EXODUS Storage Manager[1] V3.0 Architecture Overview

(Last revision: April, 1993)

1. INTRODUCTION

This document describes the architecture of Version 3.0 of the EXODUS Storage Manager. It is assumed that the reader is familiar with the Storage Manager's client application interface and server process, which are described in the [exodSM]. This document is an adjunct to [exodSM], and the authors have made an effort not to repeat here anything that is contained in [exodSM].

The Storage Manager has a client-server architecture. Part of the Storage Manager, called the *client* or the *client library*, is a library of functions that is linked with the application program. The rest of the Storage Manager, called the *server* or *servers*, is a set of Unix processes that cooperate with the client portions of application processes.

Application programs call functions in the client library to access and manipulate data in objects. The interface between the application and the client library consists of a set of functions that operate on files, root entries, objects, and indexes, and a set of options whose values are determined at run-time by configuration files or by the application through function calls.

Some of the Storage Manager's work is performed by the client library in the address (process) space of the application. The rest of the work is performed by servers. Each server manages some data; each datum is managed by one server. The client library makes requests of the proper servers as its needs

[1] The Exodus software was developed primarily with funds provided by by the Defense Advanced Research Projects Agency under contracts N00014-85-K-0788, N00014-88-K-0303, and DAABO7-92-C-Q508 and monitored by the US Army Research Laboratory. Additional support was provided by Texas Instruments, Digital Equipment Corporation, and Apple Computer.

dictate. The interface between the client library and the servers is a set of *remote procedures* that perform low-level services such as locking data, allocating and deallocating disk blocks, and transaction management.

The remainder of this document describes in detail which functions are performed by clients and which are performed by servers. First, Section 2 describes concepts that are common to clients and servers. Section 3 describes the client library. Section 4 presents the servers' architecture. Section 5 describes the interaction between the client and servers. Section 6 describes the interactions among servers.

2. COMMON CONCEPTS

2.1. Volumes

The Storage Manager stores data on *volumes*. Volumes are Unix (TM) disk partitions or files, and have fixed sizes, in *blocks*. Blocks in a volume are numbered sequentially from the beginning of the volume. All cooperating servers and all volumes that these servers manage must have been configured (or formatted, in the case of volumes) with the same size blocks.

2.2. Pages

The unit of data that is transferred between a client and a server, or between a server and a disk, is a *page*. Pages are made of contiguous disk blocks. Each page, regardless of its size, has a *page ID*, which is the number of the first block in the page.

Each page has a *page type*. The page types are:

index pages	pages used for the keys and values in an index, or for meta-data for indexes
slotted pages	pages that contain small objects, headers for large objects, and meta-data,
file pages	pages that contain meta-data to maintain files,

large-object pages pages that contain large objects and meta-data for large objects,

log pages pages that contain log records,

bitmap pages pages at the beginning of volumes that indicate which pages are free and which are in use,

root entry pages pages at the beginning of volumes on which root entries are stored, and

volume header pages a single page containing meta-data about the volume.

The Storage Manager is configured, at compile time, with a minimum page size and a size for each type of page. The minimum page size is the size of a disk block. All page sizes are a power of two and a power-of-two multiple of the disk block size.

2.3. Buffer Pools

Each client and each server has its own *buffer pool*, in which it caches data from *secondary storage* (meaning a server, if this is a client's buffer pool, or a disk, if this is a server's buffer pool). A buffer pool, is a fixed-size set of *buffers*, each the size of a disk block. Updates to data are performed in a buffer pool. Data that have been updated, but have not been written to secondary storage, are called *dirty*. Pages containing dirty data or meta-data are called *dirty*.

A buffer in the buffer pool holds a single minimum-page-size page. Several (contiguous) buffers may be required to hold a single (non-minimum-page-size) page.

2.4. Buffer Groups

The buffer manager provides the concept of a *buffer group* as proposed in the DBMIN buffer management algorithm [Chou85]. A buffer group is a collection of buffers containing *fixed* and *unfixed* pages. Each buffer group is constrained in size and has its own replacement policy. When a set of pages is to be fixed in a buffer group, the buffer group may contain unfixed pages that have to be *swapped out* to accommodate the new fixed pages. The buffer group's replacement policy determines whether the least-recently-used (LRU) or most-recently-used (MRU) unfixed pages are chosen as victims to be swapped out.

