Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

S.J. Mullender, A.S. Tanenbaum

A distributed file service based on optimistic concurrency control

Department of Computer Science / Algoritmics & Architecture Report CS-R8507 April

A Distributed File Service Based on Optimistic Concurrency Control

Sape J. Mullender

Centre for Mathematics & Computer Science
Amsterdam

Andrew S. Tanenbaum

Vrije Universiteit
Amsterdam

Principles are presented for a distributed file and database system that leaves a large degree of freedom to the users of the system. It can be used as an efficient storage medium for files, but also as a basis for a distributed data base system. An optimistic concurrency control mechanism, based on the simultaneous existence of several versions of a file or data base is used. Each version provides to the client that owns it, a consistent view of the contents of the file at the time of the version's creation. We show how this mechanism works, how it can be implemented and how serialisability of concurrent access is enforced. A garbage collector that runs independent of, and in parallel with, the operation of the system is also presented.

1980 Mathematics Subject Classification: 68A05, 68B20, 68H05 69D53, 69H22, 69H24, 69H32
CR Categories: D.4.3, H.2.2, H.2.4, H.3.2.
Keywords & Phrases: file server, data base server, distributed control, optimistic concurrency control, atomic update, serialisability, differential files.

1. Introduction

File systems play an important role in allowing information to be widely accessible, since most information is in some way or another stored on files. There are many different kinds of file systems for distributed systems, ranging from private file systems for each host to special purpose file servers for the whole network. Each kind of file system has its own characteristics concerning accessibility, complexity, protection of information against unauthorised access, speed and distributiveness.

The ideal distributed file system would be fast, files would always be near the hosts needing them, there would be protection, if necessary, to guard against unauthorised hosts or users, files could be shared among different hosts at the same time, and the system would be totally immune agains individual file server crashes or disk crashes. Unfortunately, such distributed file systems do not yet exist. Improving one aspect of a file system is nearly

Report CS-R8507 (Algorithmics & Architecture)
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

always detrimental to another. The consequence, for instance, of replicating files at several sites to improve their availability is that updating these files will become much more costly, since all copies have to be updated, and if, additionally, the changes made by different users must be synchronised, such that the changes made by one user do not interfere with the data read by another, then the cost of file operations will be increased by several orders of magnitude.

This paper goes into the design of the distributed file service for the ***Amoeba*** Distributed Operating System [Mullender85a]. We have attempted to build a file service, suitable for many different applications: ordinary 'plain' files, hierarchically structured files, replicated files, databases, source code control systems [Rochkind75], etc.

2. Design Considerations

Important in the design was the Bauer principle, governing the whole of the design of Amoeba, 'You should not have to pay for those features you do not need.' A file server, for instance, that implements atomic update on replicated files is a very nice thing to have, but a user who wants to store the output of a compiler, prior to calling a linking loader doesn't share that output with any other user; he is not interested in having his file replicated across five different network nodes for increased availability, nor is he interested in having his file atomicly updated. All the user wants is a temporary file that can be quickly accessed and changed, and just reliable enough that usually he doesn't need to compile his program all over because the file was lost. On the one hand, our file server should cater for the simple-minded user who just wants a reasonably reliable repository for his files, cheap and fast, while on the other hand, the sophisticated user should be taken into account who needs ultra-reliable storage for his files, fancy synchronisation of access by many simultaneous users, and guaranteed availability, who is prepared that it will be expensive and slow.

Another important issue in the design of a file server is that the file server be easy to understand. The interface to the file server must not only be simple, with as few commands as possible, clients must also have a simple conception of the structure of a file, and how to use it. Even if clients want highly sophisticated things done, like changing a heavily shared file atomically, they should not be burdened with the details of a five step locking protocol, or have to know just how often the file is replicated.

It is a design goal that the distributed file server should be suitable for an Amoeba environment, using the protection provided by Amoeba's ports and capabilities [Mullender85b]. We want a free-standing file server, providing disk space for the users of hosts with no, or not enough disk storage of their own.

2.1. File Servers in Open Operating Systems

In an open system, several different services may offer the same facilities, albeit in different forms. There can be several file servers, one offering ordinary linear files, another tree structured files with concurrency control mechanisms to arbitrate updates by a number of simultaneous users. The choice of which file server to use is up to the user.

The advantages of open systems over the traditional approach are obvious: operating system kernels become smaller and more maintainable, operating system services are no longer in the kernel, making them portable, and allowing multiple, equivalent, but different services to co-exist side by side.

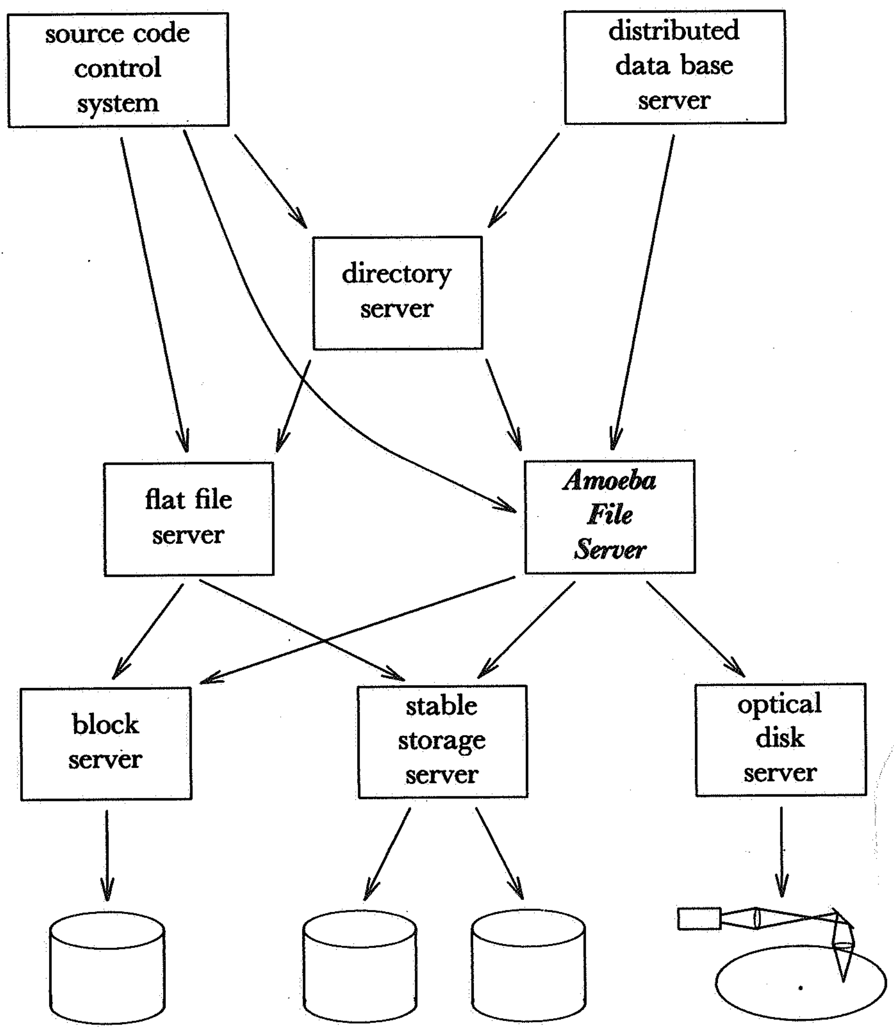

FIGURE 1. An example of a storage services hierarchy in an open system.

