The ADAMS Storage Management System

Stanley A. Janet, Jr.

IPC-TR-89-008
August 10, 1989

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903

This research was supported in part by JPL under contract #957721 and DOE Grant #DE-FG05-88ER25063.

The ADAMS Storage Management System

Stanley A. Janet, Jr.

Institute for Parallel Computation

University of Virginia

Charlottesville, Virginia 22903

ABSTRACT

This document describes the design and implementation of the ADAMS Storage Management System. The system consists of a server that manages connections with multiple clients and passes storage requests to I/O-dedicated processes. The communication channels between clients and the server are implemented through sockets, the basic interprocess communication facility in BSD UNIX†.

The interface provided to client programs consists of functions that connect and disconnect the client and server, and operations on an abstract data type called a bag, which is a collection of variable or fixed-length objects. The function interface completely hides the message passing required to support the operations and the connection. Interfaces exist for C and C++ programs.

---

† UNIX is a Trademark of AT&T Bell Laboratories.

1. INTRODUCTION

The goal of the ADAMS (Advanced DAta Management System) project [1] is to create a database system that allows processes written in traditional programming languages to simultaneously access persistent data. This report describes a storage manager that can be utilized by ADAMS to support such access, while relieving the applications programmer of the low-level details. It is assumed that the persistent data will be represented on multiple storage devices.

An important principle in the ADAMS design is that data should be able to migrate from device to device based on disk access patterns. Two of the target machines for ADAMS are powerful multiprocessors at the University of Virginia's Institute for Parallel Computation — a 32-node Intel hypercube and a 128-node Ncube hypercube. Both have powerful parallel disk access capabilities, but to take full advantage of their potential, bottlenecks must be avoided. Therefore a database system running on such systems would benefit from a mechanism for moving data based on access patterns in order to balance I/O loads.

Programmers have historically accessed persistent data through filenames and relative locations such as a byte offsets or record numbers. An unfortunate consequence of this approach is that migration of data either greatly sacrifices efficiency or is altogether unworkable; processes that need to access the data cannot keep track of its location. In addition, programmers must handle the low-level file I/O necessary to operate on the object, concurrent access to data cannot be supported without great care and operating system support. These weaknesses can largely be overcome by the client-server model. Clients refer to persistent data through some form of storage-independent handles, while a server is responsible for mapping handles to actual disk locations. Because an item's handle is invariant over its lifetime, the server can move data arbitrarily so long as its mapping rules are revised appropriately.

The ADAMS Storage Management System is based on this model. Clients are provided with an interface that allows only high-level operations, while the server performs all low-level I/O in response to client requests. This approach provides client processes with a small set of

functions that support an abstract data type called a *bag*, a persistent collection of variable-length or fixed-size objects. Individual bags are not bound to any storage device. While this implementation of the Storage Manager does not concern itself with migration, we will that show the approach used is easily extensible to provide this capability.

Client requests are received by a server, which utilizes a group of I/O-dedicated processes to perform the actual I/O. The server operates as a daemon that can multiplex many clients.

Under this system, a client uniquely identifies a data item by a (bag number, item number) pair — the latter is simply a secondary tag that uniquely identifies this particular element in the bag, but not its location on a storage device. A client can store an item by creating a bag and then inserting the item into it. Those operations return a bag number and an item number respectively; the client can later access the item by these values. Other functions allow retrieval, modification and deletion of items, and deletion of bags.

The Storage Management System is written in C and runs on a VAX† 8600 under 4.3BSD UNIX.

---

† VAX is a Trademark of Digital Equipment Corporation

2. CLIENT INTERFACE

Clients view bag operations as function calls. The function calls mask the underlying system calls required to implement the client-server interprocess communication (IPC). We use long integer bag and item identifiers, which we type as *BAGNO* and *ITEMNO*.

There are six functions provided to clients to operate on bags and items. The declarations and semantics of the functions are listed below. All functions return negative values on error. Error conditions are listed in Appendix A.

`BAGNO create_bag(long length)`

Creates a new bag and returns its bag number. If *length* is positive, the bag may contain only items of this specified number of bytes. A non-positive value specifies that the bag may contain variable-length items.

`ITEMNO insert_item(BAGNO b, char *s, long length)`

Inserts the sequence of *length* bytes beginning at address *s* into bag *b*, and returns the item number assigned to the stored bytes. If the bag was created for fixed-length items, the length must match the value originally passed to *create_bag()*.

`long retrieve_item(BAGNO b, ITEMNO i, char *s, long length)`

Retrieves the bytes stored as item *i* in bag *b*. Copies at most *length* bytes to the buffer beginning at address *s*. Returns the actual size of the item in bytes.

`long modify_item(BAGNO b, ITEMNO i, char *s, long length)`

Replaces item *i* in bag *b* with the sequence of *length* bytes beginning at address *s*. If the bag was created for fixed-length items, *length* must match the value originally passed to *create_bag()*.

`long delete_item(BAGNO b, ITEMNO i)`

Deletes item *i* from bag *b*. The item number can later be reassigned by the server when another item is inserted into bag *b* via *insert_item()*.

