The Cambridge File Server

Jeremy Dion

Computer Laboratory
University of Cambridge

1 Introduction

In a local area network such as the Cambridge ring [Wilkes79a], one of the principal benefits to be gained is the centralisation of expensive resources such as discs. Rather than each processor having a private disc, one or more computers can provide a storage service for all others on the network. The Cambridge File Server, a program controlling a dedicated minicomputer and 150 megabytes of disc storage, is an attempt to create a general storage service for the ring, and has been used to implement both filing systems and virtual memory systems in computers on the ring. This has been done for the CAP computer [Wilkes79b] and is described in a companion article [Dellar80].

The interface which a file server presents to its client machines might be placed nearly anywhere in the spectrum of functions provided by conventional filing systems. At one end lies the remote filing system which manages file directories for its clients, and transfers complete files identified by their character string names. At the other lies the remote disc which reads and writes disc blocks identified by their numerical addresses. In a network where the hosts are not all of the same type and do not provide the same operating system to their users, neither of the above approaches seems applicable. The remote filing system is too rigid if it does not allow a number of different filing systems to coexist on the same storage medium, and it does not provide a suitable interface for building virtual memory systems. The remote disc approach, on the other hand, does not impose sufficient control in an environment where its clients are not necessarily trusted filing system programs.

The interface chosen for the Cambridge file server is intermediate between these extremes [Birrell79]. It represents an attempt to hide the physical characteristics of disc storage such as block sizes and timing constraints without imposing excessive restrictions on the use of this storage. Its principal characteristics are:

- high speed transfers to random access word-addressed files.
- the ability to perform atomic updates to files.
- a capability-like access control mechanism.
- automatic reclamation of unused storage.
- a high degree of crash resistance.

2 The File Server Interface

The objects stored in the file server are of two types, files and indices. Each object is identified by a unique identifier (UID), which is chosen by the file server from a single large name space when the object is created.

A file is a random access sequence of 16-bit words whose contents can be read or written by client machines using the following operations:

- read (fileUID, offset, length): after a short delay, a number of words beginning at the selected word of the file are transmitted to the caller as quickly as possible.
- write (fileUID, offset, length): after reservation of the resources needed to receive at maximum ring speed, an acknowledgement is sent to the client. The client is then expected to transmit the data as quickly as possible to the file server.

An index is a list of unique identifiers, and is analogous to a C-list in capability machines [Dennis66]. An index can contain any UID, including its own. There are three operations provided on indices:

- preserve (indexUID, entry, UID): places UID, if valid, in the index named indexUID at the selected entry.
- retrieve (indexUID, entry): returns a unique identifier stored in an index to the caller.
- delete (indexUID, entry): deletes a unique identifier from an index.

The storage controlled by the file server thus appears to its clients as a directed graph whose nodes are files and indices. Each file or index operation is authorised by quoting the object's unique identifier to the file server, and UIDs are 64 bits long with 32 random bits. Each client, therefore, can access only some of the nodes in the graph at any time, namely those whose UIDs he knows, and those whose UIDs can be retrieved from accessible indices. Since UIDs are difficult to guess, the necessity of quoting one on every operation provides a degree of protection. Accidentally modified UIDs are unlikely to be valid, but on the other hand, only the time involved might prevent a malicious user from guessing an identifier by exhausting all 64-bit combinations.

It will be noted that there are no explicit operations for deleting objects. Instead, only those objects whose UIDs can be found by successive retrieve operations beginning at a distinguished root index are kept in existence. The fact that an object is inaccessible is usually detected by a count of index references for it falling to zero, but since cyclic structures are allowed, periodic garbage collections are also necessary. An asynchronous

garbage collector designed to be run on a different machine is described in a companion article [Garnett80].

The absence of an explicit file or index deletion operation removes from clients the burden of deciding when to delete an object. A program receiving a UID from some source need only preserve that identifier in an index which is known to be reachable from the root to guarantee that the object will be kept by the file server. When it has finished with the object the index entry can be deleted without any decisions as to whether the object itself should be deleted or kept.