Data base management systems often have their own operating systems, tailored to this particular application, because traditional operating systems provided the wrong functionality [Stonebraker81, Tanenbaum82]. An open operating system, with the right kind of file service, can support data base management efficiently, while integration with other system services is possible. A hierarchy of services, as illustrated by FIGURE 1, allows a logical layering of facilities while the development effort can be shared.

The bottom of the hierarchy is formed by the block server, which manages blocks of data of fixed size. At the next level, file services manage files – structured collections of data – and implement operations for inspecting and changing them. These operations must support the next level, where data, stored in files, is interpreted: the contents of a file may represent the state of an airline reservation system, or the contents of the bank accounts of a branch office, or a pascal program.

File services must provide the tools for the efficient implementation of as wide a set of applications as is possible. This can be realised, in part, by providing a large set of different file services, each tailored for a particular application, but, naturally, it is best to have as few as possible different file services that cover the needs of every conceivable application.

3. Related Work

Since the beginning of distributed computing, many file servers have been built. In this section we shall look at some that are closely related to our work: XDFS [Sturgis80] Felix[Fridrich81] and Swallow[Reed81]. They all have mechanisms for concurrency control. Most file servers, including the Cambridge File Server [Dion80], XDFS and Felix use *locking* [Eswaran76], while some, among them Swallow, use *timestamps* [Reed78].

XDFS is a distributed file server that uses the notion of *transactions*. *Open transaction* and *close transaction* commands bracket a series of read write commands to one or more files, and the system guarantees the *atomic property* for these transactions; that is, either all of the changes will be done, and the transaction succeeds, or none, and the transaction fails. XDFS realises the atomic property via so-called *intentions lists*, a list of changes to the file.

XDFS uses an interesting locking mechanism to guarantee serialisability: there are three kinds of locks, read locks, intention-write locks, and commit locks. When a server has locked a datum for some time, a timer expires and the lock becomes *vulnerable*. Another server, waiting on that lock, can then prod the first, requesting it to release its lock. If it is in a state to do so, it releases its lock, otherwise it ignores the prod.

The Felix file server also uses locking, although here it is at the file level. The Felix locking mechanism is combined with a *version* mechanism: when a file is examined or modified, a new version of the file is created. The version can be thought of as a copy of the file at the time of its creation, although the file is not actually copied block for block then. Sharing is supported by six access modes. Files are tree-structured. When a new version or a virtual copy is created, the whole tree is initially shared with the most recent version. When it is modified, a copy-on-write mechanism is used, leaving the original tree intact.

Like Felix, Swallow also uses a version mechanism, but the synchronisation of concurrent access is quite different. Swallow uses a timestamp mechanism, based on Reed's notion of *pseudo time*. This mechanism is used to ensure the atomic property of updates to collections of arbitrary objects (*e.g.*, files).

3.1. Advantages over Previous File Systems

The Amoeba File Server is a file server, with a version mechanism, similar to that of Felix, but in contrast to other file servers, it uses a combination of locking [Eswaran76] with an optimistic concurrency control mechanism [Kung81, Robinson82, Schlageter81]. Optimistic concurrency control mechanisms have been used in data base management systems, but we have never seen them used in a file server. Yet, an optimistic concurrency control mechanism, combined with a version mechanism provide a number of advantages, not present in other file systems.

The most important characteristic of an optimistic approach, is that the file system is always in a consistent state. Most file systems, using other mechanisms for concurrency control, need a mechanism for bringing back the file system to a consistent state after a crash. A client crash can cause parts of the file system to be inaccessible for some time, for instance, because a rollback operation must be done first to bring the file system back to a consistent state. This is no problem with the Amoeba File Service. The file system is always in a consistent state (assuming the updates themselves are consistent). Server crashes have no serious consequences: the file system is always in a consistent state, so there is no rollback, clients need only redo the update that remained unfinished because of the crash. Clients do not have to wait until the server is restored, because they can use another server to do it.

In a way, optimistic concurrency control and locking are complementary mechanisms: Optimistic concurrency control maximises concurrency and works best when updates are small and the likelyhood that an item is the subject of two simultaneous updates is small. Locking, in contrast, does not allow as much concurrency, and is more suitable when updates are large and unwieldy and when the probability of an item being subject to more than one update is significant. The Amoeba File Service combines locking and optimistic concurrency control in such a way that updates of large bodies of data (several files) use locking to prevent having to redo them if they clash with another update. Updates of small bodies of data (one file) are less likely to clash with other updates, so an optimistic approach is used here. When necessary, a *soft-locking* scheme can be used in addition to optimistic concurrency control to ward off potential conflicting updates. In all cases, the mechanisms for carrying out updates guarantee consistency of the file system at all times.

The Amoeba File Service provides the necessary mechanisms to maintain caches of data. Both Amoeba File Servers and their clients can hold data in a cache. In many file systems, it is difficult or impossible to maintain caches, because the integrity of the data in the cache cannot be assured. XDFS uses 'unsolicited messages' to tell clients to unlock cached data when it is going to be modified. This makes their caching strategy efficient only for data that is rarely modified. The integrity of the cache is checked at the start of a transaction. The cost of checking whether the cache is up-to-date is small, even for files that are frequently modified. The Amoeba File Service needs no unexpected 'unsolicited messages.'

4. The Block Server

The principle of separating the issues of file service an block service makes it easy to combine different methods of storage (*e.g.*, stable storage [Lampson79]), and storage media (*e.g.*, small fast 'electronic disks,' large slow magnetic disks, very large optical disks) in one system. Carefully designed, disk service can combine high speed with high reliability, using techniques, such as caching and dual storage, both on fast, but not so reliable storage, and slow, but very reliable storage.

We assume the block service implements as a minimum commands to allocate, deallocate, read and write fixed size blocks of data. Protection must be provided, so that a block, allocated by user A cannot be accessed by user B without A's permission. Writing a block must be an atomic action, with an acknowledgement that is returned *after* the block has been stored on disk. This property is vital for the implementation of atomic update on files.

The block server can implement a simple locking facility. Based on this, file services can realise concurrency control policies. The Amoeba File Service, for commit on a version of a file, for instance, will lock and read a block, examine and modify it, then write and unlock the block again.

We expect that the block server's clients will often use a small portion of each block for redundancy purposes. Block servers can support a *recovery* operation, which given an account number, returns a list of block numbers owned by that account. A client, *e.g.*, a file server, can then use its redundancy information to restore its file system after a severe crash.

Magnetic disks and optical disks do not usually lose their information in a crash, but it does happen occasionally. In any case, they are at least temporarily inaccessible. In order to achieve high availability in the face of disk crashes, it is necessary to store every block at least twice, on different disks, managed by different servers. Lampson and Sturgis [Lampson79] have suggested a method to use dual disk drives to implement *stable storage*. We propose a small modification to their method to make a more reliable version of stable storage.

