Mass Storage System Reference Model:
Version 4 (May, 1990)

Developed by the IEEE Technical Committee on Mass Storage Systems and Technology

Edited by: Sam Coleman
Lawrence Livermore National Laboratory

Steve Miller
SRI International

Mass Storage System Reference Model: Version 4 (May, 1990)

Contents Page

1. Preface

The purpose of this reference model is to identify the high-level abstractions that underlie modern storage systems. The information to generate the model was collected from major practitioners who have built and operated large storage facilities, and represents a distillation of the wisdom they have acquired over the years. The model provides a common terminology and set of concepts to allow existing systems to be examined and new systems to be discussed and built. It is intended that the model and the interfaces identified from it will allow and encourage vendors to develop mutually-compatible storage components that can be combined to form integrated storage systems and services.

The reference model presents an abstract view of the concepts and organization of storage systems. From this abstraction will come the identification of the interfaces and modules that will be used in IEEE storage system standards. The model is not yet suitable as a standard; it does not contain implementation decisions, such as how abstract objects should be broken up into software modules or how software modules should be mapped to hosts; it does not give policy specifications, such as when files should be migrated; does not describe how the abstract objects should be used or connected; and does not refer to specific hardware components. In particular, it does not fully specify the interfaces.

A storage system is the portion of a computing facility responsible for the long-term storage of large amounts of information. It is usually viewed as a shared facility and has traditionally been organized around specialized hardware devices. It usually contains a variety of storage media that offer a range of tradeoffs among cost, performance, reliability, density, and power requirements. The storage system includes the hardware devices for storing information, the communication media for transferring information, and the software modules for controlling the hardware and managing the storage.

The size and complexity of this software is often overlooked, and its importance is growing as computing systems become larger and more complex. Large storage facilities tend to grow over a period of years and, as a result, must accommodate a collection of heterogeneous equipment from a variety of vendors. Modern computing facilities are putting increasing demands on their storage facilities. Often, large numbers of workstations as well as specialized computing machines such as mainframes, mini-supercomputers, and supercomputers are attached to the storage system by a communication network. These computing facilities are able to generate both large numbers of files and large files, and the requirements for transferring information to and from the storage system often overwhelms the networks.

The type of environment described above is the one that places the greatest strain on a storage system design, and the one that most needs a storage system. The abstractions in the reference model were selected to accommodate this type of environment. While they are also suitable for simpler environments, their desirability is perhaps best appreciated when viewed from the perspective of the most complicated environment.

There is a spectrum of system architectures, from storage services being supplied as single nodes specializing in long-term storage to what is referred to as “fully distributed systems”. The steps in this spectrum are most easily distinguished by the transparencies that they provide, where they are provided in the site configuration, and whether they are provided by a site administrator or by system management software. The trend toward distributed systems is appealing because it allows all storage to be viewed in the same way, as part of a single, large, transparent storage space that can be globally optimized. This is especially important as systems grow more complex and better use of storage is required to achieve satisfactory performance levels. Distributed systems also tend to break the dependence on single, powerful storage processors and may increase availability by reducing reliance on single nodes.

1.1 Transparencies

Many aspects of a distributed system are irrelevant to a user of the system. As a result, it is often desirable to hide these details from the user and provide a higher-level abstraction of the system. Hiding details of system operation or behavior from users is known as providing transparency for those details. Providing transparency has the effect of reducing the complexity of interacting with the system and thereby improving the dependability, maintainability, and usability of applications. Transparency also makes it possible to change the underlying system because the hidden details will not be embedded in application programs or operating practices.

The disadvantage of using transparency is that some efficiency can be lost in resource usage or performance. This occurs because the mechanism that provides the transparency masks semantic information and causes the system to be used conservatively. High-performance data base systems, for example, may need to organize disk storage directly and schedule disk operations to gain performance, rather than depend on lower-level file systems with their own structure, scheduling, and policies for caching and migration.

There is a range of support that can be provided for distributed systems in a computer network. A system with few transparencies is often called a networked system. The simplest kind of networked system provides utilities to allow a program to be started on a specified host and information to be transferred between specified storage devices. Examples include TELNET and FTP, respectively. This type of system rarely provides support for heterogeneity. At the other end of the spectrum are fully distributed systems that provide many transparencies. An example is LOCUS. In distributed systems, a goal is for workstations to appear to have unlimited storage and processing capacities.

System and application designers must think carefully about what transparencies will be provided and whether they will be mandatory. It is possible for applications to provide certain transparencies and not others. Fundamental transparencies can be implemented by the system, saving each user from re-implementing them. A common implementation will also improve the likelihood that the transparency will be implemented efficiently.

The common transparencies are:

Access
Clients do not know if objects or services are local or remote.

Concurrency
Clients are not aware that other clients are using services concurrently.

Data representation
Clients are not aware that different data representations are used in different parts of the system.

Execution
Programs can execute in any location without being changed.

Fault
Clients are not aware that certain faults have occurred.

Identity
Services do not make use of the identity of their clients.

Location
Clients do not know where objects or services are located.

Migration
Clients are not aware that services have moved.

Naming
Objects have globally unique names which are independent of resource and accessor location.

Performance
Clients see the same performance regardless of the location of objects and services (this is not always achievable unless the user is willing to slow down local performance).

Replication
Clients do not know if objects or services are replicated, and services do not know if clients are replicated.

Semantic
The behavior of operations is independent of the location of operands and the type of failures that occur.

Syntactic
Clients use the same operations and parameters to access local and remote objects and services.

Some of the transparencies overlap or include others.

With this in mind, it is incumbent upon the Storage System Standards Working Group to identify interfaces and modules that are invariant from single storage nodes to fully distributed systems. Many sites are not likely to embrace fully distributed systems in a single step. Rather, they are likely to evolve gradually as growing system size and complexity dictate and as vendors make available products supporting fully distributed systems.

1.2 Requirements

Modern computing facilities are large and complex. They contain a diverse collection of hardware connected by communication networks, and are used by a wide variety of users with a spectrum of often-conflicting requirements. The hardware includes a range of processors from personal computers and workstations to mainframes and supercomputers, and many types of storage devices such as magnetic disks, optical disks, and magnetic tapes. This equipment is typically supplied by a variety of vendors and, as a result, is usually heterogeneous. Both the hardware characteristics and the user requirements make this type of facility extremely complicated.

To insure that the reference model applies to many computer environments, the IEEE Technical Committee on Mass Storage Systems and Technology identified the following requirements:

- The model should support both centralized and distributed hierarchical, multi-media file systems.
- The model should support the simplest randomly addressable file abstraction out of which higher level file structures can be created (e.g., a segment of bits or bytes and a header of attributes).
- Where the defined services are appropriate, the model should use national or international standard protocols and interfaces, or subsets thereof.
- The model should be modular such that it meets the following needs:
  - The modules should make sense to produce commercially.
  - It should be reasonable to integrate modules from two or more vendors.
  - The modules should integrate with each other and existing operating systems (centralized and distributed), singly or together.
  - It should be possible to build hierarchical centralized or distributed systems from the standard modules. The hierarchy might include, for example, solid state disks, rotating disks (local and remote), an on-line library of archival tape cartridges or optical disks, and an off-line, manually-operated archival vault.
  - Module interfaces should remain the same even though implementations may be replaced and upgraded over time.
  - Modules should have standardized interfaces hiding implementation details. Access to module objects should only be through these interfaces. Interfaces should be specified

by the abstract object data structures visible at those interfaces.

- Module interfaces should be media independent.

• File operations and parameters should meet the following requirements:

  - Access to local and remote resources should use the same operations and parameters.
  - Behavior of an operation should be independent of operand location.
  - Performance should be as independent of location as possible.
  - It should be possible to read and write both whole files and arbitrary-sized, randomly-accessible pieces of files.
  - The model should separate policy and mechanism such that it supports standard as well as vendor- or site-specific policy submodules and interfaces for access control, accounting, allocation, site management, security, and migration.
  - The model should provide for debugging, diagnostics, and maintenance.
  - The model should support a request/reply (transaction) oriented communication model.
  - Request and data communication associations should be separated to support high speed direct source to destination data channels.
  - Transformation services (e.g. translation, check summing, encryption) should be supported.

• The model should meet the following naming requirements:

  - Objects should have globally unique, machine-oriented names which are independent of resource and access location.
  - Each operating system or site environment may have a different human-oriented naming system, therefore human- and machine-oriented naming should be clearly separated.
  - Globally unique, distributively generated, opaque file identifiers should be used at the client-to-storage-system interface.

• The model should support the following protection mechanism requirements:

  - System security mechanisms should assume mutual suspicion between nodes and networks.
  - Mechanism should exist to establish access rights independent of location.
  - Access list, capability or other site, vendor, or operating system specific access control should be supportable.
  - Security or privacy labels should exist for all objects.

• The model should support appropriate lock types for concurrent file access.

• Lock mechanisms for automatic migration and caching (i.e., multiple copies of the same data or files) should be provided.

• The model should provide mechanisms to aid recovery from network, client, server crashes and protection against network or interface errors. In particular, except for file locks, the file server should be stateless (e.g., no state maintained between “open” and “close” calls).

• The model should support the concept of fixed and removable logical volumes as separate abstractions from physical volumes.

• It should be possible to store one or many logical volumes on a physical volume, and one logical volume should be able to span multiple physical volumes.

2. Introduction

2.1 Background

From the early days of computers, “storage” has been used to refer to the levels of storage outside the central processor. If “memory” is differentiated to be inside the central processor and “storage” to be outside, (i.e., requiring an input-output channel to access), the first level of storage is called “primary storage” (Grossman 89). The predominant technology for this level of storage has been magnetic disk, or solid-state memory configured to emulate magnetic disks, and will remain so for the foreseeable future in virtually every size of computer system from personal computers to supercomputers. Magnetic disks connected directly to I/O channels are often called “local” disks while magnetic disks accessed through a network are referred to as “remote” or “central” disks. Sometimes a solid-state cache is interposed between the main memory and primary storage. Because networks have altered the access to primary storage we will use the terms “local storage” and “remote storage” to differentiate the different roles of disks.

The next level of data storage is often a magnetic tape library. Magnetic tape has also played several roles:

- On-line archive known as “long term storage” (e.g., less active storage than magnetic disk),
- off-line archival storage (possibly off-site),
- backup for critical files, and
- as an I/O medium (transfer to and from other systems).

Magnetic tape has been used in these roles because it has enjoyed the lowest cost-per-bit of any of the widely used technologies. As an I/O medium, magnetic tape must conform to standards such that the tape can be written on one system and read on another. This is not necessarily true for archival or backup storage roles, where nonstandard tape sizes and formats can be used, even though there are potential disadvantages if standards are not used even for these purposes.

In the early 1970s nearly every major computer vendor, a number of new companies, and vendors not otherwise in the computer business, developed some type of large peripheral storage device. Burroughs and Bryant experimented with spindles of 4-ft diameter magnetic disks. Control Data experimented with 12 in. wide magnetic tape wrapped nearly all the way around a drum with a head per track. The tape was moved to an indexed location and stopped while the drum rotated for operation. (Davis 82 presents an interesting comparison of devices that actually got to the marketplace.)

Examples of early storage systems are the Ampex Terabit Memory (TBM) (Wildmann 75), IBM 1360 Photostore (Kuehler 66), Braegan Automated Tape Library, IBM 3850 Mass Storage System (Harris 75, Johnson 75), Fujitsu M861, and the Control Data 38500. One of the earliest systems to employ these devices was the Department of Defense Tablon system (Gentile 71), which made use of both the Ampex TBM and the IBM Photostore. Much was learned about the software requirements from this installation.

The IBM 1360, first delivered in the late 1960s, used write-once, read-many (WORM) chips of photographic film. Each chip measured 1.4 x 2.8 in. and stored 5 megabits of data. “File modules” were formed of either 2250 or 4500 cells of 32 chips each. The entire process of writing a chip, photographically developing it, inserting the chip in a cell, and a cell in a file module, storing and retrieving for read, etc., was managed by a control processor similar to an IBM 1800. The complex chemical and mechanical processing required considerable maintenance expertise and, while the Photostore almost never lost data, the maintenance cost was largely responsible for its retirement. A terabit system could retrieve a file in under 10 seconds.

