

https://dspace.org/

# Design of a disk-based file server involving the mapping of "pseudo-disks"

**Hildyard, Mark William**

**2015-01-27**
http://hdl.handle.net/10539/16783

Downloaded from a DSpace repository. https://dspace.org/

# DESIGN OF A DISK-BASED FILE SERVER INVOLVING THE MAPPING OF "PSEUDO-DISKS"

Mark William Hildyard

A project report submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg 1989

Declaration

I declare that this project report is my own, unaided work. It is being submitted for the degree of Master of Science in Engineering in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other university.

[signature]

8th day of May 1989

## Abstract

The Department of Electrical Engineering runs a course "Engineering Applied Computing" where it introduces computer concepts and applications to undergraduate engineering students. For this it uses a laboratory of personal computers. A network was required to link these computers with a file server to manage the complexity of supplying software and collecting projects and tests off floppy disks. The file server had to also provide the basis for the automatic analysis of student programs.

The network involved floppy disk drive emulation, where node computers interface with the network via its floppy disk controller. This unique approach held definite implications and constraints for the file server design. One specific consequence was that, to resolve space limitations, the server provides mechanisms to map any of a number of logical "pseudo" disks into or out of a client node's disk drives.

Although the physical interface was subsequently abandoned, the design is still relevant as a disk-based file server. In the course of the report I have argued that the design's very centralized characteristic is particularly suitable for its educational application, where file collection and a large software pool is required. This centralization holds promise for the security of software on the system, increasing the possibility of an educational institution being provided cheaply with a very large pool of applications software.

Acknowledgements

Much assistance, discussion and encouragement came from the other members involved in the larger project, Robert Schutz and Eddie De Souza. I would also like to extend my thanks to Mr. G.T. Gray for supervision and to Prof. A.J. Walker for supervision and motivation.

The generous sponsorship by the Council For Scientific and Industrial Research was much appreciated, as was sponsorship during my undergraduate years by the Chamber of Mines Research Organization and their release to study for my Master's Degree.

Last but not least many thanks to my family for their patience and support.

DECLARATION . . . . . i
ABSTRACT . . . . . ii
ACKNOWLEDGEMENTS . . . . . iii
TABLE OF CONTENTS . . . . . iv
TABLE OF FIGURES . . . . . viii

1 INTRODUCTION . . . . . 1.1
1.1 BACKGROUND - THE NEED FOR A FILE SERVER . . . . . 1.1
1.2 REQUIREMENTS FOR THE FILE SERVER . . . . . 1.1
1.3 EXTERNAL CHARACTERISTICS OF FILE SERVERS . . . . . 1.3
1.3.1 Basic Concepts and Terms . . . . . 1.3
1.3.2 Main Benefits . . . . . 1.3
1.3.3 Type of Objects maintained by the file server . 1.4
1.3.4 Unit of Data Access . . . . . 1.4
1.3.5 Object Management . . . . . 1.5
1.3.6 Atomic Operations . . . . . 1.5
1.3.7 Concurrency Control . . . . . 1.5
1.3.8 Level of Functionality . . . . . 1.6

2 NETWORK CHARACTERISTICS AND ITS IMPLICATIONS FOR THE SERVER 2.1
2.1 BRIEF PHYSICAL NETWORK DESCRIPTION . . . . . 2.1
2.2 MOTIVATION FOR SUCH A NETWORK . . . . . 2.2
2.3 IMPLICATIONS FOR THE FILE SERVER . . . . . 2.3
2.3.1 Space Limitations . . . . . 2.4
2.3.2 Communication with the File Server . . . . . 2.5
2.3.3 Performance Considerations . . . . . 2.5

3 HIGH LEVEL DESIGN DESCRIPTION . . . . . 3.1
3.1 THE SERVER-NETWORK INTERFACE . . . . . 3.1
3.2 THE FILE SERVER . . . . . 3.2
3.2.1 Disk Structure . . . . . 3.2
3.2.2 Access Resolution . . . . . 3.4
3.2.3 Pseudo-disk Identifiers . . . . . 3.5
3.2.4 Mapping disks to drives . . . . . 3.5
3.2.5 Operations Supported . . . . . 3.6
3.2.6 Caching Strategy . . . . . 3.7
3.2.6.1 *Prefetch Strategy* . . . . . 3.8
3.2.6.2 *Replacement Strategy* . . . . . 3.9
3.2.6.3 *Write Strategy* . . . . . 3.10
3.2.7 File Server Management . . . . . 3.11
3.3 REQUIREMENTS FOR THE EMULATOR . . . . . 3.12
3.4 SOFTWARE REQUIREMENTS AT THE CLIENT . . . . . 3.13
3.4.1 The Client Interface . . . . . 3.13
3.4.2 The User Interface . . . . . 3.13

4 DESIGN USING PROCESS-RESOURCE DECOMPOSITION . . . . . 4.1
4.1 THE PROCESS-RESOURCE METHOD . . . . . 4.2
4.2 INTERACTIVE SERVING - 1st LEVEL DECOMPOSITION . . . 4.4
4.2.1 1ST LEVEL: PROCESS BEHAVIOUR . . . . . 4.4
4.2.2 1ST LEVEL: EXTERNAL RESOURCES . . . . . 4.7
4.2.2.1 Network Request Queue . . . . . 4.7
4.2.2.2 Sector Pool . . . . . 4.8
4.2.2.3 Network Response Queue . . . . . 4.8
4.2.2.4 Data Output Resource . . . . . 4.9

**4.3 INTERACTIVE SERVING - 2nd LEVEL DECOMPOSITION . . . 4.9**
4.3.1 PROCESS BEHAVIOUR DESCRIPTIONS . . . . . 4.9
4.3.1.1 Initialization Process . . . . . 4.9
4.3.1.2 Queue-Producing Process . . . . . 4.10
4.3.1.3 Individual-Node Server Process . . . . . 4.10
4.3.1.4 Disk-Read Process . . . . . 4.12
4.3.1.5 Disk-Write Process . . . . . 4.12
4.3.1.6 Complete-Sends Process . . . . . 4.13
4.3.2 LOCAL RESOURCES . . . . . 4.15
4.3.2.1 Configuration File . . . . . 4.15
4.3.2.2 Node Queues . . . . . 4.15
4.3.2.3 HD-Attachment Interface . . . . . 4.16
4.3.2.4 Common Software Information . . . . . 4.16
4.3.2.5 User Information . . . . . 4.18
4.3.2.6 Opened Disks Tables . . . . . 4.18
4.3.2.7 Cache . . . . . 4.21
4.3.2.8 Disk-Read Queue . . . . . 4.25
4.3.2.9 Node Number . . . . . 4.26

**4.4 INTERACTIVE SERVING - 3rd LEVEL DECOMPOSITION . . . 4.27**
4.4.1 COMMON DISK OPENING PROCESS . . . . . 4.27
4.4.2 INITIALIZATION PROCESS: Further Decomposition . 4.31
4.4.2.1 Resource Preparation Process . . . . . 4.31
4.4.2.2 Configuration Process . . . . . 4.31
4.4.3 NODE-SERVER PROCESS: Further Decomposition . . 4.33
**4.4.3.1 Sub-Processes . . . . . 4.33**
4.4.3.1.1 Initialization Process . . . . . 4.33
4.4.3.1.2 Decoding Process . . . . . 4.34
4.4.3.1.3 Read-Track Request . . . . . 4.34
4.4.3.1.4 Write-Sector Request . . . . . 4.36
4.4.3.1.5 Sign-On Request . . . . . 4.36
4.4.3.1.6 Sign-Off Request . . . . . 4.38
4.4.3.1.7 Attach-Common-Disk Request . . . . . 4.38
4.4.3.1.8 Attach-User-Disk Request . . . . . 4.38
4.4.3.1.9 Detach-Drives Request . . . . . 4.39
4.4.3.1.10 Change-Operating-System Request . . 4.39
**4.4.3.2 Local Resources . . . . . 4.41**
4.4.3.2.1 Drive Map . . . . . 4.41
4.4.3.2.2 Current User Record . . . . . 4.42
4.4.3.2.3 Current Request . . . . . 4.42
**4.4.3.3 Further Sub-Processes . . . . . 4.43**
4.4.3.3.1 Attach Operating System . . . . . 4.43
4.4.3.3.2 User-Disk Attachment Process . . . . 4.44
4.4.3.3.3 Common-Disk Attachment Process . . . 4.44
4.4.3.3.4 User-Disk Opening Process . . . . . 4.45