`long delete_bag(BAGNO b)`

Deletes bag *b*. There is no requirement that the bag be empty. The bag number can later be reassigned by the server when another bag created via *create_bag()*.

Two functions are provided to allow the client to arbitrarily open and close a connection with the server. This ability provides significant flexibility to programmers concerned with robustness. For example, if the server crashes and is then rebooted, clients that were not con-

nected over this period will be unaffected.

As with the bag functions, these connection functions return negative values if they fail.

`int open_connection()`
: Opens the communication channel between the client and the server.

`int close_connection()`
: Closes the communication channel between the client and the server.

Another function allows the client to determine if the connection is open or closed. While the state can also be determined by the return values of the *open_connection()* and *close_connection()* functions, this function provides a convenient mechanism for code with multiple entry points, such as signal handlers, to test connection status:

`int connected()`
: Returns a nonzero if the communication channel between the client and server is open, and zero otherwise.

When any of the functions fail, the external variable *errno* is set to the error number. A mechanism exists so the user can access a string that describes the error:

`char *errstr()`
: Returns a pointer to a string describing a client error.

Most UNIX system calls can alter the value of *errno*, so this function should be invoked immediately after a Storage Manager function fails, as in the following code fragment:

```
if (open_connection() < 0)
    {
    fprintf(stderr,"Open_connection failed: %s\n", errstr());
    exit(1);
    }
```

A process may have no more than one connection to the server open at a given time. A call to open the connection when it is already open, or to close the connection when it isn't open will fail. A process with the connection open can exit without calling *close_connection()* — the kernel will close the client's side of the connection when cleaning up the process' open file table.

Since file descriptors are shared over a *fork()* system call, a client program that forks while connected should close the connection in either the parent or child. Since file descriptors are inherited over *exec()* system calls, a client should always close the connection before an *exec()*.

Client processes have full control to decide when bags should be created and what is to be inserted into them. The intention is that each bag should contain related items likely to be accessed sequentially, and different bags should be used for collections of data that are likely to be accessed in parallel. Later versions of the ADAMS Storage Manager will be optimized for those circumstances. Clearly, the determination of relatedness, while varying for different client applications, is better decided at a higher level.

Client processes also have full control over when the communication channel to the server is opened. A client process that infrequently needs access to data managed by the server should open the connection only when necessary. The maximum number of connections that the server can have open is finite, though large enough to handle many clients simultaneously (see Section 6.2).

The C++ user simply links his object file(s) with the object file containing the connection and bag-related functions. A file containing the type definitions for BAGNO and ITEMNO and the declarations of the functions and their arguments must be included into any client source files that call the functions to satisfy the strict type-checking requirements of *cfront*, the C++ syntax and type-checker. The C user can link with the same object file, but a C-style *include* file must be used instead of the C++-style file.

3. SYSTEM DESIGN

The ADAMS Storage Management System consists of a server process, several processes dedicated to carrying out I/O requests from the server, and a varying number of clients. Figure 1 shows the system in a typical state, with four connected clients and eight I/O processes.

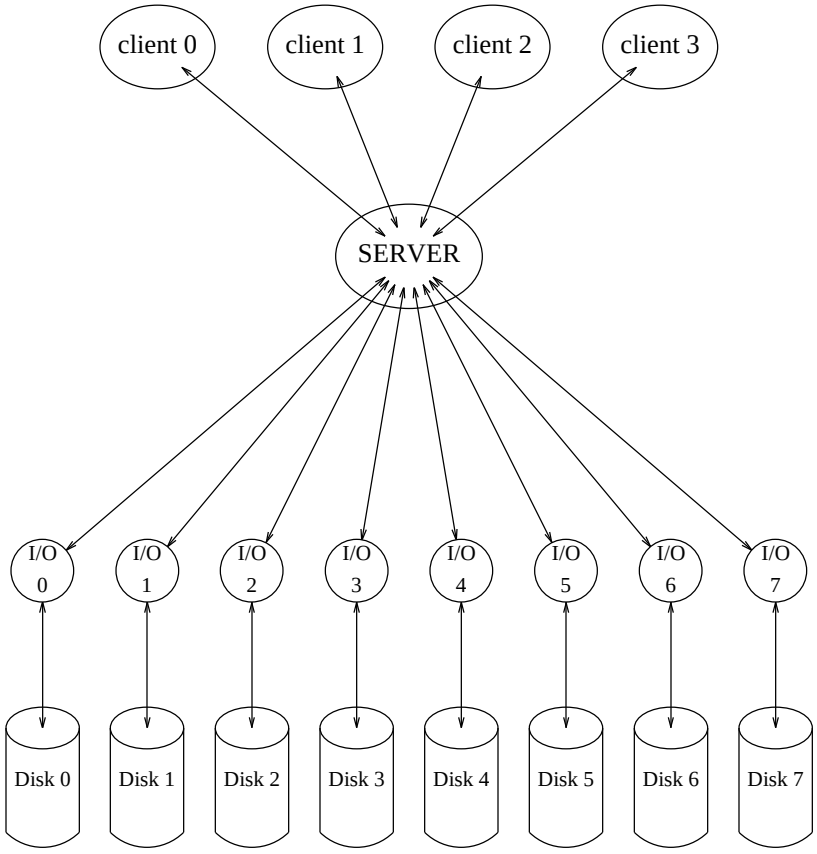

Figure 1. System Design

The number of I/O processes is set when the server is booted and reads its configuration file. The processes are identical but have different home directories. The intention is to let the configuration file specify directories that reside on different devices, thus binding I/O processes to different devices. When ported to a multiprocessor, I/O processes will reside on separate nodes and parallel storage potential can then be realized.

