DRAFT - DRAFT - DRAFT

Design for a Distributed Data Storage System

by Butler W. Lampson and Howard E. Sturgis

February 5, 1976

This document is a preliminary description of the design of a system for storing data on a number of computers. The design provides protection for the data, properly controlled simultaneous access by several users, and a high degree of immunity against loss of stored data because of system malfunctions.

XEROX

PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

This document is for internal Xerox use only.

CHAPTER 1. INTRODUCTION

... and gives to airy nothing
A local habitation and a name.
Midsummer Night's Dream V i 16

This document describes the design of the core of a distributed file system. This design is the result of about two years of intermittent work by a group consisting of the authors, Charles Simonyi, Patrick Baudelaire and Ed Fiala; there have also been various occasional participants.

Our current plan is to implement this design (or some suitably modified version of it) during 1976, together with the directory, archiving and backup facilities (collectively called *extended* facilities) needed to make it a viable alternative or supplement to the Maxc file system. The goal is to have an operational system by the end of 1976. We estimate that this will take three man-years of work by four people. Currently Ed Taft, Howard Sturgis and Barbara Hunt are planning to participate in the implementation; a fourth person is needed.

The reader should note that this memo describes only the core of the distributed file system, and not the extended facilities. We believe that the extended facilities can be implemented as a separate set of modules, depending on the system described here for storage of data, but not themselves providing any functions on which the storage system depends. The system described here will be called the *distributed data storage system* to distinguish it from a complete distributed file system.

CHAPTER 2. OBJECTIVES

The design is motivated by a number of objectives. These mainly have to do with the aspect which the system presents to a program which is using its facilities. The following short list of the objectives is followed by a more detailed discussion. The italicized keywords provide a convenient one-sentence summary of the objectives.

Data storage

The system stores files which are sequences of bytes.

It is possible to have information stored redundantly.

Distributed

The system is implemented on several co-operating computers.

The normal mode of access is remote, over the Ethernet.

Protected

Each file has protection information associated with it.

Requests are authenticated.

Consistent

A sequence of several reads and writes can be performed as an indivisible, atomic operation, in spite of crashes or simultaneous activity by other users.

User programs can maintain local caches for data stored in the system.

It is important to recognize that a number of things which are often thought of as essential properties of a file system are omitted from the data storage system. In particular,

There is no provision for symbolic names or directories.

There are no provisions for accessing data without regard for which machine is storing it (although there are facilities for finding out which machine is storing it).

There is no backup or archiving.

Data is entirely unstructured, except for the organization into files, which is intended only to provide a minimum location-independent naming facility.

As an obvious consequence of these points, the system is intended to be used by programs, not directly by people.

These things are missing not because they are considered to be unimportant, but because we believe they can be provided by separate components interposed between the user and the data storage system. We have tried to put into the data storage system only those facilities which cannot be separated out in this way.

The remainder of this section expands on each of the points in the list of objectives above. The reader should bear in mind that this section is only a description of objectives, and that complete descriptions of the various mechanisms in the system, and more thorough discussions of the alternatives which have been considered, can be found in later sections.

Here, and in the rest of this document, the word *user* means user program, since human users do not appear.

2.1 Data storage

2.1.1 Files

The objects stored in the system are files. There is no attempt to store more complicated objects, such as databases, sealed items, capabilities, or whatever. A file has several components:

an *identifier* which serves to name it; this is a 64 bit number;
a *length L*;
a sequence of *L* data bytes;
some protection information;
an *interception list*;
a *property list*;
possibly other components.

Note that a file is named by its identifier, rather than by a text name. User programs, or system facilities outside of the data storage system, may use some files to represent directories, and thus provide a symbolic naming facility.

2.1.2 Redundancy

An option is provided to store files redundantly (in some cases this may be mandatory). The weakest form of redundant storage permits the loss of a single disk page without losing any information. The strongest form of redundancy permits the loss of an entire machine or disk pack without the loss of information.

2.2 Distribution

2.2.1 Several cooperating computers

The system is implemented on several computers, communicating over some packet facility, such as the Ethernet. The net is assumed to be imperfect, in that not all transmitted messages will be received (a message received with a bad checksum is treated as not received).

The system behaves as a single organism in certain important ways. For example, the user can specify that a set of changes to the stored data should appear to occur simultaneously on all the system computers holding portions of the affected data. Also, files may move from one computer to another, possibly in response to a user request, and possibly spontaneously.

However, the user must be aware that several computers are involved. He is expected to determine which computer currently stores a particular piece of data, and to address requests for reading or writing that data to the appropriate computer. A mechanism which allows the user to direct his requests to a single computer can be provided outside the data storage system.

The system provides a way of finding out where a file is stored given its identifier. This process may be rather inefficient, however, and the system is designed on the assumption that users will normally keep track of the current (or most probable) location of a file.

2.2.2 Access is normally remote

Users normally access the system from remote computers; no provision is made for ordinary user programs to run in system computers. This access is over a packet communication facility, and takes the form of *requests* for various commands; a request is coded into packets in some way which is well-matched to the properties of the network. It is assumed that the communication facility will not always succeed in transmitting a message. This assumption affects the algorithms that users should should use when accessing the system.

It is possible that programs which are outside the data storage system, but are part of a larger file system, might reside in the same computers on which the data storage system runs. It is also possible for a personal computer to participate in the data storage system, by responding to requests for service in the same way that a system machine would respond. However, it must be recognized that the system cannot offer any guarantees about the security or correct functioning of such a personal machine.

2.3 Protection

2.3.1 Files are protected

The system provides two relatively independent protection mechanisms, hard and soft. The intention is that hard protection will provide credible, but awkward guarantees; while the soft system will provide flexibly programmed access, and hence can be programmed to give unintended access. Each request from a user must pass both barriers to be accepted.

The only entity to which protection information can be attached is a file. All the information in a file is protected identically, and different files have entirely independent protection.

2.3.2 Requests are authenticated

To support the control of access to files, each request is submitted to a series of authentication procedures. In order to gain access to more highly protected files, the request will be subjected to more extensive authentication procedures. These procedures are defined and performed independently of the actual file access. That is, a request will specify what level of authentication it requires independently of the command it requests. The result of the authentication is then checked against the protection information on the file when the request is about to be performed.

2.4 Consistency

2.4.1 Atomic commands

The system provides a mechanism called a *transaction*, which serves to package a set of file access requests, possibly including a number of reads and writes of different files, into a single action. That is, all the writes in the transaction will appear to occur simultaneously, and the information obtained by all the reads in the transaction will still be correct at the apparent time of the write. In the event of file system crashes, whether hardware or software, either all the writes of the transaction will have been performed, or none of them.

The algorithms which implement transactions have a side effect: locks are implicitly set on the data by reads and writes. These locks can lead to deadlock. The system handles these deadlocks by timeout; when a timeout occurs, the transaction involved may be aborted. Thus, users must be able to handle unexpected aborts of their transactions. Of course, system crashes would occasionally abort a transaction in any case, but deadlocks may cause aborts to occur much more frequently.

2.4.2 Caches

There is a facility which allows a user to maintain a local copy or *cache* of desired portions of files stored in the system. If the user follows an appropriate algorithm, the system guarantees the accuracy of the local copy. That is, whenever the user fetches any information from the cache, he will be informed of any changes which may have affected that information.

This facility is useful in situations where the user is reading some information frequently, and the information is updated by other users quite infrequently. For example, the information might be a directory or index into a data base, which likely to be often accessed and rarely changed.

CHAPTER 3. AN EXTERNAL VIEW OF THE SYSTEM

In this chapter we are concerned with the way the storage system appears to a user (i.e. a using program). We will describe the system in terms of the states it can be in, and the operations which the user can invoke to read and modify its state. The reader should note that in this section we are describing an *interface*, and are not saying anything about the implementation. The internal organization of the system is discussed later in this document.

The state of the system is divided into two parts, permanent or *stable* and temporary or *volatile*. Roughly speaking, stable state represents files, and volatile state represents transactions. Volatile state contains such things as interlocks on data which is being read or written by transactions in progress, and modifications to files which have been requested by transactions in progress (these modifications do not become part of the stable state until the transaction is completed). Stable state is preserved over system crashes, while volatile state is not. The stable system state is called the *S-state* or *S-view*, and the volatile state is called the *V-state* or *V-view*.

All file system commands are performed as a result of requests sent by users. When a command is complete, a response is usually sent back to the user who requested the command. A request is sent to a particular computer. In almost all cases, a request is part of a transaction, and is sent to the representative of that transaction in that computer; this representative is called a *worker*.

A request involves at least three levels of protocol: network, authentication and data. It arrives at a storage system computer as a collection of packets. The network protocol combines these packets to form a *raw* request. Then the authentication protocol converts this raw request into an *authenticated* request. Finally, the authenticated request is delivered to the appropriate worker, which makes any necessary protection checks, and if these are successful performs the requested operations on the data.

Responses travel in the opposite direction, from workers to users. We have not, as yet, given much consideration to the transformations which convert a logical response into a collection of packets.

The remainder of this section is devoted to describing these various components of the system from the user's point of view.

3.1 Protection

The model on which the protection mechanism is based is the following. Think of each system computer as a windowless building with a single door. At the door there is a series of security officers, but inside the building people and documents flow around freely. Requests written on pieces of paper arrive at the door. Each request is examined by the first security officer. He may look at the sender's name, examine the signature, check the ID of the messenger who delivers the paper, look for a password written on the paper, or even decode an encrypted message.

Based on any or all of this information, the security officer stamps the paper with a rubber

stamp which tells the people in the building how seriously to take it. The mark which the rubber stamp leaves on the paper is called an *impression*, but we will often abuse language and simply call it a stamp. The text of a stamp might be **By order of the commanding officer**, or **From the payroll department**, or **Cleared to receive secret information**. The purpose of the stamp is to provide a simple, uniform and reliable basis on which receivers of the request can decide whether to act on it or not. The security officer's function is to convert the complicated, varied and uncertain properties which the request has when it arrives from the real world into this simple form. The scheme is based on the assumption that the paper is not subject to tampering once it is inside the building.

After leaving the first security officer, the request may pass through additional officers, who scrutinize it further, and perhaps affix additional stamps. Finally it leaves the security officers, and is routed through the building to be acted upon. Suppose a file clerk receives it, reads it, and determines that it is asking for a copy of a particular document to be delivered to the originator of the request. He extracts the document from the file and looks at its cover sheet. There he finds a list of the stamped texts which can authorize copying of the document, e.g. **Cleared to receive Xerox private**. The clerk looks at the request to see whether it has a stamp with one of these texts. If so, he proceeds to send the requested information; if not, he rejects it. The texts listed on the document are called *guards*.

In order to make the security officer's job easier when he is deciding which stamps to affix, and to make it convenient for a request to carry only the minimum stamps which are needed to accomplish its intended mission, each request contains a special section labeled *Claims for Stamps*. In this section the originator of the request lists the stamps he wants to have on the request. The security officer affixes only those stamps whose names appear in the *Claims for Stamps* section, and which he determines that the request is entitled to have.

This model of the system as a building, with security enforced at the entrance, explains why we insist that storage system computers not be shared with user programs. Since the storage system has no internal protection mechanism, it depends for its security on its complete isolation from any malicious influence. If a user program is to run on the same machine, then either

- the machine must have protection mechanisms, not part of the storage system, which ensure the isolation of the storage system from the user program, or
- the user program must be trusted, or
- data stored on that machine must be recognized as insecure, and other storage system machines must treat that one as suspect.

We can imagine circumstances under which each of these alternatives might be acceptable. For our current purposes, however, we will proceed under the assumption that the storage system owns its machines, and that no user programs run in the storage system machines.

3.1.1 Levels of protection

In our view, there is a fundamental conflict between security and convenience. In order to be highly secure, a protection system must be fairly *simple*, and it must be *stable*, i.e. infrequently changed. Simplicity is important because security is only as strong as its weakest link, which a determined and ingenious enemy is likely to find. If there are many links, each one cannot be scrutinized carefully enough to ensure its strength. Stability is important because each change is an opportunity for an error. Furthermore, when changes are frequent, those responsible for specifying the changes or checking their correctness become overworked and make mistakes.

In addition, simplicity and stability are essential if the system is to be *credible* to its users, especially when it is new. No sensible man will put his trust in a very complicated and constantly changing mechanism, unless perhaps the mechanism has accumulated a great deal of operating time without a failure.

Unfortunately, these desirable properties cannot be had for nothing. A simple mechanism will be unable to represent the complex requirements for individual privacy and restricted sharing

of information which are likely to be associated with a large collection of stored information. And these requirements are constantly changing. The two temporary payroll clerks may enter information from time-cards only on Thursday afternoon, except that Thursday, July 4 is a holiday and they will work on July 3 instead. The medical department may read the work histories, and read and update the medical records, for all employees, but may not see any other information in the personnel files. And so forth. Methods are known for satisfying protection requirements like these, but they do not lead to a simple and stable system.

The data storage system reconciles the need for security with the need for flexibility by providing two levels of protection. The first level, called the *hard* protection mechanism, caters for security at the expense of flexibility. The second level provides *soft* protection, which is as flexible as we know how to make it within the general framework of the storage system. Still more elaborate protection can be provided by programs operating outside the storage system, such as a database manager.

At each level there are stamps and guards. In the interests of simplicity, each request gets exactly one hard stamp, and each file exactly one hard guard. By contrast, a request can have any number of soft stamps, and a file can be guarded by an arbitrary Boolean expression containing ands, ors, and tests for specific soft stamps.

The remainder of this section on protection presents the technical details of the protection mechanism we have designed for the distributed data storage system. We have tried to make the terminology consistent between the informal description above and the precise one below, but in case of doubt it is the precise description which governs.

3.1.2 The nature of authentication

In deciding whether or not to authenticate a request by affixing one of its claimed stamps, the authenticator has two kinds of information at his disposal: the *origin* of the request, and its *content*. These are called *indicators* of the request.

