THE FELIX FILE SERVER

M. Fridrich and W. Older
Bell-Northern Research, Ottawa, Canada

ABSTRACT: This paper describes Felix - a File Server for an experimental distributed multicomputer system. Felix is designed to support a variety of file systems, virtual memory, and database applications with access being provided by a local area network. Its interface combines block oriented data access with a high degree of crash resistance and a comprehensive set of primitives for controlling data sharing and consistency. An extended set of access modes allows increased concurrency over conventional systems.

1. INTRODUCTION

The Felix File Server has been developed at Bell-Northern Research as a part of a broader experiment in developing a distributed multicomputer technology. The technology encompasses a high-speed local communications network, a user workstation, the File Server, a network operating system, and a concurrent programming language. It thus provides the necessary building blocks for developing a variety of distributed systems.

The File Server is the storage component of the system. The principal goals of the File Server include support for the virtual memory environment of the workstation, the sharing of data, and the secure storage of data.

This paper describes the first layer of services provided by the File Server. This layer provides a base on which additional layers, such as conventional file systems with user defined directories and/or data bases, will be (and are being) built. A brief description of the implementation is also included. Emphasis is placed on the areas in which the system differs, or where it exceeds, the capabilities of systems described elsewhere ([1], [2], [3]).

2. OBJECTIVES

The objectives guiding the design of the Felix File Server include:

- support for a multiplicity of file systems
- support virtual memory
- high degree of data integrity and crash resistance
- maximal physical sharing of resources and logical sharing of data
- adequate level of security
- provision of capabilities required for building distributed file systems and distributed data bases.

The above objectives have led to the design described in this paper. The key characteristics of the File Server interface are:

- block oriented files
- atomic update of a file or a set of files
- capability based access to files
- highly flexible mechanism for shared access.

It is our belief that Felix provides a suitable base on which higher level file systems as well as data bases can be built. The key features distinguishing this File Server from other similar systems (eg. [2]) are the concepts of File Sets, File Set Commit, and the high degree of sharing possible. The relative simplicity of the interface and the underlying implementation makes support of low level functions such as virtual memory feasible. At the same time, the interface can almost directly support a higher level function such as transaction processing. Unlike [1], we have not attempted - in this layer and at this time - to solve the distributed file server problem. However, extensions in this direction are expected in the future.

3. INTERFACE DESCRIPTION

3.1 Secure Files

A Felix file is a named collection of data blocks. Every file is uniquely identified by a system generated File Identifier (FID). At this level, there is no concept of file ownership. A file is created as a result of a creation request by some client process. At this time, the system generates a FID and hands it back to the requestor.

The FID serves as a universal access capability to the file. The creator of a file can pass this capability to other clients. The capability can be passed without modifications, thus providing full access privileges (including the power to delete a file), or it can be restricted through system defined primitives to a specific access mode (e.g., read only).

Once a file is created, it is securely maintained by the system until an authorized client issues a "delete file" request. The file is then deleted and its FID is guaranteed not to be reused for a given period of time.

The use of a capability based access mechanism and the fact that Felix does not impose a specific directory structure on its clients are the chief reasons why clients must explicitly delete the files they no longer need (see the discussion of design decisions in section 6). It is expected that a proper and timely deletion of files will be automatically handled in one of the upper layers; we should note that such a higher layer is only a client with respect to the first layer being described.

A Felix file is secure in the following sense:

- System crashes never result in a file being left in an unknown, mutilated or inconsistent state. In addition, with the proper use of the multiple file commit mechanism, mutual consistency of a set of files is also guaranteed.

The size of a file is for all practical purposes unrestricted although it cannot exceed the size of the physical storage media. In addition, an implementation limit of approximately 16 GB may be also imposed.

3.2 Access to Files

To access a file the client process must first open the file in a proper mode. The open request results in the creation of an instance of the file.