The connections between clients and the server are implemented as connected pairs of communication endpoints called sockets [2,3,4,5]. Interested readers are referred to the *socket(2)* manual page for a glimpse at the power of a socket-based networking. Sockets are the basic interprocess communication (IPC) mechanism in BSD UNIX — even the *pipe()* system call is implemented internally as a pair of connected sockets. Sockets provide a reliable bidirectional communication channel between arbitrary processes. The lack of a portable mechanism for such communication had long been a major shortcoming of UNIX. Pipes can only exist between processes that are descendants of a common ancestor, and must be inherited over every process creation along the way. Sockets, which appeared in the 4.2BSD release of UNIX, allow arbitrary processes to establish and disestablish a connection at arbitrary points in their execution. The bottom line is that, after a minimal amount of groundwork, a process is returned a file descriptor through which it can use well-known UNIX system calls (such as *read()* and *write()*) to communicate with any process that is looking to do the same.

The function calls that compose the client interface are implemented much like a remote procedure call — the calls that require service from the Storage Manager set a opcode field in a message structure, pack other fields in the message based on the function arguments, send the message to the server, and wait for a reply. For example, the client function *delete_item()* is simply:

```
long delete_item(bag,item)
BAGNO bag;
ITEMNO item;
{
request.type = OP_DELETE_ITEM;
request.dat = bag;
request.dat2 = item;
if (servercall() < 0)
    return -1;
return reply.status;
}
```

While the Storage Manager was developed on a machine with a single processor, cleanly separating the tasks of the server and I/O processes will allow easier porting to a hypercube

environment.

Each I/O process communicates with the server through a pair of unidirectional pipes. Data transfer through pipes is very efficient, as the UNIX kernel buffers several kilobytes before blocking the writing process until it is read by the process at the other end of the pipe. The size of the buffer is four kilobytes on the VAX that the Storage Manager was initially implemented on. The current implementation limits item lengths to four thousand bytes, with the remaining space used for message header fields. All message transfers are therefore atomic, greatly simplifying the role of the server.

4. SERVER OPERATION

4.1. Server boot sequence

The ADAMS Storage Management System is booted by executing the server:

```
(atlas) $ server
ADAMS Storage Manager
Booted Aug  9 15:01:34
Created lock file
Opened log file
Run #847
Working directory: /at0/saj9s/server
Socket address: _SOCKET_
IO process name: child
Server pid (fg) = 10836
Random number generator seeded with 4003
Set process group
Read configuration file: 4 I/O processes
        Directory 0: /va0/adams/bags
        Directory 1: /va1/adams/bags
        Directory 2: /va2/adams/bags
        Directory 3: /va3/adams/bags
Created socket
Setup completed -- going into background
Server pid (bg) = 10837

(atlas) $ ps glxw
  PID  PPID WCHAN  STAT TT TIME COMMAND
10837     1 selwai S    ?  0:00 server
12660 10837 mbutl  S    ?  0:00 child 0 7 8 /va0/adams/bags
12661 10837 mbutl  S    ?  0:00 child 1 8 10 /va1/adams/bags
12662 10837 mbutl  S    ?  0:00 child 2 10 12 /va2/adams/bags
12663 10837 mbutl  S    ?  0:00 child 3 12 14 /va3/adams/bags
12673 27756        R    Sa 0:00 ps glxw
27756     1 u      S    Sa 0:07 -ksh (ksh)

(atlas) $ ls -l /at0/saj9s/server/_SOCKET_
srwxrwxrwx 1 saj9s  0 Aug 9 15:01 /at0/saj9s/server/_SOCKET_
```

Appendix B lists valid command line arguments that can be used to override system defaults. If the command line specifies that the the server is to operate in another directory, it immediately changes its current working directory to the one specified. The first system task is to create a lock file to prevent multiple instances of the server from running simultaneously in the same directory.

If the lock file cannot be created, the process assumes another ADAMS Storage Manager server process is already running and exits. Next it reads the system configuration file *server.cfg* which specifies the number of I/O processes for the server to create and the directories that each I/O process will operate in.

The next important stage in the boot sequence is the creation of the I/O processes and the communication channels (pipes) to and from them. The server stores the I/O process ids — if the server receives a fatal signal or some other fatal event occurs (i.e. one of the I/O processes dies), the server terminates all I/O processes before it exits. Next it receives a sequence of numbers from each I/O process representing the bags found in those processes' directories — the server inserts (bag number, I/O process number) tuples into a hash table. Later, the server maps client requests to the proper I/O process by lookups in this table.