We take the position that each separate message must be authenticated independently of any other messages, since the viscissitudes of communication preclude any guarantee of the relationship between messages. We do admit the possibility that the authenticator might have some state. For example, a sign-on dialog conducted with some machine in the recent past might cause messages from that machine to be treated with greater respect.

Origin indicators consist of

O1) the *network* on which the request arrived (supplied by the receiving hardware and software);

O2) any available information about the *route* the request has followed since it left its source (supplied by the networks it has passed over);

O3) the *source machine* (usually supplied by the network into which it was first launched);

O4) the *source socket* (supplied by software in the source machine).

These indicators have been listed in order of decreasing reliability, since each one introduces a new, possibly unreliable, supplier of information, and is also subject to corruption by the mechanisms involved in supplying the indicators above it.

How seriously each indicator should be taken, in deciding whether to affix a particular stamp, depends on a careful analysis of the properties of the various suppliers of information. It also depends on the level of confidence which the stamp is supposed to represent. Consider indicator (O3), for example. A message arriving on a dial-up telephone line carries no origin indicator of its source. One arriving on the Ethernet does carry a source machine number, but since the software on any Alto can supply any number, the indicator is not very confidence-inspiring (this is not a fundamental property of the Ethernet, by the way, but an accident of the Alto interface implementation). On the Nova MCA, however, the network

hardware supplies the source machine number, so the receiver can trust it if he is willing to discount the possibility of errors or tampering with the MCA hardware. Decisions about how much to trust the origin indicators are a matter of policy, and the protection system is constructed to accept such policy decisions as parameters.

Content indicators take more varied forms. All of them have the property that they depend only on the data bits in the received message, and not on any properties of the communication paths it has traversed (although these properties may influence the receiver's confidence in them). There are two basic types of content indicator:

C1) a *password*, which is a sequence of bits appearing at some agreed-upon place in the message;

C2) *encryption* of the message, which when decrypted contains a password.

In the case of encryption it is not particularly important that the password be secret, provided that the encryption key is secret. The main purpose of the password is to allow the receiver to be confident that the message is *recognizable*, i.e. that it was actually produced by someone who knew the key, and is not just a random collection of bits. It is important, however, that the decrypted password should depend in some complex way on all the bits in the encrypted message. Otherwise an enemy can copy the bits which determine the password from some legitimate message, and supply anything he chooses for the rest of the message.

The two schemes offer equal security if the communication path is secure. When it is not, encryption is essential, and can offer arbitrarily good security, provided a good algorithm is used and the key is changed sufficiently often. In addition to providing secure authentication, of course, encryption also performs the important function of concealing the content of the message from observers.

We feel that encryption will be an essential part of most secure distributed file systems in the real world. We do not, however, have any particular expertise in the design of encryption methods. Fortunately, a strategy for obtaining security through encryption divides naturally into two almost completely independent parts:

the encryption and decryption algorithms;

the doctrine for distribution and use of keys.

We propose to use some trivial encryption algorithm (which will be quite unable to withstand any serious attack), but to handle the keys in a realistic manner. Some later system which is nearer to a product can then readily pick up the proposed federal standard encryption algorithm and add the necessary hardware to implement it cheaply.

Initially, we propose to assign one key to each <user, system machine> pair, and to change the keys infrequently by some manual method. We have spent some design effort on more elaborate key distribution strategies, which will be described elsewhere and which might be implemented later if they seem worthwhile.

To summarize, an authentication procedure has at its disposal six indicators for the message it is considering, O1-O4, C1, and the encryption key used to successfully decrypt the message into a recognizable form. Based on this information it must make its decisions about what stamps to affix to the message. The job of the storage system is to allow the algorithm for making these decisions to be specified in a sufficiently flexible way.

3.1.3 Hard Protection

In order to make the hard protection as independent as possible of the details of how requests are processed, hard stamps are affixed to a request as soon as possible, and are checked against hard guards as late as possible. Thus, the first thing which happens to a request after it has been assembled from its constituent packets is hard authentication. The single hard stamp affixed to the request is carried along by the request wherever it goes, until the request causes a

physical disk access. At this point the stamp is checked against the single hard guard of the disk page being accessed. This guard is stored on the disk just in front of the data. The system attaches the same hard guard to every page of a file, but the checking does not depend on reading the guard from one place on the disk and the data from another. Thus the opportunity for errors which associate the wrong hard guard with the data is minimized.

The text of a hard stamp or hard guard has a very simple and somewhat awkward form. It consists of three parts:

a *compartment*, represented by an integer;

a *level*, also represented by an integer;

a *set*, represented by a bit-vector. Elements of the set are called *hard groups*, and the text is said to be *in* the groups which appear in its set.

There is a partial ordering on hard texts, defined as follows. If u and v are texts, then $u \leq v$ iff

compartment(u)=compartment(v), and

level(u)$\leq$level(v), and

set(u) is included in set(v).

When a hard stamp is checked against a hard guard, it passes the check if

text(stamp)$\leq$text(guard).

At this point, the earlier use of the word *awkward* should be clear.

The decision about what hard stamp to affix to a request is based on a comparison of the six indicators described in the previous section with a data base which parametrizes the hard authenticator. The exact form of this data base has not been specified, but it will be set up at system startup time and cannot be changed during normal operation of the system. Thus there are no commands which the user can give to the hard protection system, except to set the hard guard of a newly created file. The user's only interactions with hard protection are through certain fields which appear in every request:

In the un-encrypted or *clear* portion of the request:

(O1) delivering network;
(O2) routing information;
(O3) source machine;
(O4) source socket;
(C2) encryption key number.

In the encrypted portion:

(C1) password;
claim for hard stamp;
claim for hard principal (see below).

The hard authenticator examines O1-O4 and C1-C2, decrypts the message if necessary, consults its data base, and emerges with a maximum hard stamp and an integer h. If the claimed hard stamp is $\leq$ the maximum hard stamp, the authenticator affixes the claimed hard stamp to the request. If the claimed hard principal is equal to h, it affixes the soft stamp $\langle hp, h \rangle$ to the request. Finally, it affixes soft stamps corresponding to O1, O3 and O4. The significance of these soft stamps is described in the next section.

3.1.4 Soft Protection

Soft protection is more complicated than hard protection, both in the methods for affixing stamps to requests, and in the structure of guards on files. In particular:

a request can carry claims for several soft stamps, and hence can acquire several soft stamp impressions, provided the soft authenticator accepts the claims;

instead of having a single guard which contains the text of a single stamp, a file can have a list of soft guards, any one of which can authorize access (much like a chain of

padlocks, which can be opened by opening any one of the padlocks). Furthermore, each soft guard contains a set of soft stamp texts, and all the texts must be present on the request to satisfy the guard (much like a padlock which requires several keys to open it; actually, this arrangement is more common with safes);

a soft guard also contains a set of *permissions*, such as read and write. Each command which works on a file can require that a certain set of permissions for that file be granted before it will be carried out.

The text of a soft stamp is a pair: <*type*, *value*>. There are the following soft stamp types:

*network
*originating machine
*source socket
*hard principal
soft password
derived.

Stamps whose types are starred in the list are called *primary* stamps. All the primary stamps are attached by the hard authenticator, as described in section 3.1.3.

The value of a soft stamp is simply an integer. For primary stamps, the integer encodes the information extracted by the hard authenticator. The encoding is published, so that anyone constructing a soft guard knows exactly what soft stamp texts to include for the primary stamps he wants to demand.

A soft password stamp is affixed to any request which contains a claim for it. The claim, of course, includes the text of the stamp, which includes the integer value. The only information provided by a password stamp is that the originator knew that a particular integer was worth using; hence the name, password. Since the integers are fairly large (say 64 bits), this password gives about as much security as a typical login password for a time-sharing system. Perhaps we will apply a one-way encryption algorithm to the integer in the claim to obtain the stamp, so that a text can be compromised without compromising its password.

Derived soft stamps are objects in the system, much like files. Their purpose is to stand for groups of users. Thus, there might be a derived soft stamp called Personnel, whose text would appear on the guard list of files kept by the personnel department. The various members of the department would be given access to the Personnel stamp. The same effect could be achieved by putting the primary stamps for each person on all the guard lists, but this would be much less convenient and more error-prone, as well as being wasteful of storage.

A new derived soft stamp can be created in very much the same way that a file is created; the system guarantees that a stamp with that text has never been seen before and will never be created again. Each derived soft stamp is protected by a soft guard list exactly like the soft guard list of a file. If a request claims a derived soft stamp, the system decides whether to affix it by checking whether the soft stamps the request already has are sufficient to obtain affix permission for the claimed stamp, in exactly the same way that it decides whether to read a file by checking whether the request's stamps are sufficient to obtain read permission for the file. This process of affixing soft stamps can be iterated several times, if the primary stamps can obtain stamp *alpha*, which in turn is sufficient to obtain *beta*, etc.

There is a fixed, small *universe* of permissions (perhaps 32). The set of permissions in a soft guard is some subset of this universe (represented, obviously, by a bit vector of perhaps 32 bits). Different commands assign meaning to different permissions by published convention. Thus, for example, the ReadFile command interprets one of the permissions as read permission, and it insists that a request must obtain this permission before it will deliver any data from the file. A user who wants to allow or deny read access to his file can do so by constructing a soft guard containing the permission which ReadFile interprets as read. In this document we will simply refer to permissions by name, as we have done with read and write.

As we have already seen, the soft guard list for a file (or for a soft stamp) may contain several

guards. Each of these guards has a set of permissions which it can *grant* if it is *satisfied*. The set of permissions which the request obtains is the union of all the permissions granted by all the guards which it satisfies. A convenient way to think about this is that guards with the same permissions are "or"ed; satisfying any one of them is enough to obtain the permission.

A single soft guard contains one or more soft stamp texts, and they must all be present to satisfy the guard. Thus the guard tests for the presence of stamp *a* and stamp *b* and . . . The entire guard list, then, is roughly an "or" of "and"s of tests for the presence of particular soft stamps on the request.

There are commands to read and modify a soft guard list. These commands themselves require a **modify guards** permission.

3.2 Files

All user data stored in the file system is associated with some file. Each file is made up of a number of components; the most important component is the data, but there are other components which store the name, protection information, and other useful information. A user may read each of these components, and may modify most of them. In this section we describe the file components and their intended use. In the next section (3.3) we describe the commands available on files.

At any instant, each file has a state which completely defines its future behavior, i.e. what will happen as a result of user requests directed to the file. The entire system also has a state, which likewise defines its future behavior; the state of the system is simply the union of the states of all the files.

3.2.1 File identifier

The file identifier is a 64 bit number. No two files have the same number. The number is assigned by the system at the time the file is created, and cannot be modified. This identifier is not to be confused with a text name. One of the extended facilities (not a data storage system facility) is the ability to convert between text names and these identifiers. It is conceivable that the same file may have more than one text name, or that two different text names refer to the same file. In fact, this is likely to happen if there is more than one text naming context. File identifiers have been introduced in order to provide a unique, context independent, name for a file. An identifier may be passed around among users, and will always refer to the same file. Of course, knowing the name of a file does not grant a user access to the file.

3.2.2 Hard Guard

The hard guard contains the text of a hard stamp. This text is specified at the time the file is created, as one of the parameters in the file creation command. This stamp text will be recorded in all physical records representing the file. The text can never be changed throughout the life of the file.

All commands directed to the file after its creation must have an impressed hard stamp which is greater than or equal to, (in the partial order for hard stamp texts), the hard stamp text in the hard guard. This check will be made as each physical record is read from the disk.

3.2.3 Soft Guard List

The soft guard list is a sequence of *soft guards*, each of which is a pair consisting of:

1) a set of *soft stamp texts*, and
2) a set of *permissions*.

A permission is the right to perform a particular command on the file, e.g. to read-file-data-length. A transaction process uses the file guard list to determine which of the requested commands to perform. A requested command will not be performed, unless there is a guard <T,P> on the access list such that:

1) for each soft stamp text, *t* in T, the authenticated request carries a soft stamp with text equal to *t*; and

2) P contains a permission for the requested command.

3.2.4 File data

The file data is a sequence of bits. There are commands to determine the length of the sequence, to determine the value of particular bits in the sequence, to change the value of particular bits, and to change the length of the sequence. Ideally, these commands would accept bit addresses within the sequence. However, various efficiency arguments have been presented which suggest that the file should be treated as a sequence of bytes, of some fixed size yet to be determined. The sizes suggested vary from 1 bit, to the size of a disk page. Of course, there are several possible disk page sizes. Since this issue is still open, our description is couched in terms of bytes, and we do not specify the size of a byte. In the actual implementation this size could be as small as 1 bit, or as large as a disk page. Thus:

The file data is a finite sequence of bytes, addressed starting at 0.

3.2.5 Soft Stamp List

This is a sequence of the texts of soft stamps which have been impressed on the file. The intended use is for one user to impress a stamp on a file in order to certify to a second user that the file has some desireable property. For this certification to have the intended significance, the first user must have effective control over both

the use of the stamp which is impressed on the file, and

the contents of the file.

Otherwise, some other user may impress the stamp on some file which does not have the stated property, or the contents may change after the stamp has been impressed, and the file would no longer have the stated property.

The exact form of the list, and the exact commands available on the list, have not yet been specified.

3.2.6 File Interception List

The file interception list is at present only a suggestion. Its intended purpose is to allow selected commands on the file to be intercepted. A request for an intercepted command would not be performed, but instead would be sent to some place specified by the interception list. This might result in the command being interpreted by some other process, or simply being logged and then performed in the usual manner.

3.3 File Commands

The reader must recognize that what is presented here is only a preliminary specification for the commands. It is expected that as the design progresses, these descriptions will change. However, some general statements can be made. First, the commands naturally divide into groups, one for each of the components of a file, and one for commands on the file as a whole. Within each group there are generally commands which read the state of the component (or a portion of the state) and commands which modify the state of the component.