Dirty pages that are swapped out of a buffer group are sent to secondary storage before their buffers are reused. The buffers that are made available during the swap are put on a list of free buffers. Those buffers that are not needed for new fixed pages remain on the free list for use by other buffer groups. An example of when this might happen is this: An LRU buffer group needs one buffer to fix a minimum-size page S. The buffer group's least-recently-used unfixed page, L, is chosen as a victim for swapping, and L is the size of four minimum-size pages, so four buffers are freed when L is swapped out of the buffer group. Three pages remain on the free list after the swap is completed and the buffer group has fixed S.

2.5. Objects, Slots, and OIDs

Objects are a unit of data that the application program uses. Objects are either *small* or *large*. Small objects fit in a single slotted page. Large objects are those that do not fit in a single slotted page. Objects are identified by an OID, which contains a volume ID, a page ID, a *slot number*, and a *unique number*.

Slotted pages contain meta-data as well as objects: a page header and a slot array [Date81] at the end of the page. Each non-empty slot in the array contains the offset of the object from the beginning of the page. This mechanism allows the Storage Manager to move objects on a page without affecting the integrity of data structures that reference objects on that page. An object's location within a page may change, but its OID does not.

Objects can move from page to page and maintain their OIDs. An object that moves to a new page becomes *forwarded*. The data on its *home* page is the OID that identifies its new location.

The unique number of an OID stored in the slot array along with the object's offset from the beginning of the page. Every time an object is accessed by its OID, the Storage Manager validates the OID by comparing the unique number in the OID with the unique number in the specified slot. If the two differ, the OID is considered to be invalid. The generation of unique numbers is discussed in the Appendices to [exodSM].

Large objects occupy slots on their home pages, and they also occupy one or more large object pages. Large object pages are dedicated to a single object. The data in large objects are stored in pages whose order and location are maintained in a B+ tree. The home page of the object, a slotted page, points to (or includes, if the large object is small enough) the root of the large object's tree.

3. THE CLIENT LIBRARY

The Storage Manager client library is linked with an application program. The functions in this library provide for creating, destroying, reading, writing, inserting into, deleting from, and versioning objects. Functions are also provided for creating, destroying, accessing and modifying files of objects and indexes. Initialization, administration, and transaction support functions are included as well.

The application program calls the interface functions of the client library. It does not access the server directly. The client library functions locally perform as much work as possible, and communicate with servers when necessary.

3.1. Buffer Pool

Applications can open buffer groups in the client buffer pool for their own purposes. The client library allocates buffer groups for its own purposes, such as for logging. Applications gain access to pages that are fixed in the client buffer pool through *user descriptors*. Each user descriptor describes a range of bytes of data in an object. The bytes that it describes are contiguous in the address space of the application, even if the data do not occupy contiguous pages in the volume.

The client's buffer manager has a complex mechanism for allocating contiguous buffers for *large objects*. Different, overlapping portions of large objects can be fixed in the client's buffer pool simultaneously (through several user descriptors), and the buffer manager ensures that each such fixed portion is contiguous in the buffer pool, hence certain pages may be partially or wholly duplicated in the buffer pool. The buffer manager ensures that updates to these pages are reflected in all copies of the data.

3.2. Locking

The client contacts servers to acquire and release locks on pages. The client's buffer manager keeps track of the locks that it has on pages that are cached in the buffer pool, in order to minimize interaction with servers.

Each time the client library requests a page from a server, it also requests a lock for the page. Locks can be *exclusive*, or *shared*, depending on the use the application is making of the data. Details on the Storage Manager's lock management are found in the Appendices to [exodSM].

Lock requests are not always granted by a server. If the request would cause a deadlock, the request is denied immediately. If deadlock is not an issue, and a requested lock is held by another transaction, the lock request will wait for a time (determined by the application through the "locktimeout" option). If the request cannot be granted in that time, it is denied. A denied lock request causes the client's operation to fail. If necessary, the failed operation forces the client library to abort the transaction (rather than leave data or meta-data in an inconsistent state). If the transaction is not aborted, the application may retry the operation or may abort the transaction.

