

A Distributed File Service Based on Optimistic Concurrency Control

Sape J. Mullender

*Centre for Mathematics and Computer Science
Amsterdam*

Andrew S. Tanenbaum

*Vrije Universiteit
Amsterdam*

Abstract

The design of a layered file service for the Amoeba Distributed System is discussed, on top of which various applications can easily be implemented. The bottom layer is formed by the Amoeba Block Services, responsible for implementing stable storage and replicated, highly available disk blocks. The next layer is formed by the Amoeba File Service which provides version management and concurrency control for tree-structured files. On top of this layer, the applications, ranging from databases to source code control systems, determine the structure of the file trees and provide an interface to the users.

1. Introduction

File systems play an important role in allowing information to be widely accessible, since most information is in one way or another stored in files. Many different kinds of file systems for distributed systems exist, 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 access from unauthorised hosts or users, files could be shared among different hosts at the same time, and the system would be totally immune against individual file server crashes or disk crashes. Unfortunately, such distributed file systems do not yet exist, and improving one aspect of a file system is nearly 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 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 could be increased by several orders of magnitude.

This paper goes into the design of the Amoeba File Service, one of the three file services for the Amoeba Distributed System [Mullender85a]. Section 2 describes the considerations that led to the design of this file server and gives an overview of related work. The underlying Block Service is briefly discussed in Section 3. A detailed description of the Amoeba File Service follows in Section 4.

2. Design Principles

The Amoeba Distributed System was designed by Mullender and Tanenbaum at the Vrije Universiteit in Amsterdam [Mullender85b]. Amoeba is an open system, designed to accommodate heterogeneous hardware and software. The Amoeba Kernel, the replicated operating system running in most of the machines on the network, supports process management and interprocess communication. All other services are provided by server programs that execute in user space. A capability mechanism provides protected communication between clients and services and protected access to objects [Mullender84].

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.

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 design of the Amoeba File Server was an experiment. We wanted to try to design a layered file system, where replication, concurrency control, and database management would be in different layers. The bottom layer, the *physical*

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

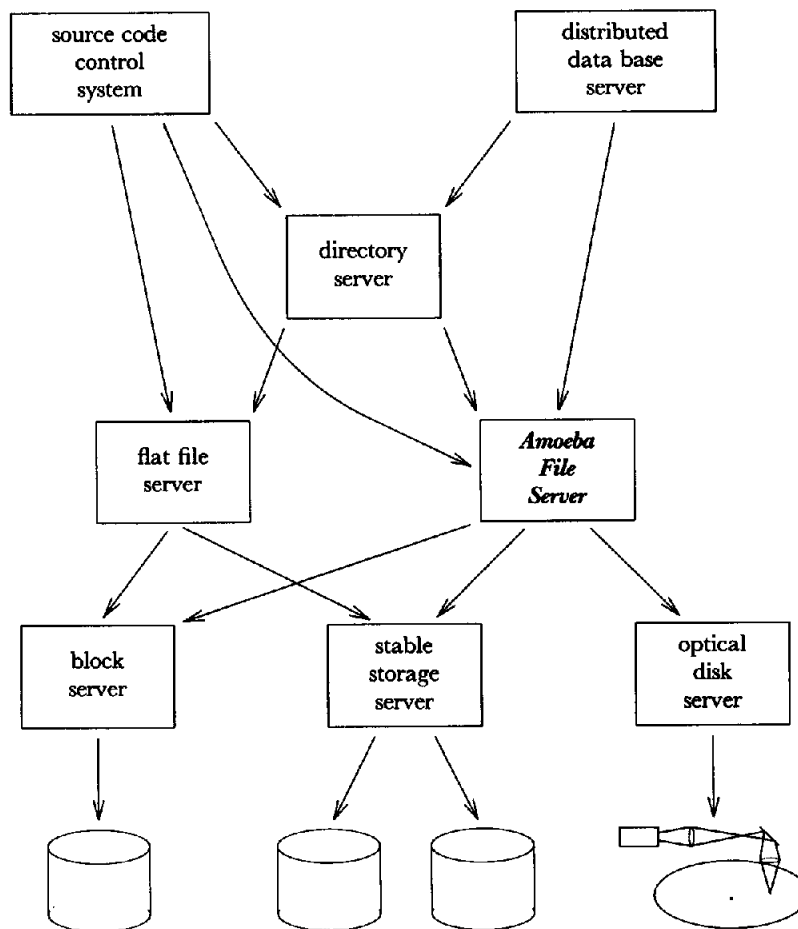


FIGURE 1. An example of a storage services hierarchy in an open system.