Files and indices are created by "create file" and "create index" operations. Both functions take as arguments the UID of an existing index and an offset in it. After creating the object to the user's specifications, the file server preserves its UID in the selected index entry before returning the new UID to the caller. This ensures that the object is reachable from some index and prevents its premature deletion.

When a new client wishes to store information on the file server, an index is created and preserved in the root index. The UID of this index is then given to the client and embedded in his programs. It is the only unique identifier he need ever remember. Thereafter, using this index as the root of his storage system, he may create a subgraph to represent his information, and may exchange identifiers with other clients.

3 Reliability Issues

The state of the storage system controlled by the file server is defined by the contents of the disc blocks which it manages. These are divided into data blocks, which hold the contents of objects, and map blocks, which define their structure. Each file and index is stored as a tree of one, two or three levels. The leaves of the tree are data blocks, and all others are map blocks containing arrays of disc addresses. Every object is initially created as a single data block pointed at directly by its UID. When an update extends beyond the length of this block, the depth of the tree is automatically increased to two levels, and may increase to three for very large objects.

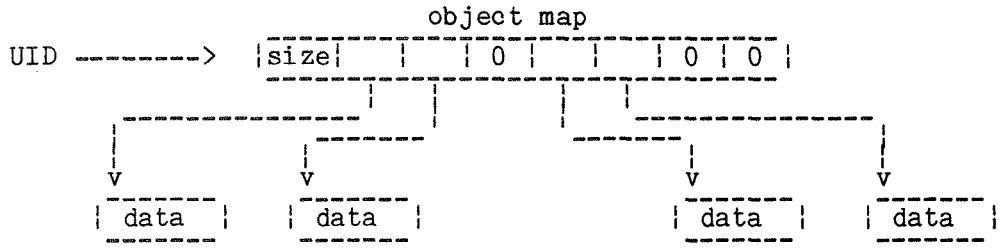

Fig 1. Structure of a two-level object

A file server transaction changes the state of the storage system by performing a sequence of disc writes. Some operations, such as reading from a file, cause no change whatsoever, but writing to a file may cause hundreds of disc blocks to be written. If the operation causes the allocation or deallocation of data blocks, then map blocks will also need to be updated. So in general, changing from one consistent state to another involves a sequence of disc writes to both map and data blocks.

In the normal course of events, the sequence will complete once started. However, it can be interrupted by a software or hardware failure in either the client or the file server. In the worst case, some of the disc writes will have been performed, some will not have been attempted, and one block will have been corrupted.

The ideal behaviour of the file server under these circumstances would be to undo the effect of all the disc writes so that the storage system reverts to its state before the start of the transaction. This would have to be done both when the file server detected a failure in the client, and also on each restart to undo operations which had failed due to a crash by the server. Thus, the fact that an operation is in progress but has not yet completed must be recorded in stable storage, and maintaining this information requires extra disc transfers.

Such a scheme would provide atomic transactions on indices and files, in that either all updates in the transaction would be performed or none would, even if a crash occurred in the client or the file server during the transaction. The benefit obtained from this mechanism must be weighed against the overhead involved in recording the additional information. For some files, such as those which hold structures defining a filing system for some client, such a facility is valuable. It would mean that the client need not take particular precautions against a file directory update being interrupted by a crash, and therefore need not make excessive checks of his filing system on each restart. For other files, however, a mechanism to perform updates atomically is undesirable if it introduces any overhead. The output from a

compilation, for instance, is so easy to reproduce that no special care need be taken in updating it.

For these reasons, the actual file server implementation is not quite so thorough about restoring the consistency of the storage system after a crash or a detected error. All map blocks are restored to their last consistent states so that the structure of the storage system is consistent in terms of block allocations, but the data in it are restored only as directed by the client.

