DFS925: A distributed file system in a workstation/LAN environment

Michelle Joy Arden

Computer Science Division
EECS Department
University of California at Berkeley

ABSTRACT

The advent of local-area networks has made possible the implementation of **distributed file systems**: file systems which physically encompass several distinct machines but which provide a global file system logically spanning the connected machines. Users who access files across this system are unaware that their files may logically reside on a machine other from their own. Distributed file systems vary widely in functionality and design, differing largely in their degree of distributed control, provisions for data consistency, and transparency.

This report describes the design and implementation of a prototype distributed file system at IBM San Jose Research Laboratory. The 925 Distributed File System (DFS925) is constructed for a network whose nodes are high-performance engineering workstations, connected by a reliable local area network. Emphasized are distributed control, the maintenance of workstation independence, and data consistency. DFS925 will be used as a research and development tool in the Computer Science Department at IBM San Jose.

May 15, 1985

Table of Contents

1.0 Distributed Systems 1

2.0 DFS925 Development Environment 2

3.0 Report Structure 2

4.0 Definition: A Distributed File System 2

- 4.1 Server Model vs. Integrated Model 3
- 4.2 Atomicity/Data Consistency 4
- 4.3 Transparency 5
- 4.4 Existing systems 5

5.0 DFS925 Motivations/Influences 6

- 5.1 Overview 8

6.0 DFS925 Implementation 9

- 6.1 The 925 Software Environment 9
- 6.2 DFS925 Structure 11
  - 6.2.1 Queueing Structure 11
  - 6.2.2 Ernestina Server Tasks 11
  - 6.2.3 Call Control Flow 12
  - 6.2.4 Sending and Optimizing Remote Requests 13

7.0 The DFS Global File System 15

- 7.1 File Access Protection 16
- 7.2 Links 16
- 7.3 Criticism 16

8.0 Transactional Support 17

- 8.1 Locking 18
- 8.2 Failsafe Guarantees 19
  - 8.2.1 Distributed Commit 19
  - 8.2.2 Transaction Abort and Checkpoint 20
- 8.3 Error and Crash Recovery 21

9.0 Performance 21

10.0 DFS925 Experience 23

11.0 DFS925: Present and Future 23

12.0 Acknowledgements 25

Bibliography 26

DFS925: A distributed file system in a workstation/LAN environment†

Michelle Joy Arden

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley

ABSTRACT

The advent of local-area networks has made possible the implementation of **distributed file systems**: file systems which physically encompass several distinct machines but which provide a global file system logically spanning the connected machines. Users who access files across this system are unaware that their files may logically reside on a machine other from their own. Distributed file systems vary widely in functionality and design, differing largely in their degree of distributed control, provisions for data consistency, and transparency.

This report describes the design and implementation of a prototype distributed file system at IBM San Jose Research Laboratory. The 925 Distributed File System (DFS925) is constructed for a network whose nodes are high-performance engineering workstations, connected by a reliable local area network. Emphasized are distributed control, the maintenance of workstation independence, and data consistency. DFS925 will be used as a research and development tool in the Computer Science Department at IBM San Jose.

1. Distributed Systems

As the need for communication between systems grows, and the cost of such communication lessens, distributed systems based upon a communications network have become a topic of increasing interest.

The motivations behind the development of such systems are well-known, being both economic and conceptual. The optimal use of storage and processing power across the network, as opposed to within one network node, becomes feasible with the development of such a system. Whereas each node maintains independence, the system is seen as a unified collection of resources by the individual user. The difficult mix of independence and cooperation which must be addressed gives the problem of design its interest: the challenge arises in taking advantage of node autonomy while maximizing the size and availability of the pool of resources which the user sees.

The introduction of the inexpensive, yet powerful, personal workstation has led to distributed systems which link these small, generally single-user systems together by networks. When a local-area network is used as a link, its high bandwidth and reliability decrease the cost of communication between network nodes; frequent contact between workstations and other, physically remote, computers becomes feasible. The workstations can exploit the capabilities of

† The design and implementation of DFS925 took place from May 1984 through January 1985: the work herein described was accomplished by the author under the supervision of Dr. J.C. Wyllie, Office Systems Laboratory, Computer Science Department, IBM San Jose Research. The views and conclusions contained in this document are those of the author, not those of the IBM San Jose Research Laboratory. Publication of this report was supported by the Computer Science Division, University of California, Berkeley.

each other and more powerful systems using the inexpensive communication provided.

The combination of such small independent clients cooperating together to offer a cohesive, powerful service through network communication characterizes an emerging breed of network-based distributed systems. This report describes the design and implementation of a file service based upon this system paradigm: a distributed file system encompassing a group of engineering workstations connected by a token-ring local area network.

2. DFS925 Development Environment

The 925 Distributed File System has been developed at the IBM San Jose Research Laboratory, in the Office Systems Laboratory. The multiprocessor MC68000-based 925 workstation, developed by research staff in the Office Systems group, is a prototype high-performance workstation. The 925 workstation offers bit-map addressability, allowing for high-resolution graphics, and can contain up to five processor boards. 925 workstations contain a local 20-40MB disk, incorporate a 2MB RAM, and are connected by a 4Mbit/sec experimental token ring local area network. Software packages developed for the 925 include WHIM, a window manager, FS925, a local file system upon which DFS925 was built, CP925, a message-based operating system; and CSS, the Communications SubSystem, through which both local communications and communications over the local area network can take place. Compilation and binding are done on an IBM mainframe, then downloaded onto the 925; the languages used are PL8 and Pascal8. To date, about twelve 925 workstations are in active use.

3. Report Structure

Section 4's exposition of the definition and design alternatives available for a distributed file system gives the necessary background for the subsequent discussion of DFS925. The system model, degree of data consistency guaranteed, and the flexibility of the data model are the main attributes of comparison between distributed file systems.

Section 5 summarizes the historical influences and motivations behind DFS925. DFS925, intended both as a research and development tool, gives particular emphasis to the maintenance of distributed control and the provision of considerable database functionality. The functionality vs. performance tradeoff is important to the design.

The development tools and environment which led to the creation of DFS925 are presented in Section 6; structural details of the design and implementation conclude that section. Location transparency is a necessary component for a useful distributed system; DFS925's implementation extends the concept of transparent distribution to the system structure. Sections 7 and 8 describe the DFS global directory structure and transactional support. Throughout the report, performance considerations are commented upon; performance measurements and their significance are included in Section 9. In closing, Sections 10 and 11 discuss the current status and future plans for DFS925.

4. Definition: A Distributed File System

The definition of a distributed file system is not a subject of general agreement: features and expectations vary greatly from implementation to implementation. Birrell and Needham [BIRR80] address the task of definition in their paper discussing the "Universal File Server". They describe the limits between which a file server may vary, from a virtual disk to a complete filing system, the latter providing group structuring, file access protection, complete crash recovery techniques, concurrency control for regulating concurrent access to files, and a naming mechanism. These parameters can be expanded into a list of design issues for a distributed file system: ease of sharing, availability, data consistency across the network, degree of distribution, workstation independence, and performance should all be considered.

Two design issues not emphasized by Birrell and Needham refine the resource pool concept. First, transparency of file location is a necessary characteristic of a genuinely useful distributed file system. The user and/or the application layers which are built upon the file system are not

required to know any information about the physical location of a file in order to access it. They may indeed be unable to gather such information, and are unable to influence file placement directly. The distributed file system is assumed to be responsible for file placement and the maintenance of a global view of the system, if any.

Second, uniform distribution of the file system implies that individual views of the file system's directory structure are globally consistent. Each node, or machine which shares the file system over the network, perceives the same directory structure as another. This is not the assumption commonly made for *network* file systems, where one node may see a distinctly different view of the directory structure than another [WELC85, KARE85, HUNT85]. Portions of a network file system may only be seen by subsets of nodes. By contrast, uniform global perception enforces equal treatment of all nodes; availability of one node is equivalent to availability of another.

For the purposes of system comparison, Birrell and Needham's above criteria and the subject of transparency can be condensed into three areas.

(1) The most structurally significant is the choice of the system design model, of which two exist: the server/client model and the integrated model. Birrell and Needham take only the server/client model as the template for their definition. This choice most directly affects the balance of independence and cooperation in the resultant system.

(2) The extent to which a file system incorporates traditional database functionality, guaranteeing data consistency and atomicity, is another criterion for comparison. Distributed databases are considerably more complex than their local cousins: guaranteeing complete data consistency may be prohibitive in the face of network or node failure. Many distributed file systems provide no database building blocks in favor of a simplified system; this may cause the correct implementation of a distributed database system using file system primitives to be impossible.

(3) Finally, the generality of file identification and organization affects the transparency the distributed system affords. A uniformly distributed system grants the same view of the file system to each member of the system. The least general distributed system should provide file location transparency; the most general will allow heterogenous nodes to share the distributed file system. The latter is a much-desired but extremely difficult goal.

4.1. Server Model vs. Integrated Model

The client/server model relies upon one or more members of the distributed system having a special role, such as file storage, name lookup, authentication, printing, or computation. In the case of a file service, separation of reliable storage from the client workstations on the network offers some strong incentives. Unlike processors, whose costs per instruction increase with power, large disk drives are much less expensive per data unit stored than their smaller counterparts. Economy of scale encourages the cooperative use of large disk drives by several workstations. Maintenance of the file servers is centralized, which is advantageous when changes are made to the file system and for backup. When one or few servers exist, data consistency considerations are less complex. A related advantage is that workstations themselves may be made less complex; local workload may decrease as there is little or no processing overhead on the workstation for file system and disk handling. Availability is proportional to the reliability of the file server: as in a time-sharing system, users may use any workstation and access their files if the file server is active.

However, the client/server model has some drawbacks. The reliability and availability of an operation which depends upon multiple servers is proportional to the product of the reliabilities of all network links and machines involved. If the client/server roles are distinct, all file system calls from the client workstation must inevitably involve at least one message across the network. While the cost of sending a message across a network is decreasing in importance as local area networks become faster and more efficient, such a call is yet substantially slower and more costly than a local procedure call. Distinct file server(s) inevitably compose a performance bottleneck for

each of the local workstations. The advantage of "owning" a local processor on a workstation disappears when productivity is limited by the bottleneck on an overloaded shared file server. According to Lazowska's studies on the performance of diskless workstations [LAZO84], under "average" request loads, a file server's CPU becomes a bottleneck when it serves 20 workstation clients regardless of packet size (performance is degraded by at least a factor of two). Welch [WELC85] has discovered that the BNCF file server becomes a bottleneck when it serves 5 workstation clients with heavy demands. Local client caching upon a small local disk yields a dramatic improvement: however, each client must then have some kind of local file system in order to allow such caching. The advantage derived from non-duplication of file system code upon the clients begins to be eroded. Finally, the useful independent existence of a client workstation is at an end, since the client cannot obtain any file information except through communication with the server machine. The advantages of node independence are decreased through the cooperative demands of the distributed system.

