# The New Filestores

George D.M. Ross

Draft of August 26, 1988

# Contents

# List of Figures

# Chapter 1

# Introduction

This report describes the Computer Science Department's new generation of file servers. The Department has around sixty 68010-based discless workstations [5,2] attached to an ethernet-like local-area network, served by five file servers. These servers fall into two types: the "old"-style filestores [4,11,17] provide basic facilities, essentially unchanged since the original filestore was commissioned in 1976; while the "new" servers, under development since 1985, provide more advanced services to both co-resident and network-based clients, including a fully hierarchic directory structure, "redirectors" which allow clients to hide the actual locations of files from users, and a protection scheme tailored for a teaching environment.

Unlike the old-style filestores which were dedicated to that task alone, the new system can co-exist quite happily with interactive users of the same machine. Indeed, the file system regards the various protocol interpreters which serve files to the network as exactly equivalent to user processes. Depending on which modules are loaded, a system can be configured to import files using various protocols, or not to import; to export files using various protocols, or not to export; and to have a file system on a local disc, or to have no local file system. These choices can be exercised independently, though obviously not all combinations are sensible. Given the presence of a local file system, there is the additional choice as to whether to allow free access to all local users, or whether to restrict access rights only to those users listed in a local authorisation database.

The systems configured to date fall into three main classes: discless workstations, relying on network servers for their file system support; workstations with a small local disc, but relying on network servers for the bulk of their storage; and systems intended primarily as servers with large local file systems, but with few resources available for local users.

The new file servers have been designed on the principle of strict partitioning of functional components into individual modules. The belief is that inter-module violations should rarely be necessary, and indeed are symptomatic of a poorly thought-out design. The only exception to this rule is that the server-statistics file system is able to interrogate directly the tables of the other server components, on the understanding that the results obtained are intended to provide a general statistical overview of the system rather than a precise snapshot of its state.

Most of the code of the system is designed to be called by concurrently-active pro-

cesses. Access to common tables is interlocked *via* semaphores, which are always claimed and released carefully in order to avoid deadlocks.

Note that both the old-style filestores and the new file system regard a file's data content as merely a sequence of bytes to be stored safely and supplied on request. Neither imposes any restriction or structure whatsoever. The file's interpretation is regarded as a matter solely for its users, as is any locking of particular file sections which applications might require (though mandatory file-grained locks are imposed by both types of server).

The construction and features of the file system are described in Chapter 2; Chapters 3 and 4 discuss the ways whereby clients and servers communicate; while Chapter 5 considers some of the system administration facilities.

## 1.1 Historical background

By 1976, when the original filestore [4,11] was built, the Department had already developed a network of high-speed reliable flow-controlled point-to-point serial links connecting together a number of INTERDATA-70 series machines, a PDP-8, PDP-9 and PDP-15, and sundry other processors and peripherals. This original filestore was based around an INTERDATA-70 processor with one CDC-9762 76 Mbyte disc drive and one point-to-point link pair to each of its clients. This network was gradually expanded, until by 1980 the Filestore, now with two disc drives, supported a cluster of ten or so discless INTERDATA-74 workstations, with another INTERDATA-70 acting as a compute-server and communications multiplexor.

Towards the end of this period two projects were begun which eventually led to the INTERDATA/link cluster being superceded: one was the development of the Edinburgh Local Area Network [7,6], an ethernet-like communications bus operating at 2 Mbaud; the other was the FRED-machine, which was put into service as the APM [5,2]. As the FRED-machine population built up, the original filestore, which was in any case beginning to show its age, was unable to handle the load, and in 1984 it was replaced by a FRED-machine-based filestore with a 160 Mbyte winchester disc on an SMD controller [17]; to all intents and purposes, however, the clients' view remained the same. There are currently around sixty APMs and five servers (four old-style, one new-style as described below) attached to the network.