3.3. Logging

When the client updates data, it logs its updates so that the updates can be *undone* (if the transaction aborts) or *redone* (if the transaction commits and recovery is necessary after a crash). The client, however, does not have its own log; rather, it ships log records to servers. Log records describe updates to data on a single page. If several pages are updated, each update to each page merits its own log record. The log record for a page is sent to the server that manages the page in question, and it is always sent before the dirty page is sent to the server. Log records vary in size and can be very small, so they are collected into pages and sent to the server when the pages are full. The size of a log page is determined by the server, and it can differ from server to server. The client library caches information about the logging characteristics of each server with which it is communicating.

Each page has a *log record count* (LRC). An update to a page causes its LRC to be incremented. The LRC can be considered to reflect, in some sense, the state of the page. If two copies of a given page have identical LRCs, the data in the pages are also identical.

Details of the Storage Manager's logging and recovery schemes are described in [Fran92].

3.4. Transactions

The application can be in one of four transaction processing states (described in [exodSM], Section 4.3.2): INACTIVE (not running a transaction), ACTIVE (running a transaction), ABORTED (the transaction is partially aborted), and PREPARED (the transaction is prepared). The client has its notion of the application's transaction state, and each server also has its notion of the same. The client and the servers' notions of this state do not always agree, because the client and servers do not always communicate on a timely basis, but eventually agreement is reached in every case.

When the application begins a transaction, the client library creates a *local transaction ID* (TID). The client library now considers a transaction to be ACTIVE, even though no server does. When the application calls a client library function that make a reference to data, the client library contacts the server or servers that manage the volumes of interest and establishes *connections* with them.

The client library now mounts the volumes of interest and begins transactions on each of the servers in question, at which time each server sends the client library a *server TID*. These servers now consider a transaction to be active, although they do not in any way recognize that they are participating in *the same* transaction. The servers are running *threads* of the transaction. When the client requests data and locks from a server, it identifies the transaction thread by the server's TID.

The application either commits or aborts the transaction. A server may also abort a transaction for its own reasons, To commit a transaction, the client library ships all the remaining dirty pages and their log records to the servers. If only one server is involved, the client library instructs the server to commit the transaction.

If several servers are involved, the client library initiates a two-phase commit protocol. It designates one server as the coordinator, and asks the coordinator to prepare and commit the transaction. The coordinator performs the two-phase commit procedure with all the other servers, and returns the result to the client library. If the commit is successful, the client returns to the INACTIVE state.

To abort a transaction, the client discards all the pages in its buffer pool and notifies the servers that it wishes to abort the transaction. Again, this will place the client in the INACTIVE state. If the server aborts a transaction, the client library is informed in the next response it receives from the server. When it receives such a response the client moves to the ABORTED state. To end the transaction and to return to the INACTIVE state, the application program must explicitly call sm_AbortTransaction( ), which notifies all the participating servers. After a transaction has been committed or aborted, the application may begin another transaction.

When an application is finished using the Storage Manager, it can shut down the client library to release all the resources that the client library allocated. The application must dismount any volumes that it mounted explicitly.

4. THE SERVER

Loosely, the Storage Manager server can be considered to be a collection of “subsystems” that provide different services, but the server really is a monolithic program. None of the subsystems stands alone, however, the distinction is useful for the purpose of this discussion. The rest of this section examines how client requests are satisfied by the server’s various subsystems: threads, disk I/O, the buffer manager, files, locking, logging, and recovery, shown in Figure 1.

figure archfigure.psf width 6i

Figure 1: The Storage Manager Architecture

4.1. Threads

In order to serve several clients simultaneously, and to perform disk I/O in parallel with network I/O, the server is multi-threaded. Server threads are units of execution similar to the coroutines provided by Modula-2. Each thread has its own stack for maintaining its execution state. A thread is always in one of the following states: executing, not in use, waiting on the ready queue for a chance to continue executing, or awaiting a resource. Resources that a thread may await include locks, latches, semaphores, completion of disk I/O, timers, and signals from other threads. Thread switching is implemented using the setjmp( ) and longjmp( ) functions in the standard C library. Threads are not preemptively scheduled.