Every file is defined by the client as normal or special at the time of creation, according to whether or not the file will be used to hold essential information. Normal files have no overhead for atomic updates, and when a data block of a normal file is to be updated, the new copy overwrites the old one. If the write fails or a sequence of writes is interrupted, the original information is lost. Special files, however, are updated using an intentions mechanism [Sturgis74] which allows any number of block allocations and deallocations to be performed indivisibly. In addition to the conventional states allocated and deallocated, the block allocation tables in the file server record two additional states intending to allocate and intending to deallocate. The course of an update to a data block in a special file is then as follows:

1) choose a deallocated block and mark it intending to allocate.
2) change the state of the old block from allocated to intending to deallocate.
3) write to the new block.

This sequence is repeated for all blocks to be updated in the operation, thus creating a number of intending to allocate / intending to deallocate block pairs. At any time, the transitions performed to date may be reversed, thus restoring the object to its original state.

When all blocks of the object involved in the transaction have been modified in this way, the operation is made irreversible by setting a commit bit associated with the object. From this point onwards, the file server guarantees that the updates to the object will eventually be performed. Finally, it is necessary to remove all intentions:

1) change all intending to allocate blocks to state allocated.
2) change all intending to deallocate blocks to state deallocated.
3) after the last intention has been removed, reset the object's commit bit.

The object is now ready to be updated again.

When restarting after a crash, it is essential to be able to correct all unfinished operations. The algorithm to do this is straightforward. The

block allocation tables in which all intentions are recorded are scanned to find the allocation state of each block. If it is in one of the intention states, then whether to perform or reverse the intention is decided by the object's commit bit, as shown in the state transition diagram below.

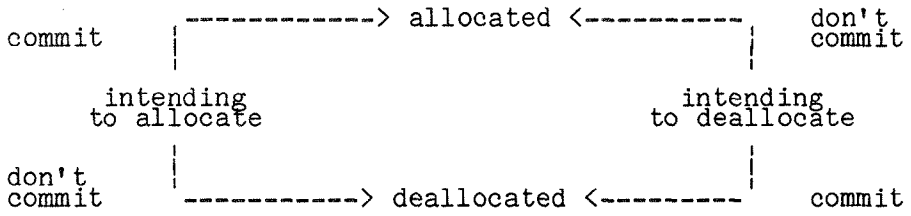

Fig 2. State transitions for blocks in intention states

There may, of course, be crashes during attempts to restart, but when the algorithm ultimately finds no intentions, the storage system is in a consistent allocation state. The net effect is that for those operations which had not yet written the commit bit, all block allocations have been reversed, and for the operations which had committed but had not finished the clearing up of intentions, all intentions have been performed.

An implicit assumption in the above argument is that allocating or deallocating a block, which involves changing its state in a block allocation table and writing a new disc address in an object map, is atomic. Unfortunately, a crash between writing an allocation table and writing an object map will produce an inconsistent state. Worse, a disc write can fail on a map block leaving it unreadable.

To make recovery from these errors possible, the file server uses two kinds of map blocks. Object maps, already mentioned, constitute the non-leaf nodes of the trees which define objects, and are simply arrays of disc addresses. Cylinder maps, so named because there is one on each cylinder of each disc, are the block allocation tables which define the current use of blocks for one cylinder. A cylinder map is an array indexed by sector number, and each entry contains the allocation state (allocated, deallocated, intending to allocate, intending to deallocate) for a block. In addition, unless the block is deallocated, the cylinder map entry contains the UID of the object to which the block belongs and defines the position of the block in the tree. The cylinder map entry for the root block of an object tree also holds the commit bit for the object.