For most components, there will be enough commands so that a user can obtain a complete representation of the state of that component, and can set the state to any desired value. In

general, several commands may be required to accomplish these goals. Further, a user must have sufficient authority (stamps on his requests).

For two components it is not logically possible to offer these abilities: the unique identifier and the hard guard text. The unique identifier never changes (it is the name of the file); hence there is no command to modify its value. Also, since it is the name of the file, it must be included in any request directed to the file. Thus, there is no use for a command to read its value. The hard guard is similar, it never changes in value, and thus there is no command to change its value. It can, however, be read.

A user requests that a command be performed by sending a request (section 3.5) to a transaction (section 3.4). Each request includes information to identify the request, along with a number of stamp impressions. In the following descriptions we suppress all of this information, and only mention those parameters required to specify an individual command.

The system returns a response to each request. The response acknowledges successful completion of the request, and also contains any information which the request returns. Alternatively, the response tells the user why his request could not be carried out. Of course, it is possible for requests to be lost, and for responses to be lost. The user must therefore be prepared to repeat his requests; section 3.4.8 describes the algorithms which users should employ to get their work done in spite of system crashes or other failures.

The description of commands below have the following form:

Name of command(parameters) **returns** results

followed by a description of the command. Parameters and results have names of the form <type><modifier>, where the optional modifier serves to distinguish different instances of the same type. The following basic types are used:

bt	byte (of data)
fi	file identifier
ht	hard text
pm	permission
sg	soft guard
st	soft stamp.

In addition, there are *type constructors* which can be prefixed to existing types to obtain new ones. The meaning of the constructors used here should be clear enough from their names:

a	address
i	index in an array or sequence (an integer).
l	length (an integer)
n	number of items (an integer)
seq	sequence. The elements are normally numbered starting from 0. A sequence includes its length; the length of a seqFoo will be a lSeqFoo.
set	set

There are several commands which takes an index i, or an index and a number n which together define a sequence of index values. For example, to read data from a file a command must specify the index of the first byte and the number of bytes to be read. Such a command normally makes a standard adjustment to ensure that the indexes it uses will not be larger than the length l of the array. This adjustment is

$i \leftarrow \min(i, l)$; $n \leftarrow \min(n, l-a)$.

In addition, when a command involves writing a sequence of items, it is possible that the request does not contain enough items; e.g. the request which says: "write 10 bytes, starting at byte 1122," might contain only 8 bytes of data. If this happens, no changes are made to the file, and an error response is returned. The descriptions of commands below make no further mention of these standard checks.

We begin with a description of the commands on a file as a whole, followed by the commands for each component, grouped into subsections numbered to correspond with the subsections of 3.2 in which the component is described.

3.3.1 Commands on the entire file

CreateFile(ht, setSt) returns fi

> Ht is the text of a hard stamp and setSt is a set of texts of soft stamps. A new file is created (i.e. a previously unused file identifier is assigned). In the new file, the hard guard contains ht, and the soft guard list contains a single pair, <setSt, pmAll>, where pmAll is the set of all permissions.
>
> No specific impressions are required on the request which authorize the use of ht and setSt. Thus, it is possible for a user to create a file which he can not subsequently access.
>
> The result is the file identifier assigned to the newly created file.

DestroyFile(fi)

> The file specified in the request is destroyed. The identifier of the file will never be used again.

SetRedundancy(fi, rd)

> Sets the level of redundancy to be used in storing the file on the system disks.

3.3.2 File identifier Commands

There are no commands on the identifier, since any request to read the identifier must be acompanied by the identifier in order to identify the file, and the identifier can not be changed.

3.3.3 Hard Guard Commands

ReadHardGuard(fi) returns ht

> Returns the hard stamp text stored in the hard guard.

3.3.4 Soft Guard List Commands

We have not made a strong committment to a particular set of soft guard list commands. The following set is believed to be logically sufficient, however:

ReadGuardListLength(fi) returns lSeqSg

> Returns the number of guards on the list.

SetGuardListLength(fi, lSeqSg)

> Sets the guard list length to lSeqSg. If this lengthens the list, the new guards are empty. If this shortens the list, guards with high indexes are discarded.

ReadGuard(fi, iSg) returns sg

> Returns the contents of the guard with address iSg. The representation to be used in the response for the value of a guard pair has not yet been chosen.

SetGuard(fi, iSg, setSt, setPm)

Sets the guard with address iSg. SetSt is the set of soft stamp texts to be placed in the guard. SetPm is the set of permissions to be placed in the guard. We have not as yet chosen representations for setSt and setPm. However, the representation will be some sequence of bits and will not, for example, include stamp impressions for elements of setSt.

As for any other command, there must be sufficient impressions on the request so that the current soft guard list permits the command SetGuard itself.

3.3.5 Data Commands

ReadFileDataLength(fi) **returns** lSeqBt

Returns the number of bytes in the file data.

SetFileDataLength(fi, lSeqBt)

Sets the file data length to lSeqBt bytes. If this is larger than the previous value, the new bytes, which are at high addresses, will have a value of 0. If it is smaller, bytes with high addresses are discarded.

ReadBytes(fi, iBt, nBt) **returns** seqBt

Reads nBt data bytes from the file, starting at byte iBt.

WriteBytes(fi, iBt, nBt, seqBt)

Writes nBt data bytes into the file, starting at byte iBt.

3.3.6 Soft Stamp List Commands

The exact commands have not yet been specified.

3.3.7 Interception List Commands

The exact commands have not yet been specified.

3.4 Transactions

In order to explain the facility we call transactions, we begin by describing the use of a system which does not have the facility. A user program interacts with this restricted system by sending individual commands to perform a single read or write. Eventually, the system sends responses to the user indicating the result of each command. While waiting for a response, the user is at liberty to send other commands.

These commands are not like subroutine calls in a programming language, since there is no way the user can tell exactly when the command is executed. There is some delay from the time the using program sends a command until the system receives it, some further delay until the command has been executed, and still further delay before the user program receives the response. If the user sends several commands, the order in which they are received by the system may not be the same as the order in which sent. Thus, they may not be executed in the same order as they were sent. Further, a command could be lost in transit, or the response could be lost in transit. Thus, if a user receives no response to a command, it is not clear whether the command has been executed.

Consider a user who maintains a data base containing several money accounts. A typical task for this user is to move some money from one account to another. As input, the user is given the name of each account, and the amount of money to move. The user will proceed by first reading the contents of the two given accounts; then computing the new values for these accounts (checking for overdrawn accounts) and finally writing the new values into the data base.

There are at least two ways in which a naive user could fail at this task. The first failure mode involves two independent users maintaining the same data base, and carrying out simultaneous updates to the data. If these two users are not "careful", they may attempt to simultaneously move money from the same account. In particular, they may each compute the new balance by subtracting some amount of money from the same original balance. The result will be that only one of the two amounts of money is actually withdrawn from the account. Thus the balance will be too high.

This failure can be prevented if each user always checks to be sure the other is not modifying an account balance. This check is usually implemented through the use of *locks*, arranged so that if one user has an account locked, the other cannot read or write it. Now a user first locks both of the accounts, then reads the current balance, computes the new values, writes the new values into the data base, and finally releases the locks. This scheme is usually refined to allow two kinds of locks, one for data which is only to be read, and the other for data to be modified. This refinement usually reduces the chance for conflict between two independent users.

The second failure mode involves an inopportune system crash. It is possible for the system to crash after a user has modified one account balance, but before the other is updated. Unless some complicated provisions have been made, the unfinished update will not be made when the system is restarted. Thus, the accounts could be left in a state where one account has been incremented and the other has not been decremented.

Now consider another user whose task is to display on a screen the current balance of several selected accounts. The selections are presumably made by an operator, and change from time to time. Certain important accounts may remain selected for long periods of time, e.g. all day. We assume it is important that an incorrect balance never be displayed. A possible scheme for this user is to continuously re-read the account balances which it is displaying. This scheme has two obvious drawbacks. In the first place it generates a lot of communication traffic, constantly re-reading the same values. In the second place, the values displayed are always just a little bit old. That is, there will be some (small) delay from the time an account balance changes until the new value is displayed; moreover, during this time

an incorrect balance is displayed.

An improved technique is to read-lock any balance to be displayed on the screen, before reading the account. However, this prevents any other program from updating the account. A better method is to mark the account so that whenever another program tries to update the balance, the display program will be informed. If the message arrives early enough, the display program can temporarily remove the account from its screen, and replace it with the new value when it is known. Some difficulties may still arise; for example the warning message may be lost, so that the display program never learns that the balance has changed.

The above discussion was not intended to prove that it is impossible to write user programs which acomplish the stated ends, but simply to convince the reader that it is difficult. Further, these programs will require some facility which provides the appropriate lock mechanisms. We could have chosen to provide such a lock mechanism, and leave the details of correct use to the users. Instead, we have decided to provide an interface which (we believe) handles all of the difficult problems mentioned above.

We describe this interface as a *transaction* interface. We use the word *transaction* in two different ways:

as a noun, which names a collection of commands from users, in which the data written is a function of the data read, and

as an adjective, to describe the various components of the system which support the interface.

The transaction interface permits the user to specify that a group of otherwise independent commands (a transaction) is to be treated as a single *atomic* action. The mechanism maintains three properties:

1) (The atomic property) Each transaction appears atomic relative to other transactions. That is, consider the set of all commands contained in all transactions which properly finish. Then there is an order of execution for these commands such that all of the commands of each transaction occur with no intervening commands from other transactions, and:

a) each command leads to the same response to the user as in the actual execution, and

b) the final state of the system is the same as in the actual execution, i.e., any subsequent reads lead to the same response to the user as they would after the actual execution.

2) (Consistency over system crashes.) Each transaction either completely finishes, so that all write commands are carried out, or aborts, so that no write commands are carried out. This property is maintained in the presence of (repeated) system crashes.

3) (Cache property.) By following a satisfactory algorithm (described below) a user can maintain a local representation (cache) of a portion of the data stored in the file system. This representation will be faithful. That is, if any user initiates a transaction to modify some data which is represented in the cache, and sends a message to the cache maintainer when the transaction is reported by the system to be complete, then at the time the cache maintainer receives this message the old value of the data will no longer be represented in the cache. More generally, there is no way for the cache maintainer to see inconsistent values for the data.

As an alternative to the above statement of the atomic property, it is tempting, but misleading, to say that there will be some "instant of truth" for each transaction. At this instant the results of any reads in the transaction are still true (i.e. the same results would be obtained by another read, unless some of the data was rewritten by the same transaction), and the effect of any writes in the transaction is true (i.e. the data which was written would be retrieved by reading

from those addresses). Although it is true that the atomic property is implied by the existence of such an instant of truth, this way of describing the atomic property is bad for two reasons:

It introduces the idea of simultaneity, i.e. of a system-wide "instant", which is otherwise unnecessary.

It is not really necessary for any "instant of truth" to exist, since it is all right to allow the locations read by transaction A to be modified by another transaction B, as long as the locations being written by A are not allowed to be read by anyone else until A's writes are complete.

The atomic property is accomplished through the use of conventional read and write locks, together with a time out mechanism. The locks serve to delay the execution of some commands, while the time outs lead to the aborting of some transactions. The commands of those transactions which are not aborted will be executed in such an order as to satisfy the atomic property.

The implementation imposes a stronger condition on the execution sequence than that implied by the atomic property. Consider the sequence of commands:

Begin transaction A
Begin transaction B
A reads p
B reads p
B writes p
A writes q
End A
End B

In this case, the implementation will delay the execution of the "B writes p" command until after "End A", even though this delay is not logically necessary.

While there is no "instant of truth" as described above, the actual implementation does impose a pseudo "instant of truth". That is, there is some instant during the completion of a transaction at which time:

The data obtained from the read commands ot the transaction is still correct. (At least for those read commands with unbroken read locks.)

The data to be modified by write commands is write locked, and will remain so until the modifications to the locked data are complete.

This pseudo instant of truth is not a real instant of truth for two reasons: the write commands have not yet been carried out, and the system my decide (later) to abort the transaction.

The property of consistency over system crashes is obtained by delaying all actual modifications to the stored data until the transaction is completed. At this time a list of all the necessary modifications is stored on the disk as a single act, and then the modifications are carried out. If the system should crash before the modifications are complete, the crash recovery mechanism will find the list of modifications, and begin again to carry them out. This idea is explained in more detail in section 6.

Except for one problem a user could implement a cache through the use of a transaction. The user opens a transaction, and obtains any information to be stored in the cache through read commands in the transaction. The atomic property of the transaction mechanism guarantees that no other user will be able to modify the original of any data stored in the cache. However, the system will not allow a transaction to hold data locked indefinitely. A read lock will eventually time out, and then if another transaction attempts to modify the locked data, the transaction holding the lock will be aborted.

A read lock which has timed out and is impeding another transaction is called "broken". Instead of aborting a transaction wich is holding a broken read lock, the user is given the oportunity to "release" the lock. By releasing the lock the user declares that no write command in the transaction is dependent on the data read under the authority of the lock. A user who is maintaining a cache will release all broken read locks, and remove the associated data from the cache.

Since it is possible for a message from the system to a user to be lost, a command is provided to test for broken read locks. A negative response from the system to this command indicates that no locks have been broken.. A user who is maintaining a cache will periodically ask the system if there are any broken read locks.

Yet one more feature is required to provide the cache property. A transaction which causes a read lock to be broken is prevented from completing until a certain time has elapsed from the moment the lock is broken. This time period is a system wide constant, T. Now if the cache maintainer has sent an inquiry to the system, and received a response indicating that there are no broken read locks, then the cache maintainer knows that the data stored in his cache is consistent for a period of time, T, beginning at the moment the inquiry was sent to the system.

There are a number of different kinds of objects involved in a transaction. These include users, requests and commands, transaction processes, read locks and write locks. Transaction processes are the active agents which reside in the file system and carry out the commands of a transaction; each one is associated with a single transaction. There are two kinds of transaction processes: *workers*, and *coordinators*. The following scenario should give a rough idea of their relationships.