TABLE 1
Interface Primitives

```
FS_OPEN (array of (FID, ACC_MODE), WAIT_TIME)
        ----> (SREF, array of FREF)
FS_COMMIT (SREF)
FS_CLOSE (SREF,COMMIT_OPTION) --> (STATS_RECORD)
CREATE (SREF of NOREF) ----> (FID)
DEL_FILE (SREF or NOREF,FID)
OPEN (SREF or NOREF, FID, ACC_MODE, WAIT_TIME)
         ----> (FREF)
COMMIT (FREF)
SAVE (SREF or NOREF, FREF) ----> (FID)
CLOSE (FREF, COMMIT_OPTION) ----> (STATS_RECORD)
VERSION (FREF) ----> (INSTV#, FILEV#)
READ (FREF, ADDR) ----> (BLOCK)
WRITE (FREF, ADDR, BLOCK)
DEL_BLOCK (FREF, ADDR)
CAP_MOD (FID1, MODIFIER) ----> (FID2)
DIR_PUT (FNAME, PASSW, FID)
DIR_GET (FNAME, PASSW) ----> (FID)
```

An instance can be thought of as a virtual copy of the file (see sections 3.4 and 3.5). A "file reference" is handed back to the client who then uses it to access the instance.

Once a file is opened, random read and write access is possible on a block basis. A request to access a block that has not yet been written results in an exception condition. The size of a block is the same as the size of the VM page (1 KB). An opened file instance can be saved - the SAVE() primitive produces a new file which is a copy of the instance specified.

The description of the file access mechanism is not complete without discussing commitment, sharing and File Set concepts. We shall do that in the three sections following.

3.3 File Sets

It is often desirable to manipulate a number of files as a single entity. For example, a client may wish to open several files, update them, and then commit all the changes in one indivisible operation. Alternately, he may decide to abort, with the implied need to discard all the changes made during this transaction.

The File Set concept provides a solution to this problem. A File Set is a dynamically named collection of mutually consistent file instances. A File Set is created by invoking the FS_OPEN() primitive which returns a "set reference" - a unique identifier for this File Set. Changes to all the files in the set are made permanent by invoking the FS_COMMIT() primitive. FS_CLOSE() closes all the files instances involved and terminates the set. Thus, FS_OPEN() and FS_COMMIT() or FS_CLOSE() serve as brackets defining a transaction.

Normally, all the files belonging to a set are identified as parameters to the FS_OPEN() primitive. However, it is also possible, by specifying OPEN(SREF,...), to "add to" a set (see Table 1). We shall call this mode of opening an incremental open. On the other hand, by invoking OPEN(NOREF,...), a file can be opened in a stand-alone mode and the resulting file instance behaves like a fixed single-element File Set. In addition, file creation, deletion and copying/saving can be also made conditional upon the commitment of a specific File Set.

3.4 Commitment

Commitment is the major tool for achieving data integrity and consistency. This capability is available in two flavours. The Single File Commit, invoked by the COMMIT() primitive, allows a number of changes to be applied to the file in one indivisible operation. The File Set Commit, invoked through the FS_COMMIT() primitive, allows changes to several files to be made in one atomic operation.

The single file commit mechanism behaves as follows. When a client opens a file in a write mode, the File Server creates an instance of the file. This instance can be thought of as a new copy of the file. The old copy is kept as back-up while the new copy is used for making updates to the file. Having performed an arbitrary number of write operations, the client issues a commit command. At that time, the update copy replaces the back-up copy in one indivisible operation. Also, a new copy for future updates is created. Alternately, if the client decides to abort, the updated copy of the file is simply discarded.

Note: Although the commit mechanism behaves as described above, its implementation differs in the sense that the unit of duplication (or sharing) is a single block and not the entire file.

The File Set Commit mechanism is an extension of its simpler brother. It should be noted that once a file belongs to a File Set, it can be only committed as a member of the set, i.e., via the FS_COMMIT() primitive.

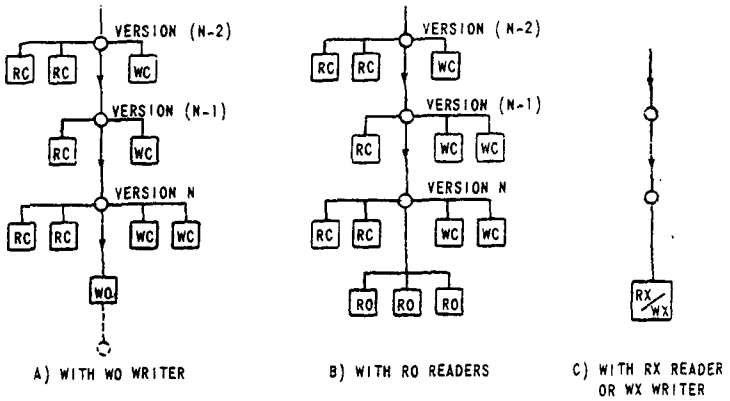
Commitment gives rise to the notion of a file version. Whenever a file is committed, its version number is incremented by one (see figure 1). Commitment is also important for file sharing, since no changes to the file are visible externally until they are committed.