layer, consists of the storage devices: electronic disks, magnetic disks and write-once optical disks. The next layer is the *Block Service*, providing *virtual disk blocks* of various kinds: fast but crash-volatile storage in memory, stable-storage disk blocks, replicated disk blocks with atomic write on all copies simultaneously, etc. The next layer up is the file system, with concurrency control mechanisms for file access, and the top layer, providing the interface to various applications, provides database management services. This paper concentrates on the middle layer: the file system.

The Amoeba File Service is a distributed file service: a request for an operation on a file can go to any one of a number of file server processes where it will be executed. The layered structure is an advantage here; the Block Service already forms an abstraction away from physical storage locations.

But the layered structure of the file system is also a potential bottleneck for performance of great magnitude: A simple query on a tiny database from a client process invokes the database service, which invokes the file service, which invokes the block service, which finds the block on disk. Caching strategies are essential at all levels of the hierarchy to avoid having to descend to the bottom level of the service hierarchy on each client request. However, caches and concurrency control mechanisms are likely to become enemies:

the administration of the caches in a rapidly changing environment can cause more inefficiency than not keeping caches at all. Obviously, thinking about caching possibilities and strategies have to be an essential part of the design process.

File services must provide the tools for the efficient implementation of as wide a set of applications as 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.

Currently, Amoeba has three file servers: a simple one, written as a student programming project, which implements simple flat files, without concurrency control; a UNIX-like† file server, which is used in combination with a UNIX emulation package for running UNIX software on Amoeba; and the Amoeba File Server, described in this paper.

An important design principle was also: '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

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

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 atomically updated. All such a 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 users who just want a reasonably reliable repository for their files, cheap and fast, while on the other hand, other users should be taken into account who need ultra-reliable storage for their files, fancy synchronisation of access by many simultaneous users, and guaranteed availability, who will be prepared to accept that it must be more expensive and slower.

Another important issue in file service design is that the semantics of file service 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 the mechanisms provided. 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 many times 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 [Mullender84].

2.1. 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], SWALLOW [Reed81], and ALPINE [Brown85]. They all have mechanisms for concurrency control. Most file servers, including the Cambridge File Server [Dion80], XDFS, FELIX and ALPINE 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 and a two-phase commit protocol.

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 client has locked a datum on a server for some time, a timer expires and the lock becomes *vulnerable*. Another client, waiting on that lock, can then prod the server, 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. A new version

is created by making a (virtual) copy of the current version; this new version can then be read and modified, and, when all changes have been made, the new version may become the new current version. Sharing is controlled using locks, providing six access modes. Files are tree-structured. A new version is created by copying a pointer to the root of the current version. When it is modified, a copy-on-write mechanism is used which leaves the current version intact. With this mechanism, only the changes between versions are stored.

ALPINE offers the user a choice between locking at the file level or at the page level. File locking is the default, but sophisticated applications are provided with mechanisms for setting and releasing various types of locks on individual pages of a file. A transaction log is kept to enable recovery from failures and deadlocks caused by conflicting locking operations. Brown *et al.* claim that transaction logs can be implemented more efficiently than a shadow-page mechanism [Brown85]. A transaction log mechanism, however, makes it more difficult to implement an efficient and simple caching mechanism, as shown in § 4.5.

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). Additionally, versions do not overlap; that is, they do not share the unmodified portions of the file.

2.2. The Amoeba File Service Compared With Other File Servers

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 a version mechanism is that the file system is always in a consistent state. Most file systems update files in place and need a mechanism for bringing back the file system to a consistent state after a crash of a server and possibly also after a crash of a client. 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. In the Amoeba File Service, the file system is always in a consistent state (assuming the updates themselves are internally consistent). Server crashes have no serious consequences: 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 their updates.

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 likelihood 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. Caching is an important concept in distributed systems [Lampson83], ITC [Satyanarayanan85] and CFS [Schroeder85] mention caching mechanisms as important parts of the system. 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. ITC was not designed for database applications and does not provide complicated machinery for concurrency control; maintaining a cache is relatively simple there. 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. In CFS, shared files do not change after creation which makes caching trivial; a version mechanism embedded in the naming mechanism is used to reflect change.