A user begins a transaction by sending an OpenTransaction command to some computer in the system. The system on that computer constructs a resident coordinator to control the transaction. The coordinator sends a response, which includes the name of the new transaction, to the user. The user then sends a create-worker command to the coordinator, and indicates on which system computer the worker should be resident, (usually the computer on which the coordinator is resident). The coordinator sends a response to the user.

The user can now send file commands (various forms of Read and Write) to the newly created worker. Further, the user can at any time command the coordinator to create a worker on another computer. We require that any command which addresses data must be sent to a worker residing on the computer which stores the addressed data. As a result, the user (or his agents) will have to create additional workers if his transaction involves data on more than one machine. The worker handles a Read command by attempting to set a read lock on the data. After this succeeds, it reads the data and sends a response to the user. The read lock remains set until the transaction terminates, (unless something unusual happens). A Write command is handled similarly, except that a write lock is set.

Finally, the user sends a CloseTransaction command to the coordinator. The coordinator then synchronizes with the various workers (of this transaction) and eventually completes the transaction. At this time a response is sent to the user.

During the transaction, various other things can happen. For example, a user of the transaction can command the coordinator to AddAUser. Another possible event is the time-out of a lock. If this is a read lock, the appropriate user is informed, generally before the lock is actually released. The user can now choose to Abort the transaction, or to re-read the pertinent information, and recompute what it should do. If this involves changing something which has already been written, the user must issue new Write commands which make the necessary changes. If a write lock times out, this generally results in a system initiated abort of the transaction.

Another possibility is for a user to send a Checkpoint command to the coordinator. The result is equivalent to sending a CloseTransaction command, and then opening a new transaction and

re-reading all of the information the user held at the end of the old transaction. The Checkpoint avoids the necessity for re-reading and re-setting the various locks. Write locks existing before the checkpoint are converted into read locks after the checkpoint.

Throughout the discussion we have used phrases such as "Read command", "Write command" and "data". In this section we blur the distinction between the various components of a file state. By a Read command, we mean *any* file command which can send information about the state of the file (including its components) to the user, without modifying the file state. By a Write command we mean any command which can result in a change in a file state. Finally, by data we mean any portion of the file state. This blurring of terminology will continue throughout this section.

For efficiency, a user sends commands in groups, called *requests.* Each request is directed to a particular file system process, and contains a set of commands together with sufficient stamp impressions to authorize the commands. In general a single request can contain commands for more than one process. These processes must all be on the same file system computer, and involved in the same transaction. For example, the request which contains the OpenTransaction command (directed to a file system machine, rather than a particular transaction process) can contain the command (directed to the as yet uncreated coordinator) which causes the creation of a worker. In the case that this worker is on the same computer as the coordinator, the request can also contain commands to be directed to the worker.

An important feature of the transaction implementation is that the detailed system state is viewed as representing two different abstract states. We call these two abstract states the V view (of the system state) and the S view. V stands for volatile and S stands for salvageable or solid or stable. After a system crash, enough of the system state will be reconstructed so that the new S view is the same as the S view before the crash. The V view, however, is completely lost, and is replaced by a new V view constructed from the S view. Thus the essential property of the S view is that *the S view is not affected by a system crash.* A large part of the system is devoted to making this property hold.

The V view of the system state contains all of the transactions which are in progress, their partial modifications to the files, and any locks they have set. The S view describes the state the system would have if all transactions currently in progress were aborted and their partial changes to the file states were forgotten. The V view after a crash is the S view with no transactions in progress.

In the remainder of this section we present a more detailed description of transaction processes, locks, and algorithms which should be followed by users to accomplish desired ends.

3.4.1 Requests

A user (section 3.4.3) makes things happen by sending *requests* to a file system computer. Each request contains one or more commands, and each request, except for the first request of a transaction, is directed by the user to a specific transaction process (section 3.4.2).

A request, as seen by a transaction process, contains the following information:

- Transaction identifier
- Transaction process identifier
- User
- Sequence number (within the (user,transaction process) pair)
- Hard stamp text which has been impresed on this request
- Set of soft stamp texts which have been impressed on this request
- Set of commands, each of which contains:
  - Command name
  - File unique identifier

Zero or more additional parameters

In a number of cases, some of these fields have no meaning. For example, the request which opens a transaction cannot name the transaction identifier, and actions which do not involve files cannot be expected to specify a file identifier. In these cases special information is required for the affected fields.

The request is delivered to the specified process. If the first command in the request results in the creation of a process on the same computer, the request (modified by the removal of the first command) is redirected to the newly created process. If this in turn results in the creation of a new process on the same computer, the request (modified again) is again redirected to the new process.

3.4.2 Transaction processes

Each transaction process lives on a single system computer. Processes do not move from one computer to another. Transaction processes are created and destroyed in response to user commands. There are two kinds of transaction process, coordinator and worker. Each transaction will have exactly one coordinator, and one or more workers. For each transaction, there will be one worker on each computer which contains data referenced during the transaction. This includes the computer which contains the coordinator.

Generally, requests containing commands which control the logical progress of the transaction will be addressed to the coordinator, while requests containing only data commands will be addressed to the worker on the computer which contains the data. Note that requests do not have to pass through the coordinator to reach the workers.

3.4.3 Users

Requests are sent from a *user* to a transaction process. A user is an abstraction invented to stand for a source of requests. Responses to commands in a request are sent to the user which sent the request. We consider the source to be a program, because we generally assume that the commands in a transaction are being constructed by some algorithm. In order to make full use of our facilities, this algorithm will have to satisfy some conditions (see [3.4.8]). A user is assumed to reside on one computer, and in many respects, it acts like a process.

A single transaction may have more than one user. These different users need not reside on the same computer. If a single transaction has more than one user, these programs may have to co-operate closely. For example, we provide no lock protection between requests sent to the same transaction by different users. The reason for taking this view of users is so that a computation which is using the storage system can itself run on several machines, using its own methods for coordinating its activities.

A single user may have several transactions in progress. Each request sent by the program identifies the transaction to which it belongs. A single computer may contain more than one user. User written programs will not reside on file system computers, for the reasons explained in section 3.1. However, some system programs, running on file system machines, will act as users.

3.4.4 Locks

The file system partitions the file state into lock regions. A command to access any information in a region results in a lock on the whole region. These regions may, for reasons of efficiency, be larger than a byte. Each lock specifies:

a region,

read or write,

the transaction which set the lock,
the user who sent the request resulting in the lock,
a time of expiration

A single region may have more than one lock. There may be more than one lock on a single region from a single transaction. In fact, there will be one lock for each independent command referencing information in the region. If any lock on a region is a write lock, however, all locks on the region must be from the same transaction.

The purpose of locks is to prevent two transactions from interfering with each other. A transaction process will be prevented from setting a lock which conflicts with a lock set by a different transaction. (Note: not merely by a different transaction process.) These locks can, and will, lead to deadlocks. To counter this possibility, we include a mechanism for forcing locks to release, even if a transaction is not yet complete.

Whenever a lock is set, an expiration time is recorded in the lock. Nothing happens until both the expiration time has been reached, and some other transaction is waiting to set a conflicting lock. At this time the lock *times out*, but is not yet released. An internal system message is sent to the transaction process responsible for the timed out lock; note that this process is guaranteed to be running on the machine which is storing the lock. It is the responsibility of this process to release the lock, and in most cases this will occur within a reasonable time. There is one situation described below, in section 6.1.1, in which the release may be delayed for a long time. In any case, a read lock will not be released until T seconds (a system wide constant) have elapsed from the instant the lock timed out.

If the timed out lock is a read lock, the transaction process immediately sends a message to the user who commanded the read which caused the lock to be set. That user then can

abort the transaction
re-read the information and re-compute the modifications it wants to make
decide that it doesn't care about the continued validity of that information and proceed regardless.

In any case, the transaction process will release the read lock T seconds after the lock timed out.

If the timed out lock is a write lock, and the transaction is not in the process of completing, the transaction is aborted, the lock released and a message sent to the appropriate user. If the transaction was in the process of completing, it will usually finish before the lock is released.

3.4.5 Request sequences

Each request is a transaction belongs to a *request sequence*. There is one sequence for each (user,transaction process) pair. When there are several users and several processes for a single transaction, probably most of these sequences will be empty. Each request specifies the sequence to which it belongs (by the pair <user, transaction process>) and its position in that sequence (numbered 1, 2, ...). In this way, it is possible to recognize, when one request is received, that a previous one is missing. Further, the final request to close the transaction is sent to the coordinator, and includes the counts of requests in all non empty sequences. Thus, it is possible to discover whether any requests have been lost.

3.4.6 Data types for transaction commands

A number of new data types are involved in the commands described in the next few sections, and they are tabulated here for reference. Section 3.3 contains a list of data types introduced so far.

co coordinator
mi machine identifier
sn sequence number

ti transaction identifier
li lock identifier
tp transaction process
us user
wk worker

3.4.7 Transaction control commands

Commands are available to control the transaction, which only indirectly affect the data stored in files. Most of these commands must be in requests directed to the coordinator (we shall indicate those which are directed elsewhere).

OpenTransaction(us) returns [ti, co]

Obtains a new ti, creates a coordinator for the new transaction, and returns to the user the new identifier and the name of the new coordinator.

This must be the first command of its request, and the request must be directed to a file system machine, rather than to a transaction process. The request may contain other commands to be performed by the new transaction (if it does, they must include at least a command to create a worker). If the request contains more than one command, the first command will be deleted from the request, and the modified request sent to the newly created coordinator.

CloseTransaction $((us_1, tp_1, sn_1), \ldots (us_k, tp_k, sn_k))$

The coordinator (through its workers) checks to see that all requests from user us_i to process tp_i containing sequence numbers less than or equal to sn_i have been received. If not, a warning messge is sent to the user specifying the lost requests. This is done for i = 1, . . . k. If this list of user-process pairs does not include all pairs with non-empty request sequences, then an error response is sent to the user.

Next the coordinator (through its workers) waits for all commands in the named requests to complete. When all are complete, all locks are released and a response is sent to the user.

CreateWorker(mi) returns wk

The coordinator creates a worker on the named machine. The coordinator then sends a response to the user with the tp of the new process.

This command must be the first command in its request, as received by the coordinator. If there are more commands in the request, this first command is deleted from the request, and the modified request is sent to the new worker.

CheckPoint$((us_1, tp_1, sn_1), \ldots (us_k, tp_k, sn_k))$

Proceed as for CloseTransaction, until all commands are complete. Then the coordinator (through its workers) converts all write locks (which have resulted from the commands in the transaction) into read locks. (This has almost the same effect as if the user closed the transaction, opened a new transaction, re-read all data read in the previous transaction, and re-read all data modified in the previous transaction. Asside from efficiency, the difference is that pending write commands from other transactions can not occur.)

Finally, a response is sent to the user.

AreYouStillThere returns seqLi

This command can only be sent to a worker, which rsponds with a list of broken read locks (perhaps null).

ClearReadLock(li)

The specified read lock is released. If the lock had been previously broken, it is removed from the set of broken read locks. Once released, it is possible for other transactions to modify the data which was locked. Therefore, the read command which caused the lock is assumed to be not included in the transaction.

AddAUser(us)

The specified user is added to the transaction. Requests sent by the new user to transaction processes will be honored.

AbortTransaction

All read and write locks are released. All write commands are forgotten A response is sent to the user. All transaction processes are destroyed.

3.4.8 Messages sent from a transaction process to a user

Responses to commands

A response is sent to the user for each command in each request. For efficiency, these responses will be batched in larger messages. If there are several users for a transaction, the response is sent to the user who sent the request. In some cases there will be information in the response (e.g. a response to a Read), in other cases the response will only signify that the command has been noted and will take place (i.e. change the S view) when the transaction finishes (e.g. a response to a Write).

Timed out read lock

Each read lock has an associated termination time. If a lock has passed this time, it is said to be timed out. Nothing happens until some other transaction is impeded by the lock. When this occurs, the lock is broken and a message is sent to the user who requested the Read which caused the lock to be set. The data covered by the lock will not actually be modified until a time T has elapsed after the lock is broken. (T is a system wide constant). The fact that the lock is broken will be recorded by the worker process involved. The transaction will not be allowed to successfully finish until the user who set the lock explicitly releases the lock.

Unexpected Abort

Each write lock has an associated termination time. If a lock has passed this time, it is said to be timed out. Nothing happens until some other transaction is impeded by the lock. When this happens, the transaction which set the lock is aborted, and a message is sent to each user of the transaction.

A transaction is also aborted if the system crashes on one of the file system machines involved in the transaction. In this case, the user may not be informed.

3.5 User algorithms

In order to obtain the full services of the transaction machinery, a user must follow certain conventions. That is, the properties claimed for the transaction mechanism assume that the user behaves in certain ways.

For the purpose of this discussion, the user commands can be partitioned into three broad classes:

Transaction control
File read
File write

The transaction control commands are all those described in section 3.4.7. The file read commands are all other commands which can not result in a change in a file state. Finally, the file write commands are those remaining.

Outstanding commands will be carried out in an unpredictable order. Thus, if it is desired that the execution of one command definitely precede another, they should be sent in separate requests, and the user should wait for an appropriate response to the first command before sending a request containing the second. This remark does not apply to the Checkpoint and CloseTransaction commands, since they automatically wait until the specified commands are complete.

3.5.1 Combining Commands into Requests

3.5.2 Basic User algorithm

If there is only one user, all the desired data is on a single file system machine and there are no unusual circumstances, then the user proceeds as follows:

Send a request to the file system machine on which the desired data is stored. This request contains the following two commands:

OpenTransaction
CreateAWorker on the same machine

The response to this request will contain the names of the co-ordinator and the worker which have been created to handle this transaction. The sequence numbers on the requests sent to each transaction process form an independent sequence. In this case there are two transaction processes, the co-ordinator and the worker. The sequence numbers on successive requests sent to the co-ordinator will increase by one, as will the sequence numbers on successive requests sent to the worker. The first request sent to each transaction process will have sequence number 1.