3.5 Sharing

Sharing of files is possible and it is supported by six different access modes. The access mode is specified at the time the file is opened. The capabilities provided include:

- Read Copy (RC) access. The requestor is given a copy of the most recent version of the file for read-only access. Concurrent writing or reading of the file by other users is allowed.

- Write Copy (WC) access. The requestor is given a copy of the most recent version of the file for read/write access. He is not allowed to commit his copy. This mode will support, among other things, the sharing of writeable VM images. As with the RC mode, concurrent writing or reading of the file by other users is allowed.



FILE ACCESS MODEL
FIG 1

- Read Original (RO) access. This mode provides read-only access to the latest version of the file. No concurrent writer (unless it is a WC writer) is permitted, hence the file cannot change behind the reader's back.

- Write Original (WO) access. This is the standard mode for write access. It allows concurrent RC readers and WC writers. These concurrent users only see the changes that have been committed by the WO writer up to the time of their open request.

- Read Exclusive (RX) access. The requestor is given exclusive read access to the file. Since no concurrent users are permitted, this mode provides a primitive semaphore capability on the file.

- Write Exclusive (WX) access. This mode is similar to the RX mode while allowing write access to the file.

An open request that cannot be satisfied immediately is queued. Table 2 shows how an open request is handled depending on the access mode employed by the existing user(s) of the file. Figure 1 gives a pictorial representation of the access model.

TABLE 2
File Sharing
(Granting a new open request as a function of existing users or queued requests)

		Mode of existing users or queued requests						
		No User	RC	WC	RO	WO	RX	WX
Mode of the new OPEN request	RC	I	I	I	I	I	Q	Q
	WC	I	I	I	I	I	Q	Q
	RO	I	I	I	I	Q	Q	Q
	WO	I	I	I	Q	Q	Q	Q
	RX	I	Q	Q	Q	Q	Q	Q
	WX	I	Q	Q	Q	Q	Q	Q

I = access is granted immediately
Q = request, is queued until all existing users accessing the file in the column specified mode quit, and all previously queued requests are satisfied.

The six access modes identified above are a combination of three locks (Copy, Original, Exclusive) and two modes (Read, Write). We shall sometimes use the term "conventional" when referring to the Original lock or a transaction using the same.

Finally, we note that a client process may open the same file more than once. It can use the VERSION() primitive to identify the file version of any one of its file instances. Each client process must close all the instances it has opened.

3.6 Directory Service

Each client is responsible for storing and remembering the FID's of the files it has created. A file can be used for this purpose - as a directory - and directories can be organized into hierarchies. However, this does not solve the problem of where to store the directory root.

The directory service provided at this layer of the File Server solves exactly this problem; it is not intended to provide a general purpose directory service. The service provides two key functions. The first one allows clients to make an entry into the directory. The entry is a pair (filename, F), where filename is a client defined string of up to 16 characters and F is a FID of a previously created file. The second function is the translation from a filename to a FID. An optional password protection is also provided.

In addition to the above, it is also possible to create a file and make a directory entry in one indivisible operation. Primitives for changing and deleting an entry are also available.

3.7 System Crashes

One of the key objectives guiding the design of Felix was to eliminate the undesirable effects of system crashes. Although the effects of a crash cannot be completely hidden, they can be limited to a single well-defined event.

Faults that cause a crash of the File Server can be classified into three broad categories: File Server failures, disk controller failures, and media failures. While disk shadowing can be employed to counteract the effect of media failures, the effect of crashes caused by the other two classes of failure is limited to the following:

- unopened file: no effect.
- opened file: the file is closed without commitment. This means that all instances are wiped out and the file is restored to its last committed version before the crash. When the File Server comes up, the clients must re-access the file.

Note that once a client receives a positive acknowledgement to its commit request it can be assured that the file has been committed. The lack of such an acknowledgement means that the file might or might not have been committed; the VERSION() primitive can be used to find out.

System crashes of the client machine will be detected by the operating system. Their effect on the selected file instances will be the same as the effect of the File Server crashes.

