Applying Database Technology in the ADSM Mass Storage System

Luis-Felipe Cabrera, Robert Rees and Wayne Hineman
IBM Almaden Research Center
650 Harry Road, San Jose, California 9512-6099
e-mail: {cabrera, rees, hiney}@almaden.ibm.com

Abstract* - *Our success in deploying the illusion of infinite storage to applications rests in the use of database technology. This paper presents the support for transactions in the ADSTAR Distributed Storage Manager (ADSM) system. For a user, ADSM offers a backup and archive service in a heterogeneous client-server environment. It also operates as a file migration repository in some Unix environments. As a storage manager, the ADSM server is a Mass Storage System (MSS) that administers storage hierarchies of arbitrary depth in which all activities are done on behalf of transactions. Its systems goals include to operate in many computing platforms, to provide highly-available metadata, to administer effectively a large amount of entities, to support continuous and unattended operation, and to support a high degree of concurrent requests.

The workload includes requests that only read system data at the server, requests that store gigabytes of user data and requests that update thousands of system data entries at the server but require no access to user data. To ensure availability of the system data we replicate it with up to three copies and also support fuzzy dumps. As the ADSM server administers from low-latency high-performance magnetic disks to optical or tape jukeboxes, for high concurrency we adopted optimistic approaches to locking, special locks for devices called leases and we do not always enforce repeatable reads.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.
Proceedings of the 21st VLDB Conference
Zurich, Switzerland, 1995

Keywords: client-server transaction system, ADSM, hierarchical storage system, mass storage system, storage hierarchies, network backup service, storage management, atomic actions, optimistic concurrency control, bitfiles, client-server backup, file backup, file archive, file migration.

1. Introduction

ADSM is a client-server backup and archive IBM product since July of 1993[10]. It is also an announced repository of migrated files in some Unix environments since June of 1995. *Users*, humans, access the system through the ADSM data capture clients, henceforth called *clients*, that operate in the node where the user is located. The system stores the data of users in the ADSM storage servers, henceforth called *servers*. Data is transmitted between the clients and the servers using a special transfer protocol.

Users can backup and restore files without operator intervention in ADSM. They can do incremental or full backups at will. The granularity of a request can range from a single file to a complete file system. Backup and archive sessions can also be centrally initiated by the system. Clients operate on PC-DOS, Windows, MAC, Novell, Sequent, Windows NT, OS/2 and different UNIX versions. Communication between a client and a server can be through TCP/IP, APPC, NetBios, Novell IPX, named pipes, PWSCS and 3270 data streams. Servers currently run on MVS, VM, AIX, OS/2, HP/UX, SUN/Solaris, AS/400 and VSE. Approximately 80% of the server source code and 75% of the client source code are platform independent.

The facilities present in the system include administration of one or more storage hierarchies of arbitrary depth, support for an open ended collection of storage devices, policy-driven data and storage administration, registration of clients and of users, remote server administration, central and distributed scheduling of activities, and non-stop operation of the service when adding or deleting physical or logical storage units[3].

The following principles guided the design and development of ADSM:

- Enable the server to preserve the physical locality that data has at its source even when clients deliver this data over long periods of time.
- Estimate the target workload and develop the system to accommodate them.
- Deploy the system in many computing platforms.
- Provide lights-out, unattended, operation with unattended recovery from failures.
- Provide continuous operation.
- Accommodate the peculiarities of a wide variety of storage devices.
- Manage the storage system through user-controlled policies.
- Minimize the periods of time in which an entity in the system cannot be retrieved.
- Use additional temporary storage space to gain concurrency and to increase the availability of user data.

The rest of this paper is organized as follows. In section 2 we describe the user view of the system and in section 3 we present the transaction technology. Section 4 has the software architecture of the server, section 5 provides the workload characteristics of the environment, section 6 discusses index management while section 7 discusses the atomicity in the server. Section 8 presents related work and section 9 contains our conclusions.

In[3] we presented how the system supports heterogeneity, policy management and unbounded storage including continuous co-location of user data.