The TBM, first delivered in 1971, was a magnetic tape drive that used 2-inch-wide magnetic tape in large 25,000-foot reels. Each reel of tape had a capacity of 44 gigabits and a file could be retrieved, on the average, in under 17 seconds. With two drives per module, a ten module (plus two control modules) system provided a terabit of storage. The drive was a digital re-engineering of broadcast video technology. The drive connected to a channel through a controller, and cataloging was the responsibility of the host system.

The Braegan Automated Tape Library was first delivered in the mid 1970s and consisted of special shelf storage housing several thousand half-inch magnetic tape reels, a robotic mechanism for moving reels between shelf storage and self-threading tape drives, and a control processor. This conceptually simple system was originally developed by Xytecs, sold to Calcomp, and then to Braegan. In late 1986, the production rights were acquired by Digital Storage Systems, Longmont, Colorado. Sizes vary, but up to 8,000 tape reels (9.6 terabits) and 12 tape drives per cabinet are fairly common.

The IBM 3850 (Johnson 75, Harris 75) used a cartridge with a 2.7-inch-wide, 770-in. long magnetic tape. A robotic cartridge handler moved cartridges between their physical storage location (sometimes called the honeycomb wall) and read/write devices. Data accessed by the host was staged to magnetic disk for host access. De-staging the changed pages (about 2 megabits) occurred when those pages became the least recently used pages on the staging disks. Staging disks consisted of a few real disk devices, which served as buffers to the entire tape cartridge library. The real disks were divided into pages and used to make up many virtual disk devices that could appear to be on-line at any given time.

Manufactured by Fujitsu and marketed in this country by MASSTOR and Control Data, the M861 storage module uses the same data cartridge as the IBM 3850; however, it is formatted to hold 175 megabytes per cartridge. The M861 holds up to 316 cartridges and provides unit capacity of 0.44 terabits. The physical cartridges are stored on the periphery of a cylinder, where a robotic mechanism picks them for the read-write station. The unit achieves about 12-second access time and 500 mounts/dismounts per hour.

A spectrum of interconnection mechanisms was described (Howie 75) that included:

- The host being entirely responsible for special hardware characteristics of the storage system device,
- the device characteristics being translated (by emulation in the storage system) to a device known by the host operating system, and
- the storage system and host software being combined to create a general system.

This has sometimes been termed moving from tightly coupled to loosely coupled systems. Loosely coupled systems use message passing between autonomous elements.

The evolution of the architectural view of what constitutes a large storage system has been shaped by the growth in shear size of systems, more rapid growth of interactive rather than batch processing, the growth of networks, distributed computing, and the growth of personal computers, workstations, and file servers.

Many commercial systems have tracked growth rates of 60-100% per year over many years. As systems grow, a number of things change just because of size. It becomes difficult for large numbers of people to handle tape reels, so automating the fetching and returning and the mounting and dismounting of reels becomes important. As size increases, it also becomes more difficult for humans to decide which devices to use for load balancing.

Because of this growth, early users of storage systems were forced to do much of the systems integration in their own site environments. Large portions (software and hardware) of many existing systems (Gentile 71, Penny 73, Fletcher 75, Collins 82, Coleman 84) were developed by user organizations that were faced with the problem of storing, retrieving, and managing trillions of bits and cataloging millions of files. The sheer size of such storage problems meant that only organizations such as government laboratories, which possessed sufficient systems engineering resources and talent to complete the integration, initially took on the development task. These individualized developments and integrations resulted in storage systems that were heavily intertwined with other elements of each unique site.

These systems initiated an evolution in storage products in which three stages are readily recognizable today. During the first stage, a storage system was viewed as a very large peripheral device serving a single system attached to an I/O channel on a central processor in the same manner as other peripheral devices. Tasks to catalog the files and free space of the device, manage the flow of data to and from it, take care of backup and recovery, and the many other file management tasks, were added as application programs within the systems environment. Many decisions, such as when to migrate a file, were left to the user or to a manual operator. If data was moved from the storage system to local disk, two host channels (one for each device) were required plus a significant amount of main memory space and central processing capability (Davis 82).

During this stage, the primary effort in design was machine-room automation to reduce the need to manually mount and dismount magnetic tapes.

The second stage (late 1970s to present) has been characterized by centralized shared service that takes advantage of the economies of scale and provides file server nodes to serve several, perhaps heterogeneous, systems (Svobodova 84).

This stage of the storage system evolution is the one that is most prevalent today. The storage system node entails using a control processor to perform the functions of the reference model in a storage hierarchy (O'Lear 82). The cost of the storage system makes it desirable to share these centralized facilities among several computational systems rather than provide a storage system for each computational system. This is especially true when supercomputers become a part of the site configuration.

This approach to providing storage has several advantages:

- The number of processors that have access to a file is larger than that which can share a peripheral device. (This type of access is not the same as sharing a file, which implies concurrent access.)

- Multiple read-only copies of data can be provided, circumventing the need for a large number of processors having access to common storage devices.

- Processors of different architectures can have access to common storage, and therefore to common data, if they are attached to a network and use a common protocol for bit-stream transfer.

- The independence between the levels of storage allows the inclusion of new storage devices as they become commercially available.

Some of the earliest systems in this shared service stage were at the Lawrence Berkeley National Laboratory (Penny 70) and Lawrence Livermore National Laboratory (LLNL) (Fletcher 75, Watson 80).

The Los Alamos Common File System (Collins 82, McLarty 84) and the system at the National Center for Atmospheric Research (Nelson 87, O'Lear 82) are more recent examples of shared, centralized storage system nodes.

The third stage is the emerging distributed system. An essential feature of a distributed system (Enslow 78, Watson 81a, 84, 88) is that the network nodes are autonomous, employing cooperative communication with other nodes on the network. The control processors of storage systems developed during this stage provide this capability.

The view of a storage system as a distributed storage hierarchy is neither a device nor a single service node, but is the integration of distributed computing systems and storage system architectures with the elements that provide the storage service distributed throughout the system. The distributed computing community has been very interested in the problems of providing file management services, albeit generally on smaller systems (Almes 85, Birrell 82, Brownbridge 82, Donnelley 80, Leach 82, Svobodova 84, Watson 81a). Probably the best known example at the workstation level is the SUN Microsystems "network file server" (Sandberg 85).

Several elements are necessary for a system to be classed as "distributed" (Enslow 78):

- A multiplicity of general-purpose resource elements,
- the distribution of these elements, logically and physically,
- a distributed (network) operating system,
- system transparency (service requests by name only), and
- cooperative communication among elements (nodes).

Achieving all of these elements sounds difficult and expensive. The motivations most often cited are extensibility, availability, and costly resource sharing (LeLann 81). Readily extensible systems permit the "hot wiring" necessary in large systems that can no longer afford downtime for cabling in new elements. Extensibility also means that individual elements can be upgraded without disrupting the entire system. System availability is obtained by replicating system elements in a way that permits graceful degradation. Sharing costly elements occurs through communications and networking.

The issues involved in designing distributed systems with the characteristics outlined above were discussed by Dr. Richard W. Watson of the Lawrence Livermore National Laboratory at the Eighth IEEE Mass Storage Symposium in Tucson, Arizona, May 1987. He stated (Watson 87) that the long-range goal is to design systems in which "mainframes, minicomputers, workstations, networks, multiple levels of storage, and input/output systems are viewed as elements of a logically single distributed computer whose resources are managed by and accessed through a single distributed operating system."

Individual operating systems have their own way of handling files. One reason for requiring a distributed operating system is to provide a single logical file and naming system. This distributed file system should be accessed by name only; that is, the naming and heterogeneous features of different component parts should be transparent to the user. Logically, the the distributed storage system should have infinite capacity and unlimited file size. This is obtained through the use of migration among the distributed storage elements that make up the storage hierarchy. The different levels of storage probably have different storage characteristics and costs.

Other design goals include high reliability and availability, high performance (low delay and high throughput), mandatory and discretionary access control, file sharing and safe concurrent access (Lantz 85), and accounting and administrative controls (Mullender 84).

It now appears that an attainable goal is to design interconnected systems, whose subsystems can be produced by a number of vendors, such that the file service is uniform from the user's local level through all levels of the on-line hierarchy to shelf storage. Internally, the distributed hierarchical storage system will consist of multiple levels of storage such as bulk semiconductor memories, magnetic disks, magnetic tape, optical disks, automated media libraries, and manual vaults. Such a system is currently under development at Lawrence Livermore National Laboratory (Coleman 84, Foglesong 90, Gary 90, Hogan 90).

2.2 Motivation

The central architectural features of the reference model and the motivation for them can be summarized as follows:

- An object-oriented description allows the identification of a modular set of standard services and standardized client/server interfaces. The reference model servers are potentially viable commercial products and are building blocks for higher-level services and recursive integration in centralized, shared, or distributed hierarchical storage systems. This integration can be done within single-vendor systems, by third-party, value-added system integrators, or by end-user organizations. The object-oriented modularity hides implementation details, allowing many possible implementations in support of the standard abstract objects and interfaces (Booch 86).

- For the storage system to be integrated with applications and operating systems supporting many different internal file structures, the abstract object visible to storage-system clients is an uninterpreted string of bits and a set of attributes.

- The separation of human-oriented naming from machine-oriented file identifiers allows integration with current and future operating systems and site-dependent naming systems. This implies separation of the name server as a separate module associated with the reference model (Watson 81b).

- The separation of access rights control as a site-specific module with a standard interface to the storage system accommodates the many operating systems and site-dependent access control mechanisms in existence.

- The separation of the request and data communication paths supports existing practices and the need for third-party control of transfers between two entities by direct data transfers from source to sink, as well as data transfer redirection and pipelining through such data transformations as encryption, compression, and check-summing.

- The separation of the site manager allows site-dependent policies and status to be managed. Provision is made for standard site-management interface functionality.

- Inclusion of a migration server within the file server allows each file server to be self-contained and file-migration policies for each server to be established separately. It also facilitates building a hierarchical storage system supporting automatic migration between servers. The general goal is to cache the most active data on the fastest storage servers and the least active on storage servers with the lowest cost-per-bit medium.

It is envisioned that the modules of the reference model can be integrated in various combinations to support a variety of storage needs from single storage systems to distributed, hierarchical systems supporting automatic file migration. Vendors can build and market individual standard modules or integrated systems supporting standard interfaces and functionality. Hopefully, the development of standards will increase markets and lead to modules and systems manufactured in larger numbers, thus reducing costs as a result of mass production economies.

To better understand the modularization and the requirements placed on interfaces but not to force a particular design philosophy, the discussion in this document does not restrict itself to external interfaces and services as might be expected of a reference model. The intent is not to standardize the internal structure of modules, since this is implementation- and vendor-specific, but to provide additional understanding to aid the model building, interface standardization, and implementation processes.

2.3 Reference Model Architecture

2.3.1 Abstract Objects

To follow the description of the reference model, there are several concepts that should first be established. These concepts employ the properties of abstract objects (Watson 81a), which have been succinctly listed as :

- Objects are an instance of a type (file, process, directory, account, etc.). As such, an object type is defined by:
  - An identifier.
  - A logical representation visible at an interface (e.g., a logical representation of a file is a set of attributes and a data segment of uninterpreted bits).
  - A set of operations or functions and associated parameters presented at the interface to create, destroy, or manipulate the object.
  - Specification of sequences of operations that are allowed.
- Objects are managed by servers. There can be many servers for a given type (e.g., there can be many file managers).
- Objects are of two basic classes, active and passive. To be manipulated, passive objects (such as files, directories, or accounts) must be acted on by requests from active objects presented at the server interfaces. Active objects, which are mainly processes, can directly change aspects of their own representation. Active objects can play either or both of two roles, a client role accessing a service, and a server role providing a service or managing a type of object.
- Objects are named via an identification scheme with a machine-oriented name that is unique throughout an environment. This identification scheme may be used in conjunction with protection and resource management schemes. Human-oriented naming is implemented by separate name servers that associate mnemonic, human-oriented names with the machine-oriented object identifiers. Higher-level file services might integrate the name and file services.
- Access to objects is controlled by the server through access lists, capabilities, or other techniques.