Latterly the Department has been building up a separate "standard" ethernet with a number of commercial minicomputer systems and workstations attached, with the intention that this network and the ELAN should be unified into one logical network. It has become clear that the facilities offered by the current APM system and the protocols used on the ELAN would make it difficult to achieve any degree of transparency. Consequently the APM system is in process of being rewritten[1] and standard protocols such as TCP and UDP [13,14,15] introduced. As the old-style filestores would also have to be substantially modified, it was decided to rewrite them from scratch too, at the same time providing the improved functionality which a more powerful server can support.

[1] Described elsewhere

# Chapter 2

# The Disc File System

In this chapter we discuss the file servers' disc file system in general terms. This is but one of the possible file systems which can coexist in any given machine,[1] though it is the primary one as far as the file servers are concerned. From the point of view of the file system, a remote client accessing files *via* some file access protocol with a local interpreter has exactly the same status as a co-resident client whose run-time support is speaking directly to the various local file systems.

The local file system is constructed in two main layers, with a user interface layer on top to stitch everything together. The lowest of these layers is the file system proper which manages the disc space and performs access control on the files held thereon. Each file is identified at this level only by a 30-bit "file-ID."[2] The facilities provided include the ability to create and delete files, to open and close them, and to read and write their data blocks.

The middle layer is responsible for maintaining the system's hierarchic directory structure. At this level the structure applies strictly to files in the local file system, though "redirectors," which are interpreted by the client, extend the user's preception of the tree to encompass a number of file systems and, indeed, autonomous servers with different types of file systems. Filenames are translated by considering each component in turn and, starting at the root of the directory tree, searching for the next in the directory whose ID has been found by translating all the previous path components in turn. The directory layer uses the "spare" two bits in the file-ID for its own purposes.

The user interface layer essentially takes each user-request and performs the necessary sequence of directory and file-system operations. In the simplest cases, such as writing a block of data, these may translate one-for-one, while creating, renaming or copying, for example, will each involve several more primitive operations. It should be noted that users interface *only* with this layer; the lower levels are not externally accessible, and are detailed here only to show how the file system is put together.

[1] The others currently comprise the server-statistics filesystem, an old-style importer and an NFS importer; an importer for the new File Access Protocol described in section 4.2 is planned.

[2] The top two bits are reserved for use by the directory layer.

## 2.1 The File System proper

The file system layer is responsible for disc management and file protection enforcement. The facilities it provides are summarised in section A.1. Each of the interface procedure has as one of its parameters an access rights record, detailing the caller's ID, privileges and group memberships, as described in section 2.1.3.

The basic sequence of operations is to open a file, to read and/or write one or more blocks, and then to close it again. This stateful approach has a number of advantages over the alternative, stateless, method whereby the file's ID would be specified with each read or write request:

- it reduces the cost for data-transfer operations, and hence the total cost of accessing more than one or two blocks of the file;
- it allows for files to be opened by suitably endowed agents on behalf of other less-well endowed clients; and
- it allows the file system to enforce file-grained concurrency control.

Supported access modes are "`read`", "`modify`" which allows any block of the file to be written, and "append", enforced here but implemented mainly in the user interface layer, which allows new data to be written beyond the end of the current file but not the alteration of existing data.

When a file is opened, the file system returns a token for the file, the size of the file (in bytes) and a flags word. The flags include some from the header, such as whether the file has been improperly closed or is marked for backup, and some which are generated, such as whether the file has world-read access (because access control is enforced by the file system, any higher-level caches must either be user-specific, or contain only world-readable information) or has a reference count greater than one.

Reading and writing blocks require only that the token issued when the file was opened be presented along with a data buffer. Files are extended automatically by writing, one block at a time, beyond the area which already contains data; for efficiency reasons blocks are allocated to the file several at a time, though only those actually written to are accessible to the user. Only the last block written may be short; an error status will be returned in those cases where an attempted write would leave a short block in the middle of the file.

Closing a file invalidates the token supplied when it was opened and removes any concurrency constraints. Optionally, the allocated space may be truncated to exactly that required to hold the data content. Files may be closed "successfully," in which case their contents are immediately accessible, or "unsuccessfully," in which case an explicit operation is required before the data can be accesssed. This capability is used by protocol interpreters and client run-time support systems to provide a measure of protection against the later use of corrupt data caused by the user program or client crashing.