2. The External View of ADSM Storage

Space to store user data is represented by *storage pools* in the server. Storage pools have names that are unique among all storage pools. A storage pool can be chained to another storage pool and form arbitrary directed acyclic graphs. Storage pools have high and low occupancy thresholds, set by administrators, that trigger data migration to their chained storage pools. Data migration may also be on demand. Storage pools may be mapped to any type of physical storage. All the devices on which a specific storage pool is mapped must have the same storage characteristics.

A *bitfile* is an uninterpreted sequence of bytes of arbitrary length[15]. When a client stores a user file as part of a backup session or a file migration, the server creates a bitfile for the data of the user file and stores, in appropriate server catalogues, some 400 bytes of system data, henceforth called *metadata*, to describe all the necessary attributes of the user file. The server has approximately 60 catalogues for its metadata. Bitfiles are never modified in the server. They are migrated complete between different levels of the storage hierarchy and may be retrieved even when being migrated. The maximum size a bitfile may have is 2^{63} bytes. The server strives to minimize the number of hardware repositioning operations required to read all the data in a bitfile.

Data and storage administration policies are expressed by users and administrators using *management classes*. Management classes are stored at the server. Each user file is associated with a unique management class. Storage policies of the management class determine the storage pool a file is first stored in, the retention period of a copy of a user file, the number of copies of a user file that should be kept and the period of time that the last copy of a deleted user file is to be retained. These policy-controlled properties are associated to bitfiles using server catalogues.

To support the backup of the ever growing amount of data stored in computers, we adopted the strategy of not requiring full backups, or full dumps, of the user data. Only a continuous series of incremental backups is necessary[3]. The performance of restore operations is maintained using on-line catalogues. Thus, restoring a user file takes time that is proportional to the time to access the on-line metadata about the file plus the time taken to retrieve the bitfile from the appropriate storage pool. In the server it takes the same time to retrieve the metadata about a file in servers with 40 million bitfiles to 1 billion bitfiles.

To enable third parties to exploit the data administration function an external Application Programming Interface (API) is available. This API was standardized through X/Open[5]. It is currently deployed in platforms in which clients operates including Windows, Novell, UNIX and OS/2.

2. Transaction Technology in ADSM

All client-server activities are accomplished by *transactions*[2,6]. The transactional capability eases system administration as software or hardware failures always leave consistent the internal state of the system. Users are unaware of transaction boundaries yet the system exploits them in the case of crash recovery.

The clients control the transaction boundaries for user-initiated requests. A backup session, for example, is done as a series of transactions that store sets of files from the client in the server. The client commits transactions and begins new ones transparently to the user during a backup session. Through system parameters a commit happens after a given number of user files have been transmitted or after a given volume of user data is transmitted. Large user files are always stored within one transaction.

The server also uses transactions to perform its own internal activities. The server implements recovery mechanisms that are necessary to preserve failure atomicity and durability, which are two of the four properties defined for ACID transactions[2,6]. Consistency, the third property of ACID transactions, is guaranteed by the server software. The server does not always guarantee isolation, also called serializability. Isolation is commonly guaranteed by traditional database management systems, but it is not semantically appropriate for our backup and archive service. The enumeration of bitfiles in a storage pool, for example, is not guaranteed to remain unchanged for the duration of a transaction.

The availability of server metadata is enhanced using *replication*. The server can maintain up to three copies of the metadata. A replicated write returns when all copies have been written. A replicated read returns when the closest copy has been read. As there are more reads than writes directed to the metadata, the run-time of the system improves when replication is being used. System administrators can enable, disable, or change the replication factor on-demand, without bringing the system down or stopping its normal operation.

The speed to bring on-line a stale copy of replicated metadata depends on the underlying input-output configuration. The worst-case scenario is to bring up a replica on the same disk arm as the data. We measured this in an otherwise idle RS/6000 model 370 with a 2 gigabyte SCSI attached disk. The data rates varied from 1.941 megabytes/second to 2.091 megabytes/second with the 75th percentile being 2.054 megabytes/second.

