A Client-Based Transaction System to Maintain Data Integrity

William H. Paxton
Xerox Palo Alto Research Center

Abstract

This paper describes a technique for maintaining data integrity that can be implemented using capabilities typically found in existing file systems. Integrity is a property of a total collection of data. It cannot be maintained simply by using reliable primitives for reading and writing single units -- the relations between the units are important also. The technique suggested in this paper ensures that data integrity will not be lost as a result of simultaneous access or as a result of crashes at inopportune times. The approach is attractive because of its relative simplicity and its modest demands on the underlying file system. The paper gives a detailed description of how *consistent, atomic transactions* can be implemented by *client* processes communicating with one or more *file server* computers. The discussion covers file structure, basic client operations, crash recovery, and includes an informal correctness proof.

1. Introduction

This paper describes a technique for maintaining data integrity that can be implemented using capabilities typically found in existing file systems. Integrity is a property of a total collection of data. It cannot be maintained simply by using reliable primitives for reading and writing single units -- the relations between the units are important also. The technique described below ensures that data integrity will not be lost as a result of simultaneous access or as a result of crashes at inopportune times.

The environment we have in mind is a collection of computers capable of sending messages to each other over a high bandwidth network [1]. One or more of these computers act as *file servers* -- they are a shared repository of data for the other machines which are called *clients*. The client machines issue read or write commands in the form of messages to the particular file server holding the addressed information. The client gets back a response from the server after the command is completed. The response is a message containing the requested information in the case of a read or an acknowledgement in the case of a write.

A *transaction* is a sequence of reads and writes by some client. Accesses may be made to several files located on different servers. From the standpoint of maintaining data integrity, it is important that the system for carrying out transactions have the following two properties.

1. The *consistency* property: although many clients may be performing transactions simultaneously, each will get a consistent view of the shared data as if the transactions were being executed one at a time.

2. The *atomic* property: for each transaction, either all of the writes will be executed or none of them will, independent of crashes in servers or clients.

There are two main approaches to building a transaction system that has these properties. The first makes the file servers responsible for ensuring that transactions are consistent and atomic. Designs for such systems have been described by Gray [2], Lampson and Sturgis [3], Israel, Mitchell, and Sturgis [4], and others. The alternative method is described in this paper; it puts the burden on the clients rather than the servers.

The client-based approach is an attractive option because of its relative simplicity and its modest demands on the file servers. It is similar to the Lampson and Sturgis single server algorithm and is significantly less complicated than their multiserver algorithm in which the transaction state is distributed over many machines. This simplicity makes the client-based approach relatively easy to implement -- an early version of the method described in this paper went through two cycles of design, implementation, and debugging in about two work-months and is now in regular use as part of an experimental office information system.

The following sections describe the client-based approach to providing transactions that are consistent and atomic. Section 2 discusses the requirements for the file servers and presents the two types of transaction files. Section 3 outlines the basic actions that are available to clients for carrying out transactions. Crash recovery, which is needed to complete interrupted transactions, is covered in section 4. A proof that the system does actually implement consistent and atomic transactions is sketched in section 5. Some loose ends concerning large data files and file creation and deletion are discussed in section 6. Section 7 suggests some possible extensions.

2. File Servers and Transaction Files

The discussion of the system begins with an overview of the capabilities required in the file servers and the structure of the two types of transaction files.

Server Capabilities

One of the merits of a client-based approach is the simplicity of its requirements which make it possible for typical existing file systems to support the transaction machinery. The server's primary role in transactions is just to provide sector-at-a-time random access to its files. In addition, the server allows a client to gain sole access to a file by locking it. The key which is returned to the client must be supplied in all subsequent accesses until the file is unlocked. If a locked file is not accessed for a period of time, the server automatically releases the lock so that a crashed client will not leave files permanently unavailable.

These are the basic file server capabilities needed for a client-based transaction system: sector at a time random access, file locks, and automatic unlocking after prolonged inactivity. For more information about a file server which satisfies these requirements and was used in an early implementation of our method, see a companion paper by Swinehart, McDaniel, and Boggs [5].

