameters in subsequent function requests to the servers (e.g., read and write). Because clients can possess machine-oriented identifiers, there must be security mechanisms to prevent the client from changing the identifier to another valid identifier and to protect identifiers against the threats of forgery, theft, and reuse. In the LSS, the machine-oriented identifiers are encrypted by servers using a DES[10] cryptographic checksum, to prevent client tampering and to protect against forgery. Encryption algorithms exist which also protect against theft and reuse,[11] but they are not used in the LSS because they were determined to be unnecessary in LLNL's secure physical environment. Because the LSS does not protect against theft and reuse, users must exercise care in storing the machine-oriented identifiers: storing indentifiers in trusted name servers is safe; storing them on workstations may be unsafe.

Performance

Separation of the naming mechanism from the other object servers has both a positive and a negative impact on performance. Separation of human naming from other object servers has had a positive impact on performance through modularity. The modularity gained from separating the naming mechanism from the object servers is observed in two important areas. First, software errors pertaining directly to the name translation code do not affect the object servers. Additionally, hardware failures affecting a given name server do not disturb other object servers on remote machines. If a name server were to go down for a period of time due to either of these causes, all applications which had previously translated their human-oriented names into machine identifiers could continue processing. Moreover, if the period of down time for the name server were short, an application may not even notice that the naming service had been interrupted.

The second modularity advantage of a separate naming mechanism is the efficiency gained in other object servers by being independent of human-oriented naming conventions. The name server acts as an intermediary between the human world and machine processes. No other object server need contain algorithms which parse the strings of human names or package them for passing in messages. Instead, the object servers deal only with machine-oriented identifiers which can be easily interpreted and utilized.

However, performance suffers when an application requires the attributes of the objects contained in a directory. This type of application, such as the UNIX *ls* utility, first must contact the name server to obtain a list of machine-oriented object identifiers. Because the name server does not contain other information about the objects, the application must then contact the object servers to obtain the desired attributes. Since the name server has been generalized to store objects of many different types, the application may have to contact several object servers. Performance improvement techniques to limit this disadvantage are being designed and im-

plemented. For example, the caching and migration of objects will help improve performance by causing both directories and other objects to be locally cached, resulting in faster access times.

The separation of the naming mechanism from the other object servers in the LSS has given the system and its clients flexibility, extensibility, and modularity not found in integral naming mechanisms. Although this separation has disadvantages, techniques exist to lessen their impact. Moreover, the system's flexibility and modularity allow it to be used under client interfaces that present the conventional, integrated view. In this environment, the user applications would not suffer from most of the disadvantages discussed above, yet the system would benefit from most of the advantages of a separate naming mechanism.

CONCLUSION

This paper discusses several key design goals and implementation areas of the LSS that support its extensibility, its modularity, and its flexibility. The separation of data and control messages has had a positive impact on performance and modularity by allowing third-party copies without the actual passing of data through the bitfile servers. Efficient management of storage media and bitfile headers has increased storage utilization and provided integrity of the header information. A network-wide locking mechanism has been designed that preserves an object's consistency when accessed concurrently by multiple applications. The separation of the human-oriented naming mechanism from the other object servers has given the system and its clients flexibility, extensibility, and modularity not found in an integral naming mechanism.

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.

REFERENCES

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

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

3. Mark Gary, "Overcoming Unix Kernel Deficiencies in a Portable, Distributed Storage System," DIGEST OF PAPERS, Tenth IEEE Symposium on Mass Storage Systems, May 1990.

4. Andrew S. Tanenbaum, "File Systems," *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987, pp. 273–277.

5. Ralph Carlson and Carole Hogan, "A Model Recharge System for the LCC," Lawrence Livermore National Laboratory, internal publication, May 5, 1988.

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

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

8. 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.

9. Samuel S. Coleman and Richard W. Watson, "Designing Archival Storage Systems for Distributed Supercomputer

Environments," submitted for *Computer*, May 1990.

10. National Bureau of Standards, Federal Information Processing Standards, Publ. 46, 1977.

11. J. E. Donnelley and J. G. Fletcher, "Resource Access Control in a Network Operating System," Proceedings, ACM Pacific Conference, November 1980.