On the Amoeba File Service, the integrity of the cache need only be 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. Furthermore, the Amoeba File Service needs no unexpected 'unsolicited messages.'

3. The Block Server

The principle of separating the issues of file service and 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 on individual blocks. Based on this, file services can realise

concurrency control policies. The Amoeba File Service, to commit a version of a file, for instance, will exclusively lock and read a block, examine and modify it, then write and unlock the block again.

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*. With minor modifications their method can be used to provide disk service which continues to be available when single-site crashes occur.

4. Amoeba File Service

The Amoeba File Service was developed for, but is not restricted to, the Amoeba Distributed Operating System. 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 conflicts in simultaneous updates.

The Amoeba File Service implements optimistic concurrency control by a version mechanism: When a client opens a file for modification, a new version of the file is 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. After commit, a version becomes immutable. Several uncommitted versions of the same file can exist at a 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.

The Amoeba File Service is a distributed service. Several server processes can be established on one or several physical machines, and each server is capable of handling updates on any file. Each version has a **manager**, the server process which created the version. Different versions of a file can have different managers. A client will typically direct all requests to the Amoeba File Service to a server process that is close to it. New versions will thus tend to be close to the clients that ordered their creation.

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 an Amoeba message transaction: 32K bytes. This ensures that pages can be read and written in one (atomic) action.* A page

* 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.

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.

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 intention 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.).

Files can be grouped together in '*superfiles*,' and superfiles can be grouped in other superfiles. Such a superfile structure is a thus also a tree structure. A superfile is, in fact, almost exactly like an ordinary file: All pages of a superfile

may contain data, exactly like an ordinary file; the root page of a superfile, however, contains references to the root pages of other files, superfiles, or both. The Amoeba File Server provides atomic update on files, or superfiles. Files or superfiles without a common root cannot be updated atomically.

The top of the tree, that is, the collection of root pages of files, is stored on magnetic random-access media, for instance, such as provided by the *stable-storage* server, mentioned in the previous section. The lower parts of the tree, that is, the collection of non-root pages of files, can be stored either 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*.

4.1. File Representation

A 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

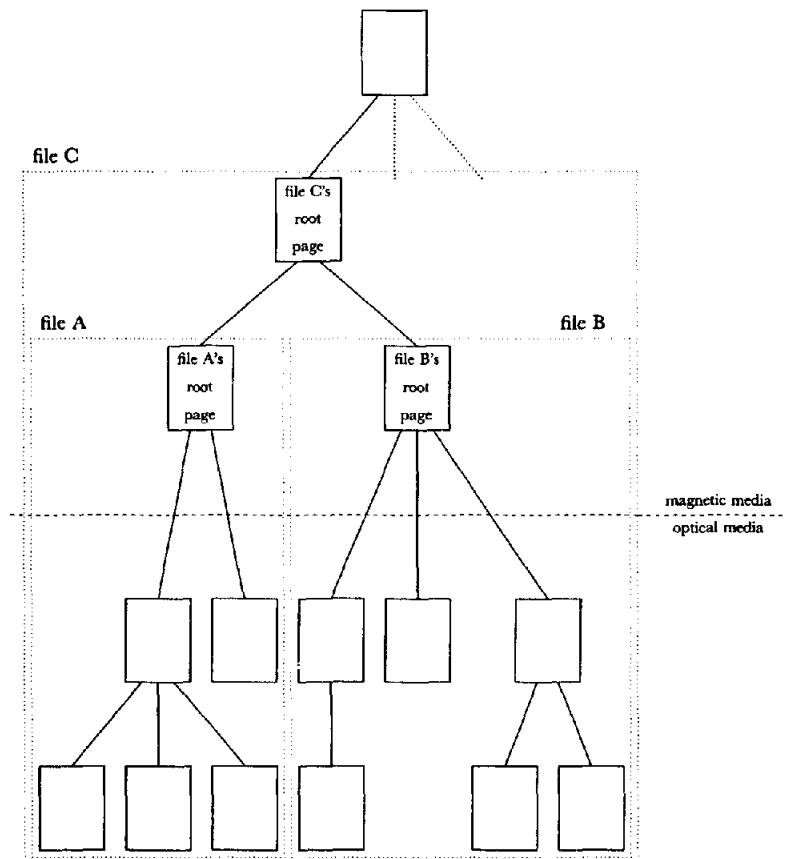


FIGURE 2. A file has the structure of a tree of pages. A superfile can be viewed as a tree of files or also as a tree of pages.