The integrated model (exemplified by LOCUS [WALK83] and the 925 Distributed File System) requires all members of the distributed system to contain the same functionality. The file service, name service, and other service software is duplicated in each of the nodes in the system; each workstation is an independent unit, able to access those files which are resident locally without cooperation with other workstations. Steps must be taken to prevent fragmentation of directory information and dangling references. If the network is rendered inoperable, at least the portion of the file system stored locally should be accessible. Similarly, the loss of use of one node on the network should only preclude the access to that portion of the file system stored on that node. This is a strong advantage over the client/server model, where the loss of the file server may spell the loss of use of the entire file system. Judicious use of file placement strategies can minimize the number of messages necessary to access a file, decreasing use of local caches and avoiding expensive cache update algorithms insofar as possible. Such strategies may use many messages themselves, however. Distributed control is implicit, which complicates the implementation of crash recovery and concurrency control. The integrated system structure therefore encourages increased availability; the problems of distributed coordination are, however, much harder to avoid than in the server/client model.

4.2. Atomicity/Data Consistency

Providing some database functionality is complicated greatly with the distribution of a file system. If data consistency is guaranteed, the user must be confident that his actions are reflected in the data, regardless of the degree of sharing allowed, the state of the network, and the reality of imperfect data transmission. These requirements lie behind the ideas of atomic transactions, automatic recovery, and concurrency control, which are addressed in the property of *atomicity*. Atomic transactions guarantee that actions seen as completed by the user are completed by the file system, regardless of the states of all the (potentially remote) participants in the action. Automatic recovery relates to the support of system crash recovery: the unavailability of any workstations or the network must not cause the data at any location to be in an inconsistent state. Through concurrency control, users are prevented from seeing inconsistent data and losing updates to their data due to update conflicts between users.

There is argument as to how many of these capabilities should be provided in the underlying file system. Applications vary in the degree of consistency required; the consistency demanded by a database system produces substantial overhead. Allowing choice in the degree of consistency provided by the file system is one option: this is the course taken by DFS925 and many of its predecessors. With the introduction of choice, simplicity suffers, and care must be taken that the perception of the integrity of the distributed file system is uniform over the user environment. Explicit involvement by a file system administrator may be necessary to enforce data integrity if too many data consistency shortcuts are allowed. Providing too little database support is another danger. Relegating the implementation of all data consistency to an application layer may result in the inability to construct a database system atop the file system with guaranteed consistency. The INGRES database system has encountered such problems with the underlying UNIX file system [STON80, STON84]. Database designers may prefer to circumvent the file system

UNIX is a trademark of AT & T Bell Laboratories

altogether, accessing the raw disk directly.

4.3. Transparency

File identification, naming, format, and organization are all facets of file system design which impact transparency of use and the extensibility of the system. The decisions made in the area are strongly influenced by the size and composition of the distributed system planned. A distributed system incorporating nodes of many different machine types, operating systems, file formats, and naming conventions would be extraordinarily useful for both research systems and the outside world. Few computer environments encompass one single type of machine; permitting file sharing across machine boundaries would allow for greatly increased efficiency of resource use. In tackling such a task, however, the developer takes on all the accumulated problems of compatibility of the last three decades. It is unfortunate that no uniformly accepted standards of data representation or remote procedure call protocol exist: the efforts to produce such standards are continual.

Given the number of translation interfaces possible, a completely heterogeneous distributed file system within the constraints of transparency is, as yet, unrealized. The server/client design paradigm carries much more potential for its incarnation, however: N interfaces must only be implemented between the client machines data representation and the file server representation (for N different types of machines) , rather than the cumulative $N^*(N-1)$ mutual interfaces which would be necessary using the integrated system design. Efforts have been made in both research and industrial arenas to create a standard file server interface for access by several of the major file systems [LYON84, DEC84]. Introduction of file server call translation and standard data representation protocols necessarily degrades performance substantially: optimization of such translation is clearly an area of current research interest.

4.4. Existing Systems

The intricacies of implementing distributed control, guaranteed consistency over a network, and location transparency have limited the number of distributed file systems which exist; the majority are within the domain of research rather than development. Most systems stop short of true transparency, only a few have attempted distributed control or the inclusion of distributed database functionality. Allowing heterogeneity is largely a dream of the future. WFS [SWIN79], CFS [MITC81], LOCUS [WALK83], XDFS [ISRA78], The V System [CHER83] and the SUN distributed file system [LYON84] are among some innovative systems which have formed the basis for past and present work in distributed file systems. Looking at these systems briefly aids in motivating the development of DFS925.

WFS was one of the first file systems to attempt distributed access. Following the client/server paradigm, it was developed at Xerox PARC, connecting a network of Alto workstations on the Xerox Ethernet. WFS's implementation was little more than a "virtual disk"; concurrency control and transaction support were left for application layers. Locking commands were provided by WFS but not enforced. Atomicity was provided only at the level of single page requests: each operation accessed at most one page. The argument in favor of WFS was that it was extremely simple, and hence flexible; each client could implement transaction, directory, and access protection services as needed. WFS file operations were optimized by a set of specialized access protocols. The small size of the system made it feasible for its workstation environment. Building these services into the client was certainly possible. However, while WFS allowed for the non-local storage and access of files by its workstation clients, it ignored the other main purpose of a file server: to facilitate the sharing of data among clients.

CFS, the Cambridge file system, accompanied WFS as one of the first distributed efforts. CFS provided significantly more functionality, allowing file-level locking and supporting single-file atomic update. CFS allowed variable levels of data consistency to be specified; files could be classified as *normal* or *special*, according to the level of guarantee desired. Many files do not contain critical data; the choice of file classes is a reasonable performance optimization.

LOCUS, developed at UCLA, is a distributed operating system based upon a distributed file system. Clients are VAX's; the network used is the Ethernet. The LOCUS design has concentrated on the issues of transparency, reliability, and availability. It follows the integrated system model; each node is both a server and a client, and provides guaranteed data consistency across the system. To date, LOCUS is the only distributed file system to have developed propagation update algorithms for replicated files, at substantial performance cost. Multiple-server, multiple-file transactions are allowed. Specialized file access protocols optimize file access; the UNIX directory service is built into the file server. Of all the file systems here noted, LOCUS provides the most complete transactional support, allowing nested transactions, which has increased the complexity of the locking algorithm considerably. The file system's close integration with the operating system has contributed largely to its acceptable performance despite the bulk and complexity of the design. LOCUS has been used successfully at UCLA for several years.

XDFS, or the Xerox Distributed File System, is a multiple-server system which has had extensive development at Xerox PARC. XDFS supports atomic multiple-file update on multiple servers. Several novel lock optimizations were introduced in order to make it efficient for clients to maintain local caches of shared data. XDFS was written to facilitate database access, allowing byte-level access and locks.

The V System, an experimental server/client system at Stanford University, is one of several UNIX-based distributed file systems currently in development (including two presently in progress at Berkeley [WELC85, KARE85, HUNT85]). Consisting of a group of SUN workstations linked by the Ethernet, the V System concentrates on efficient file transfer, leaving the question of providing data consistency and transaction support to overlying application layers. (This is typical of UNIX-based distributed file systems. The original design of the operating system makes it difficult to incorporate locking and transactional capabilities in the file system without making changes to the operating system itself.) The designers of the V System claim to have used interprocess communication calls exclusively rather than introducing special-purpose file service calls into the operating system. The V System was designed specifically for use with diskless workstations: the server(s) are the only nodes in the system which contain disks.

General heterogeneity was not a proposed goal for DFS925; the scope of the problem was too large for consideration. It is valuable to mention that a few systems exist which aspire to heterogeneity. DEC Research is working on a system which proposes to handle VMS, ULTRIX, and MS-DOS file system requests to a file system server [DEC84] ; SUN Microsystems has announced a similar product which substitutes UNIX for ULTRIX [LYON84]. Berkeley's Remote File System [KARE85, HUNT85] also intends to include heterogeneous machines.

5. DFS925 Motivations/Influences

DFS925 is a prototype distributed file system running on a group of workstations connected by a local area network. Each workstation node has an identical view of the file system, regardless of the location of the files in the system: this property may be termed uniform global distribution. Transactions and file locking provide data consistency and concurrency control; crash recovery allows the correct recovery of data upon node failure.

The development of the 925 Distributed File System responds to a perceived twofold need of the Office Systems group at IBM Research: a continuance of the study of distributed applications, and the provision of a tool to facilitate further software development upon the 925 system. It was hoped DFS would contain seeds for challenging research in the future, yet use the resident file system insofar as possible, and be of a size such that a useful system could be produced by the author over a constrained period of time.

Characteristics of the distributed file systems cited above were used as partial guidelines for design decisions. WFS is now mainly significant historically as the first trial of file distribution; it provided far too little functionality for useful sharing among clients; the one-page atomicity constraint was a very limited guarantee of consistency. The simplicity of the system was laudable: allowing some individual file system tailoring in each node seemed reasonable and would simplify implementation considerably. The policy of placing the local/remote reference interface, or *cut*,

at the virtual disk level allows the network to appear interchangeable with a disk; DFS925 follows suit by making one local/remote *cut* at the volume (logical device) level.

By supporting file classes and breakable locks, CFS and XDFS both allowed evasions of expensive complete data consistency, yet provided the latter when perceived necessary. Permitting an application to trade off performance for consistency seemed a valuable addition to a transaction-based distributed file system: this became an emphasized goal of DFS925.

LOCUS' rich transactional functionality and inclusion of data replication was attractive, but a similar system would have been far too complex for the manpower and time available. Providing comparable transactional functionality in DFS925 was desirable, but the same efficient integration with the operating system was not possible. Shortcuts and optimizations within transactional constraints were necessary for adequate performance. Replication of files on a more simplified scale for DFS925 would greatly increase availability; the potential for such a future extension in the implementation tradition of LOCUS was kept in view. DFS925 will eventually mix replication and file migration to maximize availability and minimize remote file access. DFS925 adopts the use of a specialized file access protocol as do both WFS and LOCUS.

The V System represents a much different approach than DFS925, as it includes no data consistency guarantees. Generality of use of the IPC mechanism was appealing, since the extension of the IPC mechanism in CP925 to allow distributed interprocess communication was planned; the possibility of including extended IPC in the filesystem was left open. The V System's diskless workstations encouraged deliberation on the composition of the DFS925 network: while the 925 workstations incorporated a disk, adding diskless workstations to the system was possible. It was hoped to be able to incorporate a mix of disk-bearing and diskless workstations without any change in DFS925.

The character of the software development environment in the Office Systems Group encouraged a distributed file system emphasizing workstation independence and availability. Each workstation is dedicated to one person and one office; sharing of machines is rare. Although individual work is important, integration of distributed applications is an important step towards the creation of a coherent office systems environment. The support of a distributed file system facilitates integration through transparent interaction between applications on distinct workstation nodes.

An important motivation for DFS925 is the potential for allowing the mainframe IBM 3081 machine used for development to access files on the 925 network system. DFS925, though not a heterogeneous file system, can provide some of the advantages of such a file system by allowing host-workstation sharing. Transparent access of the DFS925 file system will be possible from the IBM 3081 currently used for compilation and binding. The latter will run 925 code (compiled for use on the 3081); it may then operate on files stored on workstation nodes by simulating a workstation with no local disk. The DFS925 file system would still be homogeneous, since the code running on the 3081 is the same as that on the 925 workstations; the 3081 simulates a 925 node with extra processing power. Such an application of DFS925 would alleviate the time-consuming task of downloading from the larger machine into the workstations.