Files grow automatically as they are written. In order that they may be shortened, a procedure is provided to truncate a currently-open file. The new size is specified in bytes, and any data which existed beyond that size becomes inaccessible. For reasons of security the converse operation is not supported.

There is a separate procedure for creating files, as this was felt to result in a cleaner division of the file system's tasks than would automatic creation in the "open" procedure.[3] The caller specifies a creation name, which is recorded (possibly truncated) in the file header for the benefit of file system structure management utilities, a partition number, the initial space allocation (note that these blocks are not accessible until written) and the file-ID of a "benefactor." This last is used to supply defaults for all ownership and access fields, with the exception of the "creator" which is taken from the supplied access record. The purpose of these ownership and access fields is explained in section 2.1.3.

Each file has a reference count associated with it which is incremented or decremented in steps of one at the request of the directory layer. Files are automatically deleted when their reference count goes to zero, implying that they are no longer needed.

The procedures for querying or modifying a file's miscellaneous attributes (basically, everything except data content) take linked lists of attribute records, each of which contains an attribute code, a status, and pointers to one or two buffers. These are processed, independently of each other, in the order in which they appear in the list, the entire request being an atomic operation on the file. Files are referred to directly by ID, rather than being required first to be opened, as attribute operations are generally of a one-shot nature and do not require to be interlocked with other concurrent users. The item codes and their parameters are shown in section A.2.

The "exchange" operation is provided for the benefit of applications implementing transactions by mechanisms such as differential files. It provides a means whereby the data content of two files can be logically interchanged as an atomic operation. The operation is performed "carefully" in the sense that in the unlikely event of a total system failure partway through, the data content of both of the files is guaranteed to be preserved, albeit perhaps not with the intended IDs; the critical window is of the order of milliseconds, however.

### 2.1.1 Logical Partitions

The file system does not deal directly with raw discs. Instead, each disc is divided up into one or more logical partitions. There are a number of advantages in this approach: these include the ability to mix both file-structured and non-file-structured areas, such as the disc header (describing the partition layout on the disc) and the bootstrap area; the ability to influence disc layout on a coarse scale, for example by placing commonly-used system utilities in a partition near the centre of the disc; the provision of coarse-grained access control; and independent quota enforcement.

Each 30-bit file-ID is divided into two parts: the high-order six bits are used to identify the logical partition, while, in the case of file-structured partitions, the low-order 24 bits identify the file within the partition. The partition identification part is currently further sub-divided into two equal parts, the high order three bits identifying the disc, and the low-order three bits identifying the partition within the disc. The partition module provides a defaulting service to the directory layer, with procedures to supply missing partition and disc numbers for one file-ID based on those of a fully specified second file-ID, and conversely to strip out the partition and/or disc numbers

[3] The interface layer groups these operations together with the directory operations, presenting the whole to the user as a single operation.

from a given file-ID if they correspond with those of a second. The directory layer uses this mechanism to make files' directory entries relative to their respective parent directories, allowing directory trees to be more loosely coupled with discs and partitions.

The interface between the partition sub-layer and the file system above it takes the form of block-read and block-write requests, each of which specify the partition ID and the block within the partition.

As well as controlling coarse-grain disc layout, the partition module maintains a cache of disc blocks. This is organised in "chunks," currently of eight blocks each, read from the disc in a single operation. Writes are done one block at a time, first to the disc and then copied into the appropriate cache chunk as required: this approach is sufficient in the sense that it does not make any concurrent read from the same block any less risky. Note that it is not safe to attempt the cache update before writing to disc, as the disc driver is at liberty to schedule any subsequent chunk read request before the write, possibly giving rise to cache inconsistency. The cache replacement strategy is LRU. An special status is returned whenever a read is from the last block in a chunk; this allows the file system to detect a condition where a read-ahead would be advantageous[4] and to use the request procedure provided to initiate the operation. With a $\frac{1}{2}$-Megabyte cache, the hit rate is typically over 90%, and often in excess of 95%. Concurrent access to the cache tables is under semaphore control. Each chunk slot has an associated status word, with a wait queue for those processes attempting to access a chunk while it is still in transit from the disc.