2.3.2 Client/Server Properties

The client/server model (Watson 81a) is an *object-oriented* paradigm. Simply stated, both clients and servers are active abstract objects in which the client requests services from the server through a specified interface. The word *client* is used to mean the program that accesses some service. The word *user* is reserved to mean the human at the terminal. A client is an *agent* of a user. The *server* is a provider of a service. Access to server-supported objects or services is only through defined server interfaces, thus hiding implementation details to provide transparency. Both the client and the server may be processes or collections of processes. These processes are not necessarily associated with any particular host machine. We describe the client/server interactions in terms of messages, but it is understood that local or remote procedure calls (Birrell 82) or other communications paradigms are possible.

A server may be thought of as a collection of one or more *tasks* or *processes* (concurrently executing instruction streams). A server may include request processing and other tasks supporting concurrent handling of requests from many clients. Clients may also be constructed as many cooperating, concurrent tasks.

Client and server processes interact by sending each other messages, in the form of requests and replies. A message is the smallest unit of data that can be sent and received between a pair of correspondents for a meaningful action to take place. A client process accesses a resource by sending requests containing the operation specification and appropriate parameters from one of its ports to a server port. A given process can operate in both server or client roles at different times (Watson 84). For example, during the migration of files, a file server that manages magnetic disks can play the role of client to a file server that manages magnetic tapes. Another example is a name server that stores its catalogs in a file server.

A distinction is drawn between the words *server* and *service*. A service may include several servers (Svobodova 84). For example, a directory service might be implemented by having separate name servers for objects such as files and for other objects such as users addresses or printers. On the other hand, one might implement a universal directory to provide the whole directory service (Lantz 85), or one might choose to implement the file service defined by the ISO-OSI Virtual Filestore (DIS 8571), where this reference model serves the unstructured file segment. Thus a complete file service will likely consist of name servers and multiple file servers.

2.3.3 Reference Model Modules

The primary reference model modules, shown in Figure 1, are:

- The *bitfile*[*] *server*, which handles the logical aspects of bitfile storage and retrieval,
- the *storage server*, which handles the physical aspects of bitfile storage, and
- the *physical volume repository*, which provides manually or robotically retrievable shelf storage of physical media volumes.

Closely related to these modules are:

- The *bitfile client*, which is the programmatic agent of the user required to convert user desires into bitfile requests to the bitfile server and data transfer commands to the bitfile mover,
- the *bitfile mover*, which provides the components and protocols for high-speed data transfer,
- the *name server*, which provides the retention of bitfile IDs and the conversion of human-oriented names to bitfile IDs, and
- the *site manager*, who monitors operations, collects statistics, and establishes policy and exerts control over policy parameters and site operation.

These modules are not directly associated with any particular hardware or software products.

The modularity of the reference model defines a virtual store for bitfiles. The storage system can be implemented with many levels of storage hierarchy, including a physical volume repository. The structure of the model permits standard interfaces and multiple instances of modules, and thus should facilitate more economical implementation of many forms of storage architectures. There may be many different instances of bitfile server and storage server combinations in which storage servers need not be of the same technology and can form a hierarchy.

The bitfile client represents the program object or agent that accesses bitfiles. This agent is not the application but acts for the application. The bitfile client can take many forms depending on how the storage system is implemented and integrated into a particular user environment; it might be one or more application programs or be functionally supported within an operating system to facilitate access to storage. The bitfile client may run on personal computers, workstations, or on larger machines. The bitfile client can also be a part of a data acquisition system needing the services of a storage system. The bitfile client can locate bitfiles by use of a name server. The user's

[*]The word "bitfile" was coined by the IEEE-CS Technical Committee on Mass Storage Systems and Technology to refer to a bit string that is completely unconstrained by size and structure; it was coined to relieve those who worked on the model from being bound by any particular file management system.

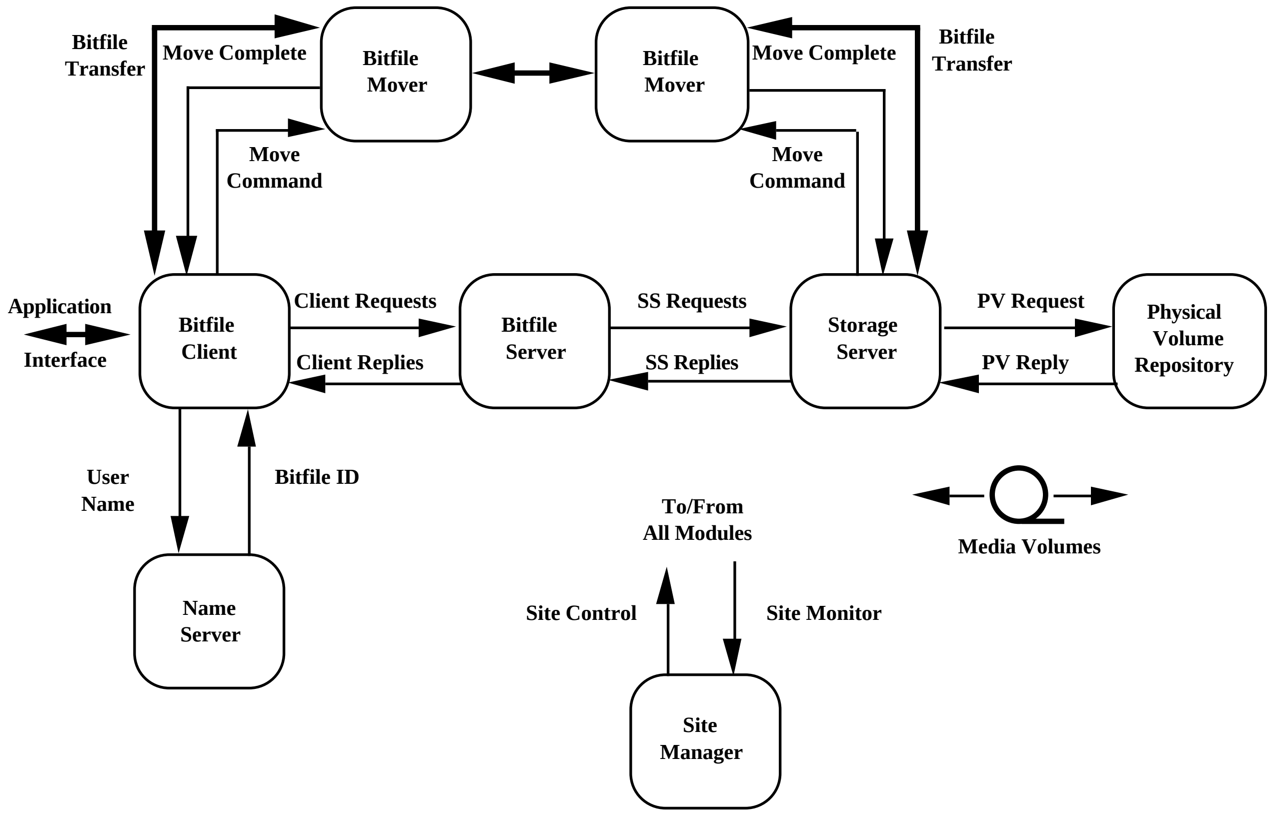

Figure 1

The Storage System Reference Model

human-oriented bitfile names are mapped to bitfile IDs and bitfile server addresses by the name server.

It is the interaction of the bitfile client with the bitfile server, the bitfile server interaction with the storage servers, and the storage server interaction with the physical volume repository that are of particular interest. There may be any number of bitfile clients in the general system environment of a site. Furthermore, bitfile servers or other entities of the total storage environment, such as name servers, site managers, or migration modules, may operate in the role of bitfile clients when they need storage service themselves.

A bitfile server handles the logical aspects of the storage and retrieval of bitfiles. The bitfile server's abstract object is the bitfile, identified by a globally unique, machine-oriented bitfile ID. A bitfile is a set of attributes (state fields) and an uninterpreted, logically contiguous segment of data bits.

A bitfile server may keep track of the bitfiles stored in one or more storage servers. A single bitfile server may control a hierarchy and need the services of several storage servers. As an alternative, a single bitfile server may handle the bitfiles in a single level of the storage hierarchy or a single storage technology; multiple bitfile server-storage server pairs simplify extensibility and evolution.

The bitfile server accepts requests from bitfile clients to create, destroy, store, and retrieve bitfiles, and to modify and interrogate the bitfile attributes needed for system management. The bitfile server contains a request processor to parse the requests and control the sequence of actions necessary to fulfill the requests. Before permitting access to a bitfile, the bitfile server authenticates the access rights of the requestor.

The bitfile server communicates action commands to the storage server. Each bitfile server contains a migration manager to prevent overflow of the storage space for which it is responsible. The migration manager knows which bitfile server is used to offload bitfiles, as established by the site manager and by migration and caching policies.

The storage server handles the physical aspects of bitfile storage and retrieval, and presents the image of perfect media to the bitfile server. (The *capacity* of the media, influenced by imperfections, may be visible to the bitfile server.) The storage server's abstract objects, as seen by the bitfile server, are logical volumes made up of an ordered set of bit string segments.

Physical volumes and volume serial numbers are not visible to the client. The site manager, however, may have privileged storage server commands where physical volumes and devices are treated as visible objects of the storage server. Volumes and devices are identified by volume and device IDs.

Bit stream segments are identified by segment descriptors. These segments represent how the storage server has allocated space for the bitfile data blocks. Each segment is identified by a descriptor generated by the storage server, and the ordered set is retained as a bitfile attribute by the bitfile server. The storage server must internally map bit string segments to real physical volumes (removable or not) and addresses where the bitfiles are stored, provide read/write (and some error management) of those volumes, and be able to access a bitfile mover to transmit and receive bitfile data blocks. One or more logical volumes may be mapped to a given physical volume, or one logical volume may be mapped to several physical volumes. Thus, a storage server contains storage devices, device-specific controllers that map bitfiles to bit string segments on physical volumes, and a means for handling and managing physical volumes on the storage devices.

The physical volume repository server manages a library that stores physical volumes such as tape reels, tape cartridges, or optical disk platters. Physical volumes, identified by physical volume IDs, are its abstract objects. A physical volume repository server can be used by one or more storage servers.

The site manager is a client process that can generate and send ordinary and privileged requests to the other servers to set policy parameters, install logical and physical volumes (import, export), obtain statistics and status, run diagnostics, etc.

The various clients and servers are interconnected through a communication service, which must handle all of the inter-process communications involved in requests and replies, as well as the high-speed transport of bitfiles. The elements of the communications service may be distributed through the many physical processors of the site. The reference model does not specify the details of this service but does assume its existence whether by procedure call, remote procedure call, or message passing. In particular, the model does not require homogeneity of this communication service. The communication service can include movement across I/O channels as well as across networks. The model assumes the ability to separate data movement from request movement. All that is required is that entities that must communicate are, in fact, able to do so and that standard interfaces are supported. The model has separated authentication details to allow each site to install the form appropriate to that site.

Existing storage systems often include a high-performance data path of some type, often specially designed, to handle the high-volume, high-speed data flow between the bitfile client and the storage server. In the reference model, the need for a high-speed data path is incorporated as part of the communication service referred to as the *bitfile mover*. This path has been separated from the request path to correspond to existing practice; to facilitate third-party control of transfers; to facilitate insertion of data transformations such as encryption, compression, and check-summing; and to support transfer redirection. If a general network is used for the communication service, it has to account for this need for high transmission speed of bitfiles as well as the communication of requests and replies.

3. Detailed Description of the Reference Model

This description of the reference model discusses the entities shown in Figure 1 in more detail. In describing the functions accepted by each module, input parameters that are common to all functions are deleted for clarity. These parameters include the identification of the client making the request or other access-control information, accounting information, and a transaction identifier. Similarly deleted are the transaction identifier and the success/fail indication common to all responses. If functions fail, error information will replace the expected responses. Optional parameters that can be defaulted are enclosed in square brackets (Miller 88a, 88b).

3.1 Bitfile Client

The bitfile client is the collection of hardware and software at a user node within the site to permit that node to use the storage system. Bitfile clients are responsible for providing the storage system interface to users at the terminal or to application processes by:

- Translating user and application requests for storage services into bitfile server requests, and

- providing communication with the appropriate bitfile servers and movers as determined by the name server mapping.