Transaction Files

Transactions involve two distinct types of files: *Data Files* which hold the information operated on by transactions, and *Intentions Files* which contain records about data files changed by a transaction and are used in crash recovery. The transaction system implements both types with standard files provided by the servers; there is nothing special about them as far as the servers are concerned.

Data files are structured by the transaction system so that a single write will convert them to a new version that may have many sectors changed. This is accomplished by providing a level of mapping from *logical* sector numbers used to address the data to *real* sector numbers as understood by the file servers. The first real sector of each data file is a *header* which contains a *sector map* indicating where in the file the various logical sectors are stored. By changing the sector map, the mapping from logical sector numbers to real sector numbers can be changed for the entire file. The transaction system directs writes to unused real sectors, so until the header is rewritten, no old data is lost.

In addition to the sector map, the header contains a free-list of unused real sector numbers and a record indicating the transaction, if any, in which this file is currently involved. A simple compactor process can periodically run as a client to relocate logical sectors to unused locations at the front of the file, update the sector map and free-list, and then truncate the file to reclaim storage.

Intentions files contain a state variable, a transaction number, a list of changes, and a sequence of headers. The transaction state alternates among STARTED, COMMITTED, and COMPLETED. While the state is STARTED, the transaction can be aborted with no significant effect on the data files; once the state becomes COMMITTED, all of the writes will eventually be done at which time the state will change to COMPLETED. The transaction number is incremented during the completion of a transaction and, along with the state variable, has an important role in crash recovery. The list of changes holds identifiers for data files that have been modified in the transaction and pointers to copies of their new headers which are saved in the remainder of the intentions file.

3. Client Actions

The basic client actions are *BEGIN TRANSACTION, OPEN, READ, WRITE, ABORT TRANSACTION*, and *END TRANSACTION*. The effects of these operations are described below. To simplify the descriptions, we assume that the client has a permanently allocated intentions file and we allow the client to have only one transaction in progress at a time. These restrictions can be removed by having BEGIN TRANSACTION allocate an intentions file and return a *Transaction Identifier* which would be passed as an additional argument to all the other routines.

BEGIN TRANSACTION

This routine is called at the beginning of each transaction to change the state to STARTED. The following routines signal an error if they are called in any other state.

OPEN

OPEN receives a file identifier and returns a *handle* for use in subsequent accesses. The identifier includes data indicating where the file is located, so the following operations can communicate with the appropriate server. OPEN locks the file, saves the key, and reads the header. If the header indicates that the file is involved in another transaction, a crash must have previously occurred preventing the completion of that transaction, so recovery is done in a manner described below and the header is then reread.

Note that the header is not written at this time; it is modified only in END TRANSACTION, and then only if there have been WRITE's to the file.

READ

The arguments to READ are a file handle, a logical sector number, and a buffer to receive the data. A local copy of the header sector map is used to convert from logical to real sector number, and the data is transferred from the file server to the buffer.

If the file lock has been broken (i.e., the file server has released the lock for some reason such as client inactivity), all files opened for this transaction are unlocked, the state changes to COMPLETED, and an error code signifying *Transaction Aborted* is returned. The transaction system makes no assumptions about how a client program will deal with such aborts. In typical usage, they will probably be rare and the client can simply treat them like a software error requiring a program restart.

WRITE

WRITE receives as arguments a file handle, a logical sector number, and a buffer containing the data to be written. It first uses the local copy of the file free-list to allocate an unused real sector. Then, the local copy of the file sector map is checked to see if the logical sector being written already exists. If so, its real sector number is recorded in a separate list to be merged with the header free-list during the

completion of the transaction. Finally, the local copy of the sector map is updated to indicate the new mapping, and the buffer is transferred to the file server. As in READ, the transaction is aborted if the file lock has been broken.