### 2.1.2 File-Structured Partitions

The key to a file-structured partition is its collection of file headers. These are grouped together into an "index file," the first block of which contains the header for the index file itself. The 24-bit part of the file ID which identifies the file within partition is subdivided into two: the low order 16 bits are used as to select the block within the index file containing the header; while the high order eight bits are used as a sequence number to catch "dangling" directory entries, being incremented by one, independently for each index file slot, each time a new file is created. The format of the file header is shown in section A.3. Each header is protected by a checksum, intended mainly to catch software bugs and occasional hardware errors as the disc controllers' ECC algorithms are considerably more powerful. No transaction mechanism is provided for file headers; instead the file system is careful not to start any operation unless it intends completing it.

A file's ID is stored in its header: if, on a header access, this is found not to match the ID specified, this is assumed to be due to a dangling directory entry, and a "no such file" error is returned. A slot-part of zero in the stored file-ID indicates that the header is free for reallocation.

The file system maintains a cache of file headers to improve performance and to make concurrent access to files easier to manage. Only one copy of any particular header is in the cache, shared amongst the zero or more users who might have the file open. When a user opens a file, both the requested access modes and a list of modes which the user is prepared to allow to other shared users of the file must be supplied. These are checked against the modes specified by any other users who might already have the file open, and

[4] Another is end-of-extent.

access is granted only if there are no conflicts. File access requests which are blocked by this test are not queued by the file system, but instead result in an immediate error reponse. "Control" access, allowing modification of a file's attributes, is always implicitly enabled for all files, except for the (short) duration of an "exchange" operation. This file-grained locking scheme is not discretionary: it is enforced for all file access requests. The header-cache replacement strategy is LRU, though only those headers which are not currently in use are candidates for replacement. Semaphores are used to maintain integrity and atomicity: each header is protected by an associated semaphore, while one common semaphore interlocks access to the global tables. In order to avoid deadlocks, no attempt is ever made to claim more than one of these semaphores at a time.

An extent-based allocation scheme is used; assuming that the partition is not too badly fragmented, this will result in a more compact representation than would be the case if each block of the file had its own pointer. Extent slots are allocated from the end of the header, growing back towards the start. Not all the allocated blocks need necessarily be used: indeed, as a file is extended new blocks will be allocated several at a time though only one will be written with each request. For security reasons, users are prevented from reading allocated blocks which they have not previously written. If possible, the file system will try to allocated new blocks contiguously with those in the final extent, and if successful will merely update the extent record to incorporate the newly-allocated blocks rather than allocating a new extent slot for them. Any unused blocks are, optionally, released when a file is closed. A cache prefetch is initiated whenever a read access pattern is noted to be sequential and the block just transferred was either the last block of the current extent or the last block of the cache chunk, as signalled by the status return from the read operation.

The "exchange" operation is performed by copying the extent records between the two files, using an additional "anonymous" file header as an intermediary. This operation is guaranteed to be atomic and risk-free, and is considerably more efficient than would be the copying of large numbers of data blocks. Both files must reside on the same partition, of course. The operation is performed as follows: the intermediary file, $I$ say, is created (but with no blocks allocated); the extent records of $A$ are copied to $I$'s header which is then flushed to disc; $B$'s extent records are copied to $A$'s header which then is flushed to disc; $I$'s extent records, originally from $A$, are copied to $B$'s header and it is flushed to disc; and finally the intermediary $I$ is deleted. Note that the data content of both $A$ and $B$ will always be preserved by this sequence of operations; even in the unlikely event of a system failure partway through, the system manager can find $I$ using the lost-files utility and either complete the exchange or delete the intermediary as required.