The bitfile client may be library routines within the application, an interface process (local or remote), or routines within an operating system kernel, and it may combine the services provided by multiple bitfile servers, bitfile movers, and name servers to form the higher-level abstract objects of an integrated storage service. The syntax and semantics of messages that flow between the bitfile client and the bitfile server should be the same regardless of the type of bitfile client. These messages identify the bitfile to be acted upon and specify which of the basic commands and parameters are to be processed.

When a bitfile is created, the bitfile ID is generated by the bitfile server and passed back to the bitfile client for retention. When the user accesses an existing bitfile by name, the bitfile client obtains the bitfile ID from a name server or some other data base system.

Several arrangements of the bitfile client are possible. In the first arrangement, the “kernel view”, all bitfiles are logically local. The bitfile client is a program in a processor with its own operating system and local storage. The storage and site-management capabilities of the local operating system are what the user sees with respect to how his bitfiles are handled. To the kernel of the operating system is added code that permits the operating system to determine if a bitfile being requested is locally stored or remote (Sandberg 85). If it is remote, this special code in the operating system kernel makes up the messages for the bitfile server and possibly for the name server. Alternatively, the local/remote distinction can be made in a library routine, and the library code can act as the bitfile client (Brownbridge 82). In either case, the bitfile is put into a local user buffer or is cached in an operating system buffer or a local bitfile and, except for a possible delay, the remote transfer is transparent to the user.

In a pure “client/server” view, all bitfiles are logically remote. Here, all references to bitfiles are translated into messages for a bitfile server, using routines in a library or other run-time support facility, such as a remote invocation system. The bitfile server might be local or remote; in a diskless workstation for example, the bitfile server is remote. Mapping human-oriented names to machine-oriented bitfile IDs is a separate, explicit step or is hidden in the run-time support facility (Svobodova 84, Watson 84).

In a third view, the systems that create and store bitfiles are separate from the systems that retrieve and process the data contained in the bitfiles. Such might be the relationship between a data acquisition system that puts bitfiles into the storage system and the systems in a data processing center that use them. While there is no name server per se, some means must be provided to retain bitfile IDs when they are returned from the bitfile server and to pass them in some understandable way (e.g. using a data base management system) to the processing systems. Such systems must take care to back up their records; if the bitfile IDs are lost, the bitfiles become lost objects in the storage system.

3.2 Name Server

The development of distributed systems has caused a much more in-depth look at schemes for identifying objects. While the advent of distributed systems brought this about, the requirements recognized are not restricted to distributed systems. They apply to all systems, especially those that grow in size. Dealing properly with naming is central to achieving the location transparency needed in a distributed system. Thus, it is advantageous to look at some of the properties of identification schemes.

There are many possible ways to designate a desired object (Watson 81b):

- by an explicit name or address (object *x* or object at address *x*),
- by content (object with value or value expression *x*),
- by source (all *my* files),
- by broadcast identifier (all objects of a class),
- by group identifiers (participants in class *x*),
- by route (all objects found at the end of path *x*),
- by relationship to a given identifier (*all previous sequence numbers*), etc.

A useful informal distinction between three important classes of identifiers widely used in system design—names, addresses, and routes—is (Shoch 78):

- The name of a resource is what we seek,
- an address indicates where it is, and
- a route tells how to get there.

One should not examine such informal definitions too closely. Names, addresses, and routes can occur at all levels of the architecture. Names used in the inter-process communication layer have often been called such terms as ports, or logical or generic addresses. A human-oriented chain or path name can be thought of as a “route” through a directory. An important idea is that identifiers at different levels of the architecture referring to the same object must be bound together in contexts, statically or dynamically. Later they must be resolved using these contexts to locate the named objects.

There are important system benefits if every bitfile ID is unique (Leach 82). Less obvious are the system-wide ramifications of the total naming system, especially the choice of the particular mechanism used to create unique bitfile IDs and the mechanism to associate application-dependent, human-oriented names with them. Of the many goals and implications of identification schemes enumerated by (Watson 81b), the goals that are the most pertinent to the reference model are abstracted and discussed below.

The naming system should:

- Support at least two levels of identifiers, one convenient for people and one convenient for machines. The latter is the bitfile identifier. The former will be handled by site or operating system specification of the name servers or by imbedding a name service in a higher level file service.

  The separation of identifier levels is very important because a storage system must be integrated with many types of heterogeneous applications and operating and storage systems (centralized and distributed), each supporting its own form of human-oriented naming scheme. The reference model provides a clean separation of mechanism for these two levels of identifiers and allows their easy integration. (When the client is responsible for the use of the bitfile ID, there is the potential to create lost objects in a system and thus mechanisms must also be included to assist the system in identifying them so that the resources they use can be reclaimed.)

- Support distributed generation of machine-oriented, globally-unique bitfile identifiers. A variety of mechanisms are available to support this need (Leach 82, Mullender 84, Watson 81b). One mechanism is to include both a bitfile server ID and a time stamp in the identifier. This structure, containing node or server boundary information, is at most a hint to applications as to where to send access requests and should not restrict migration. A machine-oriented identifier is a bit pattern easily manipulated and stored by machines and may be directly useful with protection, resource management, and other mechanisms. A human-oriented identifier, on the other hand, is generally a character string that is readable by humans and that has mnemonic value. Directory path names are a common mechanism (McLarty 84).

- Provide a storage system viewed as a global space of identified objects rather than as a space of identified host computers containing locally-identified objects. Similarly, the identification mechanism should be independent of the physical connectivity or topology of the system. That is, the boundaries of physical components and the connection among them as a network, while technologically and administratively real, are invisible in object identifiers. Further, an object's name should be independent of client or server location. Users should be able to discover or influence an object's location.

- Support relocation of objects. The implication here is that there be at least two lower levels of identifiers and that the mapping and binding between them be dynamic. For example, bitfiles are expected to migrate. Therefore, the bitfile IDs should not contain storage addresses, and there must be mechanisms for updating the appropriate context (e.g. directories and tables) when objects are moved.

- Support use of multiple copies of the same object. For example, a file may be cached on disks at one or more hosts, on staging disks, or it may be stored on an archival volume. If the value of the object is only going to be read or interrogated, one set of constraints is imposed. If values are to be written or modified, tougher constraints must be imposed to achieve consistency between the contents of the copies. Policies of enforcement of such constraints are handled using the basic locking services specified by the reference model.

- Allow multiple local, user-defined (human-oriented) names for the same object by allowing multiple mappings of a given bitfile identifier within the services of one or more name servers.

- Support two or more resources sharing a single instance of a storage object without identifier conflicts.

- Minimize the number of independent identification systems needed across and within architectural levels.

3.3 Bitfile Server

A bitfile server (Falcone 88) handles the logical aspects of bitfiles that are physically stored in one or more storage servers of the storage system. As shown in Figure 2, the major components of a bitfile server are a bitfile server request processor, a bitfile descriptor manager and its descriptor table, a migration manager, a bitfile ID authenticator, and a space limit manager and its space limit table.

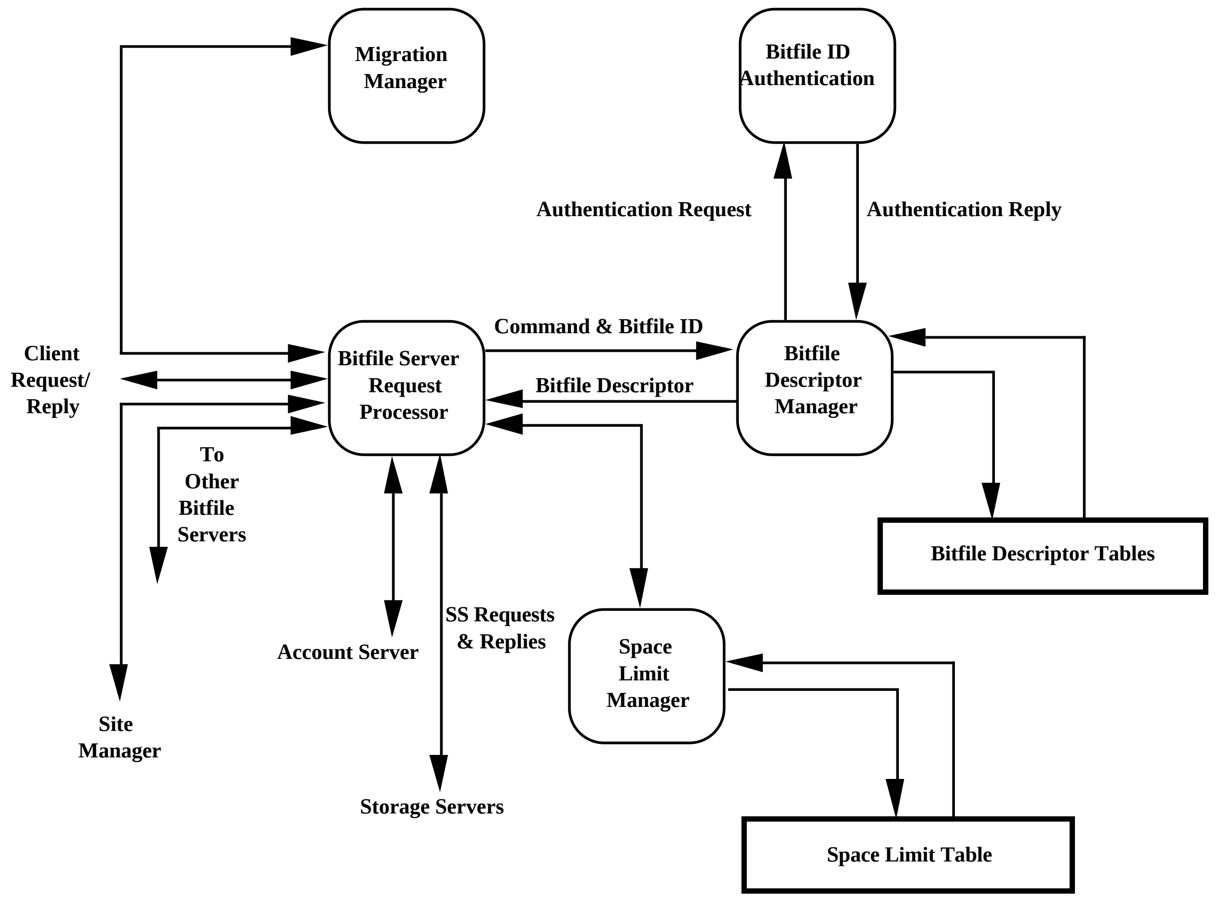

Figure 2

The Bitfile Server

The bitfile server accepts requests for service on bitfiles from the bitfile client, site managers, migration manager, and other bitfile servers. A discussion of the operations that bitfile clients can request of the bitfile server regarding the bitfile follows. The function parameters are shown in Table 1.

3.3.1 Bitfile Server Commands

Abort
The client requests that a previous request be aborted.

Create
This request is used to establish a new entry in the bitfile server's descriptor table. The requestor must prove his right to do so, and when this is established, he receives a new bitfile ID from the bitfile server, which may then be saved for use when the bitfile is accessed later.

Destroy
The client requests that a bitfile descriptor be removed from the bitfile descriptor table. The space allocated to the bitfile within a storage server is deallocated and, if the medium can be rewritten, the storage server returns it to the free-space list.

Erase
This request erases data on a storage server with the specified erasure pattern. If only a segment is to be erased, then the client must specify the length of the segment and an optional offset field displacing the length into the bitfile.

Lock
The client requests that a bitfile be locked for read access or for read/write access.

Modify
The client requests a change in one or more attributes of a bitfile contained in the bitfile descriptor table.

Query
The client requests information about a bitfile or a bitfile's attributes from the bitfile descriptor table.

Retrieve
The client requests that a bitfile or a segment of a bitfile be moved from the storage server median to the bitfile client or application buffer. If only a segment is to be moved, then the internal starting bit address (offset) and the bit string length of the segment to be transferred must be specified. (The data transfer is on a separate logical, and perhaps physical, path from the request/response path; the data block itself is <u>not</u> part of the response.)

Status
The client requests the status of a previous request made by the client. Although the way in which status is implemented is site dependent, generally a transaction ID must be generated to support the status request.

Store
The client requests that a bitfile data block be moved from a bitfile client or application buffer to the storage server medium. This request may include the ability to append a segment at the end of an existing bitfile or to update a physically specified segment of an existing bitfile. (The data block itself is <u>not</u> part of the request.)