Now read the desired data from the files on the file system machine by directing requests containing Read commands to the newly created worker.

When sufficient data has been read, compute modifications and send requests containing Write commands to the worker. These requests can contain additional Read commands if so desired. Continue to send Read commands and Write commands as desired.

When all of the desired commands have been sent to the worker, terminate the transaction by sending to the co-ordinator a request containing a CloseTransaction command. This command will contain the sequence numbers of the last requests sent to the co-ordinator and to the worker.

The co-ordinator will eventually respond with a message indicating that the transaction has been correctly closed.

3.5.4 Multiple file system machines

A user must be aware of the existence of multiple file system machines for two reasons. First, for *each* file, he needs to find the machine which contains that file. Second, in certain unusual circumstances, one file system machine may crash, and its duties be taken over by another. It has not yet been decided what mechanism will be provided for locating files. However, there will be some facility which will map from a file identifier to the machine containing the file.

If portions of the desired data are on different machines, then the above basic algorithm must be modified. Additional commands are sent to the co-ordinator to CreateAWorker on other file system machines. These commands can be included in the original request, if the identity of the other machines is known in advance. However, if the identities of other machines are determined during the transaction, these commands are included in additional requests sent to the co-ordinator after the first request.

With many worker processes involved, the user must maintain independent request sequence numbering for each transaction process. There is a sequence for the co-ordinator and a sequence for each worker. The CloseTransaction command must include the maximum sequence number on the last request sent to each transaction process.

3.5.4 Multiple users

If more than one user is to be involved in a single transaction, one of the users will open the transaction. Commands can now be included in requests to the co-ordinator which (perhaps repeatedly) AddAUser to the transaction. These users must provide their own interlocks when accessing and modifying overlapping data, since the system provides no interlocks between actions occuring in the same transaction.

Each user provides independent sequence numbers on requests to each transaction process. The CloseTransaction command must include the maximum sequence number occuring in each sequence of requests between a user and a transaction process. In principle, there is such a maximum sequence number for each (user, transaction process) pair. In fact, only non zero maximum sequence numbers need be included.

3.5.5 Read lock time out

In the event that a read lock times out, and some other transaction is actually impeded by the lock, then the system will "break" the lock, allowing the other transaction to proceed. If this happens, a message will be sent to the user whose lock is broken, and the fact of the broken lock will be recorded by the worker. A later request to the co-ordinator to close the transaction will result in the abort of the transaction.

The user may avoid the abort by sending a request to the worker to release the read lock, *before* attempting to finish the transaction. Since the data covered by the broken lock will have changed by the time the transaction finishes, the consistency guarantee is no longer valid. However, the guarantee still covers all of the other reads of the transaction. In complicated situations, the most reasonable course may be to abort the transaction and start over. Another aproach would be to re-read the affected data, after releasing the broken locks. If the user follows this policy, he must be aware of several complications.

The broken locks must be released before re-reading the affected data. This follows from two facts:

the broken locks must be released before finishing the transaction (else the transaction will abort),

if the locks are released after the re-read, then the new locks set by the re-read will also be released.

The re-read will be held up until the other transaction (which caused the locks to break) finishes. This may take some time, and other locks may time out in the interim.

If the new values of the data lead to the same modifications as the old values, then it is reasonable to finish the transaction as before. One the other hand, if the new values lead to different modifications, the user can proceed by requesting writes which both accomplish the new modifications and cancel the old modifications.

3.5.8 User Maintained Cache

One intended application for this file system is support of a user who maintains a local representation (cache) of a portion of some files stored in the system. For example, in an accounting system, such a user might maintain a display of the current balance of certain accounts.

A user maintains such a cache by opening a transaction, and then reading all the information to be represented in the local cache through the transaction. This places read locks on the data. Thus, the data will remain unchanged until a read lock is broken. Note that in general, if the local representation is to be maintained for a long period of time, all of the read locks will eventually time out. However, they will not break until some other transaction attempts to change the locked data.

If a read lock is broken, the data still does not change for the time period T. The system will send a message to the user when the lock is broken; however, the message may be lost. This difficulty is overcome through the use of the AreYouStillThere command. At regular intervals the user sends this command to each worker process involved in the transaction. The transaction responds with a message which lists any broken read locks.

Now, if the AreYouStillThere command is sent at time t_0 to a worker, and a response is returned which shows no broken read locks, then all data read through that worker is valid until time t_0+T. If no response is received before t_0+T then all information read through that worker is assumed to be invalid. If a response is received which shows certain read locks to be broken, then the data depending on those locks becomes invalid at time t_0+T.

In the event that some read locks are broken, the user program proceeds by first releasing the broken locks, then re-reading the affected data. When the other transaction finishes, its write locks will be released and the new reads will succeed. The new values of the data will be returned to the user program, which can store it in its local representation. In the meantime, it must have the affected data represented by some special flag which indicates the the correct values are unknown.

The reader should note two facts. First, the AreYouStillThere command must be sent more frequently than once per time interval T, to allow for the delay between sending the message and receiving the response. Second, the time interval T is to be counted from the time the AreYouStillThere command is sent, not fom the time the response is received. This follows from the fact that there is no way to determine for sure exactly when the system begins to count the period T, except that it must begin after the AreYouStillThere command is received by the file system machine.

3.5.9 Missing system responses

The user must always be alert to the problem of a lost message, which is usually manifested through a missing response to a user request. Such a response may be missing because

the request did not reach the file system machine

the file system machine crashed

before it could handle the request

while processing the request

after processing the request

the response itself my have been lost in transmission.

For requests to workers, simply resending the request with the same sequence number will suffice. It is essential that the same sequence number be used, since if some sequence numbers have not been seen when the transaction is closed, the transaction will be aborted. Moreover, it does not hurt to resend the same request, since repeating any read actions can not hurt, and all write actions have been designed to be repeateable without changing their effect.

3.5.10 Unexpected abort

The file system may abort a transaction at any time before it is finished. Reasons for an abort include:

timed out write locks,

broken read locks which have not been cleared,

lost requests,

system crashes.

The user may not even be informed of an abort, but may have to infer that one has occured when the file system responds to a request with "Never heard of that transaction". If a transaction is aborted, the user is expected to open a new one and try again.

A system crash may occur at any time. The usual result is an abort of all unfinished transactions. The user will not be informed. If the user has not issued a CloseTransactionCommand, he will discover that the transaction has been aborted when he receives a response of the form "Never head of that transaction".

However, if the user has sent a CloseTransactionCommand, it is impossible to discover whether the transaction completed before the crash. There are two possible solutions, and we have not selected one. One solution is for the user to perform a recognizable change to a piece of data stored in the file system which is known to be private. This data can then be checked after a system crash. The other solution is for the system to maintain a correct list of completed transactions.

3.6 Authentication

3.7 Request Assembly

CHAPTER 4. INTRODUCTION TO THE ALGORITHMS

The algorithms which implement the claims of the preceeding chapters are described in the following chapters. In this chapter we explain some of the general issues which were considered during the design of these algorithms.

4.1 Lost messages

The file system runs in more than one computer. Thus, there will be communication between activities in different computers. This communication will be accomplished by sending messages from one activity (process) to another. We admit that there is some chance (hopefully small) that the message will be *lost*. However, we do assume that no message will be incorrectly transmitted. That is, there will be sufficient error checking in the message transmission machinery to detect garbled messages (which are usually converted into lost messages).

Thus, in order to detect the occurrence of a lost message, algorithms which send messages to initiate remote actions will expect a response message from the recipient. Three kinds of response are possible: "I have completed the requested action", "I could not perform the requested action" and no response.

The last possibility, no response, can be due to any one of several events:

The message containing the request may have been lost

The system containing the recipient activity may be down.

The system containing the recipient activity may have gone down when the requested action was partly done.

The system containing the recipient activity may have gone down after the requested activity was complete, but before the recipient could send a response message.

The response message may have been lost.

The requesting activity did not wait long enough for a response.

In order to deal with the possibility of no response, the requesting process must detect the lack of a response. This is usually done by means of a time-out. Upon the occurrence of such a time-out, the request is resent, and a new time-out period initiated.

Since the first request may have been correctly received, the recipient must be prepared to deal with duplicate requests. In the case of activities which can be re-performed without affecting the system state (e.g. reads), the recipient can ignore this problem of duplicate requests, and simply re-perform the action. However, if duplicating the activity would lead to incorrect results, the recipient must *not* repeat the activity, but still must send a correct response. All writes require this treatment, for the following reason: although writing the same value *V* twice is harmless in general, it is disastrous if the user followed the sequence

write *V*
wait for acknowledgement

write *W*.
and a duplicate "write *V*" arrives after *W* has been written.

If the requested activity is lengthy, an additional protocol may be added: upon receipt of the request, the recipient will acknowledge receipt. When this protocol is used, the requestor uses a short initial time-out period. If this time-out occurs, the requestor assumes the message was lost, and re-sends. After receiving the acknowledgement from the recipient, the requestor then waits for the normal time-out period.

4.2 Error detection versus error correction

In general, different mechanisms are used for error detection and error correction. This means that *both* mechanisms must work for an error to be corrected; if the error detection mechanism fails, the redundant information which would allow the error to be corrected will never be looked at, and the incorrect value will be used. Two examples will illustrate this idea.

Messages transmitted between the various file system computers contain redundancy to allow the detection of transmission errors. In the event that an error is detected, the receiver either ignores the message (simulating a *lost* message), or informs the sender. In either case, the error is corrected by retransmission of the message. However, if an error should occur which cannot be detected from the accompanying redundancy, an incorrect action will occur, even though the correct information is still available from the sender.

Each record stored on the disk is accompanied by various amounts of error detection redundancy. This redundancy generally consists of some sort of checksum, together with the logical identity of the data and the physical address of the record. Two forms of error correction are provided. In the event that the stored data defines part of the V-view, the system is restarted as if from a crash, restoring a correct, though empty, V-view. For data which contributes to the S-view, some redundant information is stored elsewhere on the disk to allow reconstruction (see section 5). However, if an error should occur which cannot be detected from the redundant information stored with the record, this additional redundant information will never be examined.

4.3 Crash recovery

At any time, any one of the file system machines may *crash*. A crash may be caused by one of several kinds of events, including: the discovery that the V-view is inconsistent, loss of information used to maintain the V-view, an infinite loop, pressing of the Boot button, and various hardware errors. A crash on a particular machine is followed by crash recovery on that machine. The end result of crash recovery will be to restore an S-view consistent with the S-view at the time of the crash, along with an empty V-view (no in-progress transactions).

What constitutes a correct S-view after crash recovery depends on the activities in progress at the time of the crash. Any activity affecting the S-view is part of a transaction designed to make atomic changes to the S-view. That is, if any activity was in progress to change the S-view, after crash recovery either the entire change (due to the activity and all other activities in the same transaction) will appear, or no change (due to that activity or any other activity in the same transaction). This choice is made independently for each transaction in progress at the time of the crash.

This condition must also be satisfied by the crash recovery procedure itself, since a crash can occur during crash recovery. In fact, it is assumed that an arbitrary number of crashes could occur during crash recovery. The effect on other file system computers must also be considered. If two or more are performing some co-operative task, it must always be

considered that one or more of them may initiate crash recovery at any time.

After crash recovery the V-view contains no in-progress transactions. This is accomplished by either completing the transactions which were in progress at the time of the crash, or aborting them. This choice is made for each transaction individually. A transaction will be completed only if it was already in the process of completing at the time of the crash. Since the process of completing a transaction is an activity which modifies the S-view, this choice is made by the mechanism described above.

CHAPTER 5. REDUNDANT STORAGE MECHANISM

It is conceivable that a system crash would be accompanied by the destruction of some information stored by the file system (e.g. a hardware error while a disk record is being written). Thus, recovery from crashes should include some ability to reconstruct lost information. Further, this redundancy should be used to correct information lost during normal operation, e.g. by a failure to correctly read a single physical record.

Since crash recovery is only expected to reconstruct the S (salvageable) view, we need only consider information contributing to that view. (The inability to read information which belongs to the V-view is treated as a crash.) As a first step towards the desired facility, all information contributing to the S-view is stored on system disks. Hence, crash recovery need only consider the contents of disk storage, and can ignore any information stored elsewhere, e.g. in fast memory. The second step is to provide some mechanism for redundant disk storage, both for error detection and for error correction.

This mechanism provides a virtual storage system used by the rest of the file system. As such, there will be procedures for storing information and reading information. Also, there will be some miscellaneous procedures for assigning and releasing storage space. The actual storage is on the physical disk, with redundancy.

The data to be stored is assumed to be partitioned into *pages*, each of equal size. In this chapter we use the word *data* to stand for information stored for higher levels of the system. This information contains not only file data bytes, but also the other components of files, as well as the internal structure which the system uses to hold them together.

The disk is partitioned into physical records. The data page size is chosen so that a single page will fit on a single physical record, together with some additional information used to provide error checking. The physical disk records are partitioned into two groups. Records in the first group are used to store data pages (with error detection redundancy). Records in the second group are used to provide error correction redundancy for some of the records in the first group. All the mechanisms to be described use the same error detection scheme. The error check bits will be computed by a standard algorithm from the logical address of the data page, the contents of the data page and the disk address of the physical record used to store the page.

There are two different facilities which might be provided for storing a page of data. The first (called *static* writing) writes the data page at a specified physical address. The second (called *dynamic* writing) chooses an address at which to write the page, writes the page there, and returns in a response the actual address.

There are three disasters from which one may wish to recover lost information. They are listed in increasing order of seriousness:

A single physical record is lost.

All physical records at a single arm position on a single disk are lost.

All records on a single disk, or on all the disks associated with a single computer, are lost.