In our proposed method, each block is stored by two servers on two different disk drives (in contrast to Lampson and Sturgis' method which uses one server and two disk drives). On request to allocate and write a block, the receiving block server, say server A allocates a block on its local disk, then sends a request to its companion block server, server B including the data and the chosen block number. B then writes the block to disk at the address indicated by A, and sends an acknowledgement back to A. Finally A writes the data in its own block, and returns an identifier for the block to the client. Read and write requests can be sent to either block server. For reads, the block server need not consult its companion server, except when the block on its disk is corrupted. For writes, the same message exchange is used as for allocate and write.

Allocate collisions may occur when two clients allocate a block simultaneously, one on server A and one on server B, and, accidently, A and B choose the same block number. Similarly, write collisions may occur when two clients write the same block via different block servers. These collisions are detected, however, before any damage is done, because writes are always carried out on the companion disk first. When a collision is detected the companion server is warned, and appropriate measures can be taken (*e.g.*, redo the operation after a random wait interval).

After a crash, the block server compares notes with its companion, and restores its disk before accepting any requests. To this end, block servers make intentions lists for crashed companion servers. Clients send requests to the alternative block server if the primary fails to respond. Otherwise crashes are dealt with in the same manner as in Lampson and Sturgis' method.

5. Amoeba File Service

The Amoeba File Service was developed for, but is not restricted to, the Amoeba Distributed Operating System [Mullender85a]. It implements the file system as a tree of pages, whose subtrees are files, and uses a combination of an optimistic concurrency control mechanism and a locking mechanism to prevent conflict in simultaneous updates.

For concurrency control, three mechanisms stand out as the most frequently used: locking [Menasce78], timestamps [Reed78], and optimistic [Kung81]. Each method has advantages and drawbacks, and the discussion which method is best will continue for some time. Several file servers have been implemented with a concurrency control mechanism. Most of these, however, use locking as their concurrency control mechanism [Fridrich81, Sturgis80, Dion80], except a few that use timestamps [Reed81]. File servers that use optimistic concurrency control, however, are not known to us, although, as we shall see, optimistic concurrency control has some properties that make it very attractive for application in a file server.

The Amoeba File Service implements optimistic concurrency control by a version mechanism: When a client modifies a file, a new version of the file must be created, which initially behaves like a copy of the file. Then the modifications are made, and finally a *commit* operation makes the modifications permanent by replacing the previous current version with the new one. Several versions of the same file can exist at the same time. The Amoeba File Service checks on commit whether the modifications to the file constitute a serialisability conflict (see [Kung81]).

The current state of a file is contained in the **current version.** **Committed versions** represent past states of a file; **uncommitted versions** represent possible future states of the file. Files are accessed by their **file capability**, versions by their **version capability**. Atomic updates on files are bracketed by creating a version and committing a version. The current state of a file is always represented by the contents of the current version. Committing a version makes that version the current one.

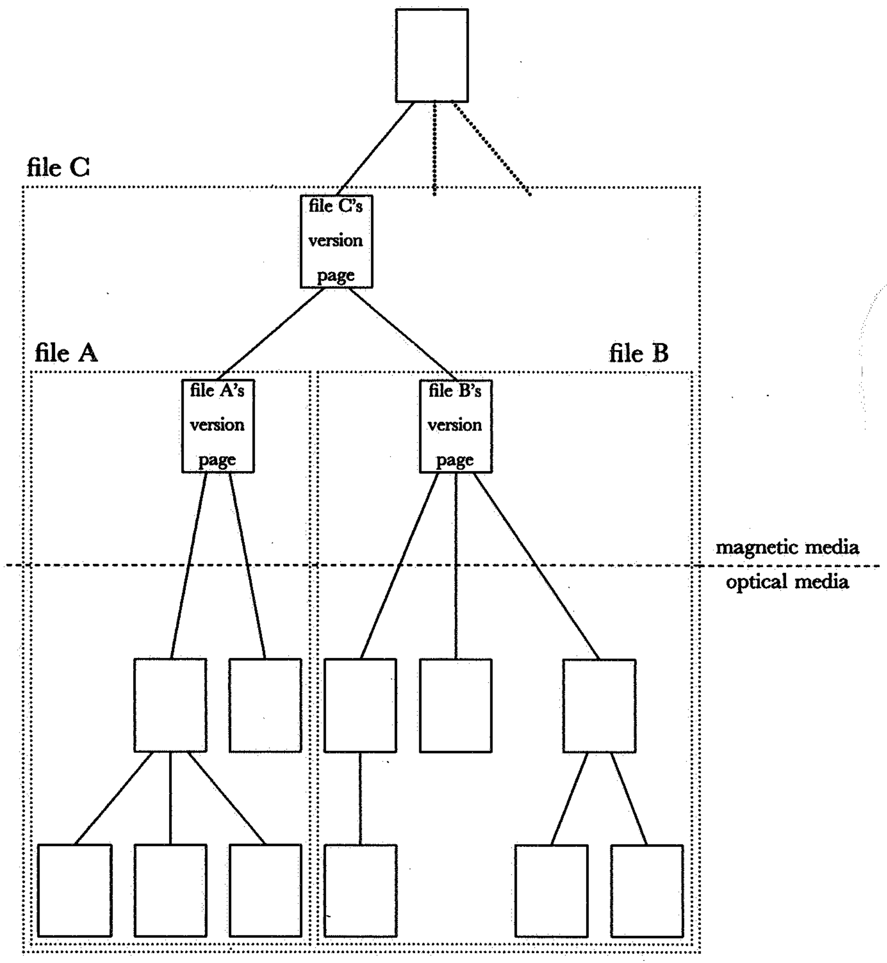

Figure 2. The file system has the structure of a tree. Files also, consist of trees of pages. The file system can be viewed as a tree of trees.

The file system as a whole is represented as a large tree of *pages*. The top of the tree (*i.e.*, near the root) is stored on magnetic random-access media, for instance, such as provided by the *stable-storage* server, described in the previous section. The lower parts of the tree can be stored on magnetic disk, or write-once media, such as optical disk. As illustrated in FIGURE 2, a subtree, whose root is in the upper part of the tree, *e.g.*, *file A*, can be viewed as a file; it can be modified atomically using the methods described below. Amoeba files, unlike files in most file systems, thus form a nested structure: A subtree whose root page is inside another subtree may be viewed as a file within another file. *File A* and *file B*, for instance, are both subfiles of *file C*. For the moment, this hierarchy will be ignored; we shall consider a file system where the upper part of the tree consists of only one page; that is, a file system containing only one file. Later, we shall return to the general situation, where the top part of the page tree forms a 'real' tree.

A version is represented as a tree of pages. Clients can read or write a page at a time. The maximum length of a page is determined by the maximum length of a message in a transaction: 32K bytes. This ensures that pages can be read and written in one (atomic) transaction.* A page may contain both data and references to pages further down in the tree. A reference consists of a block number and some flag bits that Amoeba File Service uses for concurrency control. The number of data bytes in a page is variable (per page) up to the maximum size of a page. The remaining space in a page can be occupied by references to pages in the next level of the page tree.