Timestamps note when a file was created, when it was last modified, and, if enabled for the partition, approximately when the file was last accessed. This last timestamp, which could be used by an archiving system or to delete infrequently-used files, is updated no more than one every 20 minutes or so, in order to minimise "unnecessary" disc traffic for heavily-used system utilities.

### 2.1.3 Access Control

Access control is based on 32-bit tokens: at the level of the file system there is no other form of user identification. These tokens are treated purely as bit-patterns, with any

conventional structure imposed by the system manager being for higher-level consumption only. Each user possesses a "user-ID" token, which would normally be unique but need not necessarily be so; in addition, each user may be granted the right to assert a number of additional tokens, referred to as "group-IDs" in recognition of their conventional purpose. When a file is accessed, the user's list of tokens is compared against the list associated with the file, the resulting access permission being the union of those permissions associated with tokens which matched. The mechanism for establishing the mapping between users and user- and group-IDs is described in section 2.4.

The tokens associated with a file fall into four categories:

- The "world" token, implicitly granted to all users, with its associated access rights (`world access` in figure A.3).
- The "local" token, implicitly granted to all co-resident users, with associated access rights (`local access`).
- A variable number of "group" access records (the array `access`), growing from the front of the file header towards the extent list, with associated access rights.
- The "owner" token, the "supervisor" token and the "creator" token, with `owner` access plus implicit "control" access.

When a file is created it normally inherits its access control data, with the exception of the creator-ID, from a "benefactor," chosen by the user interface layer to be either a previous version of the file, if one exists, or the parent directory in which the file is being created.

Two ownership tokens are maintained for each file, in order that that both the creator of the file and the owner of the directory (or the previous version of the file, should this be different for some reason) maintain full access rights. If only one token were maintained, one or other would inevitably be disenfranchised.

The "supervisor" token is intended as an aid to class management. The members of a particular class would be assigned to some group, with rights to the token being granted to the lecturers, tutors and demonstrators responsible for that class. This would give them full owner-access rights to the root directory of the tree and any files created as part of that tree, including the implicit right to alter a file's protection attributes. In effect, they have equal power over students' files as do the students themselves. Note that students would inherit ownership of, and hence full access rights to, any files which a supervisor might create in their directories.

In addition to the right to assert a number of access tokens, each user may also be granted various privileges. These include:

- *readall* allowing the user to read any file, irrespective of the protection set;
- *bypass* which, as its name suggests, causes the protection checking mechanism to be bypassed completely;
- *bootarea* which allows the holder to access unstructured partitions, and in particular the system's on-disc bootstrap area; and
- *admin* which enables conditional file access rights, particularly with respect to the user database files, and allows the user to change files' `owners` and `supervisors`.

### 2.1.4 Free Space Management

Free space management is performed using bitmaps, one for each partition, though as they are manipulated exclusively through a procedural interface it would be straightforward to implement some other scheme. The operations allowed are:

- to claim a specified range of blocks, returning an error if any was already allocated;
- to release a specified range of blocks, returning an error if any was not allocated; and
- to allocate a contiguous range of blocks, guided by a desired size and a suggested starting position.

Bitmaps are scanned 32-bits at a time whenever possible, starting either at the suggested location, wrapping round if necessary, or, if none is specified, from some (prime) number of blocks beyond the end of the area allocated in the previous call. The intention behind this algorithm is to make it more likely that files can be extended contiguously, for efficiency of access and to reduce the number of extent records required to describe them, the hope being that the hole which was left can be allocated at the time the file is next extended. The prime step-size should be less likely to interact with the bitmap size. If contiguous space is available then it is allocated, even if it is smaller than the requested size; if no contiguous space is available, then the requested number of blocks out of the first free area which is big enough are allocated; while if there are no free areas of the requested size, the largest is allocated.

The bitmaps are not stored on disc; rather they are built from scratch each time the file system is initialised. This simplifies manipulation, removes one possible source of inconsistency, and, incidentally, would make it easier to replace the entire free space manager with one using a different philosophy such as an explicit free list.