4. APPLICATION CONSIDERATIONS

4.1 Single File

The use of the six access modes defined earlier is intuitively obvious when a single file is considered. The RC mode is the natural mode for activities such as printing a file or generating a report. Sharing of virtual memory images is an example of a problem that may be solved by utilizing the WC mode; this mode, augmented with the SAVE() capability, also makes a good tool for a software development and management system. The RO and WO modes, if not mixed with other access modes, provide the equivalent of conventional two-phase locking. Finally, the RX/WX modes allow one to prevent other clients from obtaining a copy of the file.

4.2 File Sets - Conventional Locks

In database terminology the term transaction is used when describing a sequence of operations applied to some data. We have earlier defined (Section 3.3) transaction as the sequence of operations, bracketed by FS_OPEN() on one side and FS_COMMIT()/FS_CLOSE() on the other, applied to a File Set. In the remainder of this section we shall use the two terms somewhat interchangeably.

Transactions using only conventional (RO or WO) locks on their file set follow a conventional two phase locking protocol [6]. This maps the conditions for serializability into the conditions for deadlock freeness; in practical terms the transactions either complete successfully and are therefore serializable, or they deadlock (and were therefore potentially non-serializable).

In the case of deadlocked transactions the File Server detects and breaks the deadlock by aborting judiciously chosen transactions, and the client machines must be prepared to receive such an abort notification. In particular, automatic restart of aborted transactions is a responsibility of the client machines.

The File Server interface allows and encourages predeclaration of read sets; transactions which predeclare all resources are never aborted due to deadlock. Many simple transactions can normally do a complete predeclaration. In other cases an attractive strategy would be to predeclare only the files "normally" used, but to use incremental opens in infrequent or exceptional cases.

4.3 File Sets - Unconventional Locks

From the definition of the Copy and Exclusive access mode it is clear that a set of transactions which use only these modes are logically equivalent to a set of conventional transactions - consequently there is no increase in concurrency. The interesting case (and the advantageous one) is therefore that involving a mix of Original and Copy locks.

The first requirement in the mixed case is that a transaction using, say RC access, must get a consistent set of the desired files. This is guaranteed by the File Server provided that the requests are made simultaneously in the FS_OPEN(). In the current implementation this consistency does not extend to incremental opens in RC mode. However, at the cost of keeping, for every file, all its versions that are younger than the oldest transaction in the system, it is possible to extend this consistency to the incremental Open case as well.

For ease of presentation we will consider examples of pure read only transactions, then transactions that can use a mixture of Copy and Original locks, then transactions requiring Exclusive locks.

The simplest case of mixed mode accesses is that of pure query transactions; these may use RC access for all files, while the corresponding update transactions use WO locks. According to the locking rules of Table 2, there is then no interference at all between queries and updates. Thus, a relational query package could quite reasonably access all files of a database in RC mode and hold this lock for the duration of a query session without fear of holding up update transactions.

A slightly more complex case is that of a "two layer" database, e.g., a database directory and database "files". Normal update transactions use WO mode to update the database files and RC mode to access the directory contents. As a result, directory updates (using WO mode) may proceed without holding back normal update processing. Note that in this case a normal update may physically commit later than a directory update even though it is logically "earlier".

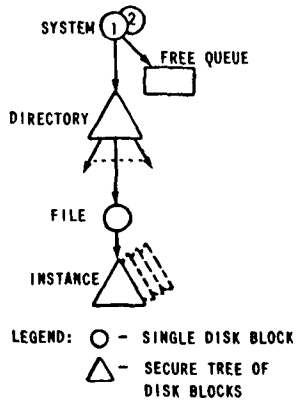
Continuing this example, let us consider an infrequent data base reorganization that modifies both the top and bottom layers, e.g. addition of a new semantic constraint followed by automatic checking that it is satisfied by the current database state. Such a transaction would need to use Exclusive locks at the directory level in order to create a consistent global division into before/after.