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 Service uses for administrative purposes.

The root of a page tree is referred to as the *version page*. The client data in a page has no predefined structure. Clients are free to write them as they see fit. The references in a page to pages further down the tree 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)					
base reference					
nrefs (number of page references)					
dsize (number of data bytes)					
client data					
page number	C	R	W	S	M
.			.		
.			.		
.			.		
page number	C	R	W	S	M

FIGURE 3. The Amoeba File Service page layout

The layout of a version page is shown in FIGURE 3. The layout of internal pages is the same, with the exception of the first five fields, which are not used. Each page is divided in two areas, a header area and the page itself; the separation is indicated by the double line. The first field in the header area of a version page is the *file capability*. This field gives the capability of the file whose root the version page is. The next field is the *version capability*, the version of the file whose root the version page is. The *commit reference* field is also used in version pages only; 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 superfile whose root is higher in the page tree; their function will be explained in a later section. The fields mentioned just now are only present in a version page. They are absent (or ignored) in other pages. The remaining fields, to be mentioned below are present and used in all pages, root pages and internal pages alike.

The *base reference* field, present in all pages of a version, 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. If this field is zero, the page is a leaf page. 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 § 4.3), 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. 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 reference tables 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.

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.

4.2. The Copy-on-write Mechanism

In this 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 independently of one another. The *S* flag tells that the references have been searched, the *M* flag tells that 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 be searched. This implies that, if a page has not been searched, the subtree of which it is the root cannot have been searched

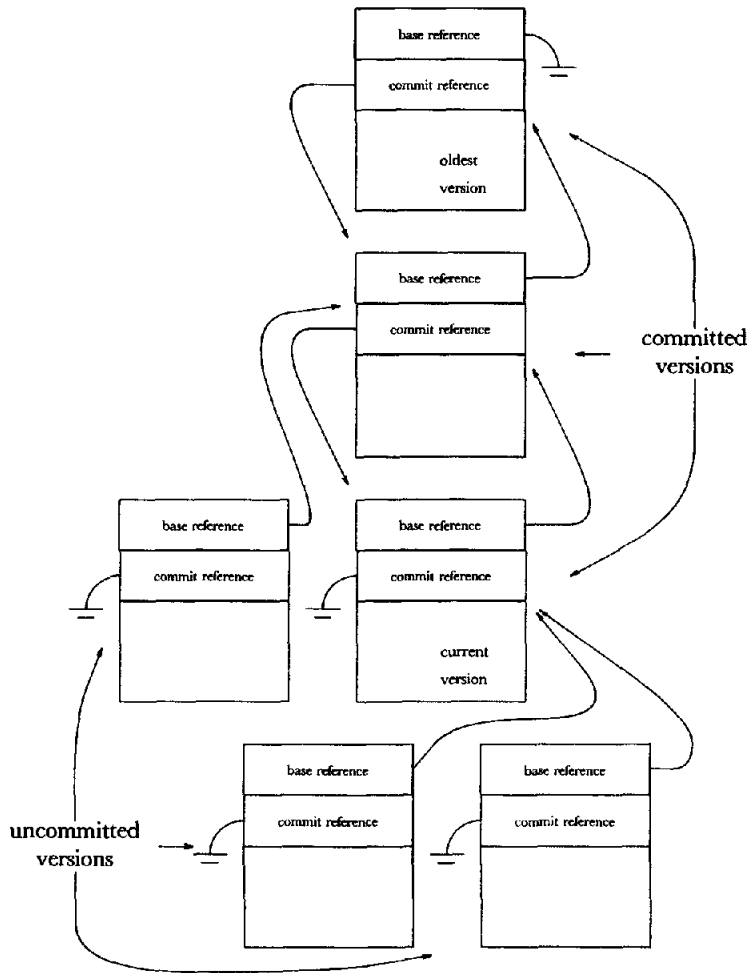


FIGURE 4. The 'family tree' of a typical file. Only the version pages are shown. The page trees descending from the version pages are not shown.

or read 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, if it is still shared with another version (i.e., it hasn't been copied-on-write yet), 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 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 always written in place, because it is never shared with another version. When a non-root page is thus copied, the *C* flag is set in the reference to it (in the parent page). A page is thus 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 its client data was written, and modified if pages were added or deleted.