The independent work habits of the workstation users caused a bias towards workstation autonomy in the design of DFS925. 925 users disappear frequently (locking their offices); their absence could not impact the productivity of the other participants in the distributed file system. The prototype nature of the 925 workstations encouraged the adoption of reliability as a feature.

The integrated system design model was chosen for DFS925. It was felt that this model best preserved the independence of the individual workstation nodes by still potentially allowing work in the presence of partial system failure. The integrated model seemed the more flexible of the two; the DFS925 may mimic a client/server system by having a mix of disk-bearing and diskless workstations, incorporating some of the advantages of both models. (Since the file system code is still resident upon all nodes, the system is not a true client/server structure.) File system calls would be automatically routed to the workstations controlling the appropriate disk. The economic advantages of large disk drives can be exploited in this manner. Small local disk drives can then

be viewed as automatic "caches" for files being frequently modified, files being moved to large disk drives, either manually or automatically, as they are less likely to be used.

FS925, the previous 925 file system, ensured local data consistency using a transaction model. Distributing the guarantee of atomicity, crash recovery, and extending concurrency control were consistent with the pursuit of reliability. Performance escape routes from consistency constraints, such as those used in CFS and XDFS, are provided by DFS925 for those applications which do not demand complete consistency. While giving support for most of the requirements of the "Universal File Server", DFS925, in the interest of simplicity, leaves the implementation of file access protection to applications.

Network transparency was a primary characteristic in the DFS925 design. It was felt that external file names should be independent of their locations; internal file identifications should be designed so that files may be moved from one location to another without having to find and update all references to them.

Integration with current and future 925 packages was of interest to the Office Systems group. No application with substantial continuous use of the CSS/LAN capabilities was previously in operation. Integration with planned distributed services such as the 925 name server, distributed CP925, and global file placement algorithms are possibilities for the future.

Finally, the desire for good performance encouraged implementation decisions such as the addition of multiple file system server tasks, the provision for caching, and optimization of network use through the use of a specialized file access protocol.

5.1. Overview

In the design of DFS925, the decision was made to initially emphasize the development of a uniformly distributed system, where data consistency is guaranteed, and workstation independence maintained to a high degree. A global directory system, linking across the network, distributed transactional capabilities, node crash recovery, and eventual caching are the means by which these goals are to be realized.

A user of the 925 Distributed File System should be able to access and link to all files resident on volumes belonging to remote workstations, unaware of their location, using a global pathname as reference. A hierarchical global directory structure allows uniform naming of all files in the file system. Preserving location transparency, the global directory is only a logical means of organization; physical file location is completely independent. Distributed transactions guarantee *atomicity* to the user; locking and concurrency control are provided transparently by the file system. Choice between levels of consistency, file types, and transactional guarantees allows the user to favor performance over complete functionality.

As noted, rather than having a workstation with special properties designated as a file server, it was decided to provide the local file system on each workstation with equal functionality, in the tradition of LOCUS. A uniform interface to each workstation promotes the development of a system with genuinely distributed control. File servers, while avoiding many of the consistency problems associated with a distributed file system, fall prey to the familiar availability problems of centralized time-sharing systems. The integrated system model has more complex control problems but was felt to better maintain the independence of the individual nodes. The integrated system is easily extensible: DFS925 coordinates added "file servers" (workstations with disks) automatically. Location transparency is extended to the implementation of the file system task design and control flow.

Transactions are distributed: they may include files in several locations. File consistency is eventually guaranteed through a distributed two-phase commit protocol. Crash recovery procedures guarantee data consistency in the case of a node failure. The user may choose to use files for which no consistency guarantees are made or choose "probable" data consistency in favor of increased performance.

Performance optimizations, implemented and planned, will shield the negative effects of "distributedness" from the DFS925 user. The file system is composed of many concurrent file

server tasks, each dedicated to a local or remote file system request as the requests appear. This both increases the efficiency of local file system processing and avoids blocking synchronous waits on the network. The performance of remote reads and writes across the network is optimized by a cooperative protocol between initiator and recipient. A relatively small penalty is paid for block reads and writes; performance measurements show that the CSS setup overhead through upon each file system call causes a significant performance loss. Planned distribution of the interprocess communication facilities of CP925 should decrease both buffer copying across interfaces and message setup overhead.

Extension of the 925 distributed file system to include optimal volume choice policies, to include directory locking and caching, and to support file replication would be the next steps. Distributed file migration algorithms to allow efficient use of storage resources would increase the performance of DFS925. A simple locating service aids in file identification/address translation; eventual integration with a planned distributed location server will allow distributed update of address tables.

6. DFS925 Implementation

The remainder of this report discusses implementation specifics of the 925 Distributed file system. Indications of design alternatives considered are given where appropriate; detailed sections are code-specific.

The software environment of the 925 [IBM84] had an important influence on the structure and functionality of DFS925. The *service request/offer* interface provided by CP925 and the features of FS925 contributed largely to DFS925 task interaction, control flow, and file system structure. CSS provides the confirmed *request/response* protocol which is the basis for DFS925' remote procedure call, as well as the unconfirmed *send* which underlies the *abort* and *checkpoint* operations. Determining the point at which the distinction between remote and local file system requests is made is a significant design decision: DFS925 chooses division at both the system call interface and the logical device level. The file system preserves file identification transparency by assigning fileids based upon locally generated sequence numbers and logical volume ids. This technique guarantees unique identification while allowing file migration between systems. Linking is handled in the same manner as normal file references.

Implementing a failsafe atomicity guarantee is complex; a distributed commit protocol is described which is also used by crash recovery routines. Distributed abort and checkpoint are optimized to avoid waiting for a response from remote partners. Complete data consistency can be circumvented in various ways for increased performance. Transaction checkpoint in lieu of commit, locks allowing multiple writers and the use of nonguaranteed *temporary* file types are among these methods.

Note: Throughout the rest of this paper, reference will be made to both "local file systems" and "remote file systems". These phrases might be misleading, and should be clarified. The DFS925 is one logically cohesive file system, whose files are distributed over a network at several locations. However, when referring to file system calls which originate at a different physical location than that node which holds the file to which the call refers, it is convenient to refer to those calls being sent from a "local file system" to a "remote file system".

6.1. The 925 Software Environment

Three software packages exist upon which DFS925 directly depends. FS925 is the local file system upon which DFS925 was modeled. CSS is the subsystem which handles communication between nodes across the local area network. CP925 is the operating system providing the mechanism through which the DFS925 file service is offered.

FS925 provided an indispensable framework for DFS925; maximal use of FS925 software was encouraged for expediency. The basic features of the two systems are the same; DFS925 development was influenced towards enhancement of FS925 through distribution rather than change of general characteristics. FS925 supports a tree-structured hierarchical file system and

resembles UNIX in style. Files are specified by pathnames, starting either at the root of the file system or at the current working directory. Files are divided into two classes: ordinary and directory files. Directory files contain names and pointers to other directories and/or ordinary files. Ordinary files contain user data, and must occur at the leaves of the file system tree. FS925 (unlike DFS925) does not support links, hence the directory is a tree rather than a directed graph. I/O is stream-oriented: files supported by FS925 and DFS925 have no internal structure unless one is imposed by an application; they are represented by a linear stream of bytes. Every file has a descriptor associated with it called a *file root*; this root is stored on disk and contains the file length, date last modified, and the physical addresses of the file's disk blocks. The file root is equivalent to the UNIX *inode*.

FS925, and its successor DFS925, is a transaction-based file system: all file access operations of FS925 must run as part of a transaction. Changes to permanent files made by a transaction may be rolled back to a checkpoint by aborting the transaction, or may be checkpointed or committed once a point is reached after which rollback is not necessary. Shadow pages and commit logs form the basis for this guarantee of consistency.

CSS, the communications subsystem, is a message-based internode communication subsystem, offering a mailbox/capability interface as a means to pass messages between local or remote processes. The subsystem interface is simple, providing the three primitives of *request/receive/send* for sending and receiving messages, and the other necessary calls to create and register mailboxes and send and receive capabilities for those mailboxes with CSS. DFS925 uses the coupled *request/receive* capability of CSS to implement the synchronous remote procedure call requesting operations on remote files. The danger of hanging remote calls upon network failure is allayed by CSS timeouts; if a call is not successfully received by the network, CSS returns an error code to the file system indicating that the network is not available for use. Asynchronous (decoupled) *send* calls are used for transaction calls with nonguaranteed delivery such as distributed *checkpoint* and *abort*, and in the protocol used for remote *buffer-handling* calls. The file system server tasks make use of asynchronous *receive* in the dual wait on local and remote file system requests.

Both FS925 and CSS, as well as WHIM, the window manager, are offered as *services* through the auspices of CP925. The operating system is kept simple by this means of system augmentation: *services* are easily added to the system, CP925 acting only as a central coordinator of their use. A *service* can be defined as a uniquely identified entity denoting a pair of queues, for *requests* and *offers*. A more precise understanding of the partners request and offer mechanism is essential to the exposition of the basic task structure of DFS925.

A task may make service requests by calling CP925 with a well-known identification for the service required. Requests are queued to await available offers of service. A task may correspondingly provide a service to the system by creating the service with some well-known identification, registering its offer for that service with CP925, then indicating its willingness to wait for a request. The task suspends itself if no waiting request exists. When its offer is matched with an incoming request for that service, the offering task is dispatched.

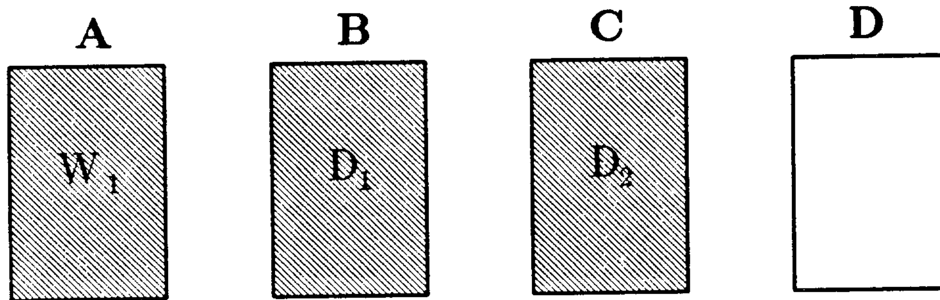
Once the offering task is done servicing the request, it *completes* the request, which allows the requesting task to proceed. Any number of tasks can request a service, and any number can offer it: requests and offers are matched first-come first-serve by CP925. A task is allowed to wait for any of a group of requests and offers to be respectively completed or requested.

CP925 therefore fulfills several roles. First, it provides a registration database for new services created and offered by tasks. Secondly, upon receiving a request or an offer for a particular service from a given task , it records the query in the corresponding queue for the service, then tries to match the query with a matching, unoccupied partner in the opposite queue. Finally, once a match is made, the corresponding pair of tasks are marked accordingly and the service provided. Once a service request has been completed, the queues are updated and the waiting task awoken. Figure 1 illustrates the registration and matching mechanisms of CP925.