Taking a transaction analysis viewpoint [4], we can observe that an Exclusive lock breaks a potential cycle in the transaction class conflict

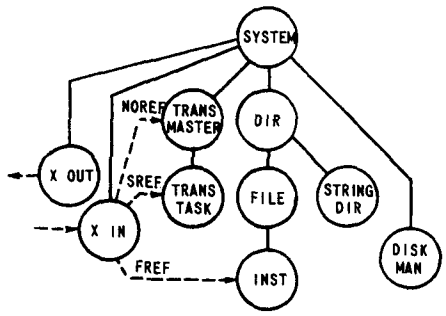
graph. The resulting acyclic diagram then allows the use of (Copy, Original) mixture locks. Readers familiar with SDD-1 [15] will recognize that there is a close analogy between SDD-1 protocols and our locks (Table 3). As in SDD-1, if certain transactions use the heavy protocol (P4 or Exclusive lock), then others can use a lighter protocol than normal (ie. P2/Copy instead of P3/Original). This suggests that our access model can benefit from SDD-1 transaction analysis while retaining a two-phase locking style of interface.

TABLE 3
SDD-1 Protocols vs Felix Locks

SDD-1	FELIX
Interclass Serialization	Original
P2	(Copy, Original) mix
P3	Original
P4	(Excl., Original) mix



DISK STRUCTURES
FIG 2



TASK STRUCTURE
FIG 3

5. IMPLEMENTATION

5.1 Assumptions

The integrity of Felix files is based on three obvious assumptions:

- No undetected hardward failures.
- Absence of media failures (or the use of a protecting redundancy scheme, eg. disk shadowing).
- Software correctness.

While our controller satisfies the first requirement, the existing implementation does not use shadowing. However, extensions in this direction are planned for the near future. To achieve software correctness, we have strived for "clean" design of minimal complexity and we have employed auditing for the specific reason of discovering potential software errors.

5.2 Overview

Felix software is implemented in an extended Pascal. The language is a superset of UCSD Pascal with extensions for parallel programming. The concurrency constructs include tasks and an intertask communication mechanism similar to the rendez-vous in ADA.

The fundamental data structure underlying the implementation is a tree of disk blocks. The organization of the disk is shown in figure 2; section 5.3 describes the integrity mechanism which extends with minor changes to the complete tree shown in figure 2. The task structure, closely resembling the disk structures, is shown in figure 3. It can be observed from the two figures that there is a natural division into four levels - system, directory, file and instance. The functions performed at the individual levels are:

- Instance. When created, the Instance task is given a copy of the file root and the identity of its client. It then facilitates access to the file. A commit request from the client is propagated up the task tree, with the acknowledgement coming from the system level down.

- File. The key responsibility of the File task is to take care of the various aspects of file sharing. This includes queueing of new open requests and creation and control of instances and versions. The fact that a file is a sequence of file versions becomes obvious at this level.

- Directory. The File Server internally employs a single flat directory. Its key function is to map a FID into the current disk address of

the file root.

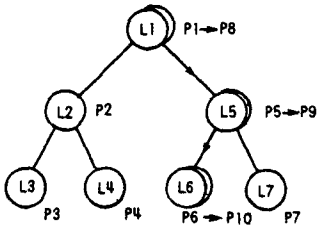
- System. The System task is the final authority in performing a commit. It also manages free storage. Two well-known disk blocks are used for alternately writing the system root to the disk. In case of a system failure, the system is automatically restarted using the latest correctly written root of the two.

The Transaction Task has the responsibility for managing a File Set. This task carries out all the SREF type commands by controlling and coordinating the opening, committing and closing of all the file instances in the set.

5.3 Secure Tree

A file is physically organized as an unbalanced tree of disk blocks. The leaves carry the data while the internal nodes only provide the access mapping. The integrity of the tree is based on a technique which makes the first write (after a commit) of a logical block to a new physical address. For efficiency reasons, subsequent writes of the same block are done in place.

As an example consider the tree shown in figure 4. The request to write logical block L6 results in the deallocation of physical blocks P1, P5 and P6, and logical blocks L1, L5 and L6 are written to a new physical address - P8, P9 and P10, respectively. Subsequent updates of L6 only rewrite P10 in place, thus requiring only one rather than three write operations. The allocations and deallocations are "temporary" - they only become permanent if and when the file is committed; in fact, they may have to be undone should the system crash or the client abort.



SECURE TREE - AN EXAMPLE
FIG 4

By now it should be obvious that the tree access algorithm must differentiate between two kinds of blocks - temporary (i.e., allocated during the current commit interval) and permanent (ie., allocated prior to the last commit). Tree nodes must therefore maintain, for each son node, a one-bit mode indicator in addition to its address.