4. Software Architecture of the ADSM Server

For backup, archive and migration the server is in charge of receiving and storing sets of user files from different clients. For restore and recall it needs to retrieve sets of bitfiles stored in its storage pools and transmit them to the appropriate clients. To support concurrent activity the server was developed as a multi-threaded, memory-sharing system. Threads are provided by a platform-independent module whose implementation is platform dependent. Figure 1 shows the principal functional software components of the server.

Activities in the system are triggered by external requests and by internal events. Registering a client, registering a system administrator, registering a user, specifying a management class, updating the specification of a management class, establishing a connection from a registered client, closing such a connection, backing up a set of user files, restoring a set of files, archiving a set of files, adding a new set of storage devices, specifying a storage pool, are examples of external requests. Two internal events are the detection (at the end of a transaction) of a storage pool that has exceeded its high occupancy watermark and scheduling a data migration operation for it, and determining that an instance of removable storage media is below its occupancy threshold and scheduling a data reclamation operation.

To understand how the components of Figure 1 fit together, let us walk through an example of backing up a set of files. The request begins with the client establishing a connection to the server. All connections are monitored by the Session Manager. The client then begins a client-server transaction that registers in the Transaction Manager (TM). Every component in the server that performs work on behalf of a transaction registers with the TM and obeys a two-phase commit protocol. The TM uses the LOG to write all the necessary log records. The Index Manager uses two kinds of B^+-Tree indices to store system data. The Logical Volume Manager (LVM) is used by the LOG and the Index Manager to access all the server system data, including log and catalogue data. Partial write detection and metadata replication are done by the LVM. The Session Manager can determine when the client end-point of a connection has failed and issue the corresponding abort action.

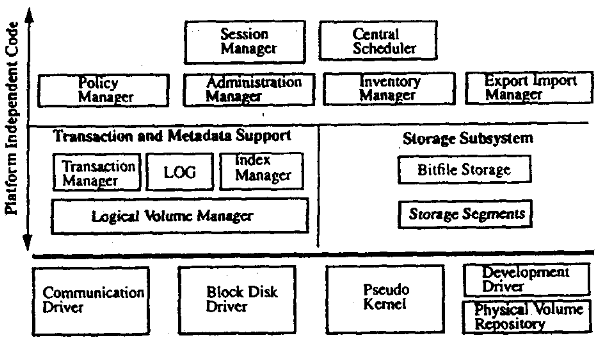

Figure 1. ADSM Server: Principal Software Components

For incremental backup, the session begins by the client requesting from the server the corresponding policies from the Policy Manager. Policies are stored in system catalogues administered by the Index Manager. The client then builds the candidate list of files for backup and requests the server to send, from the Inventory Manager, the latest information for each of the candidate files. The Inventory Manager stores all its information in catalogues administered by the Index Manager. The client then sends to the server, optionally compressing them, only those user files that have changed or have been created since the last incremental backup. The server constructs appropriate entries in the Inventory Manager and appropriate bitfiles using the Bitfile Storage component. Bitfiles are stored using the Storage Segment manager.

At the end of sending all the user files the client also sends the server the list of user files that have been deleted since the last incremental backup allowing the server to mark them and, eventually, to expire them from the system. The client then commits the transaction and disconnects from the server.

The Export Import manager is used to bulk transfer entities between servers.

The server supports a wide variety of storage devices such as disk, optical libraries, disk arrays, stand-alone tape devices or tape libraries. The server uses a common device driver model to ease incorporating new devices implemented by the Block Disk Driver. The driver model captures properties like sequential or random access and being read-write or write-once.

5. Workload Characteristics in the ADSM Environment

The workload in the network-based backup and archive system is unconventional. Transactions seldom abort yet may require half an hour to complete. Some modify minimum amounts of metadata yet store large amounts of data. Others modify large amounts of metadata and access no data. Many are read-only.

The most common initial use of the system was backup and archive. The system is evolving to also administer migrated primary data from remote file systems. When a client queries the server for the attributes of a set of user files the server processes this read-only query against its catalogues. For clients that are large file servers this query returns thousands of entries to the client.