We have investigated several proposed mechanisms to recover from these various events. This investigation resulted in two conclusions:

> The methods adequate to recover the loss of a single physical record can be modified to handle the loss of an entire arm position.
>
> The methods which are adequate to recover from the loss of an entire arm position suffer from one of two difficulties:
>
> > They require two physical records to store one data page, or
> >
> > They cannot be used to write a page statically. (This difficulty appears to be due to the constraint that the modification should appear atomic over system crashes.)

An investigation of methods for the representation of files led to the following conclusion:

> All reasonable methods required that some data pages be written statically. (These are generally *pointer* pages, which contain the disk addresses of other pages.)

Thus, the system incorporates three different redundancy mechanisms, one for static writes, one for dynamic writes, and a third to handle the loss of an entire disk or an entire machine. These three are roughly characterized as follows:

> Mechanism A requires 2 physical records to store each page of data, and the stored pages can be written statically. This method can defend against the loss of all physical records at one disk arm position.
>
> Mechanism B requires N+1 physical records to store N pages of data, for some N. However, the stored pages cannot be written statically. This method can defend against the loss of all physical records at one disk arm position.
>
> Mechanism C requires M+1 disk drives (or computers) to store M disk drives (or computers) worth of information. This method can defend against the loss of all physical records stored on one disk drive (or one computer).

The choice of mechanism, A or B, for storing a particular data page is made by higher levels of the system. Procedures are available for each mechanism.

All the redundancy methods have the property that a single bad spot on a disk (i.e. a physical record which cannot be written reliably) makes some number n of physical records unusable. In the case of method A, n=2; for method B, n=1 unless the bad spot is an XOR page, in which case n=N+1; for method C, n=(M+1)*(the n for method A or B, depending on which is being used).

There is a small improvement in the algorithms described below which avoids this loss; it is mentioned here for completeness, and is not included in the descriptions below. This improvement is to provide a list of bad records, with an alternate physical address at which the contents of each bad record is stored. In order to retain the ability to resist loss of all records at a given arm position, this alternate address should be in the same arm position as the bad record. Before a physical disk address is actually used, this list of alternates is checked, and if the address is on the list, the alternate address is used instead.

So that higher levels of the system, which depend on the redundant storage facility, can behave properly in the face of crashes, the redundancy mechanisms are designed with crash recovery in mind. That is, at any instant, the redundancy mechanisms have a number of activities in progress, in response to requests from the higher levels of the system. These requests include: read a page, write a page, allocate a record and release a record. All of these activities (except reading a page) constitute modifications to the stored information. Each mechanism has a crash recovery procedure designed to assure that all requested in-progress modifications have the property that after crash recovery they are either completed or not started.

We now proceed with a description of the three redundancy mechanisms. For each mechanism, we describe the disk representation, the interface procedures provided to higher levels of the system, and the crash recovery procedure.

5.1 Mechanism A: duplication

A portion of the disk is partitioned into pairs of physical records. The two members of each pair are in different disk arm positions. Some form of naming is provided for these pairs such that it is possible to identify the two physical records comprising a pair from the name of the pair.

5.1.1 Procedures:

AllocateFreeRecordPair

Returns the name of a free pair, the pair is marked busy.

WritePage(nPr, pg)

Writes the page pg, together with additional redundancy, on each physical record of the pair nPr. The write is not reported complete until both physical writes have completed. Since there is no essential order between the two physical writes, both may be requested simultaneously, and thus added to an existing pool of uncompleted disk requests.

ReadPage(nPr) returns pg

Choose one of the physical records arbitrarily, and read it. Compute the proper value of the error check bits and compare it against the value just read. If the check is ok, return the page just read to the requestor. If not, read the second physical record, and perform the same error check. If the check is ok, reconstruct the appropriate record for the first member of the pair and re-write it; concurrently, return the correct page to the requestor. If the second record fails the redundancy check, then the page is lost (fall back to mechanism C).

ReleasePair(nPr)

Mark as free the pair with name nPr.

5.1.2 Recording free pages

There are three possible mechanisms for recording the free pairs in the S-view. We have not as yet chosen one.

The first method is to reserve a single bit in each physical record which is included in the error check. This bit is used to mark the record as free. The system maintains in the V-view a list of the free pairs. The allocation procedure checks that a pair thought to be free is actually free, by reading the physical records. If a supposed free pair turns out to be busy, this constitutes a system crash. The free list is reconstructed during crash recovery.

The second method is to maintain an explicit bit table in the S-view. Since this bit table must be modified statically, it must be recorded using double redundancy.

The third method is implicit. A record pair is busy if it can be reached from some root pair by following pointers; otherwise it is free. In this case the system also maintains a list of free pairs in the V-view, but there is no check on the list.

5.1.3 Crash Recovery

Begin by reading all physical records, and perform an error check on each one. If exactly one of a pair passes the check, recompute an appropriate value for the other member of the pair and re-write it. If both pass the check, but contain different pages, choose one arbitrarily and re-write the other to conform (this case occurs when the system crashes while the pair is being written). If both pass the check and agree on their contents, all is well with that pair.

If there is some pair for which neither record of the pair passes the check, then it may be neccessary to consider that some information on this disk has been lost, in which case, mechanism C must be invoked. This will not be necessary if the pair can be proven to be free. The method of proof depends on the method of marking free record pairs. In particular, marking them free with a free bit stored in the record itself will not help here.

5.2 Mechanism B: parity

A portion of the disk is partitioned into groups of physical records called *correction groups.* Each record in a correction group is on a different disk arm position from all the other records in the group. One record in each group is designated as the *parity* record, and the others are called *data* records.

The general idea is that the parity record will always contain the bitwise parity of the data records in its group, not including the checksum or the *included* field (defined below). Thus the parity record contains parity bits only for the information content of the records in its group. The bitwise parity record is computed by taking a bitwise exclusive or (*xor*) of the data records; i.e. bit *i* of the parity record is the xor of bit *i* of each of the data records.

Since it is impossible to simultaneously change a data record and the parity record, an additional feature is required. A special field (called the *included* field) is set aside in the parity record. This field contains one bit for each data record in the group. A data record in the group is said to be *included* if its corresponding bit is on in the included field of the parity record. The system maintains the truth of the condition that the parity record contains the parity of all included records (except for bits in the included field, and checksum bits).

Now it is easy to change the contents of a data record. Read in the parity record and the old value of the data record to be changed. Compute a new parity record which does not include the record to be changed. (This changes the parity data, and one bit in the included field). Write out this new parity record. Write the new value of the data record. Now write out a new parity record which includes the changed record. Observe that for a short time, the data record is not included int he parity check, and hence the information in the data record cannot be recovered by the redundancy mechanism. This is the reason for the restriction against writing statically when using this mechanism.

For each page write, this mechanism requires 5 disk actions. The first two may be simultaneous, reading the old data record and the old parity record, but the others must occur in strict sequence. Moreover, the ability to resist the loss of all physical records at a single disk head position requires that disk arm seeks take place between these actions. Thus, we must be sure that these actions can be batched so as to reduce the effective cost of the seeks.

To do this, we break the actions for a single data record into two groups.

> The first group comprises the first 3 actions described above (read the data record, read the parity record, rewrite the parity record). This sequence of actions effectively frees the data record. All data records which have been released from use can be collected into a pool. At convenient times, the system can perform this sequence of actions for some of the pooled records, in particular, some subset which occur at the same arm position, and whose parity records occur at the same arm

position.

The second group of actions comprise 3 actions: write the data record, read the parity record, and rewrite the parity record. Observe that this has increased the total number of actions to 6. The first two of these last 3 actions may be simultaneous.

5.2.1 Procedures

WritePage(pg) **returns** nDr

When complete, returns the address nDr of the record on which the page has been written.

ReadPage(nDr) **returns** pg

First reads the addressed physical record and checks the internal redundancy information. If ok, the page of information is returned. If not ok, an error correction algorithm must be executed. First read the parity record corresponding to the addressed record. If the parity record fails its error check, then some information has been lost (fall back to mechanism C). Now read in all other included records, check their internal redundancy, and compute their parity. If any fail their internal redundancy check, some information has been lost (fall back to mechanism C). If all pass their internal check, xor the accumulated bitwise parity with the parity record, and the result is the information content of the originally addressed record. Return this page of information, and re-write the bad page corrrectly.

ReleaseRecord(nDr)

Read the contents of the record and its parity record. If either record fails its internal check, recompute the parity record by reading all other records in the group; when this is done, all the records can be included in the parity computation, or free records can be excluded. If both records are OK, compute the new value of the parity record as the xor of the two records, turn off the included bit, and rewrite the parity record.

These three procedures require careful disk scheduling in order to work efficiently. The actual address of a record to be written should be assigned as late as possible, so that as many records as possible are written in the same arm position. Also, their parity records should lie in the same arm position, which must be different from that of the data records.

Released records should be accumulated and freed as a background task.

5.2.2 Crash Recovery

Crash recovery begins by reading all parity records and determining which data records are to be included. Next, read all included records and form the accumulated parity for the group. If any record fails the internal check, it can be recomputed from the others. If two records fail the internal check, some information is lost: fall back to scheme C. If all records pass the internal check, but the accumulated parity does not agree with the parity record, there is a real problem. (Fall back to mechanism C?)

5.3 Mechanism C: cross-disk parity

The basic idea in this method is to form a cross-disk parity record. To this end, one disk in the group is designated as the parity disk. It is assumed that each disk stores exactly the same number of physical records. The physical records are partitioned into groups, such that each group has exactly one member stored on each disk. These groups are treated as

parity groups as in mechanism B. That is, the record in each group which is on the parity disk will contain the bitwise parity of all other records in the group. This scheme assumes either that only one disk in the same correction group is attached to each machine, or that each disk is accessible from more than one machine, so that the scheme can work when a machine goes down, as well as when a disk breaks.

A variation on this method places some of the parity records in each disk. Another variation works on a per-machine rather than per-disk basis; this makes it unnecessary for each disk to be accessible from more than one machine. We do not describe either variation here.

Not every physical record stored on a given disk participates in a cross disk correction group. In particular, only one record out of each record pair involved in mechanism A need participate. Further, the parity record in each correction group of mechanism B does not participate.

Observe that the computation of several cross-disk xors can be pipelined if the disks are on separate machines. The first machine, M_1, in the pipe line reads its representative from group 1, R_{11}, and sends it to machine M_2. Now M_1 continues to read records from groups in sequence, sending R_{12}, R_{13} etc to M_2. At the same time, M_2 reads its representative from group 1, R_{21}, and forms the xor with R_{11}. It transmits this xor to M_3. Now M_2 reads its representative from group 2, R_{22}, forms the xor with R_{12}, ands sends the result to M_3. Thus, M_i receives from M_{i-1} the xor of the representatives from each group which occur on M_1, M_2, . . . M_{i-1}. Machine M_i adds its representatives to the xors and transmits the result to machine M_{i+1}.

If all of these machines are connected by a single Ethernet, this pipeline will eventually break down. However, if there is a single connection between each successive pair (M_1, M_2), (M_2, M_3), . . . , the pipeline should run as fast as the records can be read from a single disk.

5.3.1 The algorithm

This mechanism, C, sits between the disk and mechanisms A and B. That is, it provides another virtual disk storage facility, used by mechanisms A and B. Thus, the actual file system routines see a virtual storage system (A and B) which in turn is implemented by routines which use a virtual storage system (C). Mechanism C is implemented by routines which use the real disk. Note that the implementation described below assumes that it is underlying another redundancy mechanism, such as A or B, and therefore sacrifices some security against crashes in favor of efficiency. This sacrifice does not compromise the ability of the system to maintain consistency, but it may occasionally cause information to be lost which could have been preserved.

We now give the algorithm for writing a record which participates in cross-disk parity. Assume the record to be written is record R on machine M. The algorithm is written as though the parity disk is on a different machine; the obvious simplifications can be made if this is not the case.

On machine M:

> Read the old contents of record R. Check the internal redundancy, and correct the contents if necessary. Note that this read can be avoided if record R has known contents, e.g. if it is free and has been zeroed.
>
> Send the new and the old contents of record R to the parity machine.
>
> Wait for acknowledgment from the parity machine. Resend if no acknowledgment after some suitable delay.
>
> Write the new contents into record R. This can be done in parallel with sending

information to the parity machine.

Report completion of write to the process which requested it.

Inform the parity machine that the new write is complete, repeating the message until it is acknowledged.

At the parity machine:

Receive old and new contents of record R from machine M.

Acknowledge receipt to machine M.

Read old parity record for record R.

Write new parity record for record R. Keep a record (in memory, not on the disk) that this change is in progress on machine M. This record allows the cross-disk redundancy to be preserved if machine M crashes during the write, as long as the parity machine doesn't crash.

Upon receipt of write complete message from machine M, acknowledge to machine M. If still holding a record of the in-progress write on machine M, delete the record.

5.3.2 Crash recovery

Single machine crash on machine M, not the parity machine.

Negotiate with the parity machine to find all in-progress changes to the contents of the disks on machine M. Make all the in-progress changes. Report the completion of each change to the parity machine. The reason for this procedure is to keep the parity records in good shape in spite of single machine crashes, so that the machines not involved in the crash don't lose the protection of the cross disk redundancy. Now carry out the crash recovery algorithms for mechanisms A and B. If this leaves any records in doubt, use the cross-disk parity to compute the correct values.

Contents of a disk on machine M (not the parity machine) are lost.

The parity machine computes the contents of the disk on machine M, in two steps. First the parity machine computes the bitwise parity of all records, except the one on machine M, in each correction group. This parity record is in fact the correct contents of the corresponding record on machine M. A participating machine will use the new value for any page which is in the process of being written. The second step is to use mechanisms A and B to compute the correct value of the redundant pages on machine M. The result is an image of the correct contents of the disk on machine M. The parity machine now continues the recovery procedure for a non-parity machine, and replaces machine M in the file system. When machine M is eventually repaired, it becomes the new parity machine.

Multiple machine crash, not including the parity machine

Recover machines one by one by whatever methods are available, until only one is left, then use the immediately previous procedure.

Any crash involving the parity machine.