FIGURES

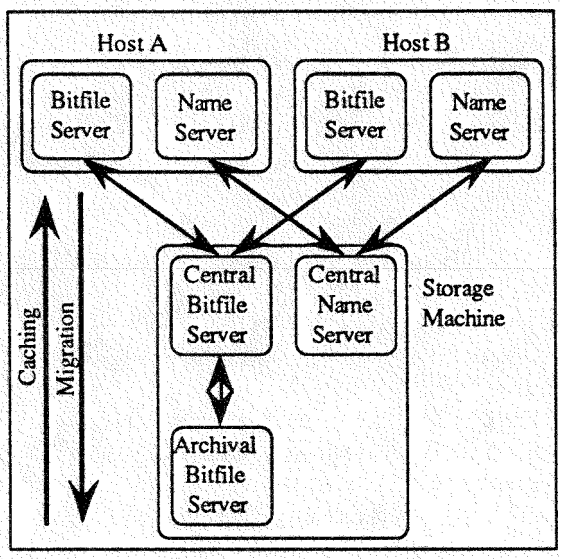


Figure 1. LSS Hierarchy

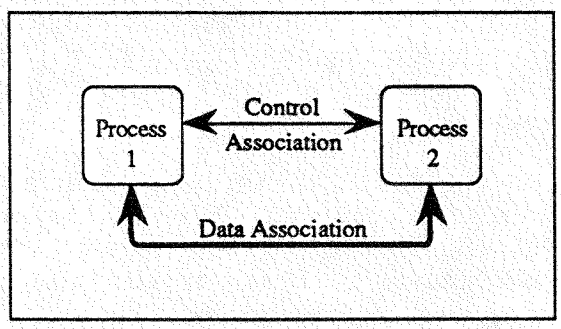


Figure 2a. Separation of Control and Data

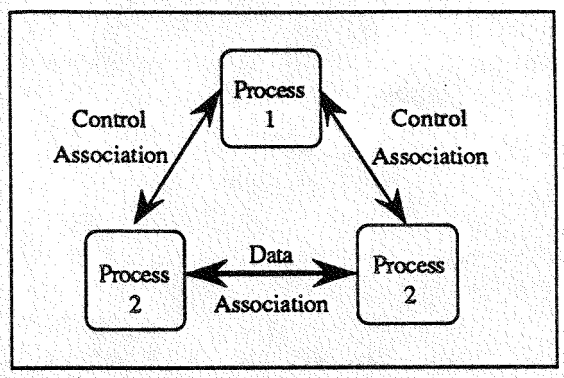


Figure 2b. Third-Party Copy

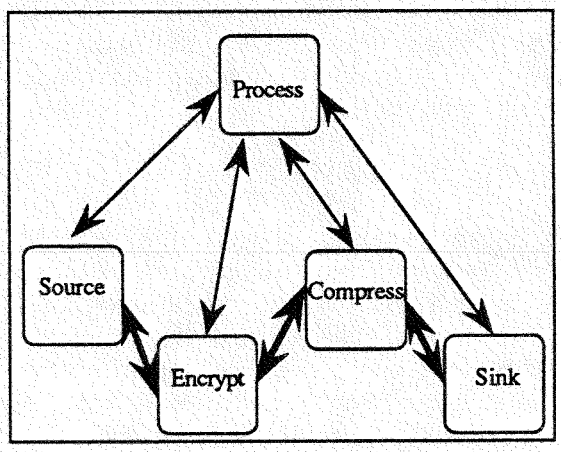


Figure 2c. Pipeline

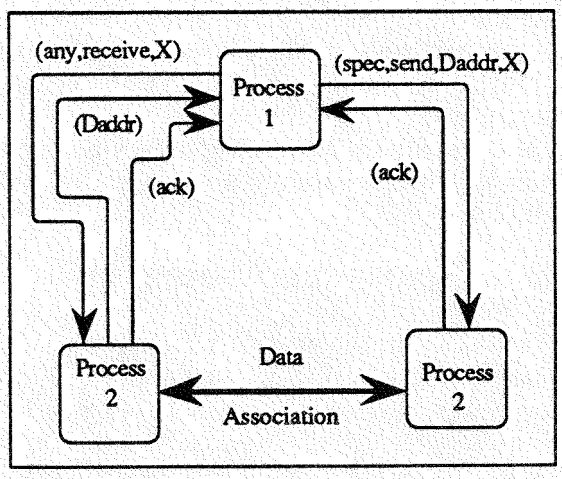


Figure 3. Third-Party Copy Protocol