Requests may require substantial processing. The transaction that stores a gigabyte user file being transmitted over a standard network will take a good fraction of an hour. The server must transfer all client data to the first appropriate storage pool. In addition, substantial amounts of data may be migrated within the storage hierarchy requiring corresponding amounts of processing. An MVS server with 3090 disks and 3480 tape drives being fed data from an RS/6000 model 530H client through a 16 megabit/second token ring and 3172 controller achieved backup throughput of 394 kilobytes/second and disk to tape migration throughput of 727 kilobytes/second. This migration throughput was unaffected when two clients were concurrently backing up data. The AIX server has been measured to receive data from an Ethernet at 9.5 megabits/second and from a 16 megabit/second token ring at 15.4 megabits/second.

The system also receives sporadic requests for single retrievals. In our Research Center, where some 450 workstations use one server for backup, there are some 20 requests per day to restore individual files. We have also confirmed that the probability of retrieving a bitfile decreases very rapidly with time. If a bitfile is not recalled within 15 days of being placed in storage its probability of being recalled individually is very low[19]. To increase the probability of fast restores we cache copies on disk when migrating down in the storage hierarchy. We then reclaim the space of cached bitfiles on-demand, not eagerly. Reclaiming the space of bitfiles on any media only requires updates to the on-line metadata.

In the system individual deletion of bitfiles is infrequent. However, as bitfiles corresponding to user files that have been deleted have expiration dates, there is the potential for expiring simultaneously large numbers of bitfiles. This happens, for example, when at a client a user deletes a complete directory and the expiration time at the server is reached. Expiration processing only requires updating the metadata of the server like deleting entries from catalogues, updating space accounting in storage pools and updating the metadata for the underlying storage media instances. Expiration processing may involve voluminous amounts of server metadata.

The backup and archive of several files is the norm. Clients seldom send individual files. The system exploits this by using transaction boundaries as units of data streaming over network connections. This approach is particularly helpful to "turn around" (without special messages) the semi-duplex connections between a client and a server.

Retrieving a large collection of related bitfiles, like the bitfiles that correspond to all files originally stored in a specific device at a client installation, poses interesting data management constraints. This infrequent user request is worth supporting efficiently as its completion time directly impacts users. The common user scenarios for this request are that a disk fails, or that a user wants to retrieve a complete file system, probably into a new environment for the data.[1]

To minimize the elapsed time of restore the system offers the option of co-locating logically-related files in a minimum number of storage media instances[3]. When co-location is enabled, user files are tagged by the server with co-location keys. At the server, bitfiles with common co-location keys are stored in close proximity in the corresponding storage pools. Co-location is preserved

1. In a large customer installation the IS organization reported one PC hard disk failure every 29 hours. Low-end disks may fail once in 30,000 hours.

when the server migrates bitfiles between storage pools and when it reclaims a storage device.

6. Index Management and Management of System Data

As the page size used by the recoverable data structures in the server may not coincide with the underlying block size of the device used to store such a structure, we implemented partial- write detection. The underlying substitution of bits occurs after the page has been isolated in the buffer pool to be passed on to the I/O subsystem. For the log, the low-level I/O recovery routines need to be privy to this data transformation so as to restore the appropriate values when returning a page to the higher levels of the system. The partial-write recovery routines were used 20% of the time we forcefully crashed a heavily loaded server.

To minimize catalogue space utilization we adopted two compression schemes, record compression and partitioned indexes. Index manager records are logically thought of as rows in tables uniquely identified by keys where each field has an internal data type. Records are compressed in a left-to-right type-dependent encoding separating the key part of the record and the data part of the record. Each encoding keeps the individual length of the encoded field. Decoding proceeds from left-to-right as individual encoded fields cannot be accessed directly. Savings in storage due to record encoding come with a run-time performance penalty as key comparisons require to decode the keys. In 1988 when this encoding strategy was established, the benefits of saving space were greater than they are today. Today we would adopt a scheme that guarantees constant access time to each field in the key and word-aligned compressed fields.