Unlock
The client requests that any locks held be released.

The interface on which this list of commands can be sent is shown in Figure 1 as bitfile client requests and bitfile server replies.

The site manager will have additional privileged requests to control allocated space limits, examine all bitfile directory fields, set access control and migration policy parameters, etc.

3.3.2 Bitfile Request Processor

The bitfile server request processor accepts, parses, and executes request messages from:

- The bitfile clients, to store, retrieve, and manage bitfiles,
- the site manager, to provide monitoring and control,
- the migration manager, to move bitfiles to other bitfile servers, and
- other bitfile servers, to support migration.

The request processor is therefore essentially the sequencer and controller of actions within the bitfile server and the interface to the other storage system modules. Requests must be scheduled to provide the best possible response to the bitfile clients while optimizing the use of the available resources. Client-specified priority, bitfile size, and storage server availability may affect the request scheduling. These actions require a significant amount of processing. In executing a request, the request processor may interact with the bitfile descriptor manager to retrieve, create, or update bitfile attribute information, and with a storage server to allocate or release logical volume space for bitfile storage and to store and retrieve bitfiles. To select a storage server when a bitfile is created, the request processor must have information about the bitfile (bitfile size, the response desired, the protection and reliability desired, the type of storage desired, etc.) and must match this information with the characteristics of the available storage servers. Bitfile clients might be able to specify a specific storage server or logical volume as well.

Table 1
Bitfile Server Functions

Function	Parameters	Response
Abort	Transaction ID	
Create	[Initial length in bits] [Maximum length in bits] [Attribute Value/Name pair list]	Bitfile ID
Destroy	Bitfile ID	
Erase	Bitfile ID [Offset] [Length] Erasure pattern	Number of bits erased
Lock	Bitfile ID Lock type	
Modify	Bitfile ID New attribute name/value pairs	
Query	Bitfile ID Attribute name list	Attribute name/value pair list
Retrieve	Bitfile ID [Offset] [Length] Data transfer sink ID	Number of bits transferred
Status	Transaction ID	Transaction status
Store	Bitfile ID [Offset] [Length] Data transfer source ID	Number of bits transferred
Unlock	Bitfile ID	

The request processor is responsible, using the bitfile ID authenticator, for the security and integrity of the access to bitfiles, and for synchronizing the sharing of bitfiles through its locking services. The bitfile request processor collects accounting data from all affected sources regarding each transaction and sends them to the account service. The request processor also communicates with the space limit manager to determine that the resources assigned to a particular account are not overdrawn.

3.3.3 Bitfile Descriptor Manager and Descriptor Table

State and attribute information for each bitfile is kept in records in a descriptor table. Each record is called a *bitfile descriptor*. A descriptor manager provides an interface for the request processor to store, retrieve, and update bitfile descriptors. Bitfile descriptors are accessed using a bitfile ID as a key which is assigned by the descriptor manager when the bitfile descriptor is created.

A convenient way to classify bitfile descriptors is by origin and usage. Typical bitfile descriptor classes and some examples are:

- **Created and used by the bitfile client.**
  - comment
  - bitfile format
- **Created by the bitfile client and used by the bitfile server.**
  - access-control information
  - account ID
  - bitfile lifetime
  - desired level of redundancy
  - family attribute
  - maximum bitfile length
  - priority
  - security level
  - service class
  - storage class
  - type of storage desired
- **Created by the bitfile server and used by both the bitfile server and the bitfile client.**
  - access statistics
  - accounting statistics
  - bitfile allocated length
  - bitfile ID
  - bitfile length
  - creation time
- **Created and used by the bitfile server.**
  - last backup time
  - last migration time
  - location of backup copy
  - lock information
  - previous location
- **Created by the storage server and used by bitfile server.**
  - last device to write bitfile
  - location of bitfile

The importance of descriptor tables necessitates that backup and recovery be supported by the descriptor manager.

3.3.4 Bitfile ID Authenticator

The bitfile ID authenticator implements a mechanism, such as an access list or DES encryption used in a capability system, which protects the bitfile ID from being forged. It may also enforce security policy based on the security level of the bitfile, the request message, or the client. The authenticator is called by the descriptor manager when the bitfile ID is created to support the authentication mechanism and, when a request for access to the bitfile is received, to authenticate the bitfile ID presented by the client. If the access control is via an access control list, an identifier of the accessing entity (principal ID) must accompany the request and be checked against an access list that is kept, at least logically, in the bitfile descriptor. If access control is via a capability system, the bitfile ID may be encrypted along with some redundant and access-right information within the capability, and decrypted by the authenticator and compared against information in the descriptor when the bitfile is accessed. It is assumed that the authentication module can be added by a site or systems integrator since access control mechanisms and security policies are site-dependent (Jones 79b, Donnelley 80, Mullender 84).

3.3.5 Migration Manager

No single storage server now available can provide both the performance and large capacity often needed by a large storage system. A successful strategy is for a number of bitfile servers and their associated storage servers to be operated as a storage hierarchy.

A migration manager is associated with each bitfile server. The migration manager is responsible for maintaining enough free space on the logical volumes managed by its bitfile server (e.g. disk storage) to ensure that requests for new bitfiles can be honored. When the migration manager initiates a migration procedure, it first queries the bitfile descriptor manager for information about all of the bitfiles that might be migrated. This information might include the bitfile priority, size, locks, activity, idle time, and client-desired response. Bitfile clients may be given different degrees of control, by various site management policies, over the placement of their bitfiles in the storage hierarchy. Using policy set by the site manager, the migration manager determines which bitfiles should be moved. Finally, the migration server sends requests to the bitfile server request processor to move the bitfiles to a bitfile server “lower” in the storage hierarchy. (Bitfiles move “up” in the hierarchy, toward higher-performance servers, as they are accessed; this movement is orchestrated by the bitfile server request processor.)

An alternate configuration may permit the migration manager to act as a third-party controller to initiate the request for a move. The separate request and data paths in the reference model allow data to move directly from a source storage server to a sink storage server, even though a third party initiated the transfer between the two bitfile servers. A request path may span two or more bitfile servers until the bitfile is located. To increase performance during retrieval, it may be desirable to establish a direct data transfer path, bypassing some storage servers, once the bitfile has been located. Such might be the case when bitfiles are accessed on very rare occasions and it is not economic to bring them back up the hierarchy.

3.3.6 Space Limit Manager

The space limit manager checks to see what logical volumes a given account, user, or user group is allowed to use; it controls space allocations, number of bitfiles allowed, or other policy parameters associated with space resource management that a given site may wish to enforce. The space limit table has entries for each account or principal ID for maximum and current space and bitfile limits.

3.4 Storage Server

A storage server (Savage 88) may best be visualized as an intelligent storage controller and its suite of storage devices. A storage server consists of a physical storage system (containing the physical bitfile-storage medium), a logical-to-physical volume manager, a physical device manager, a means of command authentication (unless it is a trusted component of the storage control processor), and some part of or intimate connection to the bitfile mover. A diagram of the storage server is shown in Figure 3.

The abstract objects of the storage server that are visible to the bitfile server are logical volumes and bit string segment descriptors. The descriptors of the space occupied by a bitfile form an ordered set of bit string segments identified by descriptors, each of which contains the logical volume ID, the starting point of the segment on the logical volume, and the length of the segment. The bit string segment descriptors are created by the storage server and stored in the descriptor tables of the bitfile server.

Each logical volume is considered to be a logical image of flawless media usable for storing bitfile data blocks, thus providing the separation of physical and logical space. Separation of logical and physical volumes supports segment relocation when media fail, where new storage devices are introduced, and when space utilization or transfer rate are optimized. Any media area that is unavailable for data storage because of flaw areas, formatting, control tracks, etc., is excluded from representation in the logical volume by the logical-to-physical mapping function.

The list of operations supported by the storage server are listed in Table 2. The site manager has a number of privileged operations including create, destroy, modify, and query of logical volumes, physical volumes, and physical devices.

3.4.1 Physical Storage System

A *physical storage system* consists of the devices used to read and write volumes and the drivers to control those devices (to position heads properly in relation to the media before reading or writing, etc.). The available physical storage systems cover a broad spectrum of characteristics in terms of random or sequential access, rewritable or write-once media, capacity, and performance.

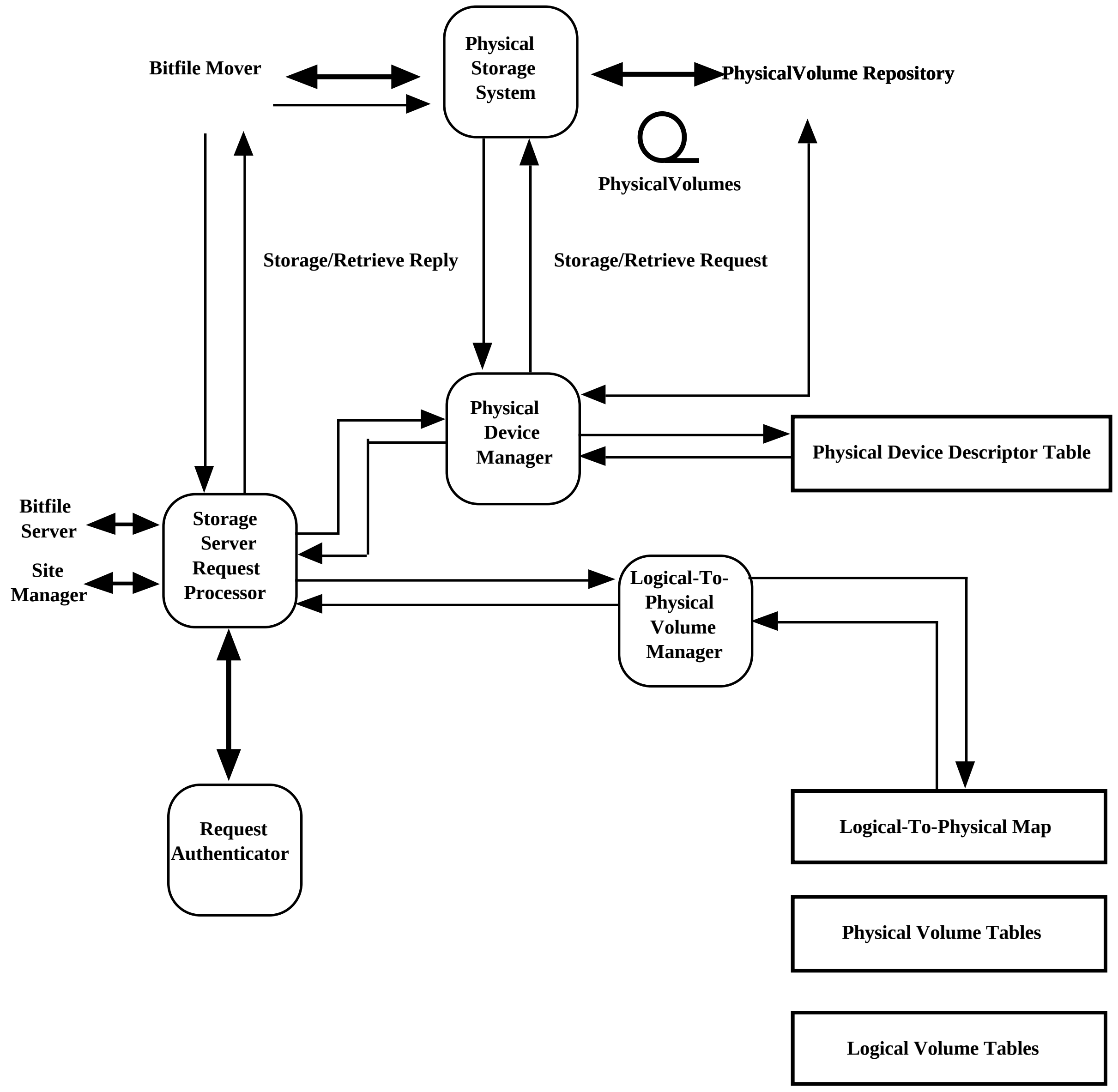

Figure 3

The Storage Server

3.4.2 Physical Device Manager

The *physical device manager* communicates with drivers in the physical storage system to load, unload, and position media volumes (it is the bitfile mover that controls the actual transfer of data).