Notice that all of the useful information from before the transaction is left untouched. Only local copies of headers are changed, and writes go to unused real sectors. The addition of sector numbers to the header free-list is delayed until END TRANSACTION so that subsequent writes during this transaction will not use them.

ABORT TRANSACTION

All files opened for this transaction are unlocked and the state of the transaction is changed to COMPLETED.

END TRANSACTION

There are three cases to consider for END TRANSACTION depending on whether zero, one, or more than one file was written. In all cases, the transaction is immediately aborted if a broken lock is discovered before the state becomes COMMITTED.

If no files were changed, END TRANSACTION simply unlocks all the files that were opened, changes the state to COMPLETED, and returns.

If a single file was modified, then after the read-only files are successfully unlocked (i.e., their locks were not found to be broken), the modified file has its header written and its lock released. In both this case and the previous one, the intentions file is not required and no accesses are made to it.

If multiple files were written, END TRANSACTION must leave around enough information so that if it crashes after committing to make the changes someone else will be able to complete them. The intentions file is the place for this information. The sequence of steps is as follows:

1. Unlock all of the read-only files.
2. Lock the intentions file. (This is the first action involving the intentions file for this transaction; up to this point all the activity has been with the data files.)
3. Mark each modified file as being changed in this transaction by placing in its header the new transaction number and the identifier for the intentions file.
4. Write copies of the new headers to the intentions file. The header free-lists are first updated to include the real sectors which are now unused.
5. Write the list of changes to the intentions file with the transaction number updated and the state variable set to COMMITTED. There is now a committment to completing the transaction rather than aborting it. If a broken lock is discovered after this point, crash recovery will be invoked to finish the transaction.
6. Write the new headers to the data files saying that they are not involved in any transaction. Unlock each file after writing its header.
7. Write the intentions file header with the state variable set to COMPLETED, and then unlock it.

Note that the state recorded in the intentions file is either COMMITTED or COMPLETED. The STARTED state is not recorded since the intentions file information is solely for crash recovery, and the recovery mechanism does not need to distinguish between STARTED and COMPLETED -- it is only invoked for a transaction that was left in the COMMITTED state.

At this point it is possible to summarize the overhead for the transaction mechanism in terms of extra interactions with the file servers in addition to the data accesses. For a transaction with N files opened and M of them modified (M greater than 1), there is an extra lock and unlock for the intentions file, N extra reads to get the data file headers, and 3M+2 extra writes. The writes, in the order that they occur, are to mark each of the M files as in the transaction (step 3), to write the M headers to the intentions file (step 4), to write the list of changes and set the intentions file state variable to COMMITTED (step 5), to write the M headers to the data files (step 6), and to reset the intentions state to COMPLETED (step 7). Large data files will require additional reads and writes for extra header information -- see Section 6 for details.

4. Crash Recovery

In a client-based system, crash recovery for transactions is done "on demand" rather than immediately after restarting. The recovery procedure is triggered when a file that was involved in an interrupted transaction is next accessed. There are two indications that a crash has previously occurred: an unlocked data file may claim to be involved in a transaction, or the state variable of an unlocked intentions file may be COMMITTED rather than COMPLETED. The former case is dealt with by recovery code in OPEN; the latter is handled by finishing the transaction in a manner described below.

Recovery in OPEN

Recall that OPEN locks each data file involved in a transaction and reads its header before any READ or WRITE access to that file is made. Thus, the READ and WRITE operations are guaranteed to come after any crash recovery performed by OPEN.

When OPEN finds a data file claiming to be in a transaction, it first checks whether the header points to an intentions file that is currently in use by this client. If so, the crash must have happened after marking the header in step 3 of END TRANSACTION and before the intentions state could be set to COMMITTED in step 5. Otherwise, if the state had become COMMITTED, the transaction would have been completed when the intentions file was first reacquired. Therefore, the previous transaction was never COMMITTED, and the file currently being opened can simply be marked as not in a transaction.