When defining an index one can specify that **p** of the **k** fields that make the key, with **p** < **k**, is the partition prefix. The so-called partitioned indices enforce that all records stored in a leaf node share the same prefix of **p** fields. The prefix is stored once in the node saving space at negligible processing cost. The structural modification logic of partitioned indices is of the same complexity as that of the standard B^+-Tree indices. In[3] we justify why it is not necessary for our indices to support duplicate keys and how we achieve compact indices in selected cases by encoding record ranges with one record.

In the server we optimized several database-related operations. The small number of data types required in the index schemas requires only a rudimentary type system. This simplifies the interface and processing of index-level operations. A second simplification made was record management. We imposed a left-to-right access to the fields in each of the two logical parts of records. This is in contrast with database management systems like Starburst[13] that require constant access time to any field. Our encoding avoids storing an offset per field, a substantial space savings for records with a handful of short fields.

7. Atomicity in the ADSM Server

To support transactions the system has its own log manager, lock manager and transaction manager. The lock manager supports all the modes found in database management systems including intention locks. The server requires that all components in the system that administer permanent storage register as participants of transactional activities and follow the presumed abort two-phase commit protocol[6]. When a transaction needs to commit, or abort, the transaction manager contacts each of the participant components to do the appropriate presumed abort protocol actions. This processing follows the same steps that an external network transaction would follow[9]. The underlying recovery algorithm is WAL ARIES[18].

The two recoverable data structures used for metadata are a bit vector and a B^+-Tree. The preferred mode of use of storage devices is "raw" mode. However, the system can be configured to use a file system when synchronous file I/O is supported.

7.1 Logging

The structural modifications of indices, B^+-Tree node splits and node merges, are preserved irrespective of the fate of the transaction that caused the structural change using the compensation log record[2,18] technique. A special log record is written that encircles all the log records pertaining the complete structural modification.

The recoverable bit vector uses value logging achieving a granularity of single block allocation and deallocation. To support isolation efficiently an extent-oriented memory data structure tracks allocations as seen by inflight transactions. Allocation and deallocation requests keep, per transaction, intentions lists of all blocks allocated and deallocated thus far. At the prepare stage, the intentions list is flushed in the log and at commit the disk bit vector is updated. During the analysis pass of recovery intentions lists are built for the inflight transactions. Allocations and deallocations may occur during recovery as indices may split and merge.

The server makes a checkpoint whenever 250 kilobytes of log space have been consumed. Using the same RS/6000 model 370 as before and running a 'heavy backup workload' we observed that the distribution of force times, in milliseconds, of a batch of log records had a mean of 32 (a minimum of 22, a maximum of 500 and a standard deviation of 44.71), that the distribution of log batch sizes between forces, in bytes, had a mean of 16270 (a minimum of 0, a maximum of 94643 and a standard deviation of 17609), that the distribution of log write batches, in pages, had a mean of 6 (a minimum of 1, a maximum of 14 and a standard deviation of 4.43) and that most times the system appended a log record in less than 0.02 milliseconds.

Non-intrusive fuzzy dumps of the metadata were implemented following the design of[17]. The server continues its normal operation during fuzzy dumps. Our implementation provides point-in-time and forward recovery, based on full and incremental fuzzy dumps. A consistent image of the system can be built beginning from any full fuzzy dump and restoring it to the most recent point in time using the incremental fuzzy dumps and the on-line log. Using the same RS/6000 model 370 as before, we measured two runs of the full fuzzy dump of a database of 25.88 megabytes, obtaining read rates of .796 and .785 megabytes/second and write rates of 1.572 and 1.556 megabytes/second respectively.

7.2 Locking

Repeatable reads is enforced by transactions only when needed. As the server initiates transactions for its internal activities, such as migration of bitfiles between storage pools and space reclamation of storage media instances, enforcing repeatable reads for all transactions is inappropriate. Common client-induced query operations, for example, would be unduly delayed by bitfile migration and space reclamation.