Clients have explicit control over the shape of the page tree. Pages within a file are referred to by a *pathname* which is constructed as follows: The root page has an empty pathname. The pathname of a page that is not the root, is the concatenation of the pathname of its parent page with the *index* of its reference in the array of references in the parent page.

This file representation has been chosen with the express intent of giving clients (file systems, data base systems, source code control systems, etc.) as much control over the shape of files as possible. Using the file structure provided by the Amoeba File Service, objects ranging from linear files to *B-trees* can easily be represented.

The Amoeba File Service provides a set of commands for the management of files and versions. There are commands to read and write the pages of a version and commands to manipulate the shape of a version's page tree (split pages into two, move subtrees to another part of the tree, etc.).

5.1. File Representation

A file — in this section we should perhaps say '*the* file' — is a collection of versions, ordered in time. When a new version is created, it behaves as if it were a copy of the current version. In fact, when it is created, a new version shares its page tree with the current version, and only when a page is changed is the page duplicated. The Amoeba File Service file representation is therefore a differential file representation, similar to that of FELIX.

Pages are stored by the block server in such a way that they can be read and written as atomic actions. Associated with each page is a small header area that the Amoeba File

* Arbitrarily long pages can be written atomically by writing them back-to-front as a linked list, whereby the head block is (over)written last, and the other blocks in the list are allocated from the pool of free disk blocks. After writing, the blocks making up the previous linked list can be freed.

Service uses for administrative purposes.

The root page of a version tree is referred to as the **version page.** The data in a page has no predefined structure. Clients are free to write them as they see fit. The references in a page are for internal use by the Amoeba File Service and can only be read and written by servers.

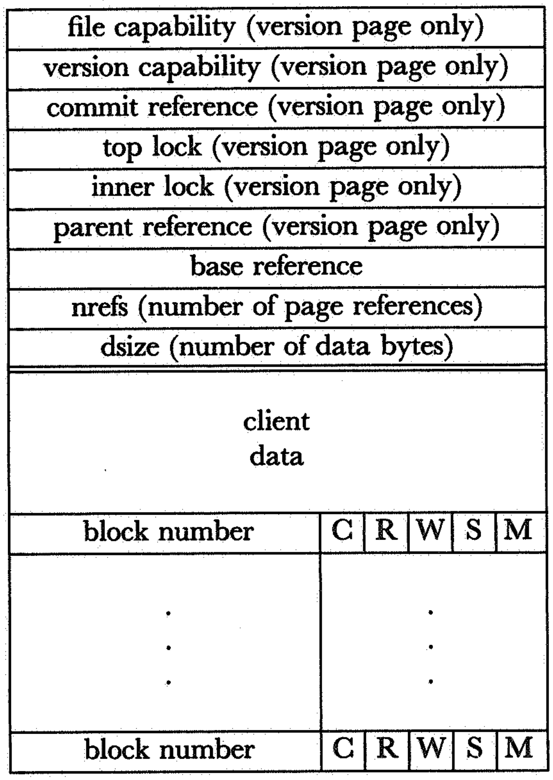
file capability (version page only)
version capability (version page only)
commit reference (version page only)
top lock (version page only)
inner lock (version page only)
parent reference (version page only)
base reference
nrefs (number of page references)
dsize (number of data bytes)
client data
block number
C
R
W
S
M
block number
C
R
W
S
M

FIGURE 3. The Amoeba File Service page layout

The lay out of a page is shown in FIGURE 3. The page is divided in two areas, the header area and the page itself; the separation is indicated by the double line. The first field in the header area is the *file capability*. This field gives the capability of the file whose root the page is. The next field is the *version capability*, the version of the file whose root the page is. The *commit reference* field is only used in version pages; its use will be explained presently. The *top lock* and *inner lock* are used to tell whether a page is currently involved in an update of a file whose root is higher in the page tree. In this section we have assumed there is only one file in the system, so these fields are not used here; their function will be explained in a later section. The *parent reference* gives the name of the parent version block. *Parent references* can be used to ascend the upper part of the page tree to the root. The fields mentioned just now are only present in a version page. They are absent (or ignored) in other pages. The *base reference* field is the block number of the page that this page was based on (copied from). The *nrefs* field holds the number of page references this page contains. The *dsize* field gives the number of data bytes. The page itself contains the *reference table*, with an entry for each child page, and the data area where the client data is kept.

The reference table is an array of *page references*, which contain a *block number*, and five flags, C, R, W, S, and M. The page reference points to a page in the next level of the page tree, the C flag, when set, indicates that the page was copied and is no longer shared with the version it was based on. The R flag indicates whether the data of that page has been read (it is needed to decide if an uncommitted version may be committed as explained in section 5), the W flag indicates whether the data in the page was written (changed), the S flag tells if the references have been used (searched), and the M flag indicates whether the references were modified (*insert page*, *remove page*, *make hole*, *remove hole*). As we shall see, it is not possible to access a page without copying it, nor is it possible to modify the references without looking at them. This reduces the number of flag combinations to 13, which allows encoding the flags in four bits. Amoeba uses 28 bits for a block number and four bits for the flags.

Pages are accessed from their parent page by the *index* in the reference table. An arbitrary page in a version can thus be accessed from the root by indexing into the references of several pages starting at the root (version page) of the page tree. Pages thus have path names consisting of a string of n-bit numbers. These path names are visible to clients, giving them explicit control over the structure of their files.

A file is made up of a sequence of committed versions and possibly a collection of uncommitted versions. The version pages of the committed versions form a doubly linked list. Each committed version's base reference points to the version it was based on (its predecessor) and its commit reference points to the next committed version. The current version's commit reference and the oldest version's base reference are nil.

The uncommitted versions are attached to the list through their base references, which point to the version they were based on; note that this is always a committed version. A typical file could look like the one in FIGURE 4, where we have just shown the version pages and their base and commit references.

In the next section we shall discuss the mechanisms that are used to implement atomic update and guarantee serialisability, but before we go into that subject, a proper understanding of the copy-on-write mechanism and the R, W, S and M flags in the page table is needed.

The R, W, S and M flags are needed primarily for deciding about committing versions. In order to be able to serialise two simultaneous updates to a file, the Amoeba File Service must know which parts of the file were read and which parts were changed (written). When set, the R flag indicates that the data in the referred-to page was read. The W flag indicates its data was written. The two flags operate independent of one another. The S flag tells that the references have been referenced, the M flag tells whether the references have been changed. These flags are not independent. When the M flag is on, the S flag must also be on; it is not possible to modify the references without consulting them.

When a page is read, the pages on the path to it must also be read. This implies that, if a page has not been searched, then the subtree of which it is the root cannot have been searched either. Hence, a cleared S flag indicates that the descendants of the referred to page have not yet been accessed.