### 2.1.5 Quotas

These have not been implemented yet. The intention at the moment is that each partition will have quotas enforced separately.

### 2.1.6 The Disc Drivers

Though not strictly part of the file system, the disc drivers are described here since the local fixed discs are universally accessed through one or more file system partitions. Although all drivers present a common interface, each is required at present to "know" the geometry of the drives attached to its controller. Thus each system configured has a slightly different version of the driver tables built in. This slightly unsatisfactory situation has arisen because the only real drive-independent choice for on-disc geometry tables, *viz* sector zero of track zero of cylinder zero, or at any rate one of the low-numbered sectors, had already been allocated to the processor's primary boot firmware. Two interfaces and four drive types are currently supported: "standard" SMD, using a NEC controller chip on a dual-ported memory interface, with Fujitsu 2284 and 2294 drives; and ST506, using an Ambit Pace controller board on a parallel interface, with Fujitsu 2243 and Rodime 204E drives. Any other type of drive could easily be configured.

Disc read and write request messages are ordered by the driver process according to disc address, and are scheduled using an "elevator" algorithm. Transfers can be of any size, though the partition module will not at present ask to read more than eight blocks at a time, nor write more than one. The driver and controller co-operate to continue transfers across track and cylinder boundaries as required. Verification can be enabled for the SMD driver, independently for read and write transfers: only write verification is currently enabled.

Each driver consists of three sections: an outer envelope contains the definitions of the disc geometry and `%includes` a controller-dependent section and a controller-independent section. The "independent" section interprets requests and performs queue management and scheduling, calling on the "dependent" section through a standard interface to perform specific requests.

## 2.2 Directories

The directory layer is responsible for maintaining the correspondence between user-supplied filenames and filesystem-generated file-IDs. It presents the user with a fully hierarchic intra-filesystem directory structure, and provides "redirectors" which can be acted upon by the user's client system to unify the view of the various servers' directory structures. The B-Tree module described in section 2.3 below is used to provide key management and data storage facilities. Individual filename components can be up to 127 characters in length; case is preserved but ignored. The interface to the directory module is shown in section A.4.

Filename paths are passed to the directory layer as a linked list, already split into their constituent components by the user's run-time support package or by the protocol interpreter. This has a two-fold benefit: the maximal number of path components is not constrained by any limitation on the size of any buffer holding the whole, unsplit, filename; and the choice of meta-character to use as path separator can be made to suit each run-time environment individually.

Multiple file versions are permitted. These are numbered in relative terms, rather than absolute, with the most recent being version zero, the next being version $-1$, then $-2$ and so on. Directories and redirectors are constrained to exist in only one version; hence the version number of only the final path component need be considered. At present there is no automatic version limit mechanism: instead the directory module simply refuses to allow a version beyond a fixed reasonably large limit to be created. This facility will be provided in a future release.

The directory layer layer uses the two "spare" file-ID bits for its own purposes: bit-31 indicates whether a key (i.e. filename) translates to a single file-ID, in which case the translation can be used directly, or whether it translates to multiple versions or a redirector, in which case the translation is presented as a token to the B-Tree data storage section; while bit-30 is used to indicate that the file-ID corresponds to a directory (this latter is also known to the user interface layer, which treats attempts to read or write directories as special cases).

Three separate lookup procedures are provided for the benefit of the user interface layer. The basic primitive is `directory lookup one`, which attempts to translate the key supplied in the indicated directory. A successful translation as a file-ID results in a zero

status, a translation as a redirector results in a positive status, while failure is indicated by a negative status. The remaining two procedures call the first repeatedly for each element in the path list, starting with a known directory-ID (the "root" directory) and using the result of one step as the input for the next. The operation is terminated should any non-zero status be returned, with the status, the textual translation (if any) and the number of successful translations being passed back to the caller. Note that redirectors are not processed in the directory layer, but instead are passed back to the run-time support to be dealt with. The reason for having three different "lookup" procedures is that this makes the operation of the user interface layer considerably clearer.