The server then goes into the background automatically and disassociates from its control terminal. It initializes its connection table to an empty state, creates the socket through which it will accept client connections, and associates the socket with a UNIX domain address†. This process returns a file descriptor to the server, hereafter referred to as the connection descriptor.

4.2. Normal server operation

After completing the boot sequence, the server enters a loop reading, processing, and replying to client messages. This continues until it receives a shutdown message. The following code fragment illustrates the main loop of the server:

---

† A UNIX domain address is a filesystem object like a directory, device or ordinary file. A process opens a channel with another process via the *connect()* system call by specifying this address.

```
stop = FALSE;
while (! stop)
    {
    while (check_for_request() < 0)
        ;
    if (read_request(&incoming_msg) < 0)
        continue;
    process_request(&incoming_msg,&outgoing_msg,&stop);
    reply_to_request(&outgoing_msg);
    }
shutdown_io_processes();
```

4.2.1. *check_for_request()*

The function *check_for_request()* determines (a) if any messages have arrived from connected clients or (b) if any other processes are attempting to connect to the server. The server maintains a table of file descriptors that refer to connected clients†, and *check_for_request()* inserts them into a set. A pointer to this set is passed as an argument to the *select()* system call, which blocks the server until one or more of the descriptors is ready for reading. On return from *select()*, only those file descriptors are in the set; *check_for_request()* examines it and uses a round-robin algorithm to schedule a client for service.

If the server's connection table is not full, the connection descriptor is also inserted into the set passed to *select()*. If this file descriptor is in the output set and therefore "readable", the system call *accept()* can be used to add a new client.

If both conditions exist — messages have arrived and one or more processes are attempting to connect to the server — the server acts on the latter. The justification for adding new clients whenever possible is that the maximum size of the queue of pending connection requests is small (5 on our VAX). We also assume that connection requests are rather infrequent in comparison to message arrivals, so no client will be starved of service.

The *check_for_request()* call can return without finding any file descriptors ready for reading if the *select()* call is interrupted by a signal, or if a new connection was added. The server

† Currently the server allows up to 16 simultaneous connections.

calls it until it succeeds in finding a connection with a message waiting.

4.2.2. *read_request()*

Once *check_for_request()* succeeds, the server calls *read_request()* which reads a message from one of the file descriptors found ready by *select()*. If more than one client is ready, a round-robin algorithm is used to pick one from those ready. This call can fail if the *read()* returns end-of-file.

Normally a client disconnects by calling *close_connection()*. In *process_requests()*, when a message with a *disconnect* opcode arrives, the server simply closes the file descriptor for the client that sent the message. If a client process terminates without calling *close_connection()* (through a fatal signal, calling *exit()* or reaching the end of *main()*), the file descriptor for that client will eventually *select* for reading. However the *read()* in *read_request()* will return end-of-file, whereupon the server closes the descriptor.

4.2.3. *process_request()*

Communication with I/O processes occurs in *process_requests()* based on the opcode found in the message read from a client. If the opcode specifies that a bag is to be created, the server randomly picks an I/O process and thus a directory for the bag. It also picks a bag number for the bag by looking for the lowest number that isn't currently assigned to a bag. The server then sends a message to that I/O process instructing it to create the bag with that number. If the creation succeeds, the server stores the bag number in a table along with the I/O process that will handle operations on it. The other operations available to the client require the server to do a lookup in this table given a bag number and to forward the client's request to the appropriate I/O process. Bag deletion requires the bag entry to be deleted from the table following it's physical removal by the I/O process. The table is implemented through hashing with linked list chaining to resolve collisions.

If the opcode of the message specifies system shutdown, the *stop* flag is set. However all messages are replied to, so the loop is not terminated immediately.

There are several other opcodes in addition to those for bag operations, disconnecting, and shutdown. Opcode *noop* always succeeds and returns immediately from the server; no I/O processes are affected. This opcode is used for testing the rate at which the client and server can exchange messages. Opcode *register* stores the client's login id and name in the connection table. This information is determined by the client uid after a lookup in the the system password file. The uid is passed to the server in a registration message before the client's *open_connection()* call returns. This is done so that the server log file can record who connected to the server and when. The mechanism could easily be extended to exclude unauthorized users, and to prevent other users from executing privileged operations. There is no direct way to determine what user is at the other endpoint of a socket-based communication channel (although the *getpeername()* system call can be used to identify the machine at the other end of a remote connection). The names appear in the server log file, allowing users to easily find the points at which they connected and disconnect.

4.2.4. *reply_to_request()*

The *reply_to_request()* call forwards a message to the whose request was just processed. The message contains the return value for the client's function call, an error code if the call failed, and any other data to fulfill the client's request.

The server logs all connection activity and requests with sources, appropriate parameters, and a timestamp. The logging can be turned off by a command line argument.