For writing pages in a version, a 'copy-on-write' mechanism is used. When a page is written, a new block is allocated for it, leaving the old page intact. Then the page reference in its parent page is updated to point to the newly allocated page and its W flag is set. This changes that page, however, and this change must also be made by allocating a new block for it and writing the new contents of the page to that new block. Every change thus

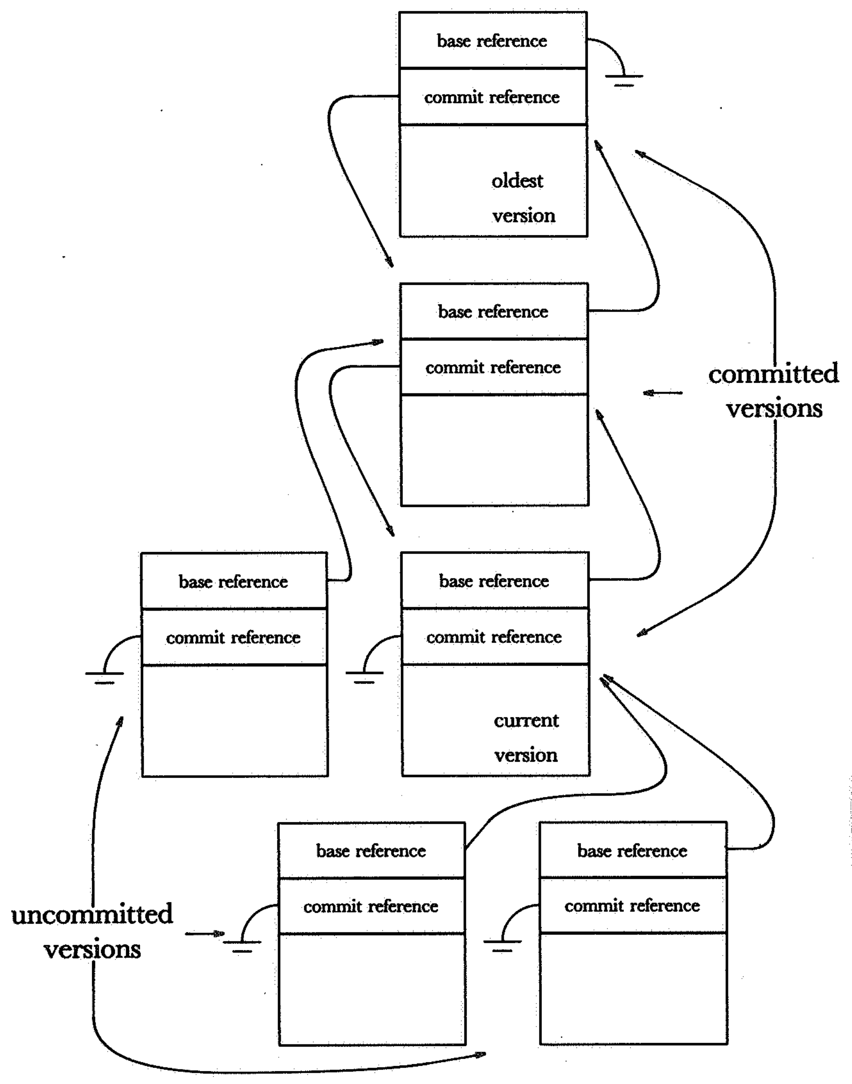

FIGURE 4. The *'family tree'* of a typical file. Only the version pages are shown. The page trees descending from the version pages are not shown.

bubbles up from the leaves of the page tree to the root page. The root page — the version page — is the only page that is written in place. When a page is thus copied, the C flag is set in the reference to it (in the parent page). Naturally, a page is only copied once; after it has been copied for writing, it can be written in place when it is written again.

It is clear now that, when a page has not been copied, its descendants can not have been copied either. Hence, a cleared C flag in a page reference indicates that the referred to page and all its descendants have not (yet) been copied, but a set C flag only indicates that the referred to page was copied. Like the S flag, it does not show whether its descendants have been copied.

A similar mechanism does not exist for the R, W and M flags. When a page is written, it and the pages between it and the root of the page tree must be copied, but the parent page of a written page is not considered written or modified, although, strictly speaking, it has changed. A parent page is only considered written if it was written itself, and modified if a client explicitly requested the page tree to be changed, for instance, by adding or deleting pages.

Page trees are usually partially shared between versions. This implies that the flags indicating access to pages are also shared even though these pages have been accessed in different ways in different versions. This presents no problem, because the serialisability test need not descend shared parts of the page tree since they have not been accessed.

The flags, indicating whether a page has been read, written, modified or copied are stored in its parent page in the page tree; the root page is therefore the only page that does not have a C, R, W, S and M flag to indicate if it was copied, read, written, searched or modified. The managing server keeps these flags separate. The root page is always copied, by the way.

When a page is first read, the C, R, W, S and M flags it contains for its child pages must be initialised to zero. This requires changing that page. The Amoeba File Service must therefore not only shadow pages that were written, but also pages whose descendants were read. As we shall see later, once a version has successfully committed, the information contained in the R and S flags is no longer needed. The Amoeba File Service garbage collector may remove pages that were copied but not written or modified and reshare the corresponding page from the version on which it was based.

5.2. The Optimistic Concurrency Control Mechanism

As long as updates are done one after the other, commit always succeeds and requires virtually no processing at all. When two updates are done concurrently, however, the server must check if commit can be allowed by testing if the two updates can be serialised. If so, the commit is allowed; if not, failure is reported to the client, and the client must redo the update.

Kung and Robinson in their paper on optimistic concurrency control divide file update into three phases: the read phase, the validation phase, and the write phase [Kung81]. The validation phase checks *serial equivalence* of transactions T_i and T_j by testing if one of the following conditions hold:

(1) T_i completes its write phase before T_j starts its read phase.

(2) The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.

(3) The write set of T_i does not intersect the read set *or* the write set of T_j, and T_i completes its read phase before T_j completes its read phase.

If one of these conditions hold, the effect of updates T_i and T_j is the same as when T_i had finished before T_j started.

The Amoeba File Service carries out updates in such a way that the critical section of the validation phase and the complete write phase are done in one atomic action. This implies that the write phases of two transactions can never overlap and the serialisability test for two updates in the Amoeba File Service reduces to

(1) Version $V.i$ commits before version $V.j$ is created.

(2) The write set of version $V.i$ does not intersect the read set of version $V.j$, and $V.i$ commits before $V.j$.

The Amoeba File Service carries out its validation test when a client process requests a version to be committed (*i.e.*, when the client process signals the end of a transaction). In the test, it is only necessary to check if serialisability conflicts will occur with versions that have already committed. In principle, the commit mechanism works as follows.

The check whether condition (1) holds, and if it holds, the write phase, are carried out as one atomic operation, described below. If condition (1) does not hold, a test has to be made whether condition (2) holds. This means that the read set of the version to-be-committed must be compared to the write set of the already-committed version. The already-committed version cannot change, so this test can be carried out without locking being needed, or critical sections. When the test succeeds, the version-to-be-committed is established as the successor of the already-committed version, and commit is attempted as if condition (1) holds.

When a client requests to commit a version that is based on the current version, condition obviously (1) holds, because it was created after the current version committed. Therefore, Amoeba File Service allows all commits of versions based on the current version. The mechanism for this is demonstrated in FIGURE 5.