**5 IMPLEMENTATION OF THE MINIMAL FILE SERVER . . . . . 5.1**
**5.1 THE iRMX OPERATING SYSTEM . . . . . 5.1**
5.1.1 Jobs . . . . . 5.2
5.1.2 Tasks . . . . . 5.2
5.1.3 Segments . . . . . 5.2
5.1.4 Mailboxes . . . . . 5.3
5.1.5 Semaphores . . . . . 5.3
5.1.6 Regions . . . . . 5.3
5.1.7 Extension Objects . . . . . 5.3
5.1.8 I/O Features . . . . . 5.3
**5.2 CODING PHILOSOPHY . . . . . 5.4**

5.3 CODING THE DESIGN . . . . . 5.5
5.3.1 Processes . . . . . 5.6
5.3.2 Scheduling of Tasks . . . . . 5.6
5.3.3 Resources . . . . . 5.7
5.3.4 Queue Resources . . . . . 5.8
5.3.5 External Resources . . . . . 5.8
5.3.6 I/O Operating System Features . . . . . 5.10
5.3.7 Use of Other iRMX Features . . . . . 5.10
5.3.8 Alterations to the Design . . . . . 5.10
5.4 FUTURE WORK . . . . . 5.11

6 POSSIBLE EXTENSIONS TO THE MINIMAL FILE SERVER . . . . . 6.1
6.1 COMPACTION AND THE CONTIGUITY OF 'DISKS' . . . . . 6.1
6.2 ANALYSIS MACHINE LINK . . . . . 6.2
6.3 PRINTING SERVICES . . . . . 6.3
6.4 FILE TRANSFER . . . . . 6.4
6.5 SHARED DATABASES . . . . . 6.5
6.6 NODE-TO-NODE MESSAGE PASSING . . . . . 6.7

7 EVALUATION AND CONCLUSIONS . . . . . 7.1
7.1 CLASSIFICATION OF THE PROPOSED FILE SERVER . . . . . 7.1
7.2 EVALUATION . . . . . 7.2
7.2.1 Performance . . . . . 7.3
7.2.2 Operating System Independence . . . . . 7.4
7.2.3 Protection Mechanisms . . . . . 7.4
7.3 CONCLUSIONS . . . . . 7.5

APPENDIX A: COLLECTIONS OF PROCESS-RESOURCE DIAGRAMS . . . . . A.1
A.1 INTERACTIVE SERVING: 1ST LEVEL . . . . . A.2
A.2 INTERACTIVE SERVING: 2ND LEVEL PROCESSES . . . . . A.3
A.3 INTERACTIVE SERVING: 3RD LEVEL . . . . . A.5
A.3.1 The Common-Disk Opening Process . . . . . A.5
A.3.2 Decomposition of the Initialization Process . . A.6
A.3.3 Decomposition of the Node-Server Process . . . A.6
A.4 INTERACTIVE SERVING: 4TH LEVEL . . . . . A.10
A.4.1 Further Decomposition: Node-Server Process . . A.10

APPENDIX B: PROGRAM LISTINGS . . . . . B.2
B.1 EXTERNAL RESOURCES: THE NETWORK-SERVER INTERFACE . . B.2
B.1.1 Network Request Queue . . . . . B.2
B.1.2 Sector Pool . . . . . B.5
B.1.3 Network Response Queue . . . . . B.8
B.2 SERVER PROCESS-BODY . . . . . B.12
B.2.1 Server Module . . . . . B.12
B.2.2 Global Literals, Constants and Types . . . . . B.13
B.2.3 String Manipulation Routines . . . . . B.15
B.3 INTERNAL (2ND-LEVEL) RESOURCES . . . . . B.17
B.3.1 Node Queues . . . . . B.17
B.3.2 HD-Attachment Interface . . . . . B.20
B.3.3 Common Software Information . . . . . B.25
B.3.4 User Information . . . . . B.28
B.3.5 Opened Disks Tables . . . . . B.30
B.3.6 Cache Resource . . . . . B.39
B.3.7 Disk-Read Queue . . . . . B.55

**B.4 2ND-LEVEL PROCESSES** . . . . . **B.56**
B.4.1 Initialization Process . . . . . B.56
B.4.2 Queue-Producing Process . . . . . B.59
B.4.3 Individual-Node Server Process . . . . . B.61
Node-Server Process Body . . . . . B.61
Drive-Map . . . . . B.63
3rd-Level Processes . . . . . B.67
Sub-processes, Common to 3rd-level Processes B.71
B.4.4 Disk-Read Process . . . . . B.76
B.4.5 Disk-Write Process . . . . . B.77
B.4.6 Track-Send Completion Process . . . . . B.79
**B.5 SUB-PROCESSES COMMON TO 2ND LEVEL PROCESSES** . . . . **B.80**
B.5 Disk-Open Process . . . . . B.80

**REFERENCES**

List of Figures


| Figure | | Page |
|---|---|---|
| 3.2.1.1 | FILE STRUCTURE OF THE SERVER'S DISK . . . . | 3.3 |
| 3.2.3.1 | THE STRUCTURE OF A PSEUDO-DISK IDENTIFIER . | 3.5 |
| 4.2 | INTERACTIVE SERVING: 1ST LEVEL . . . . . | 4.4 |
| 4.3 (a-c) | INTERACTIVE SERVING: 2ND LEVEL PROCESSES | 4.11 |
| 4.3 (d-f) | INTERACTIVE SERVING: 2ND LEVEL PROCESSES | 4.14 |
| 4.4.1 | COMMON-DISK OPENING PROCESS . . . . . | 4.27 |
| 4.4.2 | DECOMPOSITION OF THE INITIALIZATION PROCESS | 4.32 |
| 4.4.3 (a-c) | DECOMPOSITION OF THE NODE-SERVER PROCESS . . | 4.35 |
| 4.4.3 (d-f) | DECOMPOSITION OF THE NODE-SERVER PROCESS . . | 4.37 |
| 4.4.3 (g-j) | DECOMPOSITION OF THE NODE-SERVER PROCESS . . | 4.40 |
| 4.4.3.3 (a, b) | FURTHER DECOMPOSITION: NODE-SERVER PROCESS | 4.46 |
| 4.4.3.3 (c-e) | FURTHER DECOMPOSITION: NODE-SERVER PROCESS | 4.47 |
| 4.4.3.3 (f-h) | FURTHER DECOMPOSITION: NODE-SERVER PROCESS | 4.48 |
| 4.4.3.3 (i, j) | FURTHER DECOMPOSITION: NODE-SERVER PROCESS | 4.49 |

# 1 INTRODUCTION

## 1.1 BACKGROUND - THE NEED FOR A FILE SERVER

A Personal Computer Laboratory consisting of about 40 machines is currently used by the Faculty of Engineering at the University of the Witwatersrand for introductory courses in computing, specifically tailored for Engineering students. These courses teach programming languages, Operating System Principles and the use of Application Software (such as Spreadsheets and Databases). Assignments and Tests are supplied and completed on floppy disks. Each machine has disks with the Operating system and application software required for particular assignments.

The distribution of disks leads to a severe logistics problem. With approximately 200 students involved, a large amount of copying is necessary in distributing each assignment. Each time an alteration is made, all copies need to be changed and it is difficult to ensure that each student has the correct version.

Similarly, as bugs are corrected in th[illegible]ications software (which are not necessarily commercial pr[illegible] each disk needs to be updated. In addition there is a [illegible] security problem in that there can be unrestricted copy[illegible] these applications.