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 associated *C*, *R*, *W*, *S* and *M* flags in the file tree 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.

4.3. The Optimistic Concurrency Control Mechanism

As long as updates are carried out one after the other, commit always succeeds and requires virtually no processing at all. When two or more updates proceed concurrently, however, the server must check whether commit can be allowed by testing whether those 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.

When there is no concurrency, a new update will not start until the previous one has been finished; that is, a new version will not be created until the previous version has been committed. The next version is thus always based on the previous one. Updates are concurrent when a new version is created while another, uncommitted version still exists. This implies that concurrent updates are based (sometimes indirectly) on a common (committed) version.

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 whether 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 consist of 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 is committed 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 is committed 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.

When a client requests to commit a version, $V.b$, that is based on the current version, $V.a$, condition obviously (1)

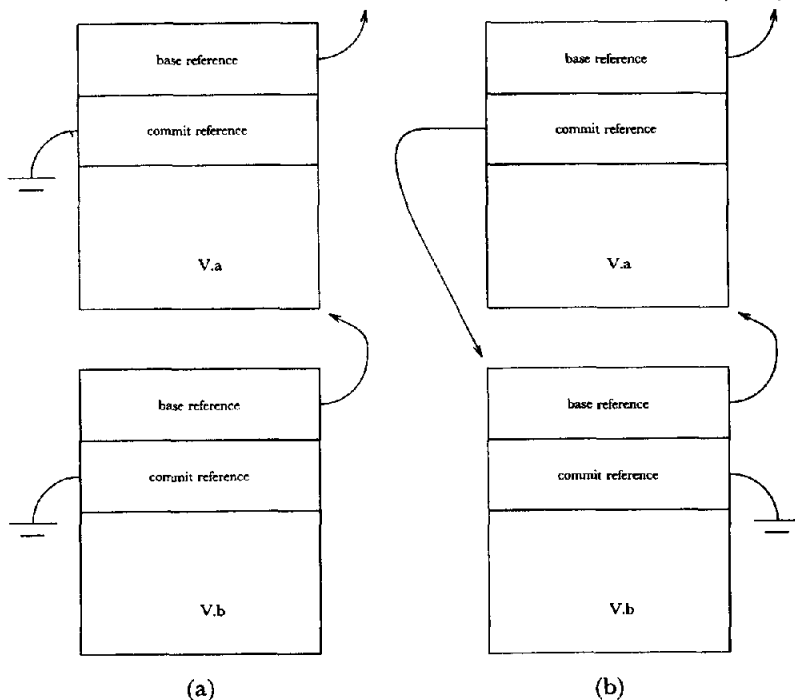


FIGURE 5. $V.b$ succeeds $V.a$ as the current version. (a) shows the situation before the commit, (b) shows the situation after the commit.

holds, because $V.b$ was created after $V.a$ was committed. Therefore, the 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. ($V.a$ is specified in the *base reference* field of $V.b$'s version page.) $M.a$ must then do the following without allowing other requests to interfere. First it must check whether $V.a$ is still the current version. If so, there is no conflict and the commit is carried out. The check for currentness is simply performed 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 page 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.

The test and set the commit reference is the only critical section in version commit. In order to make it 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. $M.a$ returns an acknowledgement to $M.b$ and $M.b$, in turn, returns an acknowledgement to C .

Let us now examine the case where $V.a$ is no longer the current version, which means that another update, concurrent with that of $V.b$, has taken place and was committed. 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 whether 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 unshadowed parts of the tree in either $V.b$ or $V.c$ need not be visited since they haven't been accessed.

While descending the two page trees, checking the serialisability constraint, $M.b$ also prepares the new current version, which must combine the updates made in $V.c$ with 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.

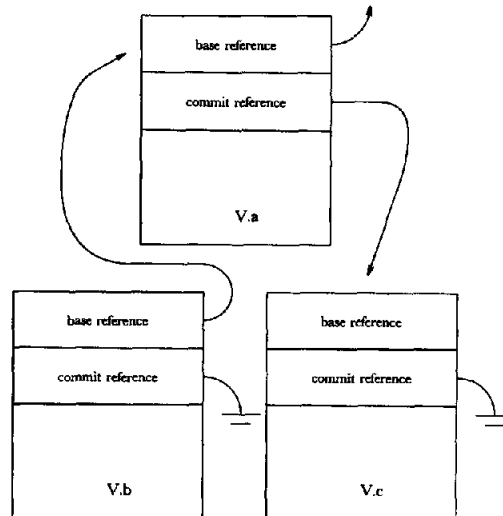


FIGURE 6. $V.b$ wants to commit, but is no longer a descendant of the current version, $V.c$.