When a client request arrives over the network, the server assigns an unused thread to handle the request and begins executing the thread. The thread runs until it has to wait for a resource, voluntarily gives up the CPU, or replies to the client's request. Threads may execute on behalf of a server task, rather than in response to a client's request. (There are threads dedicated to timing out lock requests, taking checkpoints, and timing out idle clients, for example.) Threads may also cooperate to accomplish a task. (For example, a thread makes a disk request on behalf of a client; another thread determines when the disk request is completed and informs the first thread of that fact.)

4.2. Disk I/O

Since the server supports multiple clients, it is designed to avoid blocking in Unix I/O system calls. A server performs I/O locally only when there is no need for parallel I/O i.e., only when just one client is connected and no other threads are performing I/O on the server's behalf.

A server performs disk I/O for two or more volumes in parallel by forking a process for each volume when the volume is first mounted. A server thread sends I/O requests to the appropriate disk I/O process, placing itself on a wait-list so that other threads can execute. Meanwhile, the I/O process performs disk I/O on behalf of the server. The disk I/O process reads into and writes from the server's buffer pool, which is in shared memory. Servers threads can queue several requests for I/O processes simultaneously, and each request can contain a vector of pages to be written or read.

Once an I/O request is satisfied or terminates in error, the I/O process notifies the server and awaits its next request. The server allocates an unused thread to accept the notification, wake up the thread that issued the request, and put itself back on the list of unused threads.

When the last client using a volume dismounts the volume, the server flushes all the dirty pages for that volume and issues a request to the disk I/O process to close. The disk I/O process then exits.

A server and its disk I/O processes use a combination of shared-memory queues, semaphores (see semctl(2))), timers (SIGALRM), and sockets to communicate with each other. For each volume, there is a pair of queues, one for requests and the other for responses. The server places a request in the request queue. The disk process "moves" the request-message to the response queue by changing a single pointer that divides the two queues. Each disk I/O process polls its request queue until the queue is empty, then sleeps on a semaphore.

The server polls all the response queues. In fact, the server polls the response queues only when it wakes up from a select( ) system call. The server sets a shared-memory variable to tell the I/O processes when it is blocked on select( ). When it is blocked, the I/O processes "kick" the server by sending a 1-byte message to a socket, thereby waking up the server.

4.3. Buffer Manager

The server's buffer manager has a number of important features. The buffer manager does not contain the complex large-object buffering scheme found in the client's buffer manager, because all large-object operations are performed by the client. The buffer manager enforces a write-ahead logging protocol. Since, unlike the client, the server is multi-threaded, the server's buffer manager also provides latches and semaphores for synchronizing threads' accesses to pages in the buffer pool.

The server uses buffer groups to allocate buffer space for distinct purposes. There is one large LRU buffer group used to satisfy client I/O requests. Separate smaller buffer groups are used for reading and writing the log. Smaller buffer groups are also allocated for managing the bitmap pages associated with each volume. To see how these buffer groups are used, consider the buffer group used for reading the log. Imagine a transaction that is being aborted and that has log records on 1,000 different log pages. As the log pages are scanned during the undo operation, only a few log pages are cached, subject to the size of the log-read buffer group. Without the log-read buffer group, 1,000 pages that would likely only be used once would end up being cached.

The buffer manager on the server keeps track of pages that were recently swapped to disk, along with their LRCs, which reflect the pages' states. This is used for inter-transaction page caching (described in section 5.2, below).

4.4. Lock Manager

The server's lock manager issues locks for *lock ID*s. A lock ID is a 12 byte integer. Lock IDs are generated from lock requests, and while they may include a page ID or a file ID, the lock manager does not interpret them as such.

Before a lock request is granted, the lock manager performs deadlock detection on its local waits-for-graph (containing information for the single server only). Servers do not perform any global deadlock detection among themselves.

If a lock request would result in a local deadlock, the request is not granted. If a local deadlock is not detected, the lock manager grants the request immediately, if possible, or puts the request on a wait-list if the lock is held by another transaction.