A *central file-store* accessible to the personal computers through a Local Area Network, provides a solution to these problems. This file-store will contain a pool of applications software, for which only the latest versions need be made available. User space is also provided thus avoiding the physical circulation and retrieval of floppy disks. The design of this file-store is the subject of this report. Another dissertation [E. De Souza,1], covers design of the local area network.

Centralization of the students' files, facilitates collection of their solutions. This forms the basis for the envisaged *automatic analysis of solutions*. It will be performed on a dedicated machine and includes the provision of multiple versions of problems, with each version sufficiently unique as to ensure that the correct solution could not arise from foreknowledge of the problem. Ultimately automatic marking is envisaged, where correctness and style of programs are evaluated, and suitable textual output produced. This is the subject of another report [R. Schutz, 2].

## 1.2 REQUIREMENTS FOR THE FILE SERVER

The network is to be used for student projects and tests. Therefore features provided in many networks, such as message passing and the sharing of data, (or at least unrestrained versions of these features), are undesirable. Thus the File

Server must occupy the central position in a logical Star connected system, with all requests routed to the file server or to other server machines.

The network/file-server solution must provide:

- *Shared resources* viz. User disk-space and Common software.
- A *Client-Interface* or the underlying *mechanisms* for requesting and accessing common and user software, as well as any other features provided by the file server.
- *Efficient response* to requests.
- *Protection mechanisms* for Common (Applications) software.
- *Operating System Independence* - MSDOS is the primary operating system for the computers to be networked. However, the provision of other operating systems is anticipated, and it is undesirable that the file server software should require modification whenever a new operating system is to be added. The operating system itself, should require little alteration, for it to use network facilities.
- *Restrictions* on the *Sharing of data*. Essentially, sharing should be confined to that between a user and the Analysis Machine, and should not include user to user data sharing.
- Mechanisms to enable the *Analysis Machine* to access all users' software (for students' solutions). This may entail a different Client-Interface for the Analysis Machine, or require features to be present which the Analysis Machine alone is capable of using.
- The possibility of *extending services* - a printing service is particularly necessary. However, dedicated servers may be necessary for these services.

A Large central storage is required for the shared resources. The common area could require 50 MBytes for system software and applications packages. The easy addition and modification of applications in this common area is required. At present the system must cater for 200-250 students, and a pool of 40 Personal Computers (The number would tend to augment). On-line storage the size of at least one floppy disk (360 KBytes) is required per user. Thus the On-line storage should be 200 MBytes or more.

The *efficiency* of response is especially important with the actual provision and transfer of file data, where the delay should not significantly add to delays inherent in the processing of the data by the PC's operating system. The delay in response is not as critical when resolving requests for access to user and shared software or for any requests for higher level features.

A small amount of software is to be allowed unrestricted distribution. This category would include laboratory exercises, sample programs and basic utilities. For the rest **protection** must be provided on three levels:

- Effective prevention from the copying of protected applications to floppy disk. Clearly the achievement of this is pivotal, since any further protection mechanisms can be bypassed if unrestricted copying is possible.
- Restricted access. Not all software will be available to all users, since different courses have different emphases, and the availability of certain software may be undesirable during tests etc.

- Restricted number of users. Software licensing would require that for any product, the number of copies available for use at any time reflects the number of copies purchased - although special contracts for use of the software on networks may be available. The expense of commercial software could be very large if a copy of a particular product needs to be bought for each machine on the network. This is especially inexpedient if the package is required only by a small number of users. Protection mechanisms should ensure that the number of users of a particular package do not at any instant exceed the number of copies available (purchased) for that product. This should lead to a substantial saving in software costs.

## 1.3 EXTERNAL CHARACTERISTICS OF FILE SERVERS

As previously mentioned, the nature of the education environment is such that certain features of importance in many networks (such as provision for Shared Data-Bases) are not major requirements for this network. In addition, the unique physical implementation chosen for the network (discussed in the next chapter), introduces some diverse concerns from those applicable to most file servers. However, the basic concept of sharing a storage resource is the same. Thus a brief analysis of the most salient features of file servers has been included, so that ultimately the design can be placed in context with other file servers, by fitting it into this model.

Svobodova [3] gives a comprehensive survey of the high level characteristics of file servers, as well as a more detailed internal comparison of individual file server designs - whether implemented or proposed. The main *classifications* of file server characteristics identified in this article are presented here, without attention to specific designs.

### 1.3.1 Basic Concepts and Terms

A file server is a subsystem which offers a number of basic functions to clients for manipulating a central common store. The *client* is the software on a machine connected to the network which invokes the server functions, but is distinct from the human user. The *client interface* includes the primitive mechanisms implementing the communications protocol for invoking server functions. Operations provided to clients by the interface reflect a one-to-one correspondence with functions available on the server. This interface is not suited for the human or application level. Higher level operations can be abstracted from combinations of these primitive operations and made available as additions to the operating system. Alternatively, software comprising a 'network interface shell' [4], can translate client operating system requests to functionally equivalent file server requests.

### 1.3.2 Main Benefits

Resource sharing is the basic aim of all file servers, but other benefits result or can be exploited.

- Resource Sharing - The sharing of a physical storage device, as well as software and databases.
- Automatic backup and Recovery - Relief from the burden of preventative measures against storage failure and user errors can be provided.
- User Mobility - Independence of a user from a particular workstation without the need to physically transport disk media.
- Disk-less Workstations are possible.
- Other Servers - The file server forms the basis for providing other types of services, and sharing other resources e.g. a high-speed printer.

### 1.3.3 Type of Objects maintained by the file server

High level servers have the file as their basic object for manipulation, with a correspondence between the files on the server and files available to the client. Operations provided in the client interface are file-based, generally including creating, reading, writing and deleting files and perhaps higher level operations for the protection, organization and concurrent access of files.

Conversely, low level servers have as their basic object for manipulation, blocks of storage. The server manages the allocation and de-allocation of blocks of storage for clients. No structure is imposed on these blocks, requiring that the client's file management system group these blocks into abstract files. Such servers function as remote virtual disks and the correspondence between the server's and clients views of files does not exist. Thus Operations provided in the client-interface involve the manipulation of blocks.

### 1.3.4 Unit of Data Access

For low level servers the unit of data access could be the size of the blocks of storage allocated, or some sub-block within it. Higher level servers whose basic object of storage is the file, can provide different granularities for retrieving and updating data in a file. Levels of data access include:

- Unstructured fixed-size block retrieval
- Entire file retrieval. Entire files are fetched from the server and cached on a local disk or downloaded into memory. Such 'File Transfer Servers' should have characteristics resembling those of Mass Storage Systems [5], except for the scales involved - i.e. the size of the storage, volume of transfer and frequency of retrieval.
- Page-level file access. Entire file transfers can be undesirable for very large files, and impracticable for Disk-less workstations. Page-level access alleviates these problems.

- Byte-level file access allows random access to small fields within a file and is particularly suitable for database systems.

### 1.3.5 Object Management

This concerns the facilities available for clients to organize and protect their objects. Organization is provided through directories, protection through some form of access control.

*Directory Service*
If supported, each user is supplied with a list of the names of available objects. The directory system may also abstract the presentation of objects to the user from the internal naming on the server in which case the directory maps textual names to those of the object-ID's at the server. If the server is file-based, a full-scale filing system might be implemented.

*Access Control*
Mechanisms can be supplied which in general enforce the isolation of users' objects, yet allow sharing of certain objects. Access control can be *Identity* based or *Capability* based.

In the identity based approach, the server maintains access lists for each object. A user needs to obtain an authenticated connection through some form of password protection, before access can be gained.

Capability based access requires that a user present the file server with a capability (usually the Object ID). This capability is a unique identifier chosen from a sparse name space. The sparseness can be chosen to make the possibility of forgery (whether deliberate or accidental) sufficiently improbable. Capability based access is generally preferred [6], [3,pp 372]. However, it does not easily allow different users different access rights to a particular object (such as read-only access for one user and read-write access for another) [6,pp 452].

### 1.3.6 Atomic Operations

The implementation of atomic operations together with recoverable files ensures the consistency of data. Files are recoverable, where previous consistent versions are available. Operations are atomic if the file is only altered when the entire operation completes successfully. If the operation fails, the file remains unchanged or reverts to a previously consistent version.