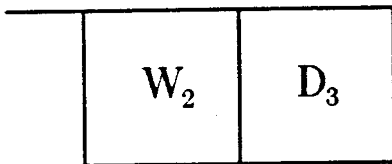
Service	Id
WHIM	1
DFS	2
CSS	3

Id	Request	Offer	Status
1	W_1	A	Busy
2	D_1	B	Busy
2	D_2	C	Busy
3		D	Idle

Registration Database



Tasks offering Service



Queued Requests

W_i , D_i : requests for WHIM and DFS

Service/Request interface

Figure 1

The *service request/offer* model is used by CP925 as the basis for the construction and integration of applications into the 925 system. Parallels can be drawn between this model and a common queueing system. User programs may be the *customers* in that model: requesting a service from the *server*, or application program offering that service. Servers may be customers of other servers ; hence the need for two queues in the model. E.g., CSS, which offers communication services, must request the services of the WHIM window handler when placing a debugging window upon the 925 screen.

6.2. DFS925 Structure

DFS925 attempts to extend the transparency of the user's view into the system structure. This effort was made partly for historical reasons. DFS925 was built on top of the local file system, FS925. Isolating the old file system code from references to the network allowed maximal use and minimal change of the previously-written code. Network transparency also decreases the number and complexity of interfaces internal to the file system: encapsulation encouraged simple call control flow and task interaction.

DFS925 is composed of three logical levels. The *server task* level intercepts system calls from the local file system or the network. Calls are then dispatched to the *processing* level, which executes the internal operations necessary to complete the call. Processing the system call may involve reading or writing physical data: the *device* level is accessed in these cases. The device used may be either the network or the local disk. Remote file system calls may be initiated from either the processing or device layers.

Transparency to the network is present in two ways: the file system is unaware of the origin of its requests; interaction with the network can appear identical to interaction with the local disk. These are applicable at different levels in the processing of a local and a remote call. DFS925 makes use of CP925's *service request/offer* mechanism at the server task level to receive remote requests using the *system call* interface; the processing of remote procedure calls follows the same call control flow as local calls. *Buffer-handling* calls add an extra cut at the *device* level to improve performance while preserving the simplicity of the remote procedure call/return interface.

6.2.1. Queueing Structure

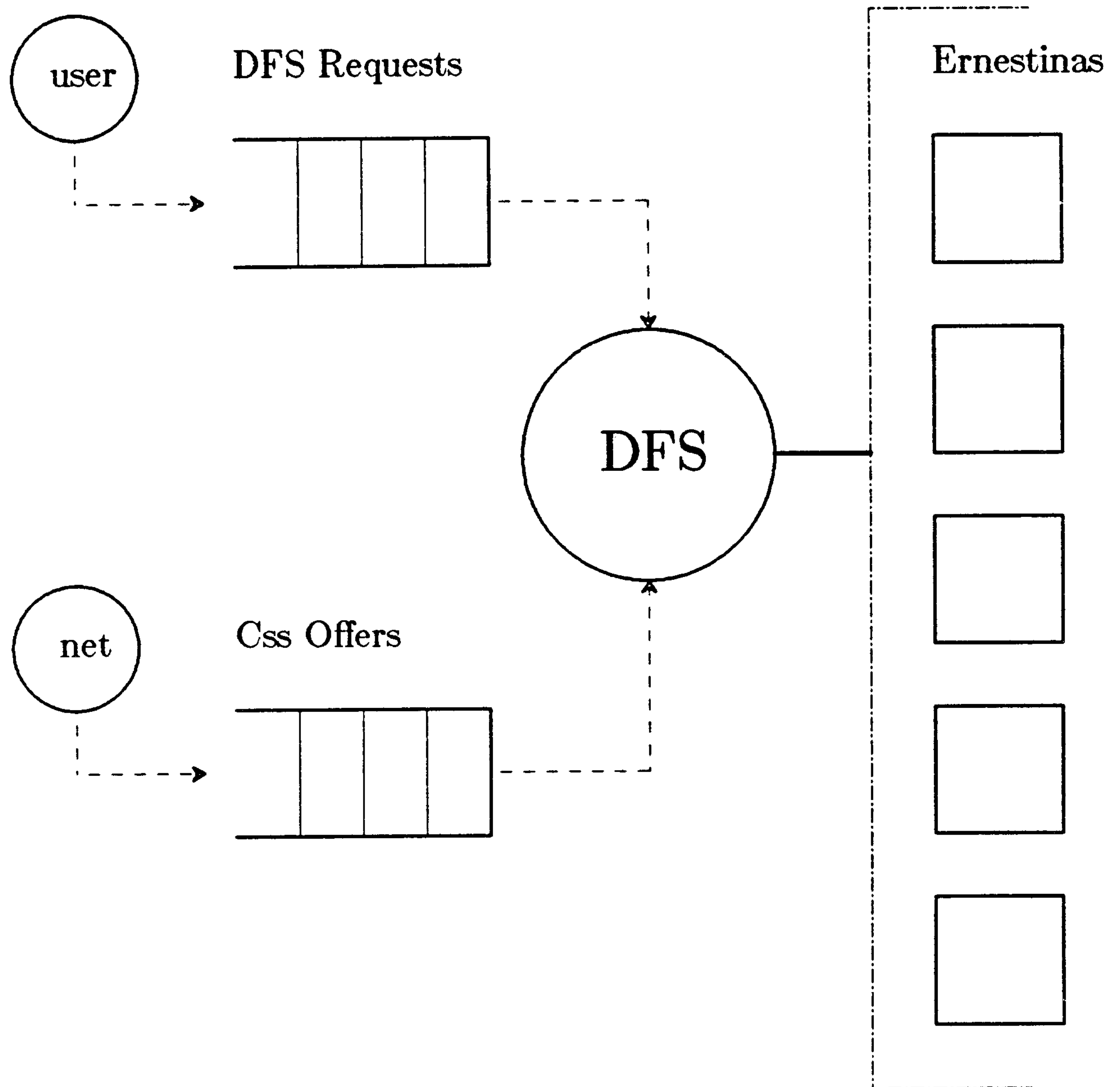
DFS925 uses a two-queue model allowing synchronous request and offer of a service in order to handle both local file system requests and remote file system requests.

A DFS925 server task services local file system calls by creating and offering the DFS service through the registration capability of CP925, then performing cyclical waits for requests. As requests are initiated through file system calls, CP925 matches them with the ready offer of DFS service; completion of the request results in the renewal of offer readiness by the DFS925 service task. The server task receives remote file system calls by being a client to CSS at the same time as it offers the DFS service. The request for messages from CSS and the readiness to offer DFS service are combined into a synchronous wait. Whichever event is satisfied first is completed by the DFS925 server task. Completion of the request for file system service or completion of CSS message processing results in the renewal of both the DFS service offer and the request to the CSS service.

There are hence two queues into the DFS queueing system: the queue of waiting requests from file system calls, and the queue of waiting messages from the network. This may be approximated by a M/M/2 queueing structure. Figure 2 gives an idea of the queueing structure with the addition of Ernestina tasks, discussed in the following section.

6.2.2. Ernestina Server Tasks

Calls upon the file system are synchronous, which creates a performance difficulty for a distributed system. A file system call may now involve waiting for the return of information from a remote file system over the network. Preempting additional file system activity while the single server task waits for a remote message is clearly unacceptable. The file system could be gainfully



DFS Queueing System

Figure 2

occupied in processing another local file system call during the wait. Reassigning the server task to handle another request would force the file system to duplicate the task scheduler function of the operating system.

This inefficiency was avoided by the use of a multiple server model for DFS925. Rather than creating only one DFS925 server task, a fixed number N of server tasks (dubbed Ernestinas*) are created at the outset of file system initialization on each workstation. Each Ernestina waits simultaneously upon both a request for messages from CSS and its offer of the DFS service. Whichever of those is first satisfied, the Ernestina task completes the call requested, then returns the information to its origin (either a local user or a remote requestor). Each workstation's local file system thus has the potential of handling N local and remote requests at the same time. Ernestinas share a common heap, but have individual stack segments.

Since each Ernestina must have the ability to access global volume descriptors, free space maps, and other global data structures, the Ernestinas are synchronized through the manipulation of a global semaphore. An Ernestina must capture the semaphore before being able to run, and releases it before performing I/O or waiting for a request from the network. Performance of local as well as remote file system calls should be enhanced as well; disk processing in the background frees a server task to process another request.

Several file system calls may be thus performed synchronously on one file system through the the Ernestina server tasks, data conflicts being prevented by global semaphores and file locking. A server task is dedicated to a call from the time that the request is matched to an offer to the time of its completion. No file server tasks are created once a system has been initialized: there is a maximum configurable limit upon the concurrency of the file system. This prevents task creation overhead during file system call processing.

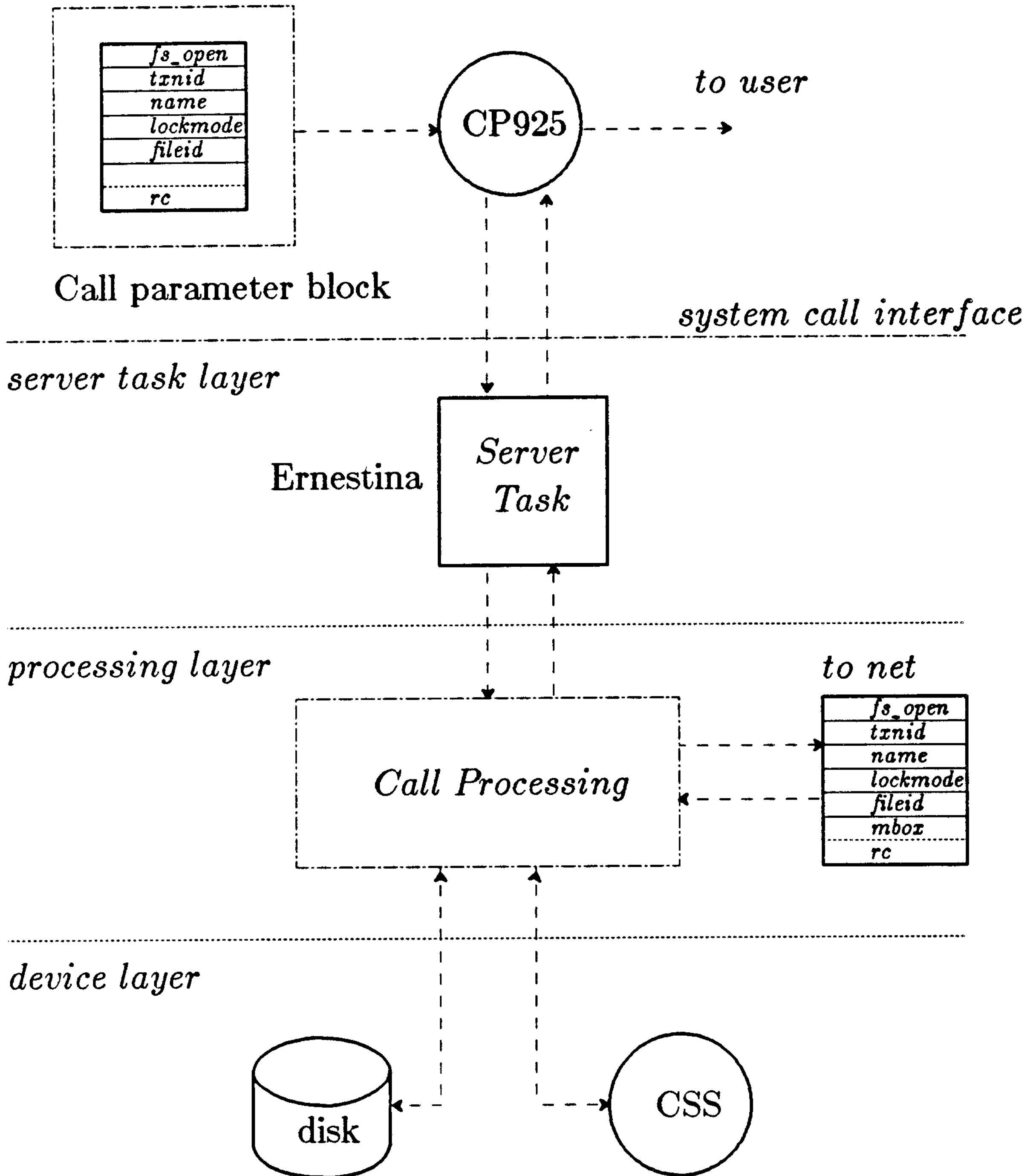
6.2.3. Call Control Flow

Remote and local procedure calls requiring no user buffer handling are differentiated at the server task level only. Control flow and processing of a call through the underlying layers of the file system are unchanged by the origin of the call.