If the data file being opened names an intentions file other than one currently in use by this client, OPEN now attempts to lock that other intentions file, waiting if necessary until it can acquire the lock. OPEN cannot simply assume that this data file is okay on the basis of someone else having locked the intentions -- the client who has the intentions locked may be currently waiting to get at this file in order to complete the transaction! To resolve this potential conflict, OPEN unlocks the data file while it is waiting to lock the intentions file. After acquiring a lock for the intentions file, OPEN relocks the data file and rereads its header to see that the file still claims to be in the transaction. (There is unfortunately still a potential for deadlock here, since the client who has the other intentions locked may be waiting in exactly the same place in OPEN trying to lock our intentions file. It is a very unlikely situation, but not an impossible one.)

After locking the intentions file, OPEN checks to see that the intentions state variable is set to COMMITTED, that the data file is referenced in the intentions change list, and that the transaction number in the intentions is the same as that in the data file header. If these conditions are not all met, the original transaction was never COMMITTED, so the data file is marked as not in a transaction and the intentions file is unlocked. If all of the conditions are met, OPEN causes the transaction to be completed according to the following procedure.

Finishing a Transaction

To complete an interrupted transaction (i.e., one that was COMMITTED but not COMPLETED), it is necessary to process each data file listed in the intentions change list. The operation for each one begins by acquiring a lock, waiting if necessary for someone else to unlock the file. It would be an error to skip a locked file, since it might be locked by a client who is trying to get at our intentions file to complete the same transaction. If so, that client will soon unlock the data file so that we can get it. Note that here we do not unlock the intentions file while waiting to get a data file, whereas in OPEN we do unlock a data file while trying to get its intentions. This asymmetry serves to resolve potential deadlocks regarding responsiblity for recovering from a particular crash -- everyone defers to the client who has the intentions file locked.

After locking the data file, we read its header and check to see whether it still claims to be involved in this transaction -- in other words, whether it contains this intentions file identifier and this transaction number. If not, we can simply unlock it since it must have been completed already, either in the original transaction or in a previous crash recovery attempt. Otherwise, we write the new header for the data file from the intentions, unlock it, and go to the next file listed in the change list.

5. Sketch of Correctness Proof

There are two main properties to be considered regarding the correctness of an implementation of transactions: the atomic property and the consistency property. Recall from the Introduction that the atomic property is satisfied if either all the writes take place or none of them do, while consistency is achieved if clients get a view of the data base such as would happen if the transactions for all the clients occurred sequentially rather than overlapped. Consistency will be considered first.

Consistency Property

It has been shown (see Eswaran, et al., [6]) that transactions will be consistent if they are *well-formed* and *two-phase*. A transaction is well-formed if it locks each file before accessing it and ultimately unlocks them all. Two-phase transactions begin with an initial phase during which all locks are acquired followed by a final phase in which all locks are released. No locks are released during the initial phase, and none are acquired during the final phase.

> As an example to clarify why well-formed transactions must also be two-phase if they are to be consistent, suppose there is a directory file containing identifiers of mailbox files for a set of clients, and Client 1 wants to send some mail to Client 2. To do this, Client 1 locks the directory, reads it to find the mailbox identifier for Client 2, locks the mailbox, and puts new mail in it. Client 2 however has decided to trade in his old mailbox for a shiny new one at just this moment! To do this, he locks the directory, locks his old mailbox, takes out any mail that is there, and changes the directory to point to his new mailbox.
>
> If these transactions are not two-phase, there is a danger of inconsistency. If Client 1 unlocks the directory before locking the mailbox, Client 2 can squeeze in to perform his transaction and leave Client 1 sending mail to an obsolete destination. There is no danger of inconsistency if the transactions are two-phase: if Client 1 locks the directory first, he will send his mail to the old mailbox and Client 2 will pick it up before changing to the new one; if Client 2 locks the directory first, he will change it before Client 1 reads it, and Client 1 will put his mail in the new mailbox. (Thanks to Howard Sturgis for this example.)