Physical device managers vary from simple modules associated with fixed-media devices, such as Winchester disks, to complex modules that deal with manually mounted volumes, as in systems with standard magnetic tape drives or automatically mounted volumes, such as in the IBM 3850 and the STK 4400 Automated Cartridge Library. In automated systems, the physical device manager communicates with a physical volume repository to request that physical volumes be mounted. The physical device manager maintains a mounted volume table to optimize mount requests. It schedules and executes requests in a manner that attempts to give the desired response to its

Table 2
Storage Server Functions

Function	Parameters	Response
SS-Allocate	logical volume IDs existing segment descriptors desired length	new segment descriptors
SS-Deallocate	existing segment descriptors	
SS-Retrieve	segment descriptors starting offset bit string length sink descriptor	number of bits transferred
SS-Store	segment descriptors starting offset bit string length source descriptor	number of bits transferred

clients while at the same time making the best use of the storage system and communication resources. For example, it may be desirable to give higher priority to transfers for which volumes are already mounted or to small bitfile transfers, to limit the number of concurrent large bitfile transfers, or to use a client-specified priority.

3.4.3 Logical-to-Physical Volume Manager

The logical-to-physical volume manager maintains descriptors of attributes for each logical and physical volume and a set of tables to permit mapping the bit string segment descriptors of the logical volumes onto physical space in one or more physical volumes. The bit string descriptors include volume serial number, starting point address, and attributes for each logical and physical volume. Examples of attributes are creation time, size, security level, and physical volume attributes.

The logical-to-physical volume manager understands the characteristics of the actual physical volumes used by the storage server. Its main functions are to allocate and deallocate space and to convert logical bit string segments to physical bit string segments for that bitfile. The logical-to-physical volume manager also maintains a flaw map to map, for example, defective tracks on a magnetic disk to spare tracks (some device controllers maintain flaw maps, making duplicate maps in the volume manager unnecessary). Similarly, it maintains a map of disk tracks or magnetic tape block numbers that are used for control and formatting and that are thus unavailable for data storage. When data is moved within the storage system because errors start to occur or new physical devices or volumes are introduced, the map must be changed.

A map of the free and used space is maintained for each logical and physical volume. Space summary information for each volume may be retrieved to aid in the volume selection process. This volume information is retained in the storage tables, which must have the same reliability and performance as the directory in the bitfile server, i.e., it must be backed up and recoverable or it must be possible to build the information from other records. All of this information is available to the site manager interface.

3.5 Physical Volume Repository

The physical volume repository (Coleman 88, Savage 85), shown in Figure 4, stores physical volumes. It may be manual or mechanical.

The physical volume repository is responsible for managing the storage of media volumes and for mounting these volumes onto drives managed by the physical device manager. Volumes may be stored in an automated library that includes a robot capable of mounting the volumes or stored in a vault and mounted manually.

The architecture of the physical volume repository is that of a server that manages abstract objects called *physical volume*s. A physical volume consists of a media volume and a volume descriptor. (A physical volume is similar to a bitfile in that both include a resource and a resource descriptor.) The volume descriptor contains at least the following fields:

- The current physical location of the media volume. The volume might be in a vault, in a storage cell of an automated device, mounted on a drive, or held by a robot.
- A human-readable label by which an operator can identify the volume.
- The media type. One physical volume repository might manage both magnetic and optical media, different varieties of magnetic tape, etc.
- Information to identify the owner of the volume.
- Access-control information to validate requests. In a capability-based system, this information might be an encryption key. In other systems, this information might be a list of clients authorized to access the volume.
- Various statistics associated with the volume, such as the number of times the volume has been mounted and the time of the last mount.

Associated with each physical volume is a *volume identifier*. This identifier, when included in a request, allows the physical volume repository to locate the descriptor for the media volume and, in a capability-based system, proves that the client is authorized to access the volume. The format of the volume identifier is not specified by the reference model. If the medium is optical disk and only one side of a physical disk can be read at a time, there may be a unique volume identifier associated with each side of a disk.

The physical volume repository maintains the volume descriptors on a storage device to which it has access. The physical volume repository cannot maintain the volume descriptor on the volume itself because:

- The reference model does not specify the format of the data on a volume. In some implementations, the physical device manager may be able to read volume labels (using a bitfile mover), but if unlabeled volumes are allowed, only the bitfile client or the ultimate application can interpret the contents of the volume.
- Most types of archival media do not support “update in place”, preventing the physical volume repository from maintaining dynamic information, such as the time of last access, on the volume itself. Information on WORM optical disks, once written, cannot be modified. Some volumes, such as CD-ROMs, cannot be written at all.
- One of the important pieces of information in the volume descriptor is the physical location of the volume. One can hardly access the volume to determine where it is!

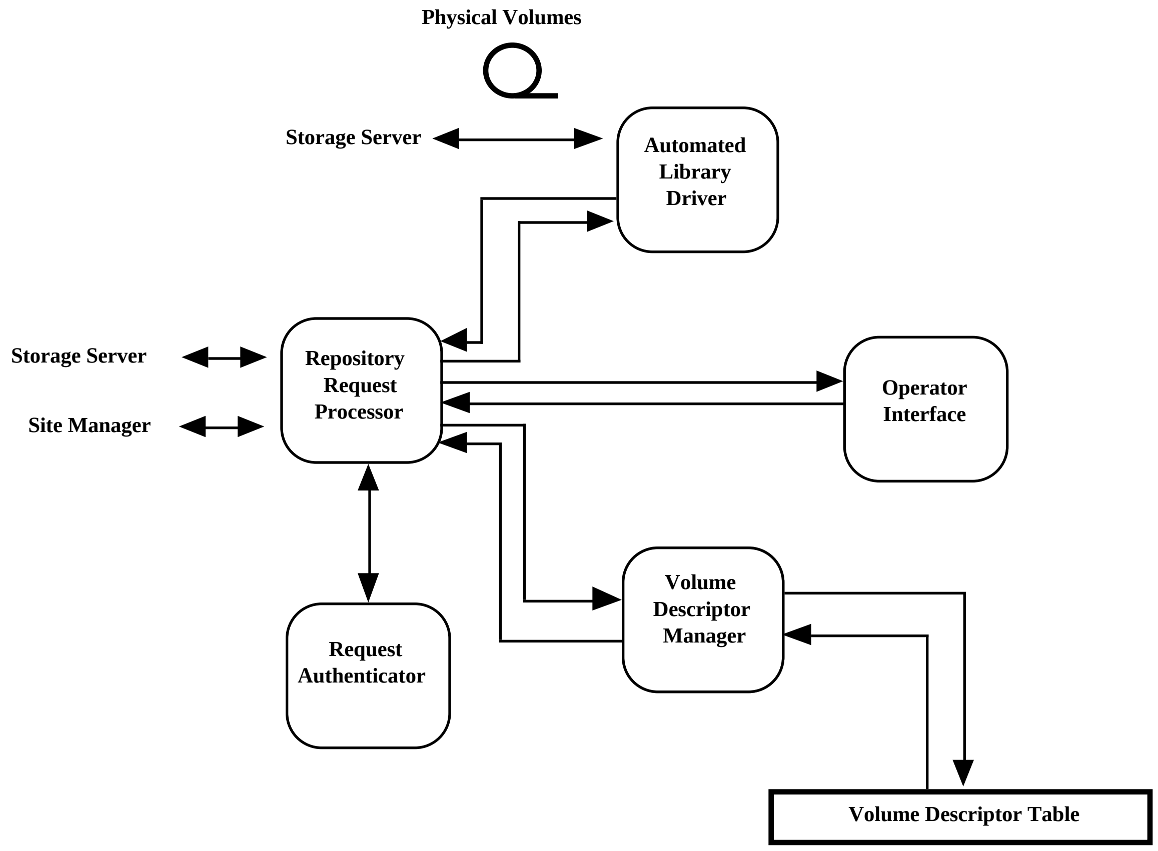

Figure 4

The Physical Volume Repository

The client interface consists of the operations necessary to allow the physical device manager to mount and dismount volumes and to allow the site manager to query and change the state of the repository. The operations and parameters that are unique to the physical volume repository are listed in Table 3 and described in the following paragraphs.

PVR-Dequeue

Any queued request for the specified volume with the specific write-protect mode that includes the specified drive as an acceptable drive is cancelled.

The dequeue function is routinely used by the physical device manager to remove requests for manually mounted volumes. Even though the physical volume repository maintains the queue of requested volumes, the physical device manager may be the only module able to detect that a volume has been mounted on a drive not accessible to the physical volume repository. If an operator inserts a requested volume into an automated library, the physical volume repository will mount the volume on an available drive; if the physical volume repository can identify the volume by reading an external label, and a request for this volume is queued, the physical volume repository will choose a drive acceptable for that request. Otherwise, the physical volume repository will choose any drive capable of handling the volume. In any event, the physical volume repository will not remove the request from the queue until it receives a dequeue command from the physical device manager.

Table 3
Physical Volume Repository Functions

Function	Parameters	Response
PVR-Dequeue	volume ID write-protect mode drive ID	
PVR-Dismount	volume ID drive ID	
PVR-Eject	volume ID reason	
PVR-Locate	volume ID	current volume location
PVR-Mount	volume ID write-protect mode list of acceptable drives	drive ID or queued for manual mount
PVR-ReadQueue	queue offset maximum number of entries to send	
PVR-ReadStatus	device ID type of status desired	device status
PVR-SetStatus	device ID type of status desired value	

PVR-Dismount

The media volume on the specified drive is dismounted and stored in a location selected by the physical volume repository.

The volume identifier is included in the dismount command to allow the physical volume repository to update its records; if the physical volume repository has a mechanism to identify the volume itself (by reading an external label), the volume identifier serves to confirm the physical volume repository's records and to detect anomalies.

PVR-Eject

The volume is removed from the domain of the physical volume repository. The “reason” parameter is an optional string to be sent to the operator explaining why the volume is being ejected.

In an automated system, the Eject command will probably result in the volume being moved to a port accessible by the operator.

PVR-Locate

The PVR-Locate function is used to determine volume locations when, for example, an automated library has failed and volumes are being accessed manually.

PVR-Mount

A media volume is mounted on a drive. Volumes queued for manual mounting are displayed on an operator console if the physical volume repository controls such a device (remotely controlled consoles can use the PVR-

ReadQueue command described below). Some physical volume repository implementations may allow concurrent requests in which no volumes of a group are mounted until all can be mounted, or requests with a choice of volumes.

PVR-ReadQueue
For each request queued for a manual mount, the volume identifier, list of acceptable drives, and the write-protect mode are returned.

Providing a queue offset and a maximum number of entries to send in the PVR-ReadQueue command allows the client to receive only the number of entries that it can handle. This function also supports operator displays not under the control of the physical volume repository.

PVR-ReadStatus
The amount and type of status information is dependent upon the devices controlled by the physical volume repository and upon their configuration. Status information might include the on-line status of the device, the volume identifier of the volume mounted on the device, current or previous error information, configuration information, etc.

PVR-SetStatus
The particular status values are dependent on the devices controlled by the physical volume repository. This function is used to bring devices on-line, take them off-line, set diagnostic or manual modes, etc.

3.6 Communication Service

The communication service includes the capability to communicate request messages as well as the bitfile mover (Kitts 88) for high-speed transfer of bitfile data blocks (Allen 83).

A bitfile mover is a set of modules that move data from one source/sink to another. A storage system includes at least two bitfile movers (Figure 1), one controlled by the bitfile client and the other controlled by the storage server. Additional movers may be required for more complex routing. Figure 5 shows the control and data paths necessary to move data from source to sink.

A source or sink can be defined as:

- A memory buffer, local or remote,
- a media extent, such as on local or remote disk, or
- a channel interface connected to a device.

These definitions do not limit the methods of data transport used by the bitfile mover or the ability to transform data during the move. Because the mover's source and sink interfaces depend on the devices, network interfaces, and network protocols used by the site, the reference model does not specify them. The bitfile mover's control interface to the source or sink manager, however, can be specified.

The Move operation supported by the mover is shown in Table 4.

The source and sink descriptors may describe network interfaces, buffer addresses, or device descriptors (device addresses, block information, etc). One descriptor is sent by the bitfile client; the other is provided by the storage server.

The transformation description may specify translation, compaction, compression, encryption, and/or check-summing to be performed by the mover.