Lock management was another area of simplification. In the server, locks are requested by the callers of components. As the caller of a lower level component has complete knowledge of the intended operation, like the migration of all bitfiles in a given co-location cluster to a lower level of the hierarchy, or the expiration of all bitfiles with a given storage date in a management class, it requests locks at a high logical level such as on predicates or on the most significant (sub)parts of a key. This allows the low-level components, like the index manager, to not acquire locks on behalf of its callers and thus to not have to deal with the notion of lock escalation.

To support a high degree of concurrency and accessibility to the user data we adopted optimistic concurrency control policies. Exclusive locks are held for a minimum amount of time. When migrating sets of bitfiles within the storage hierarchy, for example, we retrieve the required metadata, we release locks on it, we optimistically copy the data, and only then reacquire the locks on the metadata to reflect the current changes. This maximizes the time that data is available for users at the cost of temporarily retaining more copies than required. We found that by locking logical entities within the server lock escalation was unnecessary.

To fully utilize the devices with large latency times we provide mutual exclusion with a reservation mechanism that can be used by more than one transaction. This capability allows the system to exploit the devices for other transactions that require it even if they have not committed, maximizing the use of these devices without performing unnecessary work.

7.3 Locking optimizations

Our workload is a challenge for concurrency. Maximizing the user access to their data drove us in minimizing the time transactions held exclusive locks on bitfiles and on metadata. We adopted the optimistic policy of not holding metadata locks while a data transfer or a data copy was in progress. The traditional two-phase locking strategy would be an unacceptable inhibitor of access to the user data. We use additional space to always 'copy ahead' the data. For example, when migrating data between storage pools, we lock the metadata in shared mode, retrieve it, unlock it, do all the necessary data copying between storage pools, lock the metadata in exclusive mode again to verify that the world has not changed under us, and only then proceed to change all the metadata to reflect the data movement that has already occurred. As our bitfiles are immutable the worst scenario is that a bitfile that is being migrated down the hierarchy is concurrently deleted by an expiration process. As deletions are rare, this event does not concern us.

Because devices like tape jukeboxes have enormous latency times, holding locks on them on a per-transaction basis is not appropriate. As the time to acquiring use of such a device is so long once it is put to work on behalf of one transaction it should be exploited on behalf of other transactions that require it, irrespective of the fate of each individual transaction. Optimistically, if none aborts then no work is lost. To serialize activities on long latency devices we introduced the notion of leases. A lease is a mutual exclusion handle to access a device. Different transactions access the device using a lease. This technique is exploited when migrating a cluster of bitfiles to a given storage media instance like a tape or an optical disk in a jukebox. Once the processing is done on behalf of the transaction that triggered the use of the media then the system inspects its metadata and schedules the transfer of data of all clusters that need to migrate bitfiles into that media. Leases expire when no outstanding transactions require the resource.

Deadlock management was reimplemented when we found a computing platform where it was impossible to roll-back once a resource had been allocated. We adopted a deadlock avoidance approach. The server tags requests with the externally mountable resources they need. For example, the data import activity requires at most one tape mount point while the tape reclamation activity requires two mount points. Resource allocators use this information to allocate resources as aggressively as possible.

8. Related Work

Database technology has been applied before to distributed systems[9,11], to file systems[1,4,7,8], to operating systems[9], to message queueing systems and to network I/O subsystems[14]. To our knowledge we are the first ones to apply transactions to a mass storage system that administers a storage hierarchy. The fundamental difference with all other systems is the need to support high degrees of concurrency for a variety of transactions in the presence of devices with enormous latency times.

A second difference is that our server can replicate its metadata with up to three copies. No other backup system or file system we know of has this function. This and the fuzzy dumps are the important differences with the recovery work done at IBM Almaden in the QuickSilver distributed system[9], in which the distributed file system and all other system services were transactional.