### 1.3.7 Concurrency Control

If file sharing is supported, with write access available to at least one user, then the synchronization of concurrent accesses is required. Concurrency control involves preventing new accesses to a file or region within the file until a sequence of

operations on the file has been completed. This makes the sequence of operations atomic, in that no other operations on the file can be interleaved with it. The primary mechanism for control is a two-phase locking protocol. 'Locking' obtains exclusive access to a region, while 'unlocking' releases the region. The scope of the region for which exclusive access is required, may be the whole file, portion of the file or a database record or field. Use of locks can lead to the 'deadlock' problem which concurrency control must address [3,pp 359], [4,pp 144].

### 1.3.8 Level of Functionality provided through the Client-Interface

Svobodova identifies two extremes for the complexity of file servers [3,pp 355]. A server offering only low end functions, appears to the client interface as a backing store device, allocating blocks of storage to clients and being responsible only for space management on the central store and data transfers across the network. Higher level functions (such as a suitable filing system) must be provided by software on the client's side, through combinations of primitive operations. At the opposite extreme file servers can provide a very high level interface, implementing a full-scale filing system, controlling concurrent accesses of files, providing protection mechanisms and supporting hierarchical organization of files.

The trade-offs thus involve the relative requirements for software on the server and at the client. A low complexity server requires a large amount of software at the client to provide suitable file management. Its advantage is flexibility, allowing the simultaneous use of different filing and access control methods. The high complexity server has a full-scale filing system as its interface, providing high level facilities while requiring very little software on the client machine.

Birrell and Needham [6], introduce the concept of the *Universal file server*, as one which combines the potential advantages of both low and high level servers. This server is built about a low level 'backing store server', which can be accessed directly by clients running private filing systems or merely requiring backing storage. In addition the server provides a common filing system which uses this backing store server. Clients can thus choose between the high level file-based interface, or the low level block-based interface.

# 2 NETWORK CHARACTERISTICS and ITS IMPLICATIONS FOR THE SERVER

The unique characteristic of the network is that client computers are interfaced to the communications bus, using **Floppy Disk Drive Emulation**. From the hardware perspective at the client machine, the network is a floppy disk drive. The advantage of such an approach is its transparency to already existing non-network software, allowing applications to run without modification in the network environment, yet without limiting the client to a particular operating system. Such an implementation means that the file server cannot be file-based and that a floppy disk *Track* will be its basic object for manipulation. This results in some special constraints for the file server on its efficiency, and in its communication for operations beyond that of data transfer.

## 2.1 BRIEF PHYSICAL NETWORK DESCRIPTION

A detailed analysis of the network can be found in E. De Souza's dissertation [1].

The network consists of approximately 40 client machines, connected to a file server machine via a 'star-bus' communication link. (Groups of clients are connected along a common bus and these busses are connected to the server in a star formation.) The logical topology is that of a file server at the centre of a fully star-connected network. Data is broadcasted openly (no encryption). The link allows transmission rates of 1-1.5 Mbits/s, and error detection and re-transmission is supported. The client machines are Olivetti-M24 personal computers, which are 'IBM-PC type' microcomputers with MS-DOS the primary operating system. Other operating systems are available, and will be supported.

Client computers access the network via a Floppy Disk Drive Interface. The Interface card is connected to the computer's floppy disk controller as if it were another floppy disk drive. This card consists of a Floppy Disk Drive Emulator, a memory bank in which a Cache of Disk Tracks is implemented and an interface to the communications link. The drives being emulated use double sided, double density disks with 40 tracks per side - although the floppy disk controller supports up to 80 tracks per side. Emulation involves providing the data in bit form to the floppy disk controller, exactly as if it were being read off a disk. Since data on disk is MFM-encoded, this involves encoding data required by the disk controller, and decoding data from the disk controller.

Data transfer is performed as follows:

An application requests file operations. In processing a request, the operating system requires sectors from a disk device and requests a sector (or number of sectors) from the floppy disk controller. The disk controller steps from the current track to the required track and the emulator follows

this stepping. For the reading operation, the emulator has no knowledge of the actual sector(s) required, but only of the track (and side) on which the sectors are found. The track-cache is then searched and if the track is in the cache, emulation begins immediately. Otherwise a request for the track is sent along the communications link to the file server, and emulation begins once the track has been received. The disk controller monitors the data coming from the emulator until the required sector(s) is/are received, which it then transfers to a buffer supplied by the operating system. The writing operation is similar, except that the emulator is aware of the particular sector(s) being written. The track-cache is accordingly updated, and each sector's data is sent as a separate operation to the file server.

The Network Interface Unit at the file server will have the responsibility of polling the clients for requests. It will have, in addition to its own local memory, direct access to the server machine's memory. This prevents the inefficiency of relocating large blocks of data (tracks and sectors), between the two memory banks.

## 2.2 MOTIVATION FOR SUCH A NETWORK

Having a customized network is generally attractive, since the detailed documentation on the network would always be available - something which is not normally the case with commercially obtained systems. This aids flexibility, enabling the system to evolve in a direction suitable to the specific application. Such development, also provides the basis for further research.

Assuming the desirability of a customized system, the chosen implementation has advantages to both physical and development costs, because of the level of transparency which is achieved. The approach utilizes an already existing high speed interface to the computer (its floppy disk controller), although a high speed interface to the network is still required. The additional hardware is completely transparent to the client machine, since it behaves wholly as a floppy disk device. Hardware modifications are not necessary and there are no additional bus arbitration considerations. The principle justification for the particular physical implementation is therefore economic.

The implementation also has software advantages. A requirement for the file server was that it should approach operating system independence, allowing new operating systems to be employed, without requiring modification to the server. In addition software written in these operating systems should run without modification in the network environment. This is best achieved by having the file server *emulate a client's disk* (i.e. providing the client with the image of a disk and not to be confused with the disk drive emulation previously considered). A major part of any operating system is involved in abstracting basic disk storage into files and providing functions for manipulating these files. By presenting the file server to the operating system as the image of a disk, the client will immediately have available

to him a full scale file management system (FMS) with naming conventions and a directory structure. This simplifies the file server in that it does not have to provide a filing system to clients. Nor do the client operating systems have to be altered so as to make use of the server's filing service. These advantages are applicable to all file servers based on disk emulation, whether the disk commands at the client are trapped in software at the disk driver level [Hudson, 7], or in hardware by physical disk drive emulation. In our system, given the physical disk drive interface at the client, the most natural approach for the file server would be disk emulation.

Other file server systems attain transparency and allow well-behaved applications - which perform I/O through the proper operating system interface - to run unmodified in the network environment, while providing a *file-based* interface. This is achieved through extending the client operating systems to include a Network Operating System. A 'Network Interface Shell' intercepts calls to the operating system for manipulating files and directories. If the requests reference files stored locally, they are passed on to the operating system for processing. If the requests reference files maintained by the file server, they can be handled in two different ways:

- The requests can be *immediately redirected* to the file server. The approach is often used in systems based about a particular operating system [8]. In such an approach, the file server is a superset of all the operating systems which it serves [4]. For each new operating system to be supported, the Network Interface Shell redirecting calls must be designed and the file server must be augmented to process the new requests.
- Alternatively the server provides a *standard set of filing functions*, while each operating system available to clients is extended with software, which implements operating system functions using these standard server functions.

Such systems are clearly more operating system dependent than disk-based file servers, requiring a fair amount of alteration to support new operating systems.

## 2.3 IMPLICATIONS FOR THE FILE SERVER

It was decided that with the physical interface being based on disk drive emulation, the correct role for the file server would be that of disk emulation. (However, it is possible for a file-based interface to be constructed, by having the client operating system utilize the disk drive emulator merely for communication with the file server, and not as a disk device. Such communication is relatively inefficient, while the interface cannot exploit the benefits of simplicity and operating system independence, discussed in the previous section. Given the physical basis, such an approach was considered inappropriate). Disk emulation entails presenting the client with the image of a disk, and mapping references to this 'pseudo-disk' to its actual location on the file server's disk. The file server will not be

responding to requests for data from the client operating system (which references sectors on the pseudo-disk), but to requests from the disk drive emulator (which requires entire tracks of the pseudo-disk for read operations, while providing single sector data from write operations).