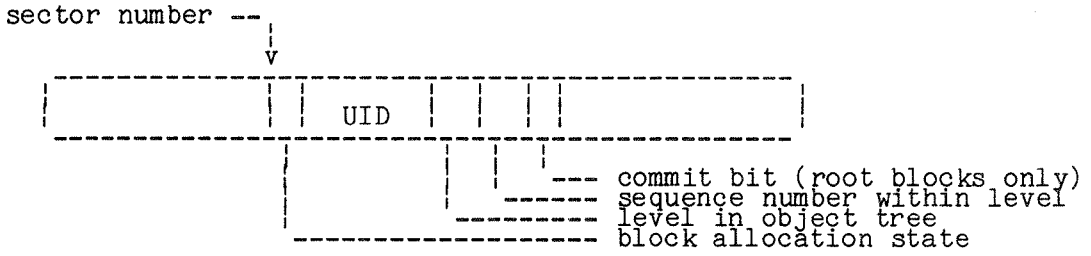

Fig 3. Cylinder Map Format

The value of this redundancy lies in the fact that each map block is now reconstructible. If an object map block is destroyed, the cylinder maps can be examined to find the blocks which were its children in the object tree, and so the disc addresses which the map contained can be reconstructed. If a cylinder map is destroyed, then by traversing the directed graph beginning at the root index (examining only object map and data blocks for indices), all useful objects can be found, and by enumerating each object tree, the cylinder map entry for every block can be reconstructed. At the end of this process, the cylinder map entries not visited can be marked deallocated. The structure of the data on disc is thus recorded twice, once in the hierarchy of object maps, and once in the list of cylinder map entries. Because of this, both cylinder maps and object maps can safely be updated in place.

The complete algorithm for performing an atomic update is now as follows:

1) As above, create block pairs with intentions by marking entries in cylinder maps held in core. The block allocation policy tends to cluster the blocks of an object by initially allocating all blocks near the root block, and then by trying to replace existing blocks with new blocks on the same cylinder. Buffered copies of the object maps are also updated with the new disc addresses.
2) Write all modified cylinder maps to disc.
3) Write all modified object maps to disc.
4) Set the commit bit, held in the cylinder map entry for the root of the object tree. Write the cylinder map to disc.
5) Inform the caller that the operation has been done.
6) Perform all intentions by modifying the cylinder map entries written in step 3 and write them again.
7) Reset the commit bit, and write the cylinder map to disc.

The operation is made irreversible at step 4. Any failure before step 4 causes the intentions to be reversed on restart, and any failure afterwards causes them to be performed. Because of this, it is safe to reply to the caller immediately after the commit bit has been written to disc.

The overhead involved in this sequence depends on the number of cylinder maps on which intentions are created, and the number of object map blocks changed to point to new data blocks. A large three-level object, for instance, can have many object map blocks and its data blocks on several cylinders. Where n cylinder map and m object map blocks are changed, 2n+m+2 extra disc transfers are needed compared with writing in place to data blocks, and n+m+1 of these transfers are synchronous with the client.

Normally, where the disc is not fully loaded and the transaction involves a few data blocks, all intentions are created on the cylinder which holds the root of the object tree. In this case, an optimisation can be made. Since there is no need to ensure that updates on several cylinders are made indivisibly, the three changes to the same cylinder map in steps 4, 6 and 7 can be made in a single transfer at step 4. With this optimisation, the usual overhead for an atomic transaction is three disc transfers: one to write the cylinder map with intentions, one to write the object map containing the new disc addresses, and one to perform all intentions in the cylinder map.

The restart algorithm must of course ensure that after all intentions have been reversed or performed, the cylinder maps and object maps agree on the state of every disc block. A crash between steps 3 and 4, for instance, will have left the object tree "ahead" of the cylinder maps. In consequence, when the cylinder maps are searched for intentions, if all the cylinder maps are found to be readable, their contents take precedence over the object maps. Whenever an intention is found on a block, the corresponding object tree is forced into agreement. If a cylinder map is found unreadable, however, then it must have been corrupted in one of steps 2, 4, or 7. In each case, the object maps are correct and can be used to rebuild the lost cylinder map.

In the current implementation, it takes less than thirty seconds for the restart algorithm to scan the cylinder maps for 150 megabytes of storage. Rebuilding a cylinder map takes about thirty minutes, but has only once been necessary in a year of operation.

4 Performance Issues

A file server intended for use as a backing store device in a virtual memory system must not have noticeably worse performance than that of a local disc. In spite of the extra expense of atomic transactions, a number of factors combine to make the file server relatively efficient.

The intentions mechanism comes into play only when blocks are allocated and deallocated. About half of all operations performed read information from the

file server and thus cause no extra work on this account. Of the remainder, most update operations take place on normal objects which do not use the intentions mechanism once blocks have been allocated to them. Only the indices and a relatively small number of files are special. Thus the majority of file server operations generate no disc traffic due to intentions.