The pioneering work at Xerox Parc[12] led the formalization multi-node atomic actions. We differ with the file system work done at Xerox Parc by Brown[1] and Hagmann[7] in that our logged actions can be undone. The redo-only logging techniques used in the above two systems pose memory constraints inappropriate for ADSM. We differ with the Journaled File System (JFS) present in AIX[4] in that our recoverable data structures have granularity of updates down to one bit of information. Our demands for concurrency do not allow us to even use the rather small 128 byte lines of recoverable units exploited in the hardware-assisted implementation of JFS. Network file systems like Swift/RAID[14] and Zebra[8], that stripe data over a set of file servers used as storage servers, use atomic network I/O operations in the context of striped file systems but none of them supports a storage hierarchy.

The server ability to do co-location of bitfiles over time allows it to never require full dumps to achieve data clustering[3]. This differentiates ADSM as a backup and archive service from the offerings of Legent[20], Legato, Cheyenne, Harbor[21], Epoch[22] and Palidrome. The server storage management at the bitfile level differentiates it from backup services like Harbor that use the hierarchical storage management services present in mainframes. These backup services cluster sets of user files into backup files and migrate them. We differ with the traditional work on storage hierarchies[24] in that we provide the atomicity property of transactions.

9. Conclusions

We built and deployed ADSM, a mass storage system providing the abstraction of unbounded storage. The system may administer several storage hierarchies. In it all operations are done on behalf of transactions. Using database technology simplified handling failures and providing continuous, unattended, lights-out operation. Transaction boundaries are also exploited to stream data over communication connections. To enhance the reliability of the server data the system can replicate it keeping up to three copies constantly synchronized and supports non-intrusive fuzzy dumps.

The workload in the network-based backup and archive system is unconventional. Transactions seldom abort yet may require half an hour to complete. Some modify minimum amounts of metadata yet store large amounts of data. Others modify large amounts of metadata and access no data. Many are read-only. The common activity in the system is storing collections of related files, seldom retrieving them and sporadically deleting them. A single retrieval may involve thousands of files.

To support a high degree of concurrency we adopted optimistic concurrency control policies. When migrating sets of bitfiles within the storage hierarchy, for example, we retrieve the required metadata, we release locks on it, we optimistically copy the data, and only then reacquire the locks on the metadata to reflect the current changes.

To fully utilize the devices with large latency times we provide mutual exclusion with a reservation mechanism that can be used by more than one transaction. This allows the system to exploit the devices for other transactions that require it even if they have not committed.

ADSM put IBM on the map of client-server hierarchical storage management. The Gartner Group[16], placed ADSM (and IBM) at the top in leaders and visionaries. PC Week named ADSM for OS/2 product of the week in July of 1994.

Acknowledgments. Many thanks to our partners in building and deploying ADSM. In particular, thanks to Paul Bradshaw and Nick Tabellion, as well as to those who currently evolve ADSM and service it. Thanks to Mike Penner for developing the low-level I/O subsystem, to Chip Coy for his server work, to Michael Kaczmarski and his team for their contributions to the server, to C. Mohan and Inderpal Narang for teaching us their inventions, to Norm Pass for being the mentor and manager at Almaden and to Jim Carlson for being the patriarch and project director at SSD software in San Jose.

Special thanks to Phil Bernstein for his many positive suggestions on how to improve the presentation of this work.

10. References

1. Mark R. Brown, Karen N. Kolling and Edward A. Taft, "The Alpine File System." *ACM Transactions on Computers*, Vol. 3, No. 4, November 1985, pp. 261-293.

2. Luis-Felipe Cabrera, John A. McPherson, Peter M. Schwarz and James C. Wyllie, "Implementing Atomicity in Two Systems: Techniques, Tradeoffs, and Experience." *IEEE Transactions on Software Engineering*, Vol. 19, No. 10, October 1993, pp. 950-961.

3. Luis-Felipe Cabrera. Robert Rees, Stefan Steiner, Wayne Hineman and Michael Penner, "ADSM: A Multi-Platform, Scalable, Backup and Archive Mass Storage System." *Proceedings of IEEE COMPCON 95, San Francisco, March 1995.* Also *IBM Research Report RJ 9936*, February 1, 1995.