Transactions that are not committed do not change any significant data and hence do not effect consistency. If a transaction is committed, it is eventually completed, either normally or by crash recovery. Crash recovery does not endanger consistency since no READ or WRITE accesses to involved data files are allowed until the interrupted transaction is either completed or aborted. Finally, if a lock is broken, the transaction is either aborted or completed by crash recovery depending on whether or not it had been committed. Thus, to demonstrate consistency, it suffices to show that any transaction that is completed normally is well-formed and two-phase. Well-formedness is clear since OPEN locks the files, READ and WRITE accesses only come after OPEN, and END TRANSACTION unlocks the files. The transactions that complete normally are two-phase since all locks are acquired before END TRANSACTION and released during it.

Atomic Property

To prove that transactions satisfy the atomic property, it is necessary to show that even after a crash at any time during a transaction or during an attempted crash recovery we can still eventually make all of the changes or none of them. If the crash occurs before END TRANSACTION is called, none of the file headers will have been modified, so none of the writes will be apparent. If the crash occurs during END TRANSACTION, we must to consider the exact type of transaction and its stage of completion at the time of the crash.

If no files were written during the transaction, the crash simply interrupted the unlocking operation which will complete automatically when the servers cause the locks to time out. If a single file was written in the transaction, then in case the crash occurs before writing its header, no change is visible; otherwise, the crash occurs after writing the header, so the transaction is complete as soon as the lock times out -- no extra recovery is needed.

In a transaction that has modified more than one file, there are three possibilities to consider: crashing before setting the intentions state variable to COMMITTED, crashing after setting the state to COMMITTED but before setting it to COMPLETED, and crashing after setting it to COMPLETED. Crashes prior to changing the state to COMMITTED will result in the transaction being aborted with no significant change to the data files; crashes after that point will result in completing the transaction.

If the crash occurs before setting the state to COMMITTED, we may have left data files claiming to be involved in the transaction. The next time a client tries to OPEN one of those files, the crash recovery procedure will discover that the transaction was aborted since the transaction number will be different, the data file will not be listed in the intentions change list, or the intentions state will not be COMMITTED.

If the crash occurs after setting the state to COMMITTED but before setting it to COMPLETED, the crash recovery code will take over the next time a client tries to access a data file that still claims to be in this transaction or when this intentions file is next used. In either case, data file headers that were not previously written will be updated at this time from the copies saved in the intentions file.

If the crash occurs after setting the state to COMPLETED, all of the important work has been completed -- the intentions lock will time out automatically and there will be no need for crash recovery.

The proof is completed by noticing that a crash while attempting to do crash recovery has the same effect as a one in END TRANSACTION after setting the intentions state to COMMITTED. The necessary information is still around in the intentions file for someone else to use later to complete the transaction.

6. Some Loose Ends

The discussion up to this point has left out some details in the interest of simplicity. In particular, we have ignored creating and deleting files, and we have not shown how to deal with data files whose header information is too large to fit in a single sector.

Creating and Deleting Files

Files are created by a special call to OPEN. If the creating transaction is aborted, it would be nice if the file would automatically disappear. This can be made to happen by running a client "garbage collector" process which looks for transaction data files whose header indicates that their creation has been interrupted. OPEN sets such a flag in the header when it creates the file, and END TRANSACTION resets it.

DELETE is another action available to clients during a transaction. When this routine is called, it makes an entry in the transaction change list telling END TRANSACTION to delete the file. This information is written to the intentions file as part of the change list, and the delete is done only after setting the intentions state to COMMITTED. This ensures that the file will not be deleted if the transaction is aborted. The crash recovery procedure checks to make sure the file has really been deleted. (If file identifiers were reused, a slightly more complex scheme would be necessary to avoid incorrectly deleting a new file -- the file to be deleted would have its header marked saying that it is involved in this transaction in the same way modified files are marked. Crash recovery would not redo a delete unless the file still claimed to be in this transaction.)

Multisector Headers

For large data files, a single sector is not big enough to hold all of the header information. It would be too wasteful to preallocate the maximum number of header sectors that might ever be needed, so the header information is structured to allow expansion.