A local file system call passes its information to the file server task through the auspices of CP925. The information included in the call is first converted into a file system call parameter block by a stub routine (i.e. *fsopen*). The stub routine requests the DFS service from CP925, including the parameter block as part of the request. When one of the Ernestina server tasks is free, CP925 matches the outstanding request from the user with the DFS service offer from that Ernestina. The parameter block is passed to the Ernestina task specified. The server task passes the parameter block to a call processing routine to process the request, returning the altered parameter block to CP925 when the request has completed. CP925 passes the parameter block back to the user, then updates the DFS request and offer readiness queues and allows the server task to reinitiate its service offer and network request. The local user routine has waited synchronously for the call to be completed. Figure 3 illustrates the possible path of a local file system call through the levels of DFS925 .

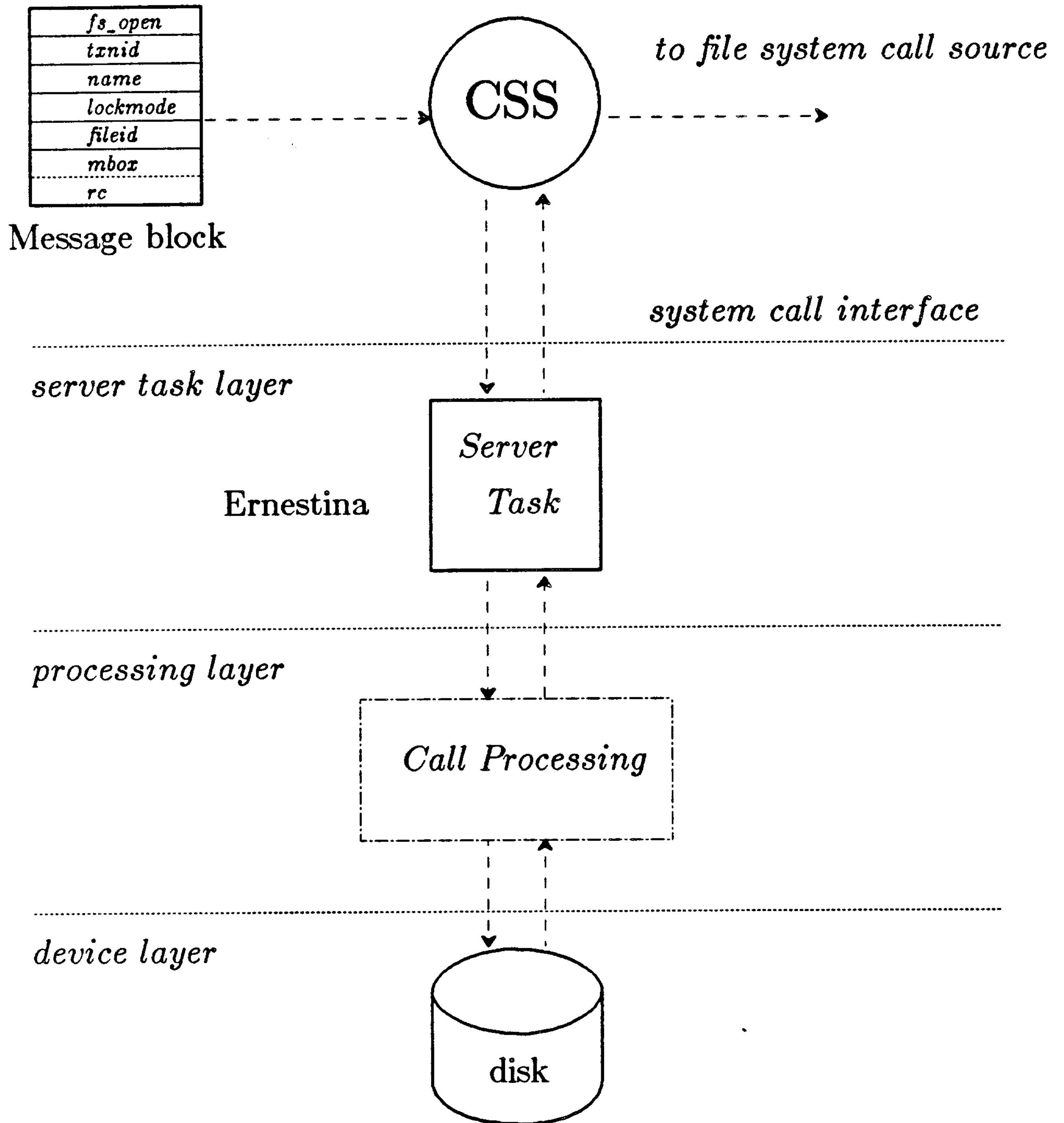
In the case of the request arriving via CSS, the message received is a parameter block identical in format to the parameter passed through CP925 in the local procedure call. The parameter block is passed to the call processing routine as in the local case. The Ernestina task assigned to the call has the responsibility for sending the altered parameter block back to the point of the call's origin through CSS. The system call interface at the server task level is preserved in both local and remote cases by the consistent use of the system call parameter block. Figure 4 shows the reception of a remote request by a local file system; such a request may access only the local disk at the device level.

* The name *Ernestina* was derived from comic Lily Tomlin's portrayal of a switchboard operator called "Ernestina". A telephone operator's function is to coordinate and dispatch calls: receive both local and remote telephone requests, connect them to the correct line, and disconnect them when the call has been terminated. The functions are similar to that of the DFS925 server tasks.



DFS925 Call Control Flow

Figure 3



DFS925 Remote Request Receipt

Figure 4

Returning remote procedure calls can be distinguished from new initiations by their destinations. Since many Ernestina tasks may wait simultaneously on remote requests, remote file system calls must wait for their returning parameter at a unique CSS address distinct from the well-known address, else the returning parameter is directed to a free Ernestina task. All remote procedure calls contain a return address for the source: the initiator of the call generates a unique CSS address (a locally unique mailbox, and CSS net and host identifications) and sends it with the file system parameter block over the network to the destination file system. A send capability for that address is generated by the remote file system when the parameter block returned to the call source. The generation, registration, and deletion of CSS mailboxes and capabilities takes place with each remote system call: it is a cause of substantial inefficiency.

In summary, a DFS925 server task (Ernestina) is required to wait upon two events: a request by the local file system and an offer from the communications subsystem. Once the system call parameter block has been received, either through CP925 or CSS, request processing proceeds by the same sequence of calls regardless of call origin. Multiple server tasks allow for simultaneous processing of calls of both local and remote origin (and destination); synchronization of server tasks prevents degradation of global information.

6.2.4. Sending & Optimizing Remote Requests

As described, the reception of both local and remote requests is done through the system call interface at the server task level. The distinction between a local and remote request is acknowledged by the Ernestina server task only; the call is processed and the results placed in the parameter block alike, regardless of the call's source.

For constructing and sending remote file system requests, a one-to-one local/remote system call translation can be imagined. Each file system call which accessed a remote file could be translated into an equivalent remote call by the local call processing routine; results would be passed back to the user with the total combined overhead of one local and one remote file system call.

This one-to-one system-level cut model is complicated by unresolved factors: the need for local processing and maintenance of state on the source file system, and the transmission of data buffers to and from user-space buffers. While the system cut model described above fits *simple* file system calls, which translate one local system call into one remote file system call, two other types of calls are common: *compound* calls and *buffer-handling* calls. Table 1 contains the DFS925 file system calls and their classifications.

Compound calls are those which demand changes in local state information; they may translate into several internal, and hence potentially remote file system calls. Most file system calls are compound.

The cost of maintaining transparency at the system call interface at both the remote call initiator and at the call recipient is the passage of any call through more system layers in both file system participants; in compound calls it is magnified by the number of remote calls made. A *simple* call accessing a remote file passes through the local CP925 request queue, the local server task and processing levels, the network, the remote CSS request queue, the remote server task and processing levels, and possibly the disk driver at the device level before returning over the network to the waiting requestor. Direct communication between layers using a specialized transfer protocol is considerably more efficient. Such an optimization was considered worthwhile in the case of *buffer-handling* calls.

Two *buffer-handling* calls exist: *fsput* and *fsget*; they include local buffer addresses from which or to which data is passed. The requestor wishes either to move data from a user buffer into the remote file, or to receive data in the user buffer from the remote file. A *buffer-handling* call may also be transformed into several block moves from disk or across the network. Numerous obstacles arise to such a call being divided into several system calls: the most obvious being the performance burden of treating these calls as separate, acknowledged, instances.

Table 1	
DFS File System Call	Type
fsbegin_transaction	*Compound*
fscheckpoint_transaction	*Compound*
fscommit_transaction	*Compound*
fsabort_transaction	*Compound*
fscheckpoint_freemap	*Simple*
fscreate_file	*Compound*
fscreate_temp_file	*Compound*
fstruncate	*Simple*
fslink	*Compound*
fslock	*Compound*
fsset_directory	*Compound*
fscreate_directory	*Compound*
fsremove_directory	*Compound*
fsdelete	*Compound*
fsopen	*Compound*
fsget	*Buffer-handling*
fsput	*Buffer-handling*
fsdate	*Compound*

To avoid breaking up all buffer-handling calls into several remote call instances, data movement across the network mimics the transfer of data from a local disk. Unlike remote procedure calls, buffers sent over the network are not separately acknowledged. Remote data transfer time is correspondingly improved. The remote/local *cut* is placed at the device level.

The communicating remote and local tasks create their own CSS addresses and pass those addresses over the network to their partners. This establishes a private conduit through which the data can be sent without the overhead of a system call construction or an acknowledgment. Once that conduit has been established, whenever a call requiring actual disk access is necessary, the file system need only send or request a buffer directly from its partner through the network. No remote disk addresses are maintained in active transaction or file information structures. The logical volume number in the fileid is the original piece of information which indicates the location of the remote file to the requestor; the remote file system call sent acts as a conduit "set-up" call. The CSS addresses created by the partners are the subsequent private addresses for direct data transfer.

Aside from the performance advantage of establishing a direct network link for the entire data transfer, burying the cut internally has logical appeal. The network fills the role of an intermediary device through which the disk is accessed. Use of remote files is indistinguishable from that of local files until a very low level. The uncomplicated RPC protocol at the server task level remains unchanged.