On shutdown, the server send a message to each of the I/O processes instructing them to exit. It then sleeps for two seconds and sends a termination signal to them in case they were somehow in a state that they couldn't receive the message. Finally, the server deletes the socket and lock file from the filesystem and exits.

An unresolved issue is determining the best design through which the Storage Management System can realize true parallelism — the server's main loop supports only sequential processing

of client requests. This implementation certainly could have forked the server after reading a message; the child would process and reply to the request, and the parent would read another one. Ironically, forking is an expensive task that would likely slow the server down. In any case, such an approach is clearly not suited for a parallel environment. There are at least two promising strategies. Perhaps the server should read all requests that arrive simultaneously, then loop, routing as many as possible to the appropriate I/O processes and replying, until all requests have been processed. Another approach would involve splitting the server into multiple cooperating processes. Both strategies would have complicated the server more than necessary on this current implementation. Furthermore, we have no way of determining which strategy is superior until the system is actually ported to a parallel environment. Certainly this design issue should be of primary concern at that point.

5. I/O PROCESSES

An I/O process begins its execution by examining the command line arguments provided by the server when it created the process. These arguments are I/O process number, the file descriptor numbers that represent its pipes to and from the server, and the directory it will operate in. The I/O process then changes its current working directory to the one one specified, and sends the server a list of the bag numbers already in existence there.

The process then enters a loop reading messages from the server and replying to them, until it receives a message with a *shutdown* opcode. All messages from the server are acknowledged, except the shutdown message which cannot fail.

In the current implementation, a bag is represented by four files in the working directory of one I/O process. The base name of every file is the bag number it corresponds to, left-padded with zeros to a constant width.

1. Bag header file:
   The bag header file contains a bag's invariant parameters, i.e. whether it contains fixed-length objects, and if so, the length. The suffix for files of this type is *hdr*.

2. Bag data file:
   The bag data file contains the actual byte streams that have been stored in the bag. The suffix for bag data files is *dat*.

3. Bag item table:
   The bag item table is in a file with a *tbl* suffix. The table is a sequence of 3-tuples which specify if the table entry is defined, and if so, the offset and length of the item in the data file. An item number is a record number in this file.

4. Bag hole table:
   The bag hole table is in a file with a *hol* suffix. A hole is a sequence of bytes in the bag data file that no longer contains an item. This table is a sequence of 3-tuples which specify if the table entry is defined, and if so, the offset and length of a hole.

Figure 2 shows illustrates the relationships between the dynamic files. Item table entries 2, 3 and 5 are keeping track of items with lengths 3, 4 and 2 respectively. Hole table entries 1, 3 and 6 are keeping track of holes with lengths 4, 8 and 5 respectively. All other item and hole table entries are unused.