An exception to this rule is the creation of objects. As mentioned above, when an object is created, its UID is preserved in an index. This preservation (like all index updates) is done atomically because losing the contents of an index might cause a number of objects to become unreachable from the root index and thus eligible for deletion. The work involved in this preservation in fact exceeds that needed to create the object. In addition, in practice it is found that the overwriting of the index entry usually causes the implicit deletion of the object whose UID was preserved there because most objects have only one index reference to them. Deletion, which occurs by the mechanism described above, is also relatively expensive, although it is performed asynchronously.

The lightweight protocol used on the Cambridge ring for communication with the file server is perhaps the largest single contributor to its efficiency. All file server transactions other than read file and write file consist of the client sending a single request packet of perhaps 20 16-bit words, and receiving a reply packet of about the same size. Read file and write file involve additional transmissions of a potentially long sequence of data packets, each up to 1K words long. No initial connection is needed to establish contact with the file server beyond sending the request packet, and the protocol provides no handshakes for error control or flow control. All data transfers are defined to take place as fast as the transmitter can send to avoid explicit flow control, and if the underlying packet protocol detects an error in a received packet, no error control facilities are provided beyond retrying the entire operation. This protocol has been found to permit transfers at about 80% of the maximum point-to-point ring bandwidth, with infrequent retries. Under light loading conditions, typical access times for 256 words in a file are 50ms to read and 65ms to write, including communication overheads in both client and server.

Since the file server maintains no state between transactions, some other mechanism must be used to take advantage of the locality of repeated operations on the same object. For this reason the file server maintains a cache of disc blocks on a least recently used basis. This cache of about fifty blocks is organised to favour the retention of cylinder maps and object maps over data blocks. Since cylinder maps are the equivalent of allocation bit maps in other systems, keeping them in the cache is particularly beneficial.

The method for resolving a unique identifier also contributes to quick access to objects. Each identifier consists of 32 bits of random data and the disc address of the root of the object tree. Validation of a unique identifier requires reading the disc block and its cylinder map. The cylinder map is then checked to ensure that the block is a root for an object tree with the same unique identifier, and the block is checked to ensure that the operation accesses words within the object limits. If these checks succeed, as is nearly always the case, the object is then directly accessible through its root block. For small objects whose contents are held directly in the root block, no further disc accesses need be made. The penalty paid for this form of unique identifier, as opposed to one resolved by indirection through a large table, is that the root of the object can never be moved, because its address is contained in the identifier. In practice, this has not been found to be a problem.

The file server is currently used for several purposes. Two different operating systems use it as their only storage device. Microprocessors such as the name lookup server use it to archive important data. Its relatively quick response has allowed it to be used to record and play back digitised speech in real time. The interface provided seems both simple and suitable for a variety of purposes.

References

[Birrell79] A.D. Birrell and R.M. Needham. "A Universal File Server". Accepted for publication in IEEE Transactions on Computing.

[Dellar80] C.N.R. Dellar. "Removing Backing Store Administration from the CAP Operating System". Operating Systems Review, this issue.

[Dennis66] J.B. Dennis and E.C. Van Horne. "Programming semantics for multiprogrammed computers". CACM 9 3, March 1966.

[Garnett80] N.H. Garnett and R.M. Needham. "An Asynchronous Garbage Collector for the Cambridge File Server". Operating Systems Review, this issue.

[Sturgis74] H.E. Sturgis. "A Postmortem for a Time-Sharing System". Xerox Palo Alto Research Center technical report CSL74-1, January 1974.

[Wilkes79a] M.V. Wilkes and D.J. Wheeler. "The Cambridge Digital Communications Ring". Proc. Local Area Communications Network Symposium, Boston, May 1979. U.S. National Bureau of Standards Special Publication.

[Wilkes79b] M.V. Wilkes and R.M Needham. "The Cambridge CAP computer and its operating system". Operating and Programming Systems Series, Elsevier North Holland, 1979.