The first real sector of a data file holds the initial part of the sector map and free-list. When necessary, it also contains a pointer to a tree of extension sectors. The first sector alone is adequate to hold header data for files of up to 220 logical sectors of 512 bytes each. As a file grows beyond that size, header extensions are added up to a maximum of 145 extra sectors for the biggest possible data file which can contain over 32,000 logical sectors. (The maximum size is set by the use of 16 bit real sector numbers and the potential need for two real sectors for each logical one during a complete rewrite of the file.)

In END TRANSACTION, modified extra header sectors are written to unused file locations and the pointers to them are updated. It is still the case that only the header information for the first real sector needs to be saved in the intentions file since it contains the root pointer to the extensions.

7. Discussion

We have described a client-based approach for carrying out consistent, atomic transactions in an environment with multiple clients and servers. The method makes relatively simple demands on the servers, so it should be possible to use it with many existing file systems. Moreover, since servers are essentially passive in this scheme, treating transactions that involve multiple servers is no more difficult than the single server case -- file identifiers include location information and clients simply send their messages to the appropriate server.

The system as described above can be extended in various ways. Two important possibilities concern allowing multiple readers and providing recovery from media failures. With a single type of lock, read-only access to a data file excludes other readers. However, if the file server has separate read and write locks, it is straightforward to modify the transaction machinery to take advantage of them to allow multiple simultaneous readers. No major changes are necessary since the transaction procedures do not modify even the header of read-only files -- such files are simply locked by OPEN, read by READ, and unlocked by END TRANSACTION.

Providing for recovery after a media failure such as a server disk crash can be handled by keeping a *change log*. It is assumed that the file server can be restored to a recent previous state by means of some kind of dump mechanism. The change log for the server holds a sequential history of writes, file creations, and file deletions since the last dump. The log could be kept on tape by the server itself, on another server, or on a *log server*. After a file server failure that has resulted in loss of stored data, the log would be used to redo all of the recorded actions. The transaction machinery would log all of its write, create, and delete actions, so recovery after media failures would become no different than recovery after a server crash not involving loss of data. Note that the change log does not need to hold any information regarding the beginning or end of transactions or even the identity of the transaction for which an action is performed; this is because the normal transaction crash recovery will take care of cases in which a transaction was in progress when the server crash occurred. Thus, other clients who are not using the transaction machinery can still make use of the change log facility to protect themselves against media failures.

Acknowledgments

The paper by Butler Lampson and Howard Sturgis [3] provided the initial inspiration for this work. Conversations with them and with Peter Deutsch were also valuable sources of ideas. Tom Boynton implemented a version of the system and helped to debug my thinking about it. Jay Israel and the referees made valuable comments on an earlier version of this paper. Jim Gray pointed out some oversimplifications in the correctness proof, directed my attention to the problem of media failures, and tactfully made me more aware of the roots of this work in data base operating systems.

References

1. R. M. Metcalfe and D. R. Boggs. *Ethernet: distributed packet switching for local computer networks*, CACM 19 (July 1976) pp. 395-404.

2. J. N. Gray. "Notes on data base operating systems," in *Operating Systems, An Advanced Course*, American Elsevier, 1978.

3. Butler Lampson and Howard Sturgis. *Crash Recovery in A Distributed Data Storage System.* unpuplished paper, Xerox Palo Alto Research Center, 1977; revised version to appear in CACM.

4. Jay E. Israel, James G. Mitchell, and Howard E. Sturgis. *Separating Data From Function in a Distributed File System.* in the Proc. of Second International Colloq. on Operating Systems, IRIA, October 1978.

5. D. C. Swinehart, G. A. McDaniel, and D. R. Boggs. *WFS: A Simple Centralized File System for a Distributed Environment*, unpuplished paper, Xerox Palo Alto Research Center, 1979.

6. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. *The Notions of Consistency and Predicate Locks in a Database System.* CACM, Nov 1976, Vol 19, Num 11, pp. 624-633.