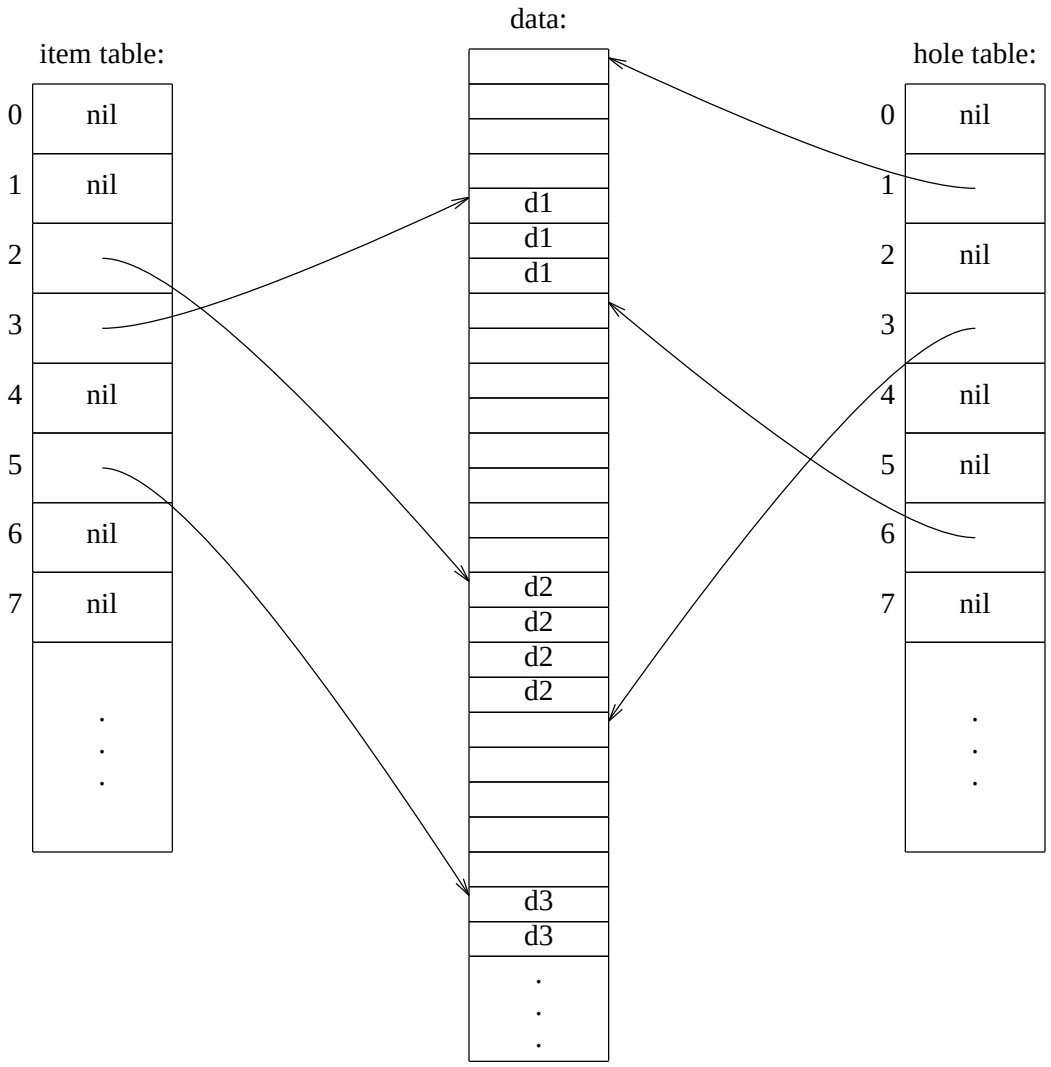

Figure 2. Bag Implementation.

When adding new items, appropriate holes are found (if any exist) for reallocation to items by scanning the hole table using a first fit algorithm. Items are added to the item table by taking the first 3-tuple that is not defined for another item. A hole is added to the hole table either by taking a 3-tuple that is not defined for another hole or by coalescing the hole with an adjacent one, whichever is encountered first. Buffered I/O is used to scan the tables to minimize disk requests. Summaries of the bag operations follow.

1. Bag creation:
   The four files are created. All files, except the bag header file, are initially empty.

2. Item insertion:
   The I/O process searches through the hole file looking for a hole large enough to contain the new item. If such a hole is found, the item is written at the hole's space in the data file. Otherwise, the item is stored at the end of the data file. The item's position and length are then added to the item table; if no unused item table entry can be found, a new one is created at the end of the item file. The item number returned to the client is the item table entry number.

3. Item retrieval:
   The I/O process seeks to the item table entry number specified by the client to find the item's offset and length in the data file. It seeks to that offset, and reads the item into a reply message buffer. If the length specified by the user is less than the length of the item, fewer bytes are read.

4. Item modification:
   The I/O process seeks to the item table entry number specified by the client to find the item's offset and length in the data file. If the length of the item is greater than or equal to the new length specified by the client, the space is overwritten with any extra space becoming a hole. If the length of the item is less than the new length, the old space becomes a hole and new space is located as in *insert_item()*. The item's table entry is updated to the proper offset and length in any of the cases.

5. Item deletion:
   The process seeks to the item table entry number specified by the client, marks the entry undefined, and stores the space that was allocated to the item in the hole table.

6. Bag deletion:
   The I/O process removes the four files that represent the bag.

Currently fixed-length bags are treated as a special case of variable-length bags, with operations disallowed if the incorrect length is specified. Length fields are ignored when searching for holes, but both fixed and variable-length operations invoke the same function.

Like UNIX directories, the files never shrink in size. Although 4.3BSD UNIX provides a file truncation system call, the capability is not portable. Keeping track of unused areas within the data file, therefore, is the better approach.

In the ADAMS Storage Manager, a hole refers to a sequence of bytes in the data file that once held an item(s). In UNIX terminology, the same term is sometimes used to refer to a part of a file where nothing has been written. UNIX holes are created by writing data after seeking past end-of-file, and no disk block is allocated to parts of such a hole that would completely span the

disk block. In this system, however, holes can be part of one or more disk blocks.

All messages from the server are logged with opcode, appropriate parameters, and a timestamp. All replies to the server are logged with return value, error code if the status is negative, and a timestamp.

6. DISCUSSION

6.1. Extensibility

A prime concern in the design of the ADAMS Storage Manager has been ease of extensibility to meet evolving needs of the project. The system can easily be modified to provide new services to clients when the operations are deemed useful. For example, one could readily add a *bag copy* operation to the system by following these simple steps:

1. Global changes:
   Add a new opcode for the operation.

2. Changes to the client function library:
   Add a new function to the client object file that takes a bag number as an argument and returns the bag number it is copied to. The function needs to set the opcode and bag number fields of the client-to-server message struct, ship the message, and wait for a reply.

3. Changes to the server:
   Modify the *switch* statement in the server that handles each opcode by adding a *case* for the new operation. It should perform a lookup on the bag routing table to determine the I/O process number handling the specified bag, choose a new bag number and select an I/O process to handle operations on it. Then the server should send a request to the first I/O process passing the bag numbers and a destination directory, and wait for a reply. If the operation succeeds, insert an entry for the new bag in the lookup table. Finally send a reply to the client that initiated the operation.