All machines recover as if the parity machine did not exist. Then recompute the contents of the parity machine.

If mechanism C is to be used, the small improvement suggested just before the descriptions of mechanisms A and B becomes fairly essential. This improvement is to keep a record on each machine of bad physical records and use substitute records. If this is not done, a single bad record on one machine will make all the corresponding records on the other machines useless.

CHAPTER 6. TRANSACTION ALGORITHMS

6.1 Introduction

In this chapter we present the algorithms used to obtain the properties of transactions which were described in section 3.4. These properties can be summarized by three notions:

interlock concurrent transactions,

make transactions atomic over system crashes, and

synchronize simultaneous activities on multiple file system computers.

Different ideas contribute to each of these objectives. A lock mechanism is used to interlock the transactions (see section 3.4.4, and below). The concept of an *intentions list* is used to provide atomicity over system crashes (see below). Finally, messages sent between computers are used to synchronize the checkpoint and finish actions for a single transaction involving multiple system computers (see section 4 for some discussion of the pitfalls).

6.1.1 Locks

The lock mechanism we use has a number of ramifications. The first issue is grain size. We have divided the information content of a file into various sized partitions (we have not as yet made a final choice of partition sizes). Each partition is individually locked. When a portion of a file is to be read or modified, all partitions containing affected information must be appropriately (read or write) locked. Moreover, the chosen partitions are smaller than the unit of physical storage (one disk page). Also, the mechanism for modifiying a portion of a page involves creating a new copy of the page. Thus it is neccessary to read-lock the contents of the whole page when modifying a portion of the page. These read locks are in addition to the write locks on the portion of the information to be actually modified.

The second issue related to locks is that of time-outs. In order to prevent a single user from holding information locked indefinitely, each lock on a partition has an associated time limit. Nothing happens when this time limit is reached, until the lock is actually impeding the progress of some other transaction. What happens when this occurs depends on whether the timed-out lock is a read or a write lock.

If the timed-out lock is a read lock, the transaction holding the lock is informed (via a BrokenReadLock message) and the transaction desiring access to the information (it must be write access, or there would be no conflict with the read lock) is allowed to write-lock the partition and proceed; in effect, "breaking" the read lock. However, the current time + T (a system wide constant) is recorded in a variable *tmEf* (time of earliest finish), in the private data of the transaction breaking the read lock. This transaction is not allowed to finish (effectively carry out its requested modifications) until the current time exceeds *tmEf*. Thus, the old values of the locked data remain effectively correct until time *tmEf*.

A user makes use of this facility by means of the AreYouStillThere command. If a user, at

time T_1, transmits to a transaction worker a request containing this command, and receives a response which indicates that there are no broken read locks, then all of the information which the user has read through that worker is correct until at least time T_1+T.

If the timed-out lock is a write lock, a different policy is followed. As in the case of a read lock, nothing happens until the lock is actually impeding some other transaction. Then a message (TimedOutWriteLock) is sent to the transaction worker holding the lock. The lock remains in effect. The worker holding the lock is expected to release the lock as soon as possible. Note that since the lock and the worker are on the same machine, transmission delays, or a crash which affects only one of the parties, cannot delay the worker's response. There is only one situation, described below, in which the worker's response can be delayed indefinitely.

If the worker is not currently involved in a checkpoint or finish command, the worker will abort the transaction, thus releasing the lock. In this case, the worker informs its coordinator that the transaction is to be aborted, and proceeds (on its own) to perform the abort locally. Thus, in this case the lock is released immediatly.

If the worker is involved in a checkpoint or finish command, the worker's activity in response to the TimedOutWriteLock message depends on the exact phase of the transaction (see subsequent discussion of multiple machine crash recovery). If the transaction has not yet reached the ready phase, the worker proceeds as above, informing its coordinator that the transaction is to be aborted and performing the abort locally. If the transaction has passed the ready phase, (has reached the finishing or checkpointing phase), then the write lock will be released shortly as a consequence of termination of the checkpoint or finish action. If the action is in the ready phase, the worker may not (locally) decide to abort. In this case, the worker sends a message to its co-ordinator requesting an abort. The co-ordinator will initiate an abort unless it has passed the r-wait phase (and thus may have sent a get-finished message to one of its workers).

In the case that the worker is in the ready phase, and the machine on which the co-ordinator resides has crashed, there may be some delay before the write lock is released. This occurs because there is no way to determine whether the transaction has begun to finish, or not. The worker must wait until the crashed machine has recovered.

An important detail of the algorithm is that a worker attempting a write will not break any read locks until there are no impeding write locks. Thus, read locks which have been set to cover the remainder of a partially modified page will never be broken.

6.1.2 Intentions

The goal of the crash recovery algorithm is to make transactions appear atomic over crashes. That is, either all or none of the file changes requested between the last checkpoint and the current checkpoint or finish should appear in the stored data. The central idea behind the algorithm is that of an *intentions list.* We first describe the idea assuming that only one computer is involved, and then later we extend it to cover multiple file system computers.

An intentions list is a list of the actions which must be taken to finish (or checkpoint) a transaction. We assume that the intentions list itself may be written as a single atomic act. If it requires several pages of disk space, simply write all but the first page, wait for completion of these writes, and then write the first page.

The actions in an intentions list must be such that they can be repeated several times and lead to the same result. This situation occurs when the file system crashes during the completion of the transaction, followed by crashes during crash recovery. More precisely, any sequence of actions composed from successive intial segements of the given list, followed by the complete given list should result in the same effect as the original list. For example, if the given intentions list is:

$A_1, A_2, A_3, A_4, A_5, A_6$

then the following sequence of actions should lead to the same result:

$A_1, A_2, A_3, A_1, A_2, A_1, A_1, A_2, A_3, A_4, A_1, A_2, A_3, A_4, A_5, A_6$

More generally, since it is desirable to carry out these actions concurrently, they should be such that they can be carried out in any order, with any number of repetitions. If each action is of the form: Store data D at address A, then these conditions are satisfied. This is the only satisfactory form for the actions that we have discovered.

In order to apply this idea, we have chosen to represent files by one or more pointer pages, each of which contains pointers to data pages. In this context, data pages contain all of the information needed to define files as described in section 3.2. Thus, what we are calling data pages here contain the file data described in section 3.2, as well as all the other information contained in the file.

The pointer pages are stored on the disk using redundancy mechanism A (which allows static writing). The data pages are stored using mechanism B (which does not allow static writing). A modification to the file is carried out by writing new data pages for the ones to be modified, and then rewriting the pointer pages to contain modified pointers which point to the new data pages. In effect, we swing the pointers to the new data pages.

The intentions list contains a list of the pointers to be changed, and their new values. Thus, to carry out a succession of changes within a single transaction, proceed as follows:

As each modification is received, determine which data pages in the file are to be changed, read in their old values and compute the new values.

From time to time, write out the new data page values on free disk pages (using redundancy mechanism B). This results in no actual change to the files, since no pointers are changed.

Form a tentative list of the pointers which must be changed.

When the transaction finishes, and all new data pages have been written, write the list of pointer changes on the disk, using redundency mechanism A; these are the intentions.

Now read in the old pointer pages, compute new values and re-write them in place.

Crash recovery proceeds as follows:

Perform crash recovery for the redundancy mechanisms.

Find all lists of intentions and carry them out. When a particular list has been completely carried out, write an empty list on top of it.

There is an alternative scheme which we have not examined fully. In this alternative scheme an *undo list* is maintained during the transaction; it contains the old values of the pointers. In this scheme, the actual pointers can be changed to the new values during the transaction. The step of completing the transaction consists of simply erasing the undo list. If the system should crash any time before the transaction is complete, then crash recovery would reset the values of the pointers to their old values, resulting in un-modified files. We shall not consider this scheme further.

6.1.3 Multiple system computers

The preceeding discussion ignored the possibility that there may be more than one system computer involved in a single transaction. In this case, a number of situations can arise. First, any one, or more than one, of the system computers may crash. Second, a write lock may time-out on one of the computers. In this case, it is desirable (for speed) to release the lock without consulting any other system computer.

The method used for multiple computers is to identify a single computer as the controller of the transaction. This identification is accomplished by introducing a process to co-ordinate the transaction. The machine on which the co-ordinator resides controls the transaction. The processes which actually carry out the work of the transaction are termed workers.

Associated with each worker is an intentions list. There is also an intentions list associated with the co-ordinator. The worker intentions lists are as described above, except that an additional variable (*sPhaseWk*) is stored with each list (in the S-view, i.e. on the disk). The value of this variable will signify one of three things:

working: the transaction has not yet finished,

finishing: the transaction has finished, and the intentions in this list should be carried out after a crash,

readyToFinish: ask the co-ordinator if the transaction is finished.

Associated with the co-ordinator is also an intentions list and a variable called *sPhaseCo*. The entries in the co-ordinator's intentions list are the disk address (and machine) for each worker intentions list. The actions of the co-ordinators intentions list are to set *sPhaseWk* of each workers intentions list to finishing. Thus a single variable (*sPhaseCo*) determines whether the transaction is finished, and setting this variable to finishing constitutes the logical act of finishing the transaction.

The basic idea for crash recovery is to scan the disk for intentions lists. If a worker's intentions list is found, its *sPhaseWk* is examined. If *sPhaseWk*=working, the transaction has not started to finish, and the list is destroyed. If *sPhaseWk*=finishing the transaction is finished, so the intentions are carried out, and the list destroyed. In the third case, *sPhaseWk*=readyToFinish, the appropriate pages are write locked, and a worker is started which attempts to communicate with its co-ordinator to discover if the transaction is finished. (In fact, the second case is also handled by write locking the appropriate pages and starting up a worker process to carry out the intentions.)

If a co-ordinator's intentions list is found, its *sPhaseCo* is examined. If *sPhaseCo*=idle, the transaction is not yet finished, and the list is destroyed. If *sPhaseCo*=f-wait the transaction is finished, and then a co-ordinator is started which attempts to inform each of its workers that the transaction is finished, and which responds appropriately to worker messages inquiring about the state of the transaction.

In the case that a machine M crashes, and is restarted with some process missing, due to a definitely unfinished (or definitely finished) transaction, M will eventually receive messages from other processes in the transaction. In this case, M will respond with a NeverHeardOfx message. In all cases, the process sending the original message is then able to decide for itself whether the transaction should be finished or aborted.

6.1.4 Progress of a transaction

The true state of a transaction is determined by the phase of the co-ordinator. However, the phase of each worker is stepped forward with the phase of the co-ordinator, so that for each worker phase only certain coordinator phases are possible, and vice versa. Thus, the phase of a worker can be used to determine the possible phases of the co-ordinator. If the possible phases of the co-ordinator do not affect the action to be carried out by a worker, the co-ordinator need not be consulted.

The general pattern is that the co-ordinator changes phase, then sends a message to each worker. Upon receiving such a message, each worker carries out some activity, then changes phase and sends a message to the co-ordinator. The following table shows the possible configurations which can be reached during the normal flow of the algorithm (i.e., ignoring write lock time-outs, user requested aborts and system crashes). The state of an individual process is described by the value of two variables, vPhase and sPhase. As their names suggest, one is in the V-view and the other in the S-view. For each possible state of a

co-ordinator, we list the possible states of a worker in the same transaction. We use abbreviations for the phase names.

Abbreviations for worker phases:

W	for working
PTF	for preparingToFinish (vPhase only)
PdTF	for preparedToFinish (vPhase only)
RTF	for readyToFinish
F	for finishing
m	for missing (the process has been destroyed)

Abbreviations for co-ordinator phases:

I	for idle
P	for p-wait (vPhase only)
R	for r-wait (vPhase only)
F	for f-wait
m	for missing

In the following table, each pair (*p*, *i*) stands for vPhase = *p* and sPhase = *i*.

Co-ordinator	*Worker*
(I, I)	(W, W)
(P, I)	(W, W) (PTF, W) (PdTF, W)
(R, I)	(PdTF, W) (RTF, RTF)
(F, F)	(RTF, RTF) (F, F) m
m	(F,F) m

A transaction is logically finished at the moment the co-ordinator changes from state (R, I) to state (F, F).

In order to convince oneself that the algorithms presented below work correctly in the presence of timed out write locks, system crashes and user requested aborts, it is necessary to consider the possible combinations of co-ordinator and worker states. For each such combination, examine the algorithim and see that it leads to the correct result.

For example, consider the time-out of a write lock. If we examine the algorithm, we see that this is handled in one of three ways. If the worker is in one of the states (W, W), (PTF, W) or (PdTF, W) then the worker performs a local abort, and signals the co-ordinator that it is aborting. In these cases it is known that the co-ordinator has not yet reached state (F, F); thus the local abort is legitimate. Moreover, even if the message to the co-ordinator is lost, the worker will eventually disappear, and messages from the co-ordinator will be responded to by "Never-Heard-of-That-Worker", which will in turn lead the co-ordinator to abort.

If the worker is in state (RTF, RTF), then the co-ordinator could be either in (R, I) or (F, F). In the first case the transaction has not yet finsished. Moreover, it could be a substantial period of time before it does finish, since the co-ordinator is waiting on some worker, and the worker may be waiting on some user. Thus, if the co-ordinator is in state (R, W) it is desirable to abort and release the timed out write lock. On the other hand, if the co-ordinator has reached state (F, F), the transaction is finished and the write locks will be released soon. Since there is no way the worker can determine which case holds, the

worker sends an abort request to the co-ordinator.

Possible occurrences of a system crash are analyzed in a similar fashion, paying particular attention to the situation occuring after processes are deleted. Sometimes a deleted process signifies that the transaction is finished, and other times it signifies that the transaction is to be aborted.

A more formal analysis could be obtained as follows. Define a configuration as a set consisting of one co-ordinator state and one or more worker states. A transaction is in a particular configuration if the co-ordinator has the state specified in the configuration, and there is at least one worker in each worker state included in the configuration. Now a configuration change diagram can be constructed which shows possible transitions among the configurations, due to various events. The particular configuration { (I, I), (W, W) } is the start configuration. (This represents the co-ordinator in state (I, I) and each worker in state (W, W).) One now checks that any path in the diagram starting with the start configuration either leads to all intentions being carried out, or none.