Let us assume client *C* sends a request to commit version *V.b*, which is based on version *V.a* to *V.b*'s managing server, *M.b*. Server *M.b* then proceeds as follows. First it ascertains that all of *V.b*'s pages are safely on disk. Then it sends a **set commit reference** request to *M.a*, the manager of *V.a*, the version that *V.b* was based on. *M.a* must then do the following without allowing other requests to interfere. First it must check if *V.a* is still the current version. If so, there is no conflict and the commit is carried out. The check for currentness is simply done by examining *V.a*'s commit reference. If it is nil, *V.a* is the current version, and the commit reference is set to the block number of *V.b*'s version page. This makes *V.b* the current version, and automatically the updates made to *V.b* are made permanent.

This is the only critical section in version commit: test and set the commit reference. In order to make this an indivisible action, only one server may be allowed to read the version block, test the commit reference, set it, and write it back. If the disk server implements a test-and-set operation, any server can be allowed to carry out a commit.

FIGURE 5(a) shows the situation before commit, FIGURE 5(b) after the commit has successfully been carried out. *M.b* returns an acknowledgement to *M.a* and *M.a*, in turn, returns an acknowledgement to *C*.

Let us now examine the case where *V.a* is no longer the current version, but another update, concurrent with that of *V.b*, has taken place. Let us assume the situation of FIGURE 6; *C* sends a request to *M.b* to commit *V.b*. However, *V.c* is now the current version, also based on *V.a*. First, *M.b* proceeds as before, and sends a **set commit request** to *M.a*; only this time, discovering *V.a*'s commit reference is already set, *M.a* does not carry out the commit, but returns *V.a*'s commit reference instead. This is the block number of *V.c*'s version page.

M.b must now check if the concurrent updates of *V.b* and *V.c* are serialisable; that is, test if condition (2) holds. *V.c* has already committed, so if the two updates are serialisable, *V.b* must come after *V.c*. This implies that there must be no overlap of *V.c*'s write set (the pages written during the update of *V.c*) and *V.b*'s read set (the pages read during the update of *V.b*). Since *M.b* received the block number of *V.c*'s version page, it can descend *V.c*'s and *V.b*'s page trees in parallel to examine if there is a serialisability conflict. This is tested using the *R*, *W*, *S*, *M*, and *C* flags in the page references. Note that uncopied parts of the tree in either *V.b* or *V.c* need not be visited since they can neither have been read nor written.

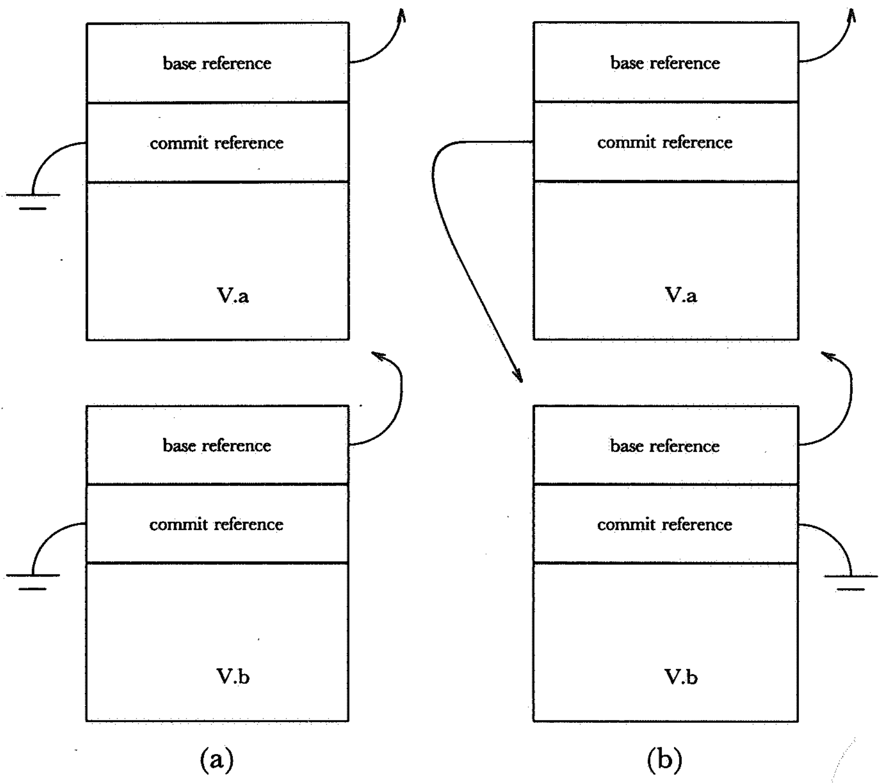

FIGURE 5. *V.b* succeeds *V.a* as the current version. (a) shows the situation before the commit, (b) shows the situation after the commit.

While descending the two page trees, checking the serialisability constraint, *M.b* also prepares the new current version, which must contain the updates made in *V.c* and those made in *V.b*. This is done by replacing unaccessed parts in *V.b*'s page tree by corresponding written parts in *V.c*'s page tree.

Both the serialisability test and the combination of the changes made by two concurrent updates are made in one pass over the page tree. Unvisited branches in either page tree are not descended, which makes the serialisability check quite fast when at least one of the concurrent updates is small.

An important property of the serialisability test is that it can be carried out in parallel with other updates of the file. While the routine *serialise* descends *V.b*'s and *V.c*'s page tree, other versions are allowed to commit, and other serialisability tests can also be carried out.

If *serialise* returns TRUE, *V.b* is ready to become *V.c*'s successor as the current version, and a *set commit reference* command is sent to *V.c*'s manager. If *V.c* is still current, this succeeds; if not, the serialisability test is repeated for *V.c*'s successor. This repeats until either the *set commit reference* command succeeds or *serialise* returns FALSE.

In the latter case, when *serialise* returns FALSE, the concurrent updates are not serialisable, and *V.b* is removed, and its owner notified. The update can be retried on another version.

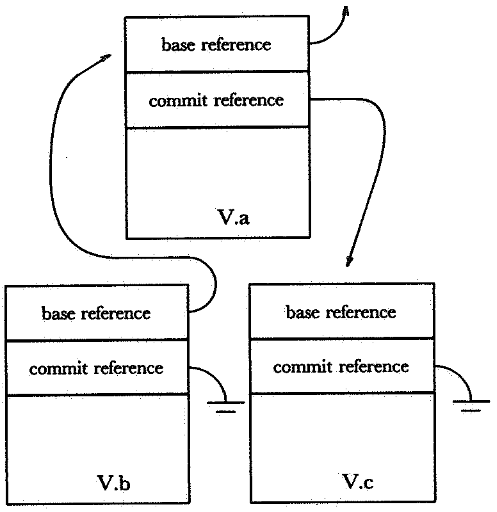

FIGURE 6. *V.b* wants to commit, but is no longer a descendant of the current version, *V.c*.

5.3. The Locking Mechanism

In the previous section we have assumed the upper part of the file tree consists of only one version page. In this section we describe the mechanisms for updating files when the upper part of the tree consists of more than one version page.