4. Changes to the I/O processes:
   Modify the *switch* statement in the I/O process that handles each opcode by adding a *case* for the new operation. It should copy the four files associated with the first bag to the location specified in the incoming message, giving them appropriate names. Finally reply to the server.

A *bag move* operation could be implemented similarly. The data migration mentioned in Section 1 could be supported by such an operation if the movement is from one device to another. The bag files on the destination device would retain the names of the files on the source device, and the latter files would be deleted. The server would simply modify the I/O process field in the routing table entry for the bag, thus binding the bag to the I/O process on the destination device.

Due to the centralized nature of the server, the system is easily extensible to a resource manager. For example, the ADAMS run-time system will need a mechanism for generating unique tags called *unique ids* for many data objects. This operation could be implemented over the bag interface by storing the last unique value generated under a constant bag and item number. However, even ignoring the lack of safety in allowing user processes to read, increment, and then modify this value through the server (presumably locks would prevent a race condition), the ensuing bottleneck would be a serious impediment to high performance. A better mechanism would be to let the server manage the value in memory. Adding another client function call and an operation in the server that generates and returns a unique id would be trivial. This implementation would require no disk I/O and no communication with the I/O processes.

6.2. File descriptor limitations

The server uses file descriptors to communicate with the I/O processes and with clients, as well as for monitoring the socket for new connections, for holding onto the lock file, and for logging. File descriptors are a finite resource for a process, so the number of I/O processes and clients is limited. The following table shows the maximum number of file descriptors available to processes on various machines at the University of Virginia:

atlas (VAX 8600, 4.3BSD)	64
uvacs (Sun 4,SunOS)	64
ice (Intel iPSC/2, SYSV)	100
babbage (AT&T 3B15, SYSV)	20
n10 (Ncube/10, Axis)	1000+

These limits are high enough to allow many clients to be connected to the server simultaneously. The maximum number of clients that can be supported will grow larger when the system is ported to a hypercube, where the communication channels between the server and I/O processes will not cost the server two file descriptors each.

6.3. Performance

The server can be used by various modules in the ADAMS system. The most important performance statistics will be the ones seen by the ADAMS run-time system in a parallel

environment. However clients can typically expect real-time responses like those shown in the following table:

Connect/disconnect	24 msec.
Bag creation	380 msec.
Bag deletion	270 msec.
Item insertion	39 msec.
Item deletion	37 msec.
Item retrieval	27 msec.
Item modification	25 msec.
Server noop	8 msec.
I/O process noop	14 msec.

The times were calculated by timing 1000 operations of the same type with the server utilizing four I/O processes. All item lengths were 16 bytes.

The relatively high bag creation and deletion times are a consequence of the high UNIX overhead in creating and deleting files — a process that simply creates 1000 files in its working directory took 71 seconds on our VAX. The average bag deletion time was almost constant from twenty to 1000 bags. The average bag creation time was 260 msec. for twenty bags and rose steadily to 380 msec. for 1000 bags.

We expect bag creation and deletion to be rather infrequent in comparison to the other bag operations, and the latter will be the easiest to optimize. Obviously, improvements in all times can be expected when the system is run under less highly loaded systems, and when the four context switches required to get each message from clients to I/O processes and back are eliminated.

The latter two operations were used to ascertain the overhead in message passing alone.

6.4. Socket implementation

The choice to implemented client-server IPC using sockets is typical of most (if not all) daemons in the 4.2 and 4.3BSD releases of UNIX. The implementation is aesthetically clean because a process can use well-known UNIX system calls (*read(), write(),* etc.) to send data to another process using a file descriptor. Previously UNIX had generalized file descriptors to allow

I/O on devices, pipes, and directories.

Socket interfaces exist on many other operating systems in addition to BSD releases 4.2 and greater and BSD-derived operating systems such as SunOS. They include AT&T System V Release 4, SCO Xenix, VMS, and MS-DOS. Furthermore, a socket-based implementation will allow a trivial extension of the server to support inter-machine communication. Older System V releases offer several different IPC mechanisms — message-passing, shared memory, and semaphores. However they are not portable to BSD implementations, and apparently will be phased out by AT&T in favor of another mechanism, *streams*.

A communication system implemented through the filesystem, such as a system of mailboxes, would be less efficient but more reliable than a socket implementation with in-memory message copying handled by the UNIX kernel. Under a mailbox system, the server would have to continuously poll the filesystem to watch for requests, as opposed to blocking until the UNIX kernel recognizes that data has arrived. Furthermore, the one-second resolution file modification field in UNIX inodes would not allow immediate recognition that a new message has arrived. A mailbox implementation might be more reliable because clients would be unaffected by a server crash — the server, after being re-booted, could process the client requests that arrived during and since the crash.

6.5. Network service

Extending the storage management system to allow access to remote clients is not difficult. First another socket, this one with an Internet domain, would have to be created and bound to a machine address (a port number). It could then be monitored for connection requests in the same manner that local requests are — by inserting the Internet socket's file descriptor into the set passed to *select()* whenever the local connection descriptor is.