4. Albert Chang and Mark F. Mergen, "801 Storage Architecture and Programming." *ACM Transactions on Computers*, Vol. 6, No. 1, February 1988, pp. 28-50.

5. David M. Choy and Ashok Saxena, "A Model for Backup, Archive and Restore - A Proposal to X/Open." *IBM Almaden Research Report, RJ 9620*, November 24, 1993.

6. Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques.* Los Altos, CA: Morgan Kaufmann, 1992.

7. Robert Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit." Proceedings of the 11th ACM SOSP, *Operating Systems Review, Vol. 25, No. 5, December 1987, pp. 155-162.*

8. John H. Hartman and John K. Ousterhout, "The Zebra Striped Network File System." Proceedings of the 14th ACM SOSP, *Operating Systems Review*, Vol. 27, No. 5, December 1993, pp. 29-43.

9. Roger Haskin, Yoni Malachi, Wayne Sawdon and Gregory Chan, "Recovery Management in QuickSilver." *ACM Transactions on Computers*, Vol. 6, No. 1, February 1988, pp. 82-108.

10. IBM ADSTAR Distributed Storage Manager, publication G520-6928-02.

11. William F. Katz, *PC Week/Netweek, July 25, 1994, pp. N11-N15.*

12. Butler W. Lampson and Howard E. Sturgis, "Crash Recovery in a Distributed Data Storage System." Technical Report, XEROX Palo Alto Research Center, Computer Science Laboratory, 3333 Coyote Hill Road, Palo Alto, CA 94304, 1976.

13. Tobin J. Lehman, Eugene J. Shekita and Luis-Felipe Cabrera, "An Evaluation of Starburst's Memory-Resident Storage Component." *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992, pp. 555-566.

14. Darrell D. E. Long, Bruce R. Montague and Luis-Felipe Cabrera, "Swift/RAID: A Distributed RAID System." *Computing Systems*, Vol. 7, No.3, Summer, 1994, pp. 333-359.

15. IEEE Mass Storage System Reference Model: Version 4 (May, 1990). Reference Model for Open Storage System Interconnection (draft, April 20, 1993).

16. G. McDermed, Gartner Group 1993 Data Center Conference. Large Computer Strategies Products, P-806-1479. *LCS Research Note*, November 8, 1993, pp. 1-15.

17. C. Mohan and Inderpal Narang, "An Efficient and Flexible Method for Archiving a Data Base" *Proceedings of 1993 SIGMOD*, Washington, May 1993, corrected version of August 4, 1993.

18. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh and P. M. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Roll-backs Using Write-ahead Logging." *ACM Transactions on Database Systems*, Vol. 2, No. 1, March 1977, pp. 91-104.

19. Alan J. Smith, "Disk Cache - Miss Ratio Analysis and Design Considerations." *ACM Transactions on Computer Systems*, Vol 3, No, 3, August 1985, pp. 161-203.

20. Mark Friedman, "Storage Management." *Demand Technology Inc.*, November 1993.

21. Harbor, "*General Information Manual.*" New Era Systems Inc.

22. Robert K. Israel, "The Growing Challenges in Client/Server Backup and Recovery." *Epoch Systems Inc.* white paper, August 1993.

23. C. Mohan and Dick Dievendorff, "Recent Work on Distributed Commit Protocols and Recoverable Messaging and Queueing." *IEEE Data Engineering*, Vol. 17, No. 1, March 1994.

24. P. A. Franaszek and B. T. Bennet, "Adaptive Variation of the Transfer Unit in a Storage Hierarchy." *IBM Journal of Research and Development*, Vol. 22, No. 4, 1978.

Trademarks: ADSM, MVS, VM, VSE, AIX, OS/2, AS/400 are trademakrs of IBM Corporation. MAC is a trademark of Apple Corporation. PC-DOS and Windows are a trademark of Microsoft Corporation. HP/UX is a trademark of Hewlett Packard Corporation. UNIX and Novell IPX are trademarks of Novell Corporation. SUN and SUN/Solaris are trademarks of SUN Microsystems corporation.