Thus the server fits essentially into the class of low-level or block-based *file servers* (*cf:* section 1.3). Alternatively, it can be classed distinctly from file servers, as a disk server - or more precisely as an intelligent disk server, since it will also provide higher level mechanisms for organization and security [4,pp 140]. As with all *non file-based* servers, the ignorance of what constitutes a file in the client's view means that it is difficult to support certain high level functions which are essentially file-based. These functions include atomic transactions and concurrency control for shared data-bases. Even printing services are more easily implemented on a file-based server.

Other consequences result specifically from the physical mechanisms of the network. These include the space limitations of pseudo-disks, how to achieve communication between the client and the file server, and the emphasis on the speed of response.

### 2.3.1 Space Limitations

As a result of interfacing with the network using the floppy disk controller, the immediate physical appearance provided by the existing device driver, is that of a single floppy disk of approximately 360 Kbytes. (Two sides, 40 tracks and assuming formatting of 9 sectors per track, with each sector 512 bytes long). Some method for providing access to a large amount of storage is required.

The size of the disk - and hence the number of sectors - presented to the operating system, could be expanded by increasing the size and number of tracks. The track-size is limited by the bandwidth of the floppy disk controller. However, it is *conceptually* possible to map the 160 tracks (80 tracks and 2 sides), which are directly addressable via the floppy disk controller, to a much larger set of tracks at the file server. This could be attained by dividing the total tracks into groups and maintaining a record of the current group. All references to tracks are within the current group - to reference tracks outside this group, a message must be passed to the file server, which changes the current group. One directly addressable track (e.g. track 79, side 1) would have to be used for this communication. The approach requires replacing the existing device driver and would allow the network to be presented to the client operating system as a hard disk. Client operating systems may have limitations in their support of hard disk devices - e.g. in MSDOS 2, the maximum disk size is limited to 32 MBytes [9].

A simpler solution was chosen - that of providing the client with a large number of virtual floppy disk devices all of which utilize the *single* floppy disk interface. Development of the floppy disk interface therefore included writing a new device driver, which functions exactly as the former floppy disk driver,

but allows 25 devices [De Souza, 1]. (This means that with one actual floppy disk unit remaining, 26 drives are available to each client - in MSDOS, drives A to Z). The disk drive emulator specifies the device when requesting tracks from or transferring sectors to the file server, and the file server has to maintain disk images for each device at each node.

Having 25 devices does not in itself provide sufficient space. If disk images for these devices were fixed, there would be approximately 9 MBytes for user files and common applications. The file server will rather contain a large number of floppy disk volumes or pseudo-disks, and provide mechanisms for these volumes to be mapped into and out of a client's devices. A directory service is required for presenting and ordering the pseudo-disks available to the client. However the client's operating system is still responsible for the internal ordering of files within the pseudo-disks.

### 2.3.2 Communication with the File Server

The existence of the network and file server, is completely hidden from the client operating system, which is aware only of disk devices. Data retrieval is from these disk devices, using already existing operating system mechanisms. However, some form of communication is required between a client and the file server, to allow the switching in and out of pseudo-disks and to support any high level functions, such as user security. Since tracks 0 to 79 are available and only tracks 0 to 39 are viewed as part of the disk, absolute accesses to track 40 can be used to support communication.

A request for an operation is written to track 40, sector 1, and if a response is required, an immediate read of track 40, sector 1 is performed. The mechanisms to support this and the structure of messages corresponding to functions offered by the file server must be provided for each client operating system, as the client interface. The disk drive emulator extracts the message from the sector (without requiring knowledge of the message), and transmits it to the server, which having processed it responds (if necessary) with status and other information. The disk drive emulator composes a track from the response message and allows the reading of track 40 to proceed. The immediate read operation following a request to the server, allows for atomic operations, in that the client machine cannot proceed until the request has been processed. (The disk drive emulator delays the read operation until the message from the server has been received). The communication mechanism is inefficient in that an entire sector must be written and read, whereas the message-length may only be a few bytes. However, efficient use of the communications link is maintained, since only the messages are transmitted.

### 2.3.3 Performance Considerations

In considering performance, typical values for certain disk characteristics have been used (*cf*: Shiell and Markoff [10]). Due to the physical floppy disk drive interface, the data transfer rate is limited to 250 Kbit/s or one track in 200 ms. The full

but allows 25 devices [De Souza, 1]. (This means that with one actual floppy disk unit remaining, 26 drives are available to each client - in MSDOS, drives A to Z). The disk drive emulator specifies the device when requesting tracks from or transferring sectors to the file server, and the file server has to maintain disk images for each device at each node.

Having 25 devices does not in itself provide sufficient space. If disk images for these devices were fixed, there would be approximately 9 MBytes for user files and common applications. The file server will rather contain a large number of floppy disk volumes or pseudo-disks, and provide mechanisms for these volumes to be mapped into and out of a client's devices. A directory service is required for presenting and ordering the pseudo-disks available to the client. However the client's operating system is still responsible for the internal ordering of files within the pseudo-disks.

### 2.3.2 Communication with the File Server

The existence of the network and file server, is completely hidden from the client operating system, which is aware only of disk devices. Data retrieval is from these disk devices, using already existing operating system mechanisms. However, some form of communication is required between a client and the file server, to allow the switching in and out of pseudo-disks and to support any high level functions, such as user security. Since tracks 0 to 79 are available and only tracks 0 to 39 are viewed as part of the disk, absolute accesses to track 40 can be used to support communication.

A request for an operation is written to track 40, sector 1, and if a response is required, an immediate read of track 40, sector 1 is performed. The mechanisms to support this and the structure of messages corresponding to functions offered by the file server must be provided for each client operating system, as the client interface. The disk drive emulator extracts the message from the sector (without requiring knowledge of the message), and transmits it to the server, which having processed it responds (if necessary) with status and other information. The disk drive emulator composes a track from the response message and allows the reading of track 40 to proceed. The immediate read operation following a request to the server, allows for atomic operations, in that the client machine cannot proceed until the request has been processed. (The disk drive emulator delays the read operation until the message from the server has been received). The communication mechanism is inefficient in that an entire sector must be written and read, whereas the message-length may only be a few bytes. However, efficient use of the communications link is maintained, since only the messages are transmitted.

### 2.3.3 Performance Considerations

In considering performance, typical values for certain disk characteristics have been used (*cf*: Shiell and Markoff [10]). Due to the physical floppy disk drive interface, the data transfer rate is limited to 250 Kbit/s or one track in 200 ms. The full

time involved for a transfer now includes - in addition to the access, disk latency and transfer times of a floppy disk - the delay for the emulator to request and receive a track from the file server. Network transfer time for a 4.5 Kbyte track at 1 Mbit/s is 37 ms, and file server delay could include disk access time, and a *queuing delay* while other clients' requests are being processed. The maximum possible total data requirement, occurring when all 40 clients are accessing 5 tracks every second, is 8 Mbit/s. For the network devices to approach floppy disk speed, the number of requests needing to wait for a network transfer must be minimal, and peek-load transfers must be spread over a longer period. Thus extensive caching at both the Emulator and the file server is essential.

*Disk Contiguity*

The critical resource limiting performance at the file server is its disk. Typical mean access times for hard disks range from 30 to 50 ms and in addition there is an 8.33 ms average latency. The transfer time for 5 Kbytes, the approximate track-size at the client, is typically 6 ms - assuming that the disk is not interleaved. To make efficient use of the disk, the time spent in data transfer should not be dwarfed by access times. This entails a read-ahead strategy, where for each access to the disk a number of tracks of the pseudo-disk are retrieved and placed in the server's cache. For this to require few disk accesses, contiguity in the storage of pseudo-disks is important.