Buffer-handling calls thus set up an internal protocol mimicking device access between remote and local tasks through the use of private mailboxes. The number of messages over the network is reduced to the request/acknowledgement of the initial remote file system call, plus the number of block requests made. Copy overhead is high. Buffers must be copied from system buffers to CSS buffers for transmission, then from CSS to user space upon reception. Improvement of copy overhead could be gained with the use of distributed IPC (much like the V System) as the transport mechanism rather than CSS. Through the use of disk block pointer

references, the additional copies to and from CSS buffer space would be avoided without loss of transparency.

7. The DFS Global File System

A flat file identification space incorporating a level of location indirection is enabled by embedding *volume* ids in file and directory identifiers. Location servers allow volume/address translation. Linking to remote or local files permits the existence of duplicate file references; links can be used as shortcuts to avoid traversal of a long path name.

Movable *volumes* and volume location servers control the generation of unique file identifiers in the DFS global file system. The 925 workstation node directly controls a variable number of physical *volumes* or hard disks. Each volume is identified by a logical id which is unique across the system. Location servers on each workstation are responsible for maintaining tables associating logical volume id's with the network address of the file system which controls them. This address specifies the CSS net, host, and mailbox upon which the file system server tasks wait for the reception of remote requests.

The DFS925 file system structure is a directed graph. Files form the leaves of the tree; directories the inner nodes. Complete file and directory text string names are specified by concatenation of the individual text string names along the path by which they are reached. Internal file and directory identifiers are unstructured and guaranteed unique within the distributed system. For the creation of each new file or directory, a unique file root number is generated by the local file system; that number is combined ("stuffed") with the logical volume number of the disk upon which the file resides. Since numbers are generated in sequence by the local file system, only repeated when all references to a previous file have been deleted, and volume numbers are uniquely assigned by the system manager, the resultant fileid will be unduplicated. Figure 5 shows a sample DFS925 global file system configuration. The logical structure of the directory system is disassociated from the boundaries of the volumes holding the system.

Choosing unique internal identifiers as internal file names in the manner described is attractive for several reasons. Fileids are absolute within the system, and only indirectly related to the file's physical location in the network. They can be passed between nodes and processes without any relative translation being necessary. In order to find the physical address of a file, the file server passes the fileid to the location server, which associates the "unstuffed" logical volume number in the fileid with a CSS network address, and returns the address to the node's file server. This indirection of location information in internal file identifiers allows the easy transfer of volumes between nodes. Individual files can be moved between volumes by copying; volumes may be physically moved between nodes by notifying the location servers of the logical volume id/address change. If the network address were embedded in the fileid rather than the volume id, volumes could not be moved from one workstation to another without all references to that file being discovered and updated. This solution does depend upon continued maintenance of the location server database on each workstation. Incorrect database entries will result in the file system being unable to locate remote files.

This kind of flat identification scheme is only possible in a fairly small, homogeneous distributed system with some system administrator intervention. Assumptions made are that local file system sequence numbers do not grow too large over time, and that the number of volumes in the system will stay bounded.

It is worthy of comment that the ".." , or relative pathname problem found in distributed UNIX-based file systems which *mount* portions of their file system is nonexistent in a file system which imbeds location indications in the file identifiers. Directories in both distributed UNIX and DFS925 contain an entry indicating the fileid of that directory's parent directory, denoted in a path name as "..". The existence of the unnamed parent entry allows the contruction of path names relative to the current directory. Complications arise in a mounted distributed file system when the parent directory entry intersects a mount boundary: the location of the server upon which the parent directory exists may not be known to the current server. Since no logical

/

sys

arden

wyllie

haskin

RSinfo

calendar

bin

mail

fsinfo

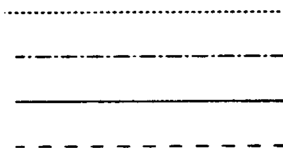
config

link

link

config

fsnotes



Lines indicate volume boundaries

DFS Global File System

Figure 5

boundaries exist in the DFS925 file hierarchy, and possession of any fileid implies the ability to find that file's location, construction of relative pathnames can be handled by normal pathname parsing. A more detailed discussion of the complications of relative pathnames in a distributed UNIX-based file system can be found in [WELC85].

7.1. File Access Protection

The DFS925 file system does not provide external file access protection; users may access and update any file in the system. Both ideological and practical concerns dictated this decision: it was felt that the DFS925 environment was not one where malicious file access was likely, and the author had insufficient leisure for its consideration. Policies which govern volume choice upon file creation can afford a kind of automatic protection directly related to performance, however. The policy for volume choice upon file creation controls the physical location of that file. Volume choice policy can differ from workstation to workstation; workstation users may be prevented from creating files on given volumes by adjustment of the volume choice policy resident to their node. Performance is clearly impacted by that choice. Should the local node not hold a disk, volume choice policy must clearly indicate a remote disk, preferably one which is the most efficient of those available and has sufficient free space to hold the file. A distributed volume policy server, using dynamic algorithms similar to load balancing algorithms could provide an optimally global choice for file creation dependent upon the state of the system. More simply, choice of the local disk (if one exists) is likely to minimize network references. Creators of files are the most likely frequent users of that file, particularly during a development phase.

7.2. Links

A link is a directory entry referring to a file or directory; the same file (together with its size, and all accompanying information) may have several links to it. Links may be used for organizational purposes: if a file is logically associated with more than one directory, or for ease of reference, if the complete or relative pathname of the file is lengthy and the file is frequently accessed. A link to a file is indistinguishable from the original directory entry; any changes to a file are effective independent of the name used to reference the file.

The implementation of file links in DFS925 was extremely simple. A link entry in a directory looks and acts identical to a file reference; the only distinction is made during file and link creation. Directory entries contain very little information, only the name of the file and its id. The "unstuffed" generated file id number represents an internal file root residing on the file's local file system. The file root contains a pointer to the physical location of the file, as well as the link count. Creating a new link to a file involves finding the file's file root through volume/address translation and the unstuffed generated file id number, updating the link count in the file root, and adding a new directory entry to the current directory with the new name of the link and the file identification. Once a link is added to a file, there is no way of distinguishing the original from the linked file. When deleting a file reference, the link count is decremented in the file root. If the link count is greater than zero after decrementing, the file root is not removed and the storage for the file not deallocated. DFS925 links are identical to UNIX hard links.

7.3. Criticism

DFS925's provision of a flat file identification space, enabled by embedded (volume) location information and location servers results in a directory system whose logical structure and physical composition are completely distinct. Such an approach ensures location transparency and volume migration; relative path names are easily handled; linking is a simple extension. Each file system node has a equivalent view of the entire file system: this symmetry satisfies the goal of uniform distribution. Wise volume choice policies can optimize file system performance and provide weak protection without the explicit implementation of file access protection.

Some problems exist with such a global structure, however. Paths which are logically hierarchical may physically span a large number of volumes and nodes; unless directory references are duplicated, the loss of a node from the network may fragment paths. Files resident on active,

connected, workstations which should be mutually accessible may become unreachable due to the inavailability of an intermediary node containing a necessary directory entry. Nodes may actually be unable to access files which physically exist on local volumes. Workstation independence and file system availability are severely degraded.

A second, performance-related disadvantage of the incremental manner in which pathnames are processed is that the file system may have to access as many remote file systems as elements in the pathname of the file accessed. Paths are traced through the file system by looking at the directories they access; each directory entry matched gives the location of the next file system to be queried. Non-centralized knowledge of the file system structure fulfills the precepts of distributed control, but is correspondingly expensive. Each remote file system access involves at least two messages across the network in order to find the next path. Avoiding incremental pathname parsing in at least some circumstances would cause substantial improvement in performance.

Enhancement of performance and file system availability should be implemented through directory information caching. Either of two methods could be followed. First, the directory structure could be shadowed on each workstation node. A workstation would always be able to access any files resident on local (or active, connected) volumes, given that the shadowed local directory structure was valid. Only local files would be allowed to be updated with the help of this information; new directory entries in inaccessible, or physically unavailable directories, could not be created. As directories are infrequently updated, maintaining this shadowed structure as a "hint" should be maintainable with acceptable overhead.

A similar, simpler caching technique is to maintain string prefix tables of directory prefixes which "cache" the locations of directories once accessed. The string prefix and cached directory location are the associated pair which make up an entry in the table. Such a system works only with absolute path names; pathnames relative to the current directory would need to be transformed into their absolute pathnames for string matching. Since only directory locations, rather than their contents, are cached, such an approach has fewer cache update implications than the method mentioned above. Unlike the full shadowing technique, there is no guarantee that the workstation user will be able to access files on all reachable volumes. The table may be an incomplete list of directory prefixes. The user will only be able to reach those files whose directory prefixes are contained in the string prefix table, or whose entire directory path refers to available volumes.

Either one of these approaches would be viable additions to DFS925; such enhancement is necessary to fulfill its goals of node independence and availability while maintaining the uniform distribution of the file system.

8. Transactional Support

DFS925 is a transaction-based file system. Transactions are a means of grouping sequences of file system calls and guaranteeing *atomic* execution of the entire sequence of calls: the actions contained in the transaction will either succeed or fail as an indivisible unit. Transactions are units of both consistency and recovery.

DFS925 extends transaction support and recovery to operate in a distributed environment. One transaction may access files on many network nodes; part of the transaction resides on each node involved. The transaction's *resident* site identifies the *master* of the transaction; the master is responsible for managing communication between transaction fragments to insure consistency. Distributed locking, commit, checkpoint, and abort algorithms are actively controlled from the master participant; commit logs on each node involved in the transaction regulate the storage of all critical state information on disk, enabling distributed recovery if necessary.

DFS925 attempts to minimize the cost of providing transactional support. Multiple-write locks and *temporary* file type are options which provide better performance at the cost of lower data consistency. Transaction checkpoint and abort make use of unacknowledged messages to remote transaction participants; local writes may at least be guaranteed if network or remote

node states are doubtful. The two-phase commit protocol is unavoidably complex; it consists of a sequence of independent actions designed to be idempotent to aid in error recovery. The crash recovery algorithm resolves unfinished commits by continuing the commit protocol where it was interrupted.

All DFS925 file system calls must be included in a transaction; every atomic sequence of calls is preceded by a *begin_transaction* system call. The *begin_transaction* call creates a *transaction descriptor* which records the transaction state for the duration of the transaction. State information includes the files read and written during that transaction session, the types of locks kept on those files by that transaction, and the physical addresses of shadow pages reflecting any file changes made during that transaction. Each node involved in the transaction contains a transaction descriptor identified by a common transaction identifier. Transaction identifiers are only created by the file system which first initiated the transaction. The initiating site is called the *master* participant; it assigns a *resident* volume to the transaction created. All other participants are known as *slaves*.

Transaction identifiers are constructed in an analogous manner to file identifiers. They consist of the combination of a generated number unique to the initiating file system with a logical volume number local to that system. This *resident* volume of a transaction will hold its commit log; it may be considered the transaction's home location. Through the same mechanism as the file identifier, the transaction id is unique system-wide.