The site manager interface can, through privileged commands, interrogate channel status and other mover statistics.

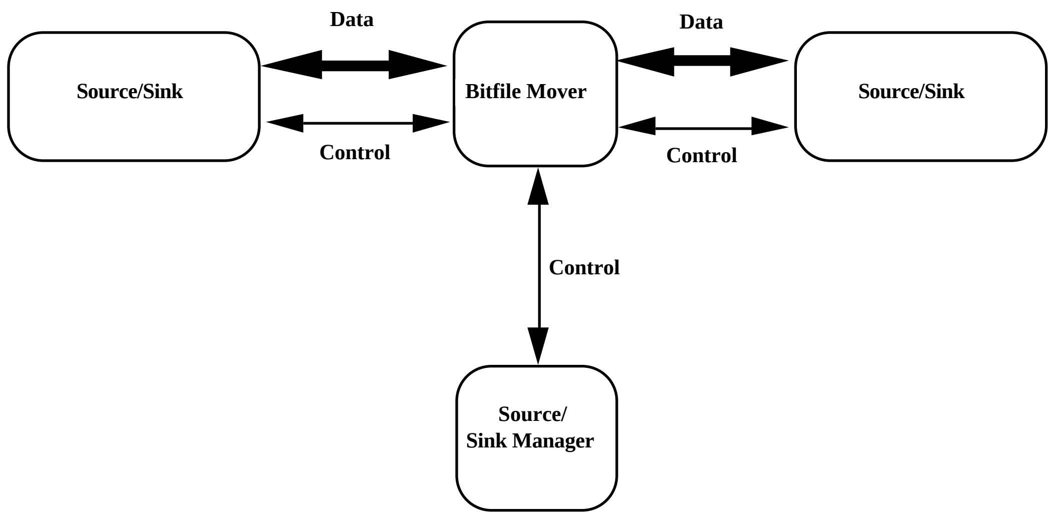

Figure 5

The Mover

Table 4
Mover Function

Function	Parameters	Response
Move source descriptor	direction number of sink bits moved sink descriptor transformation descriptor	number of source bits moved

3.7 Site Manager

Site management (Collins 88) is the collection of functions that are primarily concerned with the control, performance and utilization of the storage system. These functions are often site-dependent, involve human decision making, and span multiple servers. The functions may be implemented as stand-alone programs, may be integrated with the other storage system software, or may be policy.

Site management attempts to allocate the resources of the storage system to the best use for the overall benefit of the site. Policies for the site must be set, and the manual and automatic procedures must be developed to implement those policies. The procedures must be adaptable because the requirements will change as time progresses and because the same software may be run at a number of different sites.

For this discussion, the site management functions are grouped into seven areas: storage management, operations, systems maintenance, software support, hardware support, administrative control, and bitfile management. The format will be to state the requirements and then discuss the tools needed to satisfy these requirements for each area.

3.7.1 Storage Management

3.7.1.1 Requirements

The storage management function is concerned with providing good performance and reliability for user storage and access needs, while utilizing the storage servers in an efficient and cost-effective manner. The specific goals are to optimize the overall performance of the storage servers, to control placement of bitfiles in the storage hierarchy, to maintain sufficient free space in each storage server, to control fragmentation of space on volumes, to add and delete volumes, to recover data from bad volumes, to implement data backup policies, to enforce space allocation policies, and to determine the need for equipment. Most of the activities should be automated to the extent that the task of the human system administrator is primarily one of monitoring summary reports and using reports for planning purposes.

3.7.1.2 Tools Needed

The key to storage management for any storage system is the ability to gather and utilize information about the current state of the storage servers and statistics on their transaction histories. The total space, total free space, and distribution of free space on individual volumes define the state of a storage server. Information should be extracted from the transaction log of each storage server to give the number of bitfile accesses, amount of data transmitted, average and mean response times, average and mean data transfer rates, and patterns of access by bitfile age, activity and size. Performance monitor programs are needed to provide information such as the average wait and response times, resource utilization, demand and contention, and queue lengths for storage system components.

The migration manager component of the bitfile server is the primary tool for implementing storage management policies. The migration manager uses guidelines set by the system administrator as well as historical data and current state data to determine the amount of free space to keep available for each storage server, to decide what bitfiles (by activity, size, etc.) should be stored on each storage device, and to determine which bitfiles to move within the hierarchy to keep active bitfiles readily accessible.

The storage system should have automatic fragmentation control. Information about the amount of free space and allocated space can be used to determine when to re-pack volumes. This function may be performed by the migration manager or by some other storage server module.

Programs to analyze, initialize and label volumes should be provided, along with storage server commands to add and delete volumes. Two distinct types of volumes exist. Volumes like fixed magnetic disks are not demountable, and are usually defined to the operating system at system generation time. Demountable volumes such as tapes or optical disks are not subject to these constraints.

Storage servers and physical volume repositories should manage their demountable volumes largely without human intervention. The only human activities involved are occasional monitoring and revision of control parameters and supplying empty volumes to the storage server or physical volume repository when needed.

Two areas of storage management require the direct involvement of knowledgeable persons. The first is the recovery of data from bad volumes. Programs are needed to analyze, display and modify information on a volume, and to copy the entire contents of a volume to another volume without changing bitfile locations, skipping bad data which cannot be read after a number of retries. Data recovery may use the migration manager to evacuate a bad volume by moving individual bitfiles. The decision as to which type of recovery should be used in a particular case must be left to an experienced person.

System planning also requires human involvement. As new products are developed and old ones discontinued, changes in the storage servers are needed. In addition, changes in the network or user environment may require changes in the storage management policy or implementation. Statistical information may be used to decide when storage servers/devices should be enhanced, acquired and phased out. Changing patterns seen in usage information may be the best indicator that changes are needed in the policies of storage management.

3.7.2 Operations

3.7.2.1 Requirements

The operations functions are concerned with making sure that the storage system operates continuously and insuring that user requests are being satisfied in a timely and reliable manner. The system must be monitored and controlled to identify and resolve problems, to load/unload off-line volumes, and to verify that site management jobs have run correctly.

3.7.2.2 Tools Needed

An intelligent operations control center spanning the complete storage system is needed. Console displays need to show active requests for each server, requests queued for each server, volumes mounted for each storage server, space summary information (total number of volumes, number of empty volumes, free space, etc.) for each storage system in the hierarchy, resource status (processors, storage controllers, storage devices, communication links, volumes, etc.), and a special display for resources that are suspect or unavailable. Operator commands should be available for each server to restart or abort requests, and to set resources available or unavailable.

Storage system log information is needed. Messages that require action such as volume mounts and error messages should go to a display and/or hard copy console. All messages should be kept in a data base where they can be easily retrieved and displayed.

Job summary information is needed. Successful completion messages and error messages for system jobs should be written to a data base where they can be reviewed. When an important system job fails, such as backup of the bitfile descriptor tables, an operator action message should be issued.

The operational means to recover from temporary and permanent failures is needed. This includes the ability to isolate equipment which is failing or needs preventive maintenance (e.g. tape drives needing cleaning) and the ability to switch to backup equipment.

Automation of operations is needed to maximize the performance and reliability, and to minimize the manual effort. This includes automation of volume loading using a physical volume repository and automation of the decision-making process to minimize human errors and human delays.

3.7.3 Systems Maintenance

3.7.3.1 Requirements

The systems maintenance functions strive to maintain the performance, reliability, and availability of the storage system, and the integrity of the stored data. Performance is supported by monitoring the individual components and devices as well as the overall storage for failing components or out-of-balance conditions. The key to reliability and availability is the preservation of critical system information in an environment of possible hardware errors, software errors and system crashes. This information includes name server directories, bitfile descriptor tables, space limit tables, physical volume tables, physical device descriptor tables, logical-to-physical maps, logical volume tables, network configuration tables and transaction logs.

3.7.3.2 Tools Needed

System programmers must have the ability to quickly make changes in operating system or storage system parameters that affect the performance of the system. These tuning parameters may be available at execution and/or compile time. A “super-user” capability is needed so that a system programmer can execute all commands and have access to all system data.

Tools to maintain the integrity of information are needed. Programs must be available to back-up bitfiles and volumes, and to restore information from the backups. Additional tools must be available for important, dynamic tables and data bases where a backup quickly becomes out-of-date. One technique is to keep a secondary copy of dynamic information in addition to the primary copy. If either the primary or secondary fails, a new copy is immediately made of the good copy. Another technique is to keep a journal of the important transactions. If a failure occurs, the journal is applied to a backup to restore the information to the current level. The recovery programs needed to restore a backup to the current level following a crash must be available and well tested. Several persons should be familiar with the procedures required.

A checkpoint capability is needed to restore critical storage system tables and data bases to a consistent state if a crash occurs during a transaction that makes multiple changes (such as saving a bitfile which makes a new bitfile descriptor, updates the directory that points to the descriptor, and may update the accounting data base as well). During restart following an abnormal termination, the checkpoints are used to either complete or back-off requests so that the tables and data bases will be consistent.

Verification programs are needed to check the consistency of storage system information. These programs should be designed to run in parallel with the system so that operation may continue while verification is done.

Tools to help with problem determination are needed. These include trace capabilities, breakpoint capabilities, selected printing of formatted and unformatted dumps of data and programs, and dump analysis programs. Tools are needed to modify and repair storage system information.

3.7.4 Software Support

3.7.4.1 Requirements

For sites that develop new storage system software, facilities must be available to develop, maintain and test that software.

For customer sites, a test facility is required to verify that a new version satisfies local security and other requirements before production use.

3.7.4.2 Tools Needed

An environment must be provided to test software changes and enhancements without disrupting the production use of the storage system. The ability to run a test version and the production version of the software simultaneously is necessary. The test software may run on the same processors as the production software or run on other processors. It may share devices such as the communication systems and the physical storage systems, but it should have its own tables, data bases, volumes, etc. Instead of running a complete test version of the storage system software, a test version of a particular component (e.g., a bitfile server) could be run using components of the production system for the rest of the system.

A regression testing capability should be available so that a comprehensive set of tests can be run at any time against the production or test system to verify security, integrity, and performance. Both the running and checking of the regression tests should be automated.

3.7.5 Hardware Support

3.7.5.1 Requirements

The hardware support functions are concerned with the display, diagnosis and correction of hardware problems, and the configuration and installation of the hardware. Hardware failures and the time needed to repair failures must be minimized especially for those failures that bring down the storage system. It must be easy to reconfigure the system hardware, and install and remove equipment.

3.7.5.2 Tools Needed

Programs to report hardware errors are needed. These programs should be able to give a detailed time history of hardware errors, and show correlated summaries of both temporary and permanent errors by error-type, device-type, specific device and volume, over specified time intervals. The ability to recognize the beginning of a problem before it becomes permanent is especially important when dealing with storage devices/volumes where permanent errors generally mean the loss of data.

Programs to exercise and diagnose all hardware components of the storage system are needed. These programs should be able to analyze the errors and recommend the corrective action. Storage devices with mechanical parts, such as magnetic disk, optical disk, magnetic tape, and especially physical volume repositories, have a much higher error rate than strictly electronic hardware so diagnostic and exercise programs play an important role in storage systems.

The system should be redundantly configured so that components and paths can be isolated, removed for repair and upgraded with a minimal impact upon operation and performance. Dynamic reconfiguration capabilities, including the switching of the production software to a backup processor, should be available.

3.7.6 Administrative Control

3.7.6.1 Requirements

Administrative control covers the security, accounting, and management policies of the storage system. The security requirements are to implement the security policies of the site and to recognize if policy violations are being attempted. The accounting requirements are to gather usage information, to charge for use of resources, and to control the resources. The management requirements are to present summary information concerning the operation and performance of the system that can be used to justify operational and equipment expenditures and to set high-level policy.

3.7.6.2 Tools Needed

The storage system must implement the particular security policies of each site by building the policies into the programs or through the use of replaceable modules. In general, the policies involve verification that a user has access to the requested resources of a server. Access or capability information is stored with the resource and checked against similar information in the request. For some sites it is required that classification and partition levels be associated with bitfiles and requests, and that access be controlled based on certain classification and partition rules. A security log must be available that contains all security violations (as determined by a site) and all transactions above a specified security classification level. A log of all transactions should be kept to help diagnose anomalous situations.