Contiguity of the disk is a normal requirement when large data transfers are necessary. However, the contiguity of files on the pseudo-disks is also very important. The importance is augmented by a disparity in views - the client operating system requests and manipulates sectors, whereas the file server supplies entire tracks. A badly fragmented file may have each of its sectors on a different track. With N sectors/track, this means that N times as many tracks are transferred than would be necessary for a contiguous file. The effect is to not only degrade performance for a particular client (by increasing track access times), but to load the whole network, with an N times increase in transfers across the network and accesses to the server's critical disk resource.

The combined effect of the fragmentation of files on the pseudo-disks and fragmentation of the pseudo-disks on the hard disk, is to multiply their individual effects. As an (unrealistic) *worst-case illustration*, assume the following. At the client, tracks have 9 sectors of 512 bytes, file allocation is in single sector clusters, and due to fragmentation each sector of a file resides on a different track. At the server, sectors are the same size as those of the client, file allocation is in single sector clusters and there is absolute fragmentation. In such a case the transfer of a small file of 4.5 Kbytes would require 81 separate disk accesses! (Most of these accesses would take closer to a typical minimum access time of 15ms (7ms + 8.33ms) than to a mean access time. Nevertheless, aggregate access time would be greater than a second). In addition 330 ms of network transfer time is used, and this is for a single client.

The degradation in performance caused by fragmentation of files within the pseudo-disks, makes it attractive to provide a condense function for eliminating fragmentation. However, this operation is completely dependent on the client's file system , and should not be included in the kernel of the file server. It would nevertheless be desirable to provide this as an extended function to certain oft-used operating systems, particularly MSDOS.

*Prefetching*

Prefetching attempts to maximize the 'hit-rate' of a cache, by anticipating future requirements and fetching them before they are requested. It relies on the assumption that much data access is sequential in nature. The cache at the file server will perform prefetching in addition to its read-ahead strategy (cf section 3.6.1). The cache at the Emulator performs prefetching, by searching for the track subsequent to that currently requested by the client, and if it is not available, requesting it from the file server. Both caches rely on sequential track requests and hence the approximate contiguity of files on the client pseudo-disks.

# 3 HIGH LEVEL DESIGN DESCRIPTION

Design of the file server cannot be isolated to software requirements on the server machine, but encompasses the interface between the server and the network, software requirements at the client, and requirements for the emulator at the client's network interface.

## 3.1 THE SERVER-NETWORK INTERFACE

The Network Interface Unit at the file server machine has direct shared access to the file servers's memory and thus the interface between the server and the network is implemented in the server's memory. The file server dequeues requests in their received order but processes them asynchronously - although requests from a particular client must be kept serialized - and there may be a number of requests which are simultaneously being processed. The interface cannot efficiently consist of single input and output queues for both messages and data, since this would require the dequeueing and relocation of large blocks of data. Instead, disk write requests contain pointers to the memory block in which the sector data is stored, and responses to track requests contain a pointer to the track's location in the server's cache.

The network interface unit and file server have dual access to four memory resources. These are the:

- *Input Requests Queue* - consisting of fixed size entries, large enough to contain any request message. Disk write requests contain a pointer to an entry in the sector list.
- *Sector List* - Comprising a list of available blocks of memory the size of a client disk sector.
- *Output Response Queue* - consisting of fixed size entries, with responses to read requests containing pointers to the track's location in the track cache.
- *Track storage blocks* in the File Server's Cache.

For transfers to the server, operation at the interface is as follows. Clients send messages requesting operations - the Interface-unit at the client, prefixes the message with the client/node identity. The interface unit at the server copies the request into the first available entry in the Request Queue. If the request is for a disk write operation, an available entry is obtained from the sector list and a pointer to this entry added to the request entry. Once the sector data has been transferred, the request entry is released into the queue. The server consumes requests, immediately releasing the entry, but releases the sector entry to the sector list only once the required operation has been completed.

For transfers from the server, operation at the interface is a rough reflection of the above. The server enqueues messages (prefixed with the destination node) in the Response Queue. If it is the response to a Track-request, the message contains a

pointer to the track's location in the cache. This track must remain in the cache until it has been transferred across the network. The network interface unit consumes the message and transfers it (stripped of the destination node) together with the track-data to the required node. It must then indicate to the cache that the transfer is complete.

Clearly, operations performed by the server and the interface unit on the resources are concurrent, requiring mutual exclusion in the access and update of certain resource variables such as queue sizes and pointers. This can be achieved through the use of semaphores, but the exact approach has not yet been decided.

## 3.2 THE FILE SERVER

The file server aims to provide space for user files and present the user with a large number of common applications, which may run under different operating systems. Its primary operation involves providing data from, and resolving accesses to, pseudo-disks. Access resolution is identity-based, requiring that an authenticated connection be established by the user. The server must maintain for each client node, a permanent record of the current user's characteristics and of the pseudo-disks attached to the client's drives.

### 3.2.1 Disk Structure

It is possible to partition the server's disk storage into fixed blocks the size of a pseudo-disk (approx. 360 Kbytes). This has the advantage of the pseudo-disk being stored entirely contiguously. However such a representation would be grossly inefficient in space usage, since at any point in time most pseudo-disks will have a substantial unassigned area.

Instead pseudo-disks are stored as variable-sized files which contain only the portion of the disk which has been used. In addition to efficiency of storage, the range of file management functions provided in the server's operating system is then available for the manipulation of the pseudo-disks. References to tracks are readily translated to offsets within the file.

Blank disks must be created when a disk is accessed for the first time. This is best done by copying relevant sectors of a blank formatted disk to the new file. A file containing this information is required for each operating system supported - for MSDOS this involves the first 12 sectors, or 6 Kbytes. The file is automatically extended when tracks, which have not previously been used, are written to. Contiguity and therefore large granularity in the allocation of space to files is necessary for good performance and granules should be an integral multiple of the track-size of the pseudo-disk (*cf*: sect 2.3.3). Even operations which increase the unassigned area of a pseudo-disk, such as a user deleting files it contains, cannot reduce

the file length - thus the condense function (*cf:* sect 2.3.3) which makes files within a pseudo-disk contiguous, should also compact the pseudo-disk.

When a pseudo-disk is requested by a user, an opening connection must be created to the file. A single connection is sufficient for all users accessing a particular file, and a table contains information on each connection and open disk. It is essential that track requests which require disk accesses should not require an extra disk access to determine the sector location of that part of the file on the disk. Rather, when the open connection is created, the chain of sectors for the file should be loaded into RAM. This criteria is generally satisfied in file systems which support random file accesses. However, of three commonly used file system structures, *block chain, index block and file map* [11, pp 162), operating systems which use the block chain approach would not be suitable. (In the block chain, each block contains the pointer to the next block in the file).

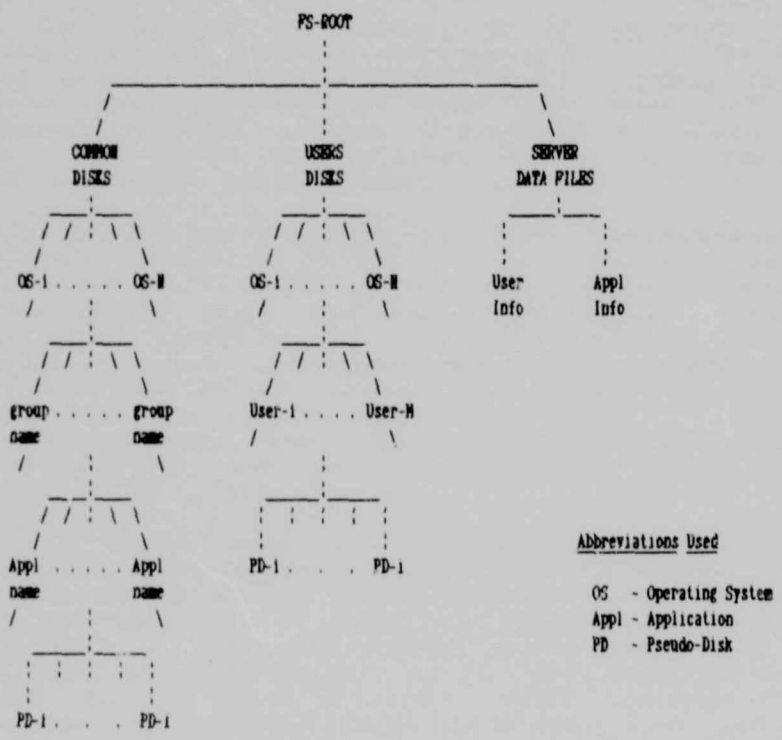


Fig 3.2.1.1 - File Structure of the Server's Disk

The hierarchical structure chosen in the storage of files on the disk is illustrated in Fig 3.2.1.1. Terminal nodes are data files (mainly pseudo-disk files) and intermediate nodes the directory

files. System data files (currently) include information on the users, common software and configurable settings such as the default operating system. An application may be a single disk or a series of disks comprising a package. Applications are grouped into categories such as programming languages, spreadsheets, word processors etc. Directories for operating systems, groups and applications, have textual names. Private user directory names are composed from the user number - i.e., Uxxxx.