Before continuing, some terms are defined to simplify discussions. The upper part of the tree, stored on magnetic media, which contains the version pages for the files in the system, will be called the **system tree**. A file whose root is a leaf of the system tree will be called a **small file**, although a 'small file' may, of course, be arbitrarily large. A file whose root is an internal node of the system tree will be called a **super-file**. A small file or super-file whose root is contained in a super-file will be a **sub-file** of the super-file. A tree that makes up a small file or super-file is a **page tree**.

Updates of small files still use the optimistic method for update: Two updates on different small files do not interfere with each other since they affect disjoint page trees. Two updates of the same small file use optimistic concurrency control, as described in the previous section, to maintain integrity.

Updates of super-files, however, must use different rules. Updates on super-files generally require larger amounts of processing and affect more pages than updates on small files. Consequently, the likelyhood of a serialisability conflict is greater for updates on super-files. Additionally, the work lost because of a serialisability conflict is usually greater in the case of super-file updates.

For these updates *locking* provides a better form of concurrency control, because it warns in advance that two updates are likely to cause a conflict. Locking has some drawbacks, however, especially with regard to crash recovery. Most systems that use locking need elaborate mechanisms to restore the system after a crash: Locks have to be cleared, files or databases may have to be rolled back, or intentions lists must be carried out before the system can resume operations. We deemed it a challenge to find a locking mechanism that requires no special recovery in case of crashes. Our method is described below.

Each version page contains two *lock* fields, the *top lock* field, and the *inner lock* field. A file is considered to be *locked* if the lock field is non-zero. Locks only have meaning in the current version. We assume it is possible to test the two lock fields for zero and set one of them in one atomic operation.

When an update is made to a super-file, the *top lock* is set in its version block, and the *inner locks* are set in visited internal nodes of the file tree that are version blocks of sub-files. When an update is made to a small file, the *top lock* is also set in its version block, but, since small files have no internal version blocks, no *inner locks* have to be set.

Updates on super-files happen in exactly the same way as updates on small files, with the exception that locks have to be checked and set while the update is in progress. As in the case of small files, a version must also be created for a super-file before updates can be made. Before a version may be created, however, the version block for the current version must be locked.

The algorithm for creating a version is the following: If the file is a super-file, check the *inner lock* and *top lock* fileds, and, if they are both zero, set the *top lock*. If one of them is non-zero, wait until it is cleared, then try again. (The waiting process will be described later; locks are made of ports, which are used to realise an automatic warning mechanism for waiting updates.) If the file is a small file, only the *inner lock* must be tested, but the *top lock* set. Thus, a small file can be subject to more than one update at the same time, using the optimistic method of concurrency control.

If an update, while descending the page tree, discovers a *top lock*, it must wait until the lock is cleared before that subtree can be entered. It is not possible to encounter an *inner lock* while descending the page tree.

The commit operation is somewhat more complicated for super-files than for small files. Commit on a small file or a super-file works as described in the previous section. However, commit on a super-file is not finished when the *commit reference* is set. After commit on a super-file, the page tree must be descended to commit the sub-files of the super-file, and clear the locks. These commits always succeed, because the locks prevent access by other clients during the update to the super-file.

It is not difficult to see that this locking mechanism gives exclusive access to any subtree of the file system, and therefore provides a concurrency control mechanism. It can also be seen that sub-files, not accessed by an update, are not locked and therefore accessible to other updates. Full concurrent update remains possible on small files, because simultaneous updates on the same small file need not wait for *top locks*.

However, it is possible to use *top locks* on small files as hints which indicate that the file is likely to change soon. An update, known to affect large parts of a small file, can thus be postponed until the file is 'idle.' In contrast to this *soft locking* scheme, it is also possible to allow more concurrency on updates of super-files. The rules for creating a version may be relaxed to allow creating a version when the version block's *top lock* is set. The optimistic concurrency control which still lurks underneath this locking mechanism will see to it that no harm is done 'concurrencywise.'

When a server process crashes in the middle of an update, no harm is done to the integrity of the file system; the optimistic method underneath sees to that. The locks remain, however, rendering some files inaccessible. Fortunately, the mechanism described above for waiting on locks also provides a mechanism for crash recovery: When the server crashes, the outstanding transactions with the server crash as well, telling all servers waiting on locks that the process holding the locks has crashed.

A server, waiting on a *top lock* proceeds as follows: If the commit reference is off, the lock can be cleared without further ado, and, when the page tree is descended, *inner locks* (with the same port, of course) can be cleared or ignored. If the commit reference is set, the version it refers to is current. The version with the lock, and the current version are traversed simultaneously, and the commit references of the sub-files are set, finishing the work of the crashed server. A server, waiting on an *inner lock* ascends the *system tree* to the first unlocked page, or a page with a *top lock*. If the page thus found is not locked, the *inner lock* can be ignored. If the page is locked, it is treated as described above.

5.4. Maintaining a Cache

An important form of optimisation is caching. It is a defect in most distributed file systems that it is virtually impossible to keep local copies of remote data around, because of the race conditions thus introduced. The decreasing cost of primary memory makes caching techniques increasingly useful both for file servers and their clients. Some file servers have attempted a solution, the most prominent of which is probably XDFS [Sturgis80] Although XDFS provides an efficient mechanism for caching files or portions of files, the designers of the file server introduced the concept of the *unsolicited message*, a prod in the form of a message from server to client, telling the client his cache entry has become invalid. We have rejected such a solution because it does not fit the client-server model: an active client, that sends requests to a passive server that merely waits for requests, and carries them out. To have to be prepared to receive unsolicited messages makes client programs unnecessarily complex.

The Amoeba File Service — by design — is especially suited for caching. A version, from the moment of its creation, behaves like a private copy of a file that cannot change without the owners consent. Both Amoeba File Servers and their clients can therefore maintain a cache which, for the most recently used versions of a set of files, contains collections of pages. When a new version of a file is created, a client or a server examines its cache to see if there are any pages of a previous version of the file that can still be used. The mechanism for this is simple, as shown below.

For each file, a server or a private client can make a cache entry, consisting of pages of the most recent version it has had locally. When a request for a new version of the file is made, a serialisability test is made between the cache entry and the current version in order to find out which blocks of the cache are still valid. If the serialisability test succeeds, all blocks are still valid, if not, the blocks that cause the test to fail must be discarded. Note, that it is not necessary to transmit pages while making the serialisability test. If the cache holder is a client, the version capability must be sent to one of the Amoeba File Servers so the serialisability test can be made, and the server returns a list of path names of pages to be discarded. The server responsible for carrying out the test can make the test itself, or it can delegate the task to the server holding the most recent version for efficiency.

Even for shared files the page cache can be quite efficient. As shown previously, the serialisability test can be made in time proportional to the size of the intersection of the set of pages of the version in the cache and the union of the sets of pages in the versions since then. The server making the serialisability test likely has parts of the most recent version in its cache, reducing the number of disk accesses and the amount of network traffic further still. But our method of maintaining a cache is even more efficient for files that are not shared: the cache entry will always be for the most recent version of a file, so the serialisability test is a null operation, and all pages in the cache will always be valid.