The storage system needs a resource-charging mechanism. Charges may be incurred for the following resources and services: amount of data stored, number of bitfiles stored, data transferred, bitfiles accessed, and any of the requests made to the bitfile servers. These charges may vary for the different bitfile servers, depending on the level of performance and the class of storage used. Requests made to the storage system should contain an account code to which the charge is to be made. An account code can be stored in each bitfile descriptor along with the storage space used, the length of time stored, and a reference time for accounting purposes. An accounting program obtains the storage and bitfile charge information from the bitfile descriptors; obtain the access, data transfer and request charge information from the transaction logs; accumulate and sort the charge information; and write the charge information to an accounting file. Another accounting program has the resource charging rates and calculates the bills. Since the account codes often change, an automatic means of updating the bitfile descriptors is needed. One approach is to have a central data base of accounts from which an accounting program can update the bitfile descriptors. This data base can also be used to validate users and to show what resources they can use.

The summary information used by management to set high-level policy needs to be extracted from all the other site management reports and data bases, and presented in a useful (usually graphic) manner. A number of vendor products are available that can be used to extract, process and display information.

4. References

Allen, I. D. (1983). The role of intelligent peripheral interfaces in systems architecture. *Proc. Nat. Computer Conf.* pp. 623-630.

Almes, G. T., Black, P., Lazowska, D., and Noe, D. (1985). The Eden system: a technical review. *IEEE Trans. on Software Engineering SE-11.* (1), 43-59.

Birrell, A. D., Levin, R., Neddham, M., and Schoeder D. (1982). Grapevine: an exercise in distributed computing. *Comm. ACM*, Vol 25, No. 4, 260-274.

Booch, G. (1986). Object-oriented development, *IEEE Trans. on Software Engineering, SE-12*. (2), 211-221.

Brownbridge, D. R., Marshall, L. F., and Randell, B. (1982). The newcastle connection. *Software Practice and Experience* 12, 1147-1162.

Coleman, S. and Watson, R. (1984). Storage in the LLNL Octopus network: an overview and reflections. *Digest of Papers*, Sixth IEEE Symposium on Mass Storage Systems, Vail, Colorado, June 1984.

Coleman, S. (1988). Physical volume repository. *Digest of Papers*, Ninth IEEE Symposium on Mass Storage Systems, Monterey, California, November 1988.

Collins, B., Devaney, M., and Wilbanks, E. (1982). A network file storage system. *Digest of Papers*, Fifth IEEE Symposium on Mass Storage Systems, Boulder, Colorado, October 1982.

Collins, B. (1988). Mass storage system reference model system management. *Digest of Papers*, Ninth IEEE Symposium on Mass Storage Systems, Monterey, California, November 1988.

Davis, J. D. (1982). Mass storage systems: a current analysis. *Digest of Papers*, Fifth IEEE Symposium on Mass Storage Systems, Boulder, Colorado, October 1982.

DIS8571. *Information processing systems open systems interconnection, file transfer, access, and management* (in four parts, draft international standard ISO/DIS8571), distributed by Omicon Information Services.

Donnelley, J. E., and Fletcher, J. G. (1980). Resource access control in a network operating system. *Proc. ACM Pacific 80 Conf.*

Enslow, P. H., Jr. (1978). What is a "distributed" data processing system? *Computer*, Vol 11, No. 1, Jan, 13-21.

Falcone, Joseph R. (1988). The bitfile server in the IEEE reference model for mass storage systems. *Digest of Papers*, Ninth IEEE Symposium on Mass Storage Systems, Monterey, California, November 1988.

Fletcher, J., G. (1975). Computer storage structure and utilization at a large scientific laboratory. *Proc. IEEE* 63 (8), 1104-1113.

Foglesong, Joy, et. al. (1990). The Livermore distributed storage system: implementation and experiences. *Digest of Papers*, Tenth IEEE Symposium on Mass Storage Systems, Monterey, California, May 1990.

Gary, Mark (1990). Overcoming Unix kernel deficiencies in a portable, distributed storage system. *Digest of Papers*, Tenth IEEE Symposium on Mass Storage Systems, Monterey, California, May 1990.

Gentile, R. B., and Lucas, J. R. (1971). The TABLON mass storage network. *Proc. Spring Joint Computer Conf.*, pp. 345-356.

Grossman, C.P.(1989). Evolution of the DASD storage control, *IBM Systems Journal*, Vol.28, No.2, 1989.

Harris, J. P., Rhode, R. S., and Arter, N. K. (1975). The IBM 3850 mass storage system: design aspects. *Proc. IEEE* 63 (8), 1171-1179.

Hogan, Carole, et. al. (1990). The Livermore distributed storage system: requirements and overview. *Digest of Papers*, Tenth IEEE Symposium on Mass Storage Systems, Monterey, California, May 1990.

Howie, H. R., Jr. and Salbu, E. (1975). Mass storage implementation approaches: a spectrum. AFIPS The Information Technology Series, *Memory and Storage Technology*.

Johnson, C. (1975). IBM 3850 mass storage system. *AFIPS Conf. Proc.* 44.

Jones, A. K. (1979). The object model: a conceptual tool for structuring software. "Operating Systems". Springer-Verlag, Berlin.

Kitts, D. (1988). Bitfile mover. *Digest of Papers*, Ninth IEEE Symposium on Mass Storage Systems, Monterey, California, November 1988.

Kuehler, J. D. and Kerby, H. R. (1966). A photographic mass storage system. *AFIPS FJCC Proc.* 29, 735-742.

Lantz, K. A., Edighoffer, J. L., and Hitson, B. L. (1985). *Toward a Universal Directory Service*. Report No. STAN-C5-85-1086, Stanford University.

Leach, P. J., et al (1982). UIDs as internal names in a distributed file system. *Proc. Symposium on Principles of Dist. Computing*, Ottawa, 34-41.

LeLann, G. (1981). Motivation, objectives, and characteristics of distributed systems. "Distributed system-architecture and implementation". Springer-Verlag, Berlin, 1-9.

Mclarty, T., Collins, B. and Devaney, M. (1984). A functional view of the Los Alamos central file system. *Digest of Papers*, Sixth IEEE Symposium on Mass Storage Systems, Vail, Colorado, June 1984.

Miller, S. W. and Collins, B. (1985). Toward a reference model for mass storage systems. *Computer*, Vol. 18, No. 7, July, 9-22.

Miller, S. W. (1988a). "A Reference Model for Mass Storage Systems", *Advances in Computers*, Volume 27, Academic Press.

Miller, S. W. (1988b). Mass storage reference model, special topics. *Digest of Papers,* Ninth IEEE Symposium on Mass Storage Systems, Monterey, California, November 1988.

Mullender, J., and Tannenbaum, A. S. (1984). Protection and resource control in distributed operating systems. *Computer Networks* 8, 421-432.

Nelson, M., Kitts, D. L., Merrill, J. H., and Harano, G. (1987). The NCAR mass storage system. *Digest of Papers*, Eighth IEEE Symposium on Mass Storage Systems, Tucson, Arizona, May 1987, pp. 12-20.

O'Lear, B. T. and Choy, J. H. (1982). Software considerations in mass storage systems. *Computer* 15 (7), 36-44.

Penny, S. J. and Alston-Garnjost, M. (1970). Design of a very large storage system. *Proc. Fall Joint Computer Conf.* pp. 45-51.

Sandberg, R. (1985). Design and implementation of the SUN network file system. *Proc. Tenth Usenix Conference*, 119-130.

Savage, P. (1985). Proposed guidelines for an automated cartridge repository. *Computer*, Vol 18, No. 7, July, 49-58.

Savage, P. (1988). Storage server as physical box. *Digest of Papers*, Ninth IEEE Symposium on Mass Storage Systems, Monterey, California, November 1988.

Svobodova, L. (1984). File servers for network-based distributed systems. *Computing Surveys, 16*, (4), 354-398.

Watson, R. W. (1980). Network architecture design for a back-end storage network. *Computer*, Vol 13, No. 2, Feb, 32-48.

Watson, R. W. (1981a). Distributed system architecture model. *Distributed Systems—Architecture and Implementation*, Springer-Verlag, Berlin, 10-43.

Watson, R. W. (1981b). Identifiers (naming) in distributed systems. *Distributed Systems—Architecture and Implementation*, Springer-Verlag, Berlin, 191-210.

Watson, R. W. (1984). Requirements and overview of the LINCS distributed operating system architecture. Lawrence Livermore National Laboratory, Preprint UCRL-90906.

Watson, R. W. (1987). Tutorial notes, Eighth IEEE Symposium on Mass Storage Systems, Tucson, Arizona, May 1987.

Watson, R. W. (1988). The Architecture of Future Operating Systems, UCRL Preprint 99896, presented at the Cray Users Group Meeting, Tokyo, Japan - September 1988.

Wildman, M. (1975). Terabit memory system: design history. *Proc. IEEE* 63 (8), 1160-1165.

5. Glossary

Authentication Request/Reply
The command to test the access rights of the requestor to a particular service.

Bitfile
A collection of data that can be created on, read from, written into, and deleted from a storage system. These data are treated as a string of bits without any particular structure.

Bitfile Authenticator
The process that checks the access rights of a requestor for service.

Bitfile Descriptor
The bitfile attribute information that is stored as an entry in the bitfile descriptor table.

Bitfile Descriptor Manager
The process that manages the bitfile descriptor table.

Bitfile Descriptor Table
The data store where the bitfile descriptors are stored.

Bitfile ID
A machine-oriented, globally unique identifier of a bitfile.

Bitfile Mover
The processes, including the high-level protocols, that control the movement of bitfiles.

Bitfile Server
The set of processes that control the creation, destruction, and access to the many bitfiles under its control.

Bitfile Server Request Processor
The portion of the bitfile server that acts upon requests and controls the request/reply communications with internal modules as well as other processes and servers.

Bitfile Transfer
The high-speed movement of bitfile data blocks.

Client Request/Reply
The list of permitted commands from a client to a server and the resulting responses.

Create
The bitfile client request to form a bitfile descriptor record in the bitfile descriptor table. The bitfile attributes to be contained in the bitfile descriptor are specified in the request.

Destroy
The bitfile client request to remove a bitfile descriptor from the bitfile descriptor table. The space allocated to the bitfile is deallocated and, if the media can be rewritten, returned to the free space list.

Lock
The bitfile client request to establish a lock for a bitfile in preparation for one or more stores or retrieves of the bitfile.

Modify
The bitfile client request to change one or more attributes of a bitfile as contained in the bitfile descriptor table.

Monitor Information
Status information from storage system modules used by the site manager to assist in the management and control of the storage system.

Move Command
The request to move a bitfile between specified devices.

Name Server
The server that converts between the human-oriented name for a bitfile and the machine-oriented name for the same bitfile.

Physical Volume
A bounded unit of storage media that is used to store bitfiles.

Physical Volume Move
The physical movement of a volume between the volume repository and a storage server or its return.

Physical Volume Repository
The place where physical volumes are stored when they are not at a read/write station.

Principle ID
Identification of the agent requesting service from the bitfile server.

PVR-Dismount
A request sent to the physical volume repository to remove a physical volume from a drive.

PVR-Mount
A physical volume ID sent to the physical volume repository with the request to mount it on a particular storage device in the storage server.

Query
The bitfile client request to obtain information about a bitfile or its attributes from the bitfile descriptor table.

Retrieve
The bitfile client request to move a bitfile or a segment of a bitfile from a storage server to the bitfile client. If only a segment is to be moved, then the internal starting bit address and the bit string length must be specified.

Site Control
Commands from the site manager for initial set up, operations, and management of the storage system.

SS-Allocate
The request to a storage server to make logical space available for bitfile storage.

SS-Deallocate
The request to a storage server to remove a bitfile from physical storage and return the space to the free space list.

SS-Retrieve
The request from a bitfile server to move a bitfile from a storage server to a bitfile client.

SS-Store
The request from a bitfile server to move a bitfile from bitfile client buffer to storage server media.

Status
The bitfile client request for the status of the bitfile server or of a previous request made by the bitfile client.

Store
The client request to move a bitfile data block from the bitfile client to a bitfile server medium. This request may include the ability to append a segment at the end of an existing bitfile or to update a physically specified segment of an existing bitfile.

Unlock
The client request to release the lock held on a bitfile.