This multi-tiered storage, reflects the structure presented to the user, but is not necessary for the operation of the file server (such as to ensure unique disk naming, since unique numbers are already associated with each disk - *cf* sect 3.2.3). It was however considered better practice to maintain the hierarchical separation of disks between operating systems, applications and users, at the server, particularly with a view towards 'house-keeping'.

### 3.2.2 Access Resolution

With the education environment, it is important that strict separation of user files is enforced, with only the owner being allowed access. Shared files are mostly confined to application programs made available to users, and users can be readily categorized according to the applications to which they have access. This makes identity-based (rather than capability-based) access resolution most suitable.

Identity-based access requires the establishment of an authenticated connection between the user and the file server. A simple password validation scheme was chosen - although more secure methods exist such as transformation passwords [12], reconstructed passwords [13], and pass-algorithms, based on the knowledge of secret algorithms [13]. Each user is assigned a unique user number and the user information file correlates user numbers with the user characteristics. This information includes the user's password, the number of pseudo-disks allotted in his private directory, and the configuration of application software to which he is allowed access.

Access to common application software is based on access lists, which refer to categories of users and not individual users. The list indicates either read-write access, read-only access or no access, for each class of user. A special category has manager status, which is automatically given *Carte Blanche* or read-write access to all applications. It is desirable to reserve one category for unofficial or casual users, allowing access to a small subset of software. These users would have no user status, and thus no private directory, but maintain their own files on floppy disk.

Access to the application is first determined. The information file for common software associates each application with a unique number and entry. This entry contains the number of the operating system under which the application can be used, the path name on the server's disk, the number of pseudo-disks in the application, the access list and the maximum number of concurrent

users. The user's current operating system must match that under which the application runs, and his user category must have the required access attribute in the application's access list.

An additional access characteristic is that of the 'load type'. An application may be specified as protected, meaning that it must only be attached to a drive if the software is to be executed immediately. Loading software at the client must allow for two types of load - one where it executes the software after attachment - and it must indicate to the file server, the type of load being performed (*cf*: section 3.4.2). Access will only be granted if the load type matches that of the application.

Further constraints are enforced for each disk required from the application. For each pseudo-disk which is to be attached, the aggregate number of attachments (current users) must be less than the maximum allowed by the disk's application. Write-access to a pseudo-disk is strictly single access. This means that a user cannot be granted write-access to an already-open disk, or read-access to a disk opened for write-access. For concurrent-update to be supported, this requires modification (*cf* Sect 6.5).

### 3.2.3 Pseudo-disk Identifiers

In addition to textual names, unique '*shorthand*' identifiers are necessary for pseudo-disk references. This is required at the file server for tabulating already open disks, at the Emulator's Cache for referencing to which disks tracks belong, and for client requests to specify the disks of an application or the whole application.

A unique two-byte identifier can be composed from the user-number (for user disks) or from the application number (for application disks) - *cf* Fig 3.2.3.1. The most significant bit distinguishes between a private user disk and a common application disk. The four least significant bits are used for the disk's number within the application or user set - if zero, the reference is to the whole application or user set. The scheme allows up to 2048 different applications and 2048 user sets each with a maximum of fifteen pseudo-disks.

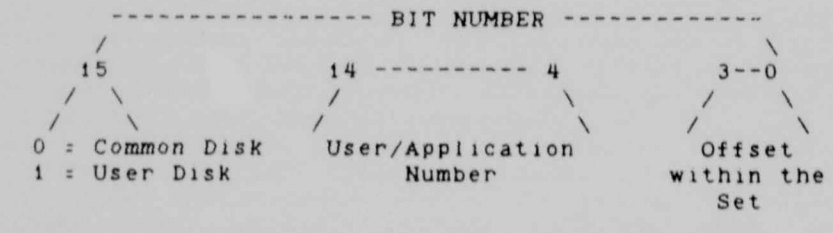


Fig 3.2.3.1 - The Structure of a Pseudo-Disk Identifier

users. The user's current operating system must match that under which the application runs, and his user category must have the required access attribute in the application's access list.

An additional access characteristic is that of the 'load type'. An application may be specified as protected, meaning that it must only be attached to a drive if the software is to be executed immediately. Loading software at the client must allow for two types of load - one where it executes the software after attachment - and it must indicate to the file server, the type of load being performed (*cf*: section 3.4.2). Access will only be granted if the load type matches that of the application.

Further constraints are enforced for each disk required from the application. For each pseudo-disk which is to be attached, the aggregate number of attachments (current users) must be less than the maximum allowed by the disk's application. Write-access to a pseudo-disk is strictly single access. This means that a user cannot be granted write-access to an already-open disk, or read-access to a disk opened for write-access. For concurrent-update to be supported, this requires modification (*cf* Sect 6.5).

### 3.2.3 Pseudo-disk Identifiers

In addition to textual names, unique 'shorthand' identifiers are necessary for pseudo-disk references. This is required at the file server for tabulating already open disks, at the Emulator's Cache for referencing to which disks tracks belong, and for client requests to specify the disks of an application or the whole application.

A unique two-byte identifier can be composed from the user-number (for user disks) or from the application number (for application disks) - *cf* Fig 3.2.3.1. The most significant bit distinguishes between a private user disk and a common application disk. The four least significant bits are used for the disk's number within the application or user set - if zero, the reference is to the whole application or user set. The scheme allows up to 2048 different applications and 2048 user sets each with a maximum of fifteen pseudo-disks.

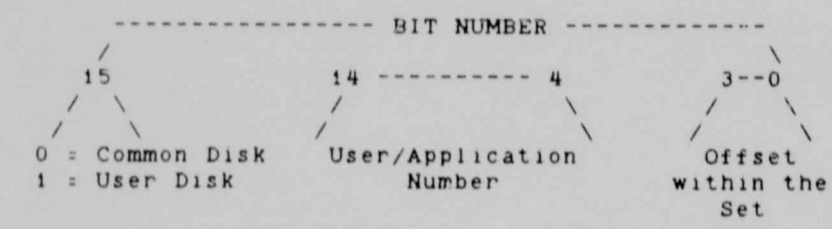


Fig 3.2.3.1 - The Structure of a Pseudo-Disk Identifier

### 3.2.4 Mapping disks to drives

The file server maintains a table for each client which maps the client's drives to pseudo-disks. There is a level of indirection in that the table does not reference the file containing the pseudo-disk, but rather an entry in a table containing information on each open pseudo-disk file. More than one client may have a drive referencing a single open file. When a new disk is attached, the first available drive is used, and if the file is not yet in use it must be opened and recorded in the table of open files. If a disk is detached from a drive and the client is the only current user of the disk, the file must be closed and removed from the table.