4.4. The Locking Mechanism

In the previous section we have described the update mechanism for a single file. In this section we describe the mechanisms for updating superfiles which may contain several smaller files.

Before continuing, some terms are defined to simplify discussions. The upper part of the tree, 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, i.e., an ordinary file, will be called a *small file*, although a 'small file' may, of course, be arbitrarily large. In FIGURE 2, for instance, file *A* and file *B* are small files. A file whose root is not a leaf node of the system tree will be called a *super-file*. In FIGURE 2, file *C* is 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 likelihood of a serialisability conflict is greater for updates on super-files. Additionally, the work lost because of a serialisability conflict is usually more 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 the drawback, however, that after a crash, locks have to be cleared 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, which indicates the version page is the root page of an ongoing update, and the *inner lock* field, which indicates an ongoing update higher in a higher super-file has affected the locked version page. A file is locked if one of the locks is on. We assume the lock fields can be tested and set in one atomic operation. When an update is made to a super-file, the *top lock* is set in its version page, and the *inner lock* in visited internal nodes in the file tree that are version pages of subfiles. When an update is made to a small file, the *top lock* is also set in its version page, but since small files have no internal version pages, no *inner locks* need 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* fields, 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 wait-

ing process will be described later; locks contain the name of the locking server, which is 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. When multiple, concurrent updates are allowed on a super-file, this rule can be used on super-files as well.

Assume, for instance, that an update of file *A* in FIGURE 2 has to be carried out. It is a small file, so only its *top lock* will be set. Other updates on file *A* can proceed concurrently: the *inner lock*, which is not set, is tested, and concurrent updates can be carried out as described in the previous section.

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.

Suppose again that file *A* is being updated, so its *top lock* is set. An update of file *C* can proceed, as long as its left subtree, which is file *A*, is left untouched. When *C*'s left subtree is descended, however, *A*'s *top lock* will be encountered, and *C*'s update must wait until *A* has been committed and its lock has been cleared.

The use of the inner locks will become clear when we assume an update on file *C* descends *A*'s page tree. This update will cause *A*'s inner lock to be set. When an attempt is now made to update *A*, the inner lock will be encountered, and the update must wait until it is cleared.

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. For-

tunately, 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* (containing the same server name, 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 page without an inner lock, or a page with a *top lock*. If the page thus found has no lock at all, the *inner lock* that the server was waiting on can be ignored. If the page thus found has a top lock, it is treated as described above.

4.5. 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 difficulties of keeping the local copies up-to-date. The decreasing cost of primary memory makes caching techniques increasingly useful both for file servers and their clients.

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 client requests a server to create a new version of a file, the client, the server, or both, examine their 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 client or a server 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 version used for 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.

If a file is not shared, the cache entry will always be based on the current version. The serialisability test for finding out if the cache entry is up-to-date is then a null test which always succeeds. 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.

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. 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 a version mechanism, 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 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 Francisco 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. Since pages of 32K bytes can be written, one such page is often 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

- [Brown85] Brown, M. R., Kolling, K., and Taft, E. A., "The Alpine File System," to appear in *ACM TOCS*, 1985.
- [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.
- [Lampson83] Lampson, B. W., "Hints for Computer System Design," *Proc. 9th SOSP*, Oktober 1983.
- [Mullender84] Mullender, S. J. and Tanenbaum, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5,6, pp.421-432, 1984.
- [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., *Principles of Distributed Operating System Design*. CWI, Amsterdam:PhD. Thesis, October 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.
- [Satyanarayanan85] Satyanarayanan, M., "The ITC Distributed File System: Principles and Design," *Proc. 10th SOSP*, December 1985.
- [Schlageter81] Schlageter, G., "Optimistic Methods for Concurrency Control in Distributed Database Systems," *Proc. VLDB Conference*, 1981.
- [Schroeder85] Schroeder, M. D., Gifford, D. K., and Needham, R. M., "A Caching File System for a Programmer's Workstation," *Proc. 10th SOSP*, December 1985.
- [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).