It is worth noting that, in contrast to other file systems, the page cache does not have to be a 'write through' cache. When a page in a version is written, it need not be written to stable storage immediately. This can be postponed until just before commit.

The Amoeba File Servers can also conveniently cache the concurrency control administration, the flag bits. This allows serialisability tests without having to read the page tree. However, the flags must also be present in the files themselves to make crash recovery possible.

5.4.1. Robustness

The potential strength of distributed file systems, in contrast to traditional centralised file systems, is that distributed file systems can be much more 'crash proof'; that is, the file system will continue to operate, even when a few of the server processes, or even some of the disks are not operational.

Note that increased crash resistance and efficient concurrency control tend to mutually exclude each other, because better crash resistance is usually obtained by replication of data, which makes concurrency control more difficult. Making the Amoeba File Service crash proof has been an important aspect of its design.

In principle, the File Service operates using a number of server processes, which, in turn, use a number of block servers for information storage. This causes a separation of reliability aspects into two distinct areas: on the one hand, accessibility and robustness of file services as such, and, on the other hand, accessibility and robustness of individual files and versions. The former is realised through replicated server processes; the latter through replicated block storage, such as, for instance, *stable storage*[Lampson79] and backup block servers.

Assuming stable storage is used, the pages of each version of each file that are on disk are, in principle, always accessible. Access paths to committed versions go through the replicated file table, and a chain of version pages on stable storage, hence version access and file access can be guaranteed as long as one or more servers are operational.

Uncommitted versions need not be salvaged in a server crash. The concurrency control mechanisms were designed such that clients must be prepared to redo the updates in a version; if a version is lost in a crash the situation is not much different. Uncommitted versions are therefore not as important as committed versions.

6. Conclusions

The Amoeba File Service combines a number of concepts from the operating systems' world, the distributed systems' world, and the database world in a novel way. To the best of our knowledge distributed file servers have not been constructed using optimistic concurrency control. Yet, it provides a number of advantages not often encountered in other file systems.

With optimistic concurrency control, the file system is always in a consistent state. After a crash, there is no necessity for recovery: no rollback is required, no locks have to be cleared, no intentions lists have to be carried out. Optimistic concurrency control allows a maximum of concurrency in accessing files. Some updates will have to be redone when concurrent updates are not serialisable, but with the unbounded potential of computing power that distributed systems offer, redoing an operation now and then is acceptable.

Still, starvation may occur, especially when a large update must be carried out on a heavily shared file. The locking mechanism, described in § 6.4.3, can be used to lock a file when it is known that the update is large, and the probability of a serialisability conflict serious.

The file system should be organised carefully to avoid that updates on super-files have to occur too frequently. To this end, each small file should be self-contained as much as possible, so most updates will be on small files. This allows a large degree of concurrency. Locking should be the exception rather than the rule.

Page caches can be maintained, both by end-user processes and Amoeba File Server processes. We believe our method is superior to that in XDFS because no unsolicited messages are necessary. These cause an unneeded additional complexity for client processes.

The version mechanism and the page tree closely resemble the mechanisms in Felix. However, Felix uses locking at the file level. The idea behind our system of not locking small files is that many updates, even on the same file, do not affect the same parts of the file. For example, changes in an airline reservation system for flights from San Fransisco to Los Angeles do not conflict with changes to reservations on flights from Amsterdam to London.

The Amoeba File Service provides mechanisms that allow both sophisticated and simple applications to use its services efficiently. We have discussed the methods for concurrency control at some length, perhaps creating the impression that simple-minded applications – such as the example, mentioned in the introduction, of a compiler that needs to make temporary files – must once again pay the price of all that complicated machinery for guaranteeing serialisability. This need not be the case at all: Pages of 32K bytes can be written. Often, one such page is large enough to contain a whole file. Writing these one-page files is efficient; no concurrency control mechanisms slow it down.

A last advantage of the Amoeba File Service is that it is eminently suitable for a file system on write-once media, such as optical disks. Optical disks show great promise for the future, because of low cost and huge capacity. Traditional file systems are not suitable for these media, because files cannot be overwritten on a write-once device. The version mechanism, coupled with a cache in which uncommitted files are kept until just before commit seems an ideal file store for optical disks.

References

[Dion80]
Dion, J., "The Cambridge File Server," *Operating System Review*, vol. 14, no. 4, pp.26-35, Oct. 1980.

[Eswaran76]
Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notions of Consistency and Predicate Locks in a Database Operating System," *Comm. ACM*, vol. 19, no. 11, pp.624-633, November 1976.

[Fridrich81]
Fridrich, M. and Older, W., "The Felix File Server," *Proc. Eighth Symp. on Oper. Syst. Prin.*, vol. 15, no. 5, pp.37-44, Dec. 1981.

[Kung81]
Kung, H. T. and Robinson, J. T., "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp.213-226, June 1981.

[Lampson79]
Lampson, B. W. and Sturgis, H., *Crash Recovery in a Distributed Storage System.* Palo Alto, CA.:Xerox PARC, 1979.

[Menasce78]
Menasce, D. and Muntz, R., "Locking and Deadlock Detection in Distributed Databases," *Proc. Third Berkeley Workshop on Distributed Data Management and Computer Networks*, August 1978.

[Mullender85a]
Mullender, S. J. and Tanenbaum, A. S., "The Design of a Capability-Based Distributed Operating System," *to appear in Computer Journal*, 1985.

[Mullender85b]
Mullender, S. J. and Tanenbaum, A. S., "Protection and Resource Control in Distributed Operating Systems," *to appear in Computer Networks*, 1985.

[Reed78]
Reed, D., "Naming and Synchronization in a Decentralized Computer System," *PhD. Thesis*, 1978, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[Reed81]
Reed, D. and Svobodova, L., "SWALLOW: A Distributed Data Storage System for a Local Network," *Proc. IFIP*, pp.355-373, 1981.

[Robinson82]
Robinson, J. T., "Design of Concurrency Controls for Transaction Processing Systems", Ph.D Thesis (CMU-CS-82-114), Carnegie-Mellon University, Pittsburgh Pa., April 1982.

[Rochkind75]
Rochkind, M.J., "The Source Code Control System," *IEEE Trans. on Softw. Eng.*, vol. SE-1, no. 4, Dec. 1975.

[Schlageter81]
Schlageter, G., "Optimistic Methods for Concurrency Control in Distributed Database Systems," *Proc. VLDB Conference*, 1981.

[Stonebraker81]
Stonebraker, M., "Operating System Support for Database Management," *Comm. ACM*, vol. 24, no. 7, pp.412-418, July 1981.

[Sturgis80]
Sturgis, H., Mitchell, J.G., and Israel, J., "Issues in the Design and Use of a Distributed File System," *Operating System Review*, vol. 14, no. 3, July 1980.

[Tanenbaum82]
Tanenbaum, A. S. and Mullender, S. J., "Operating System Requirements for Distributed Data Base Systems," pp. 105-114 in Distributed Data Bases, ed. H. J. Schneider, North-Holland Publishing Co. (1982).