A file commit resets the mode of all the blocks in the tree to permanent. It is desirable to do this without having to rewrite the majority of the blocks in the tree. Our approach to this problem uses a "virtual reset"; at commit time, the reset operation only changes the value of incore variable ROOT_MODE, while the resetting of other nodes is done concurrently, but logically ahead of, a path write.

Because of the various overlapped versions, a file is physically organized as a rather complex confluence hierarchy of disk blocks. Logically however, each instance sees only a simple tree structure and the file can be treated as a simple sequence of trees that are logically disjoint. Each instance "deallocates" disk blocks freely and independently as required by the tree algorithm. For non-committing instances (ie., WC or aborting instances), such deallocations are null operations. For committing instances, the deallocations are committed simultaneously with the instance commit, but the blocks are not "free" until they have been deallocated by all existing versions of the file.

Atomic multi-file commitment is implemented by applying the same technique at the level of the directory. The process is facilitated by the Transaction Task which prompts each file to make its own relative commitment and then asks the Directory for one bulk commit. Any file creations or deletions attributed to the same transaction are also committed at this point.

The use of the above techniques reduces the problem of atomic commitment to that of managing free disk blocks correctly and efficiently. The simplest way to do this is by "garbage collection", but we have also devised a continuous way of tracking free storage. This second technique, which employs a "history log" of events such as file open, block allocation, and file commit, might be preferable under certain operating conditions.

The problems usually associated with garbage collection or auditing of an active system are solved relatively simply by taking what amounts to a "read copy" image of the tree shown in figure 2.

6. CONCLUSIONS

The interface provided by the File Server stands half way between what could be called a "file system" interface on one side and a "data base" interface on the other. Indeed, this is a result of the objectives we set up at the beginning. The relative simplicity of the interface makes it suitable for implementing higher level file systems with character or record oriented files and user defined directory structure. At the same time, the high level of concurrency, the security,

and the File Set concept appear to be simple but effective tools for dealing with notorious data base problems such as consistency, deadlock and transaction processing in general. While the level of concurrency is similar to that of SDD-1, our access modes are simple and intuitively easy to understand.

We have made several conscious decisions with respect to the capabilities provided (or not provided) at this level. The guiding principal was to exclude anything that is not absolutely essential and to only include capabilities that would have to be done by every application anyway.

The block oriented interface is a logical choice because of the need to support virtual memory. It is also a suitable base on which conventional file systems as well as data bases can be built.

It was felt that security and sharing should be considered from the very beginning. These capabilities are very difficult to add to an existing system because of their profound impact on the design, implementation, and performance.

Finally, it did not seem reasonable that this layer should be aware of "users" and be involved in all the problems of user ID management. We also felt that we should not impose a specific directory structure on our applications. This lead to the decision of providing a simple capability based access. An implication of this decision is that clients must explicitly delete files that they no longer need.

We are now gathering experience with the existing implementation. Planning for some of the higher level extensions and for a Distributed File Server is also in progress. We hope to be reporting on these activities in the future.

7. ACKNOWLEDGEMENTS

The references and many stimulating discussions with our colleagues at BNR lead to the ideas presented in this paper. P. Cashin, N. Gammage and F. Mellor should be explicitly mentioned for making this project possible. J. Howard provided detailed comments which helped to improve the clarity of presentation.

REFERENCES

[1] Issues in the Design and Use of a Distributed File System, Sturgis, H., Mitchell, J., Israel, J., ACM SIGOPS Operating System Review, July 1980.

[2] The Cambridge File Server, Dion, J., ACM SIGOPS Operating System Review, October 1980.

[3] WFS: A Simple Shared File System for a Distributed Environment, Swinehart, D., McDaniel, G., Boggs, D., Proceedings of the 7th Symposium on Operation System Principles, 10-12 December 1979.

[4] SDD-1: A System for Distributed Databases, Rothnie, J.B., et al., Computer Corp. of America, Technical Report CCA-05-79, August 1, 1979.

[5] Concurrency Control in SDD-1: A System for Distributed Databases, Computer Corp. of America, Technical Report CCA-03-79, January 1, 1979.

[6] Locking and Recovery in a Shared Database System: An Application Tutorial, Date, C.J., Proceedings of 5th Intl. Conf. on VLDB, October 1979.

[7] On the Design of a Reliable Storage Component for Distributed Database Management Systems, Menasce, D.A., Landes, D.E., Proceedings of IEEE, 1980 - 1534 - 7/80.