There are 26 drives of which the second is an actual floppy disk drive and is permanently unavailable for disk attachment. The first drive - the client computer 'boot' drive - always contains the first system disk of the current operating system. At initialization the only disks attached are those of the default operating system, which includes network software. Once a user has 'signed-on', his drives contain the following disks in order:

- 1st system disk of the Op. Syst.
- Floppy Disk Drive
- Rest of the Op. Syst. (if it has more than one disk)
- Directory Disk
- User Disk(s)

He is then able to add and remove disks. The directory disk makes common software accessible by listing the applications and their corresponding numerical identifiers. The operating system and directory disks cannot be removed by the client and are removed only when an operating system change is requested.

### 3.2.5 Operations Supported

The file server should provide three levels of functionality - data transfer, connection management and high-level, file-oriented services.

Operations for data transfer facilitate the exchange of tracks and sectors between the emulator and the pseudo-disk files at the server. Requests reference drives, and the server must interpret to which file it refers. The speed of response is critical, with a large cache acting as intermediary.

Connection management concerns the organization of a client node's connection, and involves alterations to the drive-map and current user characteristics, and provision of information on these characteristics. Essential operations are:

- *Signon* - The request specifies a user number and password. If successful, the user becomes associated with that node and his disks are attached. Status information is returned.
- *Signoff* - All currently attached disks are closed and the connection is re-initialized. No response is issued.
- *Attach an Application* - The request specifies a disk number, the type of access and the type of load required (*cf*: section 3.4.2). The application number is extracted from the disk number and access determined. If the disk offset of the number is zero then it refers to all disks

in the application, else it refers to a single disk. The disk(s) are attached to consecutive drives and status information is returned.

- *Attach another User disk* - The request must specify the disk offset within the user-set and the type of access required. If the given offset is zero, then all disks in the user-set which are not already attached will be attached.
- *Change Operating System* - The request must specify the operating system number. All currently attached disks are removed and the required operating system, directory disk and user disks for the new operating system are attached. If the request is unsuccessful then the default operating system is attached.
- *Remove disks from drives* - The request must specify the starting drive and the number of drives from which disks are to be removed. Certain disks cannot be removed (*cf* section 3.2.4). No response is issued.

These operations together with those of data transfer form the core or minimal file server. Non-essential functions include the alteration of passwords and the provision of information such as the current user's characteristics, the state of the drive-map (disks currently attached) and details on specific applications.

High level functions are those which may be desirable but do not naturally fit into the pseudo-disk oriented architecture. Instead they are intra-disk and file-oriented, encompassing printing services, collection of student files for the analysis machine, and support for the concurrent update of data-bases. These are considered in Chapter 6.

### 3.2.6 Caching Strategy

Disk caching has been discussed (*cf*: section 2.3.3) as being essential for dealing efficiently with requests for tracks from the pseudo-disks. It involves duplicating data from slow disk-based memory in fast RAM. The objectives are twofold - a reduction in the data access-time for the clients, and the efficient use of the shared disk-resource.

A reduction in data access-time is achieved when the requested data is already in the cache, and can be transferred immediately to the client, without first having to be retrieved from disk. This avoids the large disk access-time required for the disk head to move to the required area. Most importantly, it avoids substantial queuing times awaiting availability of the hard disk - which is the critical resource limiting the file server's performance (*cf* sect 2.3.3.1).

Successful caching must also optimize the usage of the disk, in terms of the average useful data retrieved (or stored) per unit time. The average access-times for disk devices (due to head movement) are substantial compared to the transfer times of the data [10]. Thus the number of head movements, and therefore the total number of disk accesses, should be minimized.

A generalized caching of the file server's disk could be performed, with disk sectors as the basic data unit. This has the advantage that all operations involving retrieving disk data, benefit from the cache. However almost all disk requirements will be concerned with requests for pseudo-disk tracks, and data retrievals should be on the boundaries of these tracks. Thus the pseudo-disks will be cached, with a pseudo-disk track the basic data unit. This allows more specific assumptions to be made in the caching strategies. The cost is that only disk operations involving pseudo-disk tracks can benefit from the cache.

Cache design involves choosing algorithms which maximize the cache hit-rate and optimize disk usage. Jones [14] identifies three issues - the prefetch, replacement and writing strategies. Prefetch algorithms anticipate what data to bring into the cache before it is requested, while replacement algorithms determine what data should remain in the cache. The write strategy attempts to accumulate write-requests and to decouple actual disk writes from these requests.

### 3.2.6.1 *Prefetch Strategy*

Prefetching tries to predict future requirements, on the basis that most data requirements are sequential accesses to files. However, a request for a track contains no information on which file a client is accessing within the pseudo-disk. Therefore the prediction must be based on the assumption that the file he is accessing is stored on consecutive sectors of the pseudo-disk.

Three forms of prefetching are performed:

- Whenever a track must be retrieved from disk, a whole group of sequential tracks will be fetched.
- Subsequent to a read request for a track which is in the cache, or a write request to the last sector of this track, a group of tracks will be fetched unless the next track of that pseudo-disk is already in the cache .
- For a pseudo-disk which has been newly attached at a client's drive, the first group of tracks will be brought into the cache (This is restricted to protected disks, whose programs will be executed immediately - *cf*: section 3.4.2).

The number of tracks in a group to be fetched is primarily determined by a prefetch number, a characteristic chosen for each pseudo-disk. This allows prior anticipation for each pseudo-disk of how many of its tracks could be needed in short succession. A shared pseudo-disk with a large program file that is seldom modified and thus probably contiguous, would have a larger prefetch value than the user pseudo-disks which may contain a number of small files.

The prefetch number gives the suggested range of tracks which should be fetched as a group. However, if a track within this range is already in the cache, then only the tracks up to this one will be fetched. The size of the group can also be limited by the available space in the cache.

The philosophy behind retrieving a group of tracks rather than a single track is that once the time has been spent to move the disk head to a particular area of the disk, it is beneficial to retrieve as much data from that area as seems likely to be needed later - thus avoiding future head movements.

Clearly head-movement is only reduced if the pseudo-disk files are stored contiguously or approximately contiguously on the server's disk. This is in addition to the original assumption of the contiguity of files within the pseudo-disk.

3.2.6.2 *Replacement Strategy*

Whenever any data is brought into the cache, this has to displace data which is already there. Decisions for replacement should be based on the probable (future) usefulness of data blocks relative to one another.

Jones [14] identifies three types of replacement algorithm - those of random replacement, first-in/first-out (FIFO) and least recently used (LRU). The FIFO method always replaces the oldest data, whereas the LRU method displaces the data which has been unused the longest. McKeon [15] discusses a fourth method useful for small caches, and based on the frequency of access of data blocks. He claims that the LRU method is an approximation of this frequency of access algorithm, but breaks down for small caches where turnover is high.

An LRU method involving replacement of groups or clusters of tracks, rather than individual tracks, was chosen.

LRU replacement becomes desirable in large caches when the frequency of use varies for different data blocks. In our network, disk caching is also performed locally at each client node, giving the system the characteristic of having two caches in series. A consequence is that multiple accesses to a particular track at the client, do not generally result in multiple requests for the track from the file server. However, tracks in the cache would be requested multiple times due to separate clients requesting tracks off shared disks. Also all disk-write operations at a client do not only effect the local cache, but are forwarded immediately to the file server.

Another consideration is that serially prefetched data be given sufficient time to be used, before being replaced. With LRU replacement, after a track has been requested it will be removed to the rear of a replacement queue. However, prefetched tracks are not required immediately and could reach the end of this queue before being requested - even though their use could be imminent. Rather, every time a track from the cache is used, the likelihood of the subsequent ones being requested is high - and these should also be moved to the rear of the replacement queue. This form of 'prefetching' internal to the cache, can be achieved by dealing with replacement on a group basis.

Data is brought to cache in variable size groups of tracks. Group replacement involves retaining each of these groups as a single unit for the purposes of replacement. A replacement queue is

**Author** Hildyard Mark William
**Name of thesis** Design Of A Disk-based File Server Involving The Mapping Of "pseudo-disks". 1989

***PUBLISHER:***

**University of the Witwatersrand, Johannesburg**
**©2013**

***LEGAL NOTICES:***

**Copyright Notice:** All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed, or otherwise published in any format, without the prior written permission of the copyright owner.

**Disclaimer and Terms of Use:** Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page) for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.