Transactions may include operations on files which reside both locally and remotely. In the case where a transaction accesses a remote file for the first time, a transaction is created on that remote file system with the same transaction id as that on the master site. New transaction id's are only created if a *begin_transaction* request originates locally. File systems thus support transactions which have been initiated at remote locations as well as at their own location. However, only a transaction descriptor on the master site contains global information about the transaction; slave sites hold information only about resources local to their sites. The resident site of the transaction contains a list of all slave sites by volume id. Because the resident volume number of the transaction is embedded in the transaction id, a slave site may always discover the master location of an active transaction. This becomes important upon transaction commit and crash recovery.

8.1. Locking

Since multiple transactions may exist at the same time, it is possible for a file to be accessed by more than one transaction. Concurrency control is enabled by means of file locking. Locks have the effect of notifying others that the file is being used, and protect the file from modification from others: inconsistencies such as lost updates, dirty read, and unrepeatable read are avoided [GRAY78]. Whenever using a file, a transaction must acquire a read or write lock and hold it until the transaction is complete. DFS925 uses an automatic two-phase lock control: after releasing a lock, transactions may not acquire additional locks.

The degree of transaction serialization enforced by file locking controls the consistency and concurrency levels allowed. Insuring that only one transaction accesses a file at one time enforces complete transaction serialization and insures data consistency, but decreases system concurrency. DFS925 allows the trade of consistency for increased concurrency. Third-level consistency (one writer, multiple readers) [GRAY78] is the strongest guarantee of consistency provided by DFS925. File changes are only performed by one transaction; no permanent changes to a file root are made until full agreement for commit has been attained (see description of transaction commit below). DFS925 permits weaker consistency and higher potential concurrency through two optional mechanisms. Multiple-write locks allow more than one transaction to have a write lock on a file, and files are updated in place rather than through shadow pages. *Temporary* files are scratch files for which no consistency guarantee is made; updates are made in place, and no record of changes to those files is written in the commit log.

A new lock on a file must agree with all previous locks by all active transactions according to the consistency level chosen. Only the file system which physically contains the file has a

complete list of all locks kept on that file for all transactions. In order to avoid communicating with that file system for each lock request, local lists of known remote locks are maintained on each file system as a "hint" to determine lock success. If the lock requested agrees with the local list (the hint is positive), final lock confirmation takes place upon the file system containing the file's physical volume.

Currently, DFS925 does not guarantee third-level consistency for directories. Write-locking directories in a hierarchical system is problematical due to the potential of lock-out of all other transactions accessing that directory and children of that directory. It is proposed to alleviate this problem by the introduction of an incremental log table associated with each directory. Such a log table would essentially allow record-level locking of the directory; only the record of the file which was being updated by the directory would be locked for write, rather than the entire directory. File creation would add a provisional record to the log table; potentially conflicting file creations would be checked against all provisional records. Other transactions could simultaneously update other files in the directory. The log table would be saved in the commit log as are shadow pages; upon successful commit the transaction's log table would be applied to the directory concerned.

8.2. Failsafe Guarantees

DFS925 insures the data consistency granted by locking and transactions against both node and network failures. Distributed two-phase commit, distributed abort, non-guaranteed checkpoint, and individual crash recovery upon node failure provide this insurance. Such distributed guarantees must (1) allow for correct action amid error conditions on master, slave, or network and (2) minimize disk writes and messages under normal file system operation. Keeping crash recovery and commit/abort actions idempotent, limiting distributed abort to one phase, and avoiding the use of special-purpose crash recovery algorithms are among the optimizations used by DFS925 to increase performance while maintaining functionality. In conformity with the design goals allowing the tradeoff of functionality for performance, transaction checkpoint differs from its canonical definition; it gives only a probable guarantee of consistency. The number of messages required is correspondingly reduced.

8.2.1. Distributed Commit

A two-stage distributed commit protocol consists of a two-phase dialogue between the *master*, who acts as the active transaction coordinator, and the passive *slave* sites. Each phase transition in a commit request is initiated by the master through a *commit* call differentiated only by an argument indicating the current state of the master. Complete rounds of acknowledged messages ensure that no incorrect irreversible assumptions are made about the state of any site. DFS *commit_transaction* is designed to be a series of idempotent communications between the master and the slaves. Any state information essential to the preservation of consistency is hence written to disk; loss of active state information through node failure only degrades performance rather than also functionality.

The protocol is as follows:

Master:

(1) *Commit_transaction* is called at the master site. The master allocates a commit log, labels it with the transaction id, and fills it with the file identifiers and shadow page addresses of any changed files *local* to that file system, first forcing those changes from active buffers out to disk. The log is flagged *ready* and atomically written to disk. (This protocol assumes that a block write is atomic action, and that failure to complete the write successfully will result in a detectably bad block. This assumption is probably unrealistically optimistic for DFS925.)

(2) The master then sends *Commit_transaction* messages to each of the remote participants in the transaction. These messages are constructed and dispatched as normal remote procedure calls; a call argument indicates that the master is

requesting confirmation of *ready* status from the slaves.

(3) If the master receives *ready* confirmation from all slaves within a reasonable length of time, the commit log is flagged *commit* and rewritten to disk. The global commit is irrevocable from this point on. Local files changed in the transaction are updated, shadow pages replace old pages, and disk storage for replaced pages is freed. The master then sends out *Commit_transaction* messages to all slaves indicating that the *commit* phase has been reached. Once all remote calls return affirmatively from the commit, the master frees the commit log and the commit is complete.

Slave:

(1) In error-free circumstances, the slave receives two *Commit_transaction* messages from the master, indicating passage into each of the *ready* and *committed* phases. Receipt of the *Commit_transaction(*ready*)* remote procedure call entails forcing out file buffers, and constructing and writing a commit log flagged *ready*. Successful return of the commit call to the master automatically confirms slave receipt and action.

(2) After receiving the *Commit_transaction(*commit*)* message, the slave is free to update files and free storage without rewriting the commit log. Return of the remote file system call parameter block to the master is again sufficient as receipt and confirmation.

The protocol is illustrated in Figure 6. Message or commit failure is allowed for at any of the stages at the master or slave. If, during the ready phase, a master site does not receive ready confirmation from all of the slave sites (which could be caused by network or slave node failure) , the user is informed. He may choose to continue local processing with that transaction, abort, or use non-guaranteed checkpoint; should he reattempt to commit, the commit log will be reused. Once the master site has committed, the commit log will not be deallocated until all slave sites have acknowledged that commit. Protocol failure beyond the master's commit point will at least leave ready logs on uninformed slave sites and a commit log on the master site. Crash recovery protocols reinitiate the commit protocol at the point where it was interrupted, resolving any conflicts between communicating nodes.

8.2.2. Transaction Abort and Checkpoint

One phase abort and non-guaranteed checkpoint are calls which assure the continued functioning of the file system in the case of node or network failure.

Transaction abort nullifies any actions performed on *permanent* files with level three locking consistency. Upon receiving an abort request, the master site sends out unguaranteed abort messages to all slaves, not waiting for a reply to destroy the transaction. If a slave site does not receive the abort request, the slave must be assumed failed. There are two possible states of the transaction at that failed slave site: either all state information about that trànsaction has been lost (which is equivalent to that transaction being aborted) or the transaction is partially committed. Any unresolved slave *ready* commit logs for an aborted transaction are destroyed upon slave file system recovery: the slave site checks the master site for saved commit logs. If the partially-committed transaction had not been aborted, a commit log would exist on the master site. If the master is unavailable, the slave retains the partial commit log; the master will query the slave when the master once again becomes active.

Transaction checkpoint represents a non-guaranteed commit; consistent data is only guaranteed on the local file system. Like abort, non-guaranteed messages are sent out for checkpoint to all slave sites; no commit log is written. The checkpoint requestor only receives acknowledgement that locally changed files are updated and replaced pages deallocated. Non-guaranteed checkpoint is mandatory for the continued functioning of a transaction-based

master **slaves**

prepare commit log(ready)

ready *ready?* → *prepare commit log(ready)*

→

timeout?: **abort**

←

←

ready!

prepare commit log(commit)

commit --

commit?

→ *write files*

→ *delete commit log*

timeout?: **return**

←

←

write files

delete commit log

commit!

Two-phase Commit

Figure 6

distributed system. While data consistency is guaranteed through commit and crash recovery, the loss of use of one node on the system could potentially cause incomplete transactions to hang almost indefinitely. Unless it is possible, through checkpoint, to override the transactional mechanism and save changed data where feasible, dropping file locks could only be accomplished by aborting the transaction. At the price of system-wide data consistency, transaction checkpoint followed by abort saves file changes made on all active nodes.

8.3. Error and Crash Recovery

Crash recovery is initiated after a node has failed and is attempting to reinitialize the file system. The recovering file system sequentially reads all commit logs, trying to resolve all unfinished transactions by querying the other known members of the transaction as to their status. Transaction status query is accomplished through the use of the *Commit_transaction* remote procedure call. Unlike the normal commit protocol, however, both master and slave may initiate the *Commit_transaction* call during the crash recovery process.

The master reattempts to recommit the transaction only if the saved commit log is marked *commit*. If the log is marked *ready*, only the *ready* phase was completed, and the master sends out abort messages to the slaves. A slave discovering a commit log (marked *ready*) attempts to find out the status of the transaction on the master. If no record of the transaction on the master exists, the slave aborts, else the slave continues in the commit protocol.

Distributed crash recovery can be a complex process. As in the *Commit_transaction* it must be composed of an idempotent sequence of calls; repeated node or network failure may interrupt the recovery protocol without warning. Using the same *Commit_transaction* protocol for crash recovery reduces the potential for inconsistency.

9. Performance

Performance measurements made give the relative overhead of a file operation involving a remote file as opposed to one involving a local file. Timings were done for copying files and for repeated truncate, read, and write calls: they are contained in Tables 2, 3, and 4 respectively. File copy is composed of several *compound* file system calls. Truncate is a *simple* file system call, and is DFS925's closest approximation to a null system call. Read and write are both *buffer handling* calls.

Compound commands on a small scale exhibit few adverse effects in the remote case. The composition of the file copy command in particular is favorable to reasonable performance. Buffer-handling calls are the most important of the file system calls in file copy: these calls are optimized to decrease the number of CSS setup calls and messages required by the network.

Local file system calls are considerably faster than the remote operations; neither network latency nor CSS setup overhead are present. The setup overhead imposed by DFS925 for a remote procedure call through CSS includes the creation and deletion of both mailbox send and receive capabilities. Observation of the measurements for truncate, read, and write calls emphasizes the increased efficiency of buffer-handling calls over other calls requiring CSS setup overhead for each network message. Truncate follows the simple request/response model; two messages are sent across the network, and 6 CSS setup calls (2 CreateCapability calls, 1 CreateMailbox call, and the corresponding DeleteCapability and DeleteMailbox calls) are used. A *read* file system call requires the same setup calls, but may use many more messages: the minimum read message count (for 2048 bytes or less) is three. Each additional 2K block transfer across the network requires another message, but no added CSS setup calls. The difference between a 4 byte remote read and a truncate call is about 20 ms (averaged over 1000 calls): this represents the approximate cost of one network message. The average cost of a CSS setup call can be approximated by comparing the times of a remote read and a remote write call. Remote writes involve one additional message with respect to remote reads because the remote respondent to a remote write must return an additional address to the call source: the address of the mailbox to which the write buffer may be sent. A 4-byte remote write requires 10 CSS setup calls and 4 network messages: a 4-byte remote read requires 6 CSS calls and 3 network messages. Using 20

ms as the cost of a network message, a CSS setup call can be said to take (215 - 4*20)/10 = 13.5 ms. This figure checks with the summed cost of the truncation and read calls. Confirmation of the importance of the CSS setup overhead can be seen in the increased efficiency of 8192 byte transfers over 2048 byte transfers - the ratio of CSS setup calls to network messages is lower when larger read and write requests are made. This implies that overhead is caused less by network latency and buffer copies than by the setup cost of individual remote file system calls.