6.2 Data structures involved in transactions

There are a number of data structures used by the transaction machinery, which are not directly visible to a user, and hence were not described in section 3.4. We give a brief description of these. The first, a file, is not associated with a particular transaction. The others are either associated with a transaction worker process, or a co-ordinator process.

6.2.1 File

A file consists of one or more pointer pages, which contain pointers to the disk addresses of data pages. The information content of the file is stored in the data pages, while the identity and order of the data pages is determined by the contents of the pointer pages. The pointer pages are stored on the disk by redundancy mechanism A, and data pages are stored by mechanism B.

The pointer pages are designed so that individual pointers can be rewritten by: reading in the old pointer page, modifying a pointer and re-writing the page. This can be repeated indefinitely with the same result.

6.2.2 Worker

Associated with each worker process we have the following items. The local intentions list (including *sPhaseWk*) will survive a system crash, while the other items do not. In the event of a system crash, if the worker process is to be restarted, these other items are set to values depending on the value of *sPhaseWk*.

Local intentions list

This list contains a variable, *sPhaseWk*, and a sequence of pairs. The possible values of the variable are the ordered set:

working < readyToFinish < finishing,

The sequence is initialized to be empty, and *sPhaseWk* to working. Each pair contains the address of a file page pointer (given by the disk address of a pointer page, and the index of the pointer within the page), together with the address of a data page.

Each worker intentions list is stored on the disk using redundancy mechanism A, and survives a system crash.

vPhaseWk

A variable which defines current mode of the worker. It takes on the (ordered) values:

working < preparingToFinish < preparedToFinish
< readyToFinish < finishing < aborting

tmEf

Earliest time the transaction can finish, initialized to the current time.

doAbort

Initialized to false.

setRsnIcWk

A set of request serial numbers (i.e. pairs <user, integer>) which are *incomplete*. A request serial number *rsn* is incomplete if some request for the same user with a later serial number has arrived, and no request numbered *rsn* has been completely processed by the worker.

setRsnMaxWk

A set containing all the maximal rsn's on requests seen by this worker. This set has one element for each user who has sent a request to this worker.

6.2.3 Coordinator

The following items are associated with a co-ordinator process. The first, the intentions list (including *sPhaseCo*) survives a system crash. The others are reconstructed after a system crash, depending on the value of *sPhaseCo*.

Intentions List

This list contains a variable, *sPhaseCo*, and a sequence of pairs. The possible values of the variable are the ordered set:

working < f-Wait.

The sequence is initialized to empty and *sPhaseCo* to working. Each pair identifies a worker intentions list, by giving a machine name and a disk address.

Each co-ordinators intentions list is stored on the disk using redundancy mechanism A, and survives a system crash.

vPhaseCo

This variable takes on the ordered set of values:

working < p-wait < r-wait < f-wait

and for a newly created co-ordinator is initialized to working.

doAbort

Initialized to false.

6.3 Worker process algorithm

We now present the algorithm followed by a worker process. This description omits the details of the internal file structure, and only differentiates between file reads and writes. The process only acts in response to messages received from other processes. These messages may be user requests, or control messages from other file system processes, possibly on other

file system computers. This description is written in an ad hoc language. The only unusual feature of the language is the concurrent statement. Indentation is used to denote grouping.

The worker is composed of three main concurrent processes. The first two described below handle normal acitvity (i.e. user requests and lock time-outs). The third controls finishing and checkpointing. The first two spawn independent processes to handle each request, while the third is programmed as a single process.

Concurrently

Process user messages,
Process messages from other workers on this machine,
Process messages from co-ordinator

Process user messages:

Concurrently, for each user message (request) with rsn ⟨*user, sn*⟩

if (*vPhaseWk* ≥ **preparedToFinish**) **then** ignore this request

if (*vPhaseWk* = **preparingToFinish**) **then**

Compare serial number on this request with the maximum serial number encountered on requests from the same user. If the number on this request is higher than the maximum encountered, then ignore this request.

Augment setRsnIcWk and setRsnMaxWk:

Find the element in *setRsnMaxWk* for this user, and call it ⟨*user, snOldMax*⟩. If there isn't one, add ⟨*user*, 0⟩ to *setRsnMaxWk* and use that.

for *i* **in** (*snOldMax..sn*] **do** add ⟨*user, i*⟩ to *setRsnIcWk*; i.e. add the current request's rsn to the set of imcomplete rsn's, together with rsn's for any requests which seem to have been lost in transmission.

if *sn*>*snOldMax* **then** replace ⟨*user, snOldMax*⟩ in *setRsnMaxWk* by ⟨*user, sn*⟩.

Concurrently, for each command in the request

switch on type of command
(In this description, we only consider the generalized cases of read and write).

Read command:

Obtain the necessary read locks, by any desired solution to the readers and writers problem, adding appropriate entries to the set of read lock entries for this worker.

Read the data from the disk. (Watch out if the data is write locked by this transaction, as the S-view will be inaccurate.)

Send an action completed response to the user specified in the request, containing the data read from the file.

(This completes the read command.)

Write command:

concurrently, obtain the necessary locks, and prepare the write.

Obtain locks:

The necessary locks are:

Write locks for any information to be modified

Read locks for any information which is stored on the same file page as some information which is to be modified, but is not

itself to be modified.

Obtain the locks by any desired solution to the readers and writers problem, with the following additions:

If the system reaches a state in which all the impeding write locks are timed out, send WriteLockTimedOut messages to each worker holding those locks.

If the system reaches a state in which all impeding locks are timed out read locks, then (Note: it is essential that at this point there are no impeding write locks):

Send ReadLockBroken messages to each worker holding those locks.

Re-set *tmEf* to the current time + T.

Erase the timed out read locks.

Prepare the write:

Concurrently, for each file page containing data to be modified:

When all the read locks have been obtained for the data on this page, read the old version of the page and create a new version containing the modified data. Add the appropriate pointer change to the list of intentions for this transaction on this system machine.

Send a command completed response to the user specified in the request.

(This completes the write command.)

ClearReadLock command:

Remove all lock entries from the set of read locks for this worker, which match the data addresses and user specified in this request.

Remove all lock entries from the set of broken lock entries for this worker, which match the data addresses and user specified in this request.

Send a response to the user specified in the request.

(This completes the clear a read lock command.)

AreYouStillThere command:

Send a response to the user specified in the request. Place in the response a list (even if null) of all entries in the in the set of broken read locks for this worker, which designate the user specified in this request.

(This completes the AreYouStillThere command.)

(This completes the user request command switch)

Delete <*user, sn*> from *setRsnIcWk.*

(This completes the activity for a user message.)

Process messages from other workers in this machine:

Concurrently, for each message

Switch on type of message

BrokenReadLock message:

if (*vPhaseWk* $\geq$ preparedTofinish) **then** ignore this message

for each appropriate entry in the set of read locks (for this worker)
Remove the entry from the read lock set
Send an appropriate message to the user specified in the read lock
Place the entry in the set of broken read locks (for this worker)

TimedOutWriteLock message:

if *vPhaseWk* < readyTofinish then
Set *doAbort* ← true
Send Abort to Co-ordinator

if *vPhaseWk* = readyTofinish then
Send Abort to Co-ordinator

Process messages from co-ordinator:

Wait for message from co-ordinator, or *doAbort* = true
if message = Abort, then goto Abort
if message = GetPreparedToCheckpoint then goto Checkpoint
if message = GetPreparedToFinish then goto Finish
if *doAbort* then goto Abort
Error

Checkpoint:
begin checkpoint code
[program will be filled in in due time]
Resume wait for message from co-ordinator, at *Process messages from co-ordinator.*
end checkpoint code

Finish:
begin finish code
for each user specified in the message
modify *setRsnIcWk* and *setRsnMaxWk* (as described in *Process user messages* above), exactly as though a request had arrived from the user with the serial number given for that user in the GetPreparedToFinish message.

Set *vPhaseWk* ← preparingToFinish

wait until (the set of indices of uncompleted requests is empty and the current time exceeds the value of *tmEf*) or (*doAbort* = true).

if *doAbort* = true then goto Abort

if the set of broken read locks is non-empty then
(Note: an alternative would be to invite a user to release the broken read locks).
Set *doAbort* ← true
Send Abort to Co-ordinator
goto *Abort*

Set *vPhaseWk* ← preparedToFinish.
Send PreparedToFinish to the co-ordinator, including the disk address of the

```
        intentions list.
    Release all Read locks (held by this worker).
    wait for message from co-ordinator, time-out or Abort-flag = true
    if time-out then
        Send PreparedToFinish to the co-ordinator, including the disk address of
          the intentions list.
        Resume the wait for message from co-ordinator or time-out
    if doAbort = true or message = Abort or message =
      NeverHeardOfThatCoordinator then goto Abort
    if message = GetPreparedToFinish then
        Send PreparedToFinish to co-ordinator
        Resume preceeding wait for message or time-out
    if message # GetReadyToFinish then error
    Set sPhaseWk = readyToFinish, and wait for the disk write to finish.
    Set vPhaseWk = readyToFinish
    [Worker must now interogate the co-ordinator to determine if the
      transaction is finished.  Up to this point the worker knew that the
      transaction was not finished.]
ReadyToFinish:
    Send ReadyToFinish to co-ordinator
    Wait for message from co-ordinator or time-out
    if time-out or message = GetReadyToFinish  then goto ReadyToFinish
    if message = Abort then goto Abort
    if message # GetFinished and message # NeverHeardOfThatCoordinator
      then error
    Set sPhaseWk ← finishing, and wait for the disk write to finish.
    Set vPhaseWk ← finishing
    Send Finished to the co-ordinator
    [The worker now knows that the transaction is finished.]
Finishing:
    Carry out intentions.
    Release all write locks (held by this worker).
    Destroy Intentions list.
    Destroy worker.
end finish code
Abort:
begin abort code
    Release all read locks held by this worker
    Release all write locks held by this worker
    Destroy intentions list
    Destroy worker
```

end abort code

6.4 Co-ordinator algorithm

The co-ordinator is essentially a single process. However, during waits for responses from the workers, a number of subsidiary processes are spawned.

Wait for a message from a user or a worker
Switch on message type
User request message:
begin user request
for each command
switch on command
Command not directed to co-ordinator:
(Then this action, and subsequent actions in this request, must be directed to the worker on this machine)
Send remainder of the request to the worker on this machine
Resume global wait
Create new worker command:
[Code will be filled in]
Checkpoint command:
[Code will be filled in]
Finish command:
Send GetPreparedToFinish to all workers
Set *vPhaseCo* ← p-wait
concurrently for each worker
begin each worker
wait for response, time-out or *doAbort* = true
if time-out then
Resend GetPreparedToFinish to the worker, and resume waiting
if *doAbort* = true then cease processing this worker
switch on response
NeverHeardOfThatWorker response:
(sent by the system on the machine on which the worker is purported to reside.)
Set *doAbort* ← true
PreparedToFinish response:
Record the address (of the worker's intentions list) in the co-ordinator's intentions list; the address is transmitted in the PreparedToFinish response.)
Abort response:
Set *doAbort* = true
end each worker

```
if doAbort = true then goto Abort
Set vPhaseCo ← r-wait
Send GetReadyToFinish to all workers
concurrently for each worker
    begin each worker
    wait for response, time-out, or doAbort = true
    if time-out then
        Resend GetReadyToFinish to the worker
        Resume waiting
                (Note: an alternative to the preceeding code would be
                to abort).
    if doAbort = true then goto Abort
    switch on response
      ReadyToFinish response:
        finished with this worker
      PreparedToFinish response:
        Send GetReadyToFinish to the worker
        Resume waiting
      Abort response:
        Set doAbort ← true
        Finished with this worker
      NeverHeardOfThatWorker response:
      (sent by the system on the machine on which the worker is
      purported to reside)
        Set doAbort = true
        finished with this worker
    end each worker
if doAbort = true then goto Abort
Set vPhaseCo ← f-wait
Set sPhaseCo ← f-wait, and wait for the disk write to finish
[The transaction is now effectively finished.]
[Resume co-ordinator here during crash recovery]
Send GetFinished to each worker
concurrently for each worker
    begin each worker
    wait for response or time-out
    if time-out or response = ReadyToFinish then
        Resend GetFinished to the worker
        Resume waiting
    if response = Finished then finished with this worker
```

```
                    if response = ReadyToFinish then
                        Send GetFinished to the worker
                        Resume waiting
                    end each worker
                    Erase co-ordinator intentions list
                    Destroy co-ordinator process
            Abort command:
                goto Abort
            end of switch on command
        end user request
    Message (unsolicited) from some worker
        switch on type of worker message
        Abort message:
            goto Abort
        (end of switch on type of worker message)
    (end of switch on message type)
    Abort:
      Send Abort to each worker
      Destroy co-ordinator's intentions list
      Destroy co-ordinator
```

6.5 Crash recovery algorithm

Scan the disk, looking for worker intentions lists and co-ordinators intentions list.

```
for each worker's intentions list do
  switch on sPhaseWk
      < readyToFinish
          destroy this intentions list
      = readyToFinish
          start up a worker process, with
              vPhaseWk = readyToFinish
              All pages write locked which appear in the intentions list
              start the worker at ReadyToFinish
      = finishing
          Start up a worker process with
              vPhaseWk = finishing
              All pages write locked which appear in the intentions list
              Start the worker process at Finishing
  (end of switch on sPhaseWk)
```

for each co-ordinators intentions list
  switch on *sPhaseCo*
    = idle
      Send abort to each worker
      Destroy co-ordinator's intentions list
    = F-Wait
      Start up a co-ordinator process at the point labeled "Resume co-ordinator here during crash recovery".
  (end of switch on *sPhaseCo)*

7.0 Processes, Modularization