Each transaction has a lock timeout value, which determines the maximum time that the transaction will wait for a lock. If that time expires before the lock request is granted, the request is aborted, and the server replies to the client that the lock is *busy*. This is a simple way to avoid global deadlocks.

Access to objects involves the standard hierarchical two-phase locking (2PL) protocol (see [Gray78] or [Gray88]). The lock hierarchy contains two granularities: file-level, and page-level. The page that is locked when an object is accessed is the page containing the object header. There are six lock modes: no lock (NL), share (S), exclusive (X), intent to share (IS), intent to exclusive (IX), share with intent to exclusive (SIX) [Gray78, Gray88]. Files can be locked in any of the six modes, while pages are only locked in share and exclusive mode.

A table is used to determining whether two lock requests are compatible (eg., when a client holds a lock on a file and another client wants to obtain a lock on it as well). A table of the lock compatibilities is found in the Appendices of [exodSM].

A table of the lock convertibility is found in the Appendices of [exodSM].

4.5. Logging and Recovery

The Storage Manager's logging and recovery subsystem is based on the ARIES recovery algorithm [Moha89]. Details on the logging and recovery subsystems can be found in [Fran92] and Section 5.2.5 of [exodSM].

Each server has a log volume, which it mounts when it starts. If the log volume is newly formatted, the server initializes the log, otherwise, the server performs recovery. The log is managed as a circular buffer of recent log pages. When the end of the log volume is reached, log records are placed at the beginning of the log volume. Logically, the log is a sequence of log records identified by a log sequence number (LSN). LSNs contain a physical address in the log and a *wrap count* which is used to make LSNs unique.

While generating log records, a server periodically takes a checkpoint. The server makes a lists of the dirty pages in the server's buffer pool, the active transactions, the volumes that are mounted, and other state information, and writes a log record that contains these lists. Recovery, should it be needed, begins its analysis phase with the last checkpoint record written, so, as a general rule, recovery time is shorter if checkpoints are frequent.

The checkpoint frequency is determined by several factors: First, it is based on the number of log records generated. The server's "checkpoints" option limits the number of log records that the server generates between checkpoints. The option's value can be changed while the server is running.

Second, the buffer manager initiates checkpoints when it invalidates a dirty page to make room for a new page being read. This causes semi-random checkpoints to be taken when the buffer pool fills with many dirty pages.

The server has a thread whose sole responsibility is lazily writing dirty pages to secondary storage. When a checkpoint is taken, this thread is awakened if it is not already active. By writing out dirty pages when no other server activity is going on, recovery time is reduced.

4.6. File Manager

All Storage Manager objects are stored in *files*. A file is a collections of large-object pages and slotted pages, held together with a B+ tree index whose key is the page ID. The index for a file resides on file pages. All operations that involve file pages occur on the server. File pages are never sent to clients, for two reasons. First, it eliminates the need for the complex cache consistency system that would be required if multiple clients were manipulating file pages. Second, the file code uses *savepoints* [Moha89], which are not available to clients, to back out of operations in the event of a failure. More discussion of Storage Manager files can be found in [Care86,89].

5. CLIENT-SERVER INTERACTIONS

5.1. Connections

When an application initializes the Storage Manager, no communication takes place between the application process and servers. The client portion of the application process contacts servers only when data are required from one or more servers. Communication between clients and servers is initiated by the clients, which open TCP connections using the Unix (TM) Sockets interface. The TCP connection remains alive as long as the client is *active*.

Servers time out any connections left over from inactive clients. A client is inactive with respect to a server if it does not have a transaction running on that server. If an application shuts the Storage Manager down, communication with servers is severed by the client library.

If either the client or a server closes a connection or terminates abnormally, the other process is notified by Unix. If the network breaks, both processes are notified.

When Unix notifies a server that a client's TCP connection has terminated (whether due to the client's abnormal termination or a dysfunctional network), the server aborts the transaction that the client was running, if any, and frees all the resources that the client had acquired.