It should be noted that since read and write buffers are kept in core until forced out, measurements taken for repeated read and write calls are not indicative of disk latency. The calls requiring more than one 2K buffer each have the extra overhead of searching through more than one buffer descriptor to find all cached buffers required by the call: this will account for some of the time taken by both the local and remote 8192-byte read and write calls.

Considerable tuning needs to be done in order to bring these costs down to a reasonable level. The CSS setup cost per remote procedure call is currently inevitable; a distributed IPC mechanism for CP925 should be able to use well-known mailboxes for communication between nodes, largely decreasing the number of calls to CSS. Update of *compound* file system calls which make several internal file system calls (such as copy, set_directory, and so on) could make a visible improvement to performance. Performance is badly degraded both locally and remotely for operations which use several compound calls: listing a local directory with 10 entries takes 5 seconds , whereas listing a remote directory of the same size demands 55 seconds and over 300 remote file system calls.

Few performance figures are available for transaction-based distributed file systems; null remote procedure calls in the V System and the Berkeley Network Distributed File System are 5 and 11 milliseconds respectively; these systems perform no atomic guarantees or concurrency control.

Table 2 Times for File Copying			
	16K	32K	64K
Local	5 sec	5 sec	6 sec
Remote	5 sec	6 sec	7 sec

Table 3 Elapsed times for 1000 Truncates [seconds]	
Local	13
Remote	121

Table 4				
Elapsed times for 1000 repeated Reads and Writes [seconds]				
	Reads		Writes	
	4 bytes	2048 bytes	4 bytes	2048 bytes
Local	14	18	15	29
Remote	142	212	215	260

Table 5		
Elapsed times for 250 Reads and Writes [seconds]		
	8192 bytes	
	Reads	Writes
Local	28	29
Remote	110	140

10. DFS925 Experience

DFS925 was tested incrementally in two major phases. Distributed file systems first ran on one 925 workstation, writing to a disk-simulation file. Final testing placed a DFS925 file system upon each of two workstations, communications passing over the network and writing taking place to disk. The DFS925 file system currently runs in prototype form only as an additional *service* to the local file system on two workstations. The system grew from the FS925 size of 70K to the DFS925 size of 150K.

It should be observed that the consistent *service/request* interface provided by CP925 made the construction of a test system trivial; 925 system enhancement through application additions is extremely simple due to CP925's design. The basis provided by a tested underlying file system (FS925) was essential to any kind of progress in distribution. The prototype 925 workstations were, unfortunately, quite unreliable over the development period. During the implementation, the lack of a debugger running on the 925 made software testing considerably more difficult; the often-lengthy process of compiling and downloading from the 3081 was very tedious. Providing a solution to the latter irritant gave impetus to the project; the potential usefulness of a distributed file system linking the 3081 and the 925 network was incontroversible.

11. DFS925: Present and Future

This report has described the design and implementation of a distributed file system intended to emphasize uniform distribution, workstation independence, data consistency and availability. The integrated system model and the global file system best exemplify the efforts towards uniform distribution and workstation independence. Data consistency, built upon the structure offered by FS925, is provided by transactions, locking, and a collection of transaction operations such as *commit, checkpoint,* and *abort*. An idempotent sequence of actions compose a commit protocol which is used in both transaction commit and node recovery procedures. Network transparency was an essential goal: implementation strove to extend that transparency into the file system structure. Among the performance enhancement measures taken were the multiplication of file server tasks, and the introduction of a degree in choice of functionality in

varied file *types* and lock *modes*.

Some of the changes proposed for DFS925 have been described in the body of this report. Criticism of the global file space has been discussed; directory caching used as "hints" would aid to fully address workstation independence and availability. File caching, widespread in server/client-based systems, complicates the distributed maintenance of data consistency substantially. Given the independent file usage pattern in the Office Systems Group, it is felt that effective file migration policies - moving files to local or efficient disks upon frequent use - would better minimize the number of remote file references and enhance performance.

Data replication, in the tradition of LOCUS, would increase system availability, though inevitably complicating transaction guarantees. Propagation of changes to replicated files is a problem beyond the scope of DFS925. An alternative to change propagation while preserving consistency is to disallow file update when one file copy is not available. DFS925' lack of an explicit protection mechanism would always permit file duplication.

An implementation advantage of the integrated system model approach was that the control flow of local file system calls was unchanged; no additional local performance burden was added by distribution. However, the unnoticeable inefficiency of local file system calls is magnified severalfold when the call becomes remote. Redesign of common compound calls in order to minimize the number of internal file references made would increase the performance of the DFS925 file system overall. Studies have shown that performance of distributed or remote file systems can be tuned through adjustment in network packet size. Investigation of DFS925 performance gains through volume transfers and buffer read-ahead should be carried out.

Integration with further distributed applications could increase the functionality and performance of DFS925. Global file migration and volume choice policy algorithms could optimize the latent performance advantages of DFS925's global file system. Replacement of the CSS interface with the CP925's expected capacity for distributed interprocess communication would improve network performance by avoiding call setup costs for send and receive capabilities and buffer copies.

DFS925 was designed as a small distributed file system; design decisions such as a flat file and volume identification space, the maintenance of considerable state for active files and transactions, and the static limitations placed upon concurrency were possible because of the environment's homogeneity and limited scope. The integrated model was a successful choice by reasons of some of these limitations: given a more massive system and powerful special-purpose server nodes, the server/client model might be the more viable. The provision of complete transactional capability is probably not necessary in the design of a "Universal" file system. Allowing tools such as buffer-force-out and lock modes is sufficient to permit the construction of a reliable database system as an overlying application layer. Heterogeneous systems are the next challenge of the future: unless implementation biases change, the server/client model is the obvious choice. The advent of widely accepted network, remote procedure call, and data representation protocols will encourage the development of such heterogeneous systems.

12. Acknowledgements

My thanks to the Office Systems Laboratory at IBM San Jose Research for providing generous financial support and development freedom; it is my hope that DFS925 will be useful to the group in the continuance of their research. Thanks are also due my research advisor, Prof. Domenico Ferrari, for his guidance and advice, and the EECS Department at UC Berkeley for a background conducive to research and learning. Prof. Luis Cabrera offered thoughtful editorial and technical commentary.

The author would like to extend her gratitude in particular to Dr. James Wyllie for his infinite patience and invaluable help. Thanks also goes to Dr. Roger Haskin and Dr. Carl Hauser for the loan of their technical expertise and good will, and to Dr. Sten Andler for his aid in understanding CSS. Brent Welch was a sounding board and source of insight; Dain Samples supplied coffee, conversation and Unterstutz; Allen Akin provided patience and understanding.

BIBLIOGRAPHY

[BIRR80] Birrell, A.D., and Needham, R.M. : "A Universal File Server", IEEE Transactions on Software Engineering SE-6, 5, September, 1980.

[CABR84] Cabrera, L. F., Hunter E., Karels, M., Mosher, D. : "A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX 4.2BSD", Report No. UCB/CSD 84/217, University of California at Berkeley, December 1984.

[CHAM81] Chamberlin, D.D., et al.: "A History and Evaluation of System R", Communications ACM, 24(10), October 1981.

[CHER83] Cheriton, D. R., and Zwaenepoel, W. : "The Distributed V Kernel and its Performance for Diskless Workstations", Operating Systems Review, 17(5), Proceeding of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, N.H., October 1983.

[DEC84] "The DEC Heterogeneous File System". From correspondance with Songnian Zhou.

[GRAY78] Gray, J.N. : "Notes on Data Base Operating Systems", Draft, IBM Research Laboratory. 1978.

[GRAY81] Gray, J.N. : "The Transaction Concept: Virtues and Limitations", Proceedings of the Seventh International Conference on Very Large Databases, September 9-11, 1981.

[HUNT85] Hunter, E. : " Adding Remote File Access to Berkeley UNIX 4.2BSD Through Remote Mount", M.S. Report, University of California, Berkeley, December 1984.

[IBM84] Office Systems Group, IBM San Jose Research: "CSS925, FS925, CP925 and WHIM: Users and Programmers Manuals", IBM Internal Documentation, 1984.

[ISRA78] Israel, Jay E., Mitchell, James G., and Sturgis, Howard E. : "Separating Data from Function in a Distributed File System (DFS)", Xerox PARC Report no. CSL-78-5, September 1978.

[KARE85] Karels, M. : "The Berkeley Remote File System", UCB Report to be released, 1985.

[LYON84] Lyon, Bob, Sager, Gary, and members of the SUN NFS project. "Overview of the SUN Network File System, Sun Microsystems Inc. October, 1984.

[MITC81] Mitchell, James, and Dion, Jeremy : "A Comparison of Two Network-Based File Servers (CFS, XDFS)" Proceedings of the Eighth Symposium on Operating Systems Principles, 15(5), December 1981.

[OPPE81] Oppen, Derek C., and Dalal, Yogen K.: "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment", Report OPD-T8103, Xerox Office Products Division, Systems Development Department, October 1981.

[POPE83] Popek, G.J., and Thiel, G. : "Distributed Data Management Issues in the LOCUS System", Quarterly Bulletin on Database Enginering, 6(2), June 1983.

[SCHR84] Schroeder, Michael D., Birrell, Andrew D., and Needham, Roger M. : "Experience with Grapevine: The Growth of a Distributed System", Xerox Palo Alto Research Center, ACM Transactions on Computer Systems, Vol 2, No. 1 February 1984, pp 3-23.

[STON80] Stonebraker, Michael : "Retrospection on a Database System", ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980, pp. 225-240.

[STON84] Stonebraker, Michael: Class Notes, CS286, University of California at Berkeley, Spring 1984.

[STUR80] Sturgis, H., Mitchell, J., and Israel, J. : "Issues in the Design and Use of a Distributed File System (DFS)", ACM SIGOPS Operating System Review, 14(3), July 1980.

[SVOB83] Svobodova, Liba : "File Servers for Network-Based Distributed Systems: Updated Version of RZ 1186 (November 2, 1982)", IBM Zurich Research Laboratory, Switzerland, October 4, 1983.

[SWIN79] Swinehart, Daniel, McDaniel, Gene, and Boggs, David : "WFS: A Simple Shared File System for a Distributed Environment", Proceedings of the 7th Symposium on Operating Systems Principles, 1979.

[WALK83] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G. : "The LOCUS Distributed Operating System", Operating Systems Review, 17(5), Proceeding of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, N.H., October 1983.

[WALK83] Walker, Bruce J. : "Issues of Network Transparency and File Replication in the Distributed Filesystem Component of LOCUS", UCLA Report No. CSD 830905, November 1983.

[WELC85] Welch, Brent, and Ousterhout, John : "The Berkeley Network File System", To be published, May 1985.