Communication with remote clients would require some additional care to ensure that all messages are interpreted properly on the local and remote machines. Potential problems include differences in byte-ordering, data sizes and alignment, and struct packing.

6.6. Bag implementation

As it currently stands, I/O processes create all bag files in one directory for simplicity. Presumably a more efficient arrangement could be found. The structure of a UNIX directory does not allow efficient reading or updating. The BSD and System V releases maintain directories more or less as sequences of (inode,filename) pairs. When presented with a filename, the UNIX kernel searches directories sequentially to find its inode. Accessing a file at the end of a directory can be much slower than accessing one at the beginning. Directories are themselves files; maintaining more efficient data structures would therefore require much greater overhead than in-memory data structures. Most UNIX directories have relatively few files, so the added maintenance overhead would rarely pay off.

Separating files into subdirectories based on their suffix or bag number might improve performance in some cases. However every access to the four files that represent each bag would require the UNIX kernel to search the current directory to find the inode of the subdirectories, and then search the subdirectories.

One potential way to assist clients in storing data for efficient access would be to give them the ability to specify the device that any bag should be created on. This first release could have supported such an operation because each bag lies on a single device. However no such guarantee can be made in general. When the system is running on the Intel hypercube and utilizing its Concurrent File System, it will not be possible to determine if bags are on one device because the actual distribution is hidden by the filesystem.

7. SUMMARY

The ADAMS Storage Management System consists of a server which provides client processes with basic storage operations, and a set of I/O-dedicated processes. The functions operate on data structures called bags, and mask the system calls needed to implement client-server interprocess communication and disk I/O. The system can be utilized by all parts of the ADAMS run-time and compile-time systems because interfaces exist for both the C and C++ languages. The Storage Manager multiplexes up to 16 clients. It is easily extensible to suit the evolving requirements of the ADAMS project.

REFERENCES

[1] John L. Pfaltz *et al*, 1989, The ADAMS Database Language, IPC-TR-89-002, Institute for Parallel Computation, Charlottesville, Virginia.

[2] Stuart Sechrest, *"An Introductory 4.3BSD Interprocess Communication Tutorial*", Computer Science Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

[3] S.J. Leffler, R.S. Fabry, W.N. Joy, P. Lapsley, S. Miller & C. Torek, 1986, *"An Advanced 4.3BSD Interprocess Communication Tutorial*", Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

[4] Maurice J. Bach, The Design of the UNIX Operating System, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

[5] John Romkey, *"Networking With BSD-Style Sockets"*, UNIX World, July 1989, Volume VI, Number 7, pp.95-100.

APPENDIX A — ERROR CODES

If any client functions fail, the external UNIX variable *errno* is set to a number indicating the cause of the error. The client process can access a string describing the error through the *errstr()* function. Errors due to system call failure are defined by the UNIX kernel and are described in the *intro(2)* manual page. Errors specific to the ADAMS Storage Management System are defined as follows:

E_OPCODE	Bad opcode
E_BAG_EXISTS	Bag exists
E_BAG_DNE	Bag does not exist
E_BAG_NUMBER	Bad bag number
E_PACKET	Bad packet received by server
E_OUT_OF_BAGS	Out of bags
E_NOT_IMPL	Operation not implemented
E_NOT_SUPERUSER	Privileged operation
E_BAD_LENGTH	Bad length
E_BAD_SLOT	Bad item number
E_ITEM_DNE	Item does not exist
E_ITEM_UNDEF	Item is not defined
E_STDIO_ERROR	Stdio library error
E_BAG_HEADER	Bad bag header
E_FIXED_LENGTH	Bad length on fixed-length bag
E_INTERNAL	Internal error
E_FBACKUP	Couldn’t back up file pointer
E_FWRITE	Fwrite failed
E_LENGTH_WRONG	Table has wrong length for item
E_BAD_IONODE_SND	Bad I/O node send
E_BAD_IONODE_RCV	Bad I/O node receive
E_BAG_LOOKUP	Bag lookup failed
E_NOT_CONNECTED	Not connected to server
E_CONNECTED	Already connected to server
E_NO_SUCH_CHILD	No such I/O node

APPENDIX B — SERVER OPTIONS

Various command line arguments alter the operation of the server. They are listed below.

`-C`*child_program*
`-c`*child_program*

Set the pathname of the I/O process to *child_program*. The default is *child* in the server's working directory.

-D*server_directory*
`-d`*server_directory*

Set the working directory of the server to *server_directory*. The default working directory is the one it inherits when created.

`-f`

Stay in foreground. By default, the server goes into the background automatically when its boot sequence has completed.

`-h`

Print a Server Usage message and exit.

`-l`

Turn off logging of all operations. Only server errors will appear in the server log file, *server.log*.

`-r`

Do not look up clients' login ids and names in /etc/passwd. This can be expensive, and should always be turned off when measuring the Storage Manager's performance.

`-S`*socket_name*
`-s`*socket_name*

Set socket name through which connections are accepted to *socket_name*. The default is *_SOCKET_* in the server's working directory.