Clients are notified of a TCP connection's abnormal termination when the they attempt to send a request or receive a response. When this occurs, the client behaves as if the server had aborted the active transaction. The application is informed of the (partially) aborted transaction the next time it tries to use the Storage Manager. It is the application's responsibility to finish aborting the transaction by calling the client library function sm_AbortTransaction( ), which aborts the transaction on all the participating servers and cleans up local resources associated with that transaction.

5.2. Inter-Transaction Page Caching

When an application commits a transaction, the pages in the client buffer pool are marked *invalid*, but are not thrown away. When a new transaction begins, the application performs an operation on data, and the client issues a request to a server for a page. If the client's buffer pool still contains the page in question, and the page is marked invalid, the client's request to the server contains the page's LRC. The server, when it receives such a request, checks the LRC, and if the LRC is up-to-date, it informs the client that the client's copy of the page is up-to-date, thereby avoiding shipping an up-to-date page across the network.

The server keeps a list of the LRCs for pages recently swapped out, in an effort to avoid reading pages from disk only to read the page's LRC and find that the page need not be shipped to the client.

6. SERVER-SERVER COMMUNICATION

When an application uses data on several servers, the client library maintains some state information about what servers are in use for the transaction. The servers, however, have no notion that they are participating in a distributed transaction. When the application requests the Storage Manager to commit the transaction, the client library initiates a two-phase commit procedure among the participating servers. The client chooses a server (it does not have to be one of those participating) to coordinate the two-phase commit protocol. That server is sent a list of the participating servers, their Internet addresses, and the transaction-identifiers by which that transaction is known to each server. The coordinating server takes over, and when the fate of the transaction is determined, it informs the client library of the result, and terminates the interaction with other servers.

The two-phase commit protocol that is used by the servers is a variant of Presumed Abort (PA) [Moha83]. Each server is capable of being a coordinator or a subordinate, or both. The servers communicate over UDP, using the Unix (TM) Sockets interface.

When a server crashes and recovers, it uses the same PA protocol to resolve transactions that were prepared under PA before the crash. Of the recovered prepared transactions, there are two kinds. First there are the transactions that were prepared by the Storage Manager as a result of an application's request to commit the transaction. The server performing recovery crashed before these transactions were resolved. These transactions are resolved (committed or aborted) by the coordinating server.

Second, there are transactions that were prepared by the *external two-phase commit functions* (see Section 4.11.1 of [exodSM]). These transactions, after being recovered, continue to consume resources and **must** be resolved by a recovery application in a timely fashion.

7. REFERENCES

[Care86] M. Carey, D. DeWitt, J. Richardson, and E. Shekita, *Object and File Management in the EXODUS Extensible Database System*, **Proc. of the 1986 VLDB Conf.**, Kyoto, Japan, Aug. 1986.

[Care89] M. Carey, D. DeWitt, E. Shekita, *Storage Management for Objects in EXODUS*, **Object-Oriented Concepts, Databases, and Applications**, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.

[Chou85] H. Chou and D. Dewitt, *An Evaluation of Buffer Management Strategies for Relational Database Systems*, **Proc. of the 1985 VLDB Conf.**, Stockholm, Sweden, Aug. 1985.

[Date81] C. Date, *An Introduction to Database Systems (3rd edition)*, Addison-Wesley, Reading, Mass., 1981 (pg. 173).

[Fran92] M. Franklin, M. Zwilling, C.K.Tan, M. Carey, and D. DeWitt, *Crash Recovery in Client-Server EXODUS*, **Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data**, San Diego, CA, June 1992.

[Gray78] J. N. Gray, *Notes on Database Operating Systems*, **Lecture Notes in Computer Science 60, Advanced course on Operating Systems**, ed. G. Seegmuller, Springer Verlag, New York 1978.

[Gray88] J. Gray, R. Lorie, G. Putzolu, I. Traiger, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, **Readings in Database Systems**, ed. M. Stonebraker, Morgan Kaufmann, San Mateo, Ca., 1988.

[Moha83] C. Mohan, B. Lindsay, *Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions*, **Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing**, Montreal, Canada, August, 1983.

[Moha89] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, *ACM Transactions on Database Systems*, Vol. 17, No 1, March 1992.

[exodSM] *Using the EXODUS Storage Manager V3.0*, unpublished, included in EXODUS Storage Manager software release.