As well as file-IDs, which if they have the same key as an already-existing entry are inserted as the most recent version, there are two forms of textual redirectors, *viz* `internal` and `external`. These are treated identically by the directory layer but result in a different status value being passed back as the result of a translation, allowing the higher layers to handle them differently. One common entry-deletion procedure is provided, the the type of entry to delete being known from the directory itself. The insertion or deletion of a file-ID entry results in a suitable adjustment of the corresponding file header reference count, with the file being automatically deleted by the file system if the reference count goes to zero (i.e. when all paths to the file have been removed).

Directories are just the same as other files as far as the file system is concerned, with no special deletion procedure being required. However, the user interface layer knows about the "directory" bit in the file-ID, and will refuse to delete a directory unless it has been confirmed to be empty.

A list giving the contents of a directory can be obtained using the `directory contents` procedure. This is a rather *ad hoc* mechanism, though it does have the virtue that the time that the directory is required to be open is minimised.

In order to speed up the translation of commonly-used filenames, the directory layer maintains a cache of recently-used names. This is organised as a number of directory slots, identified by file-ID, each containing name/translation pairs. Directory slot reuse is controlled by a LRU algorithm, while within slots the entries at present remain permanently allocated. Because the directory layer sits at a higher level than that at which file protection is enforced, the cache mechanisms must take care not to compromise file system security. This is achieved by making a cache entry only where the directory concerned is world-readable.

## 2.3 The B-Tree Module

As has already been mentioned, the directory layer makes use of a separate B-Tree package to perform key management and data storage (the interface is shown in section A.5). The package, which is also used by other server components such as the authorisation manager, has three parts:

- the I/O interface,
- the key manager, and
- the data record manager.

As well as interfacing to the file system, the I/O section implements transactions using a form of virtual file [18,12]. Two two-level indirectory maps are held in the file along with the tree and data blocks. The one-block roots of the maps, which are stamped with an epoch number to show which is the most recent, are in blocks 0 and 1 of the file. The high-order bits of a virtual block number are used to index into the current root block; the resulting entry is the physical block number of the second-level map block; the physical block number corresponding to the original virtual block number is obtained by using the low-order bits of the latter as an index into the former. A transaction is opened by selecting the root block with the more recent epoch number, the other one being ignored meantime. Blocks are read using the current indirectory map. When a virtual block is written for the first time a new physical block is selected and the second-level map block is updated; if this was the first time that map block was updated then a new physical block will be chosen for it and the root block will be updated accordingly. The transaction is committed by flushing the map cache, then incrementing the epoch number and writing the root back to the *other* root block site; it is abandoned simply by closing the file without updating the root. Although a small degree of complication is added by this process it is more than made up for by the considerable simplification in the logic of the rest of the B-Tree module and the other system components which make use of it.

The key management section maintains a B-Tree of variable-size keys and corresponding 32-bit data. Insertions follow the usual splitting algorithm. Deletions are complicated slightly by the variable key-size: a non-leaf key deletion requires that the next-highest key be borrowed from the appropriate leaf, possibly resulting in splitting if the borrowed key is larger than that being deleted; the shape of the tree is then readjusted, starting at the leaf from which the key was borrowed. Readjustment, which is carried out only if the node is less than half full, takes the form first of an attempt to merge adjacent blocks, right or left merging being chosen according to which gives the best use of space; if merging is not possible an attempt is made to rotate one or more keys left or right, again depending on use of space; if neither merging nor rotation was possible the attempt is abandoned, the assumption being that some subsequent readjustment will recover the wasted space. Note that empty blocks will never arise, as it would always be possible to merge or rotate in this case. No attempt is made to carry out tree manipulation operations in a "safe" manner, as the transaction system in the I/O interface allows any alterations to be backed out if required.

The data storage section provides a means whereby variable-size records may be stored in the same database file as the keys. When a record is inserted a token is returned to the caller (usually to be stored itself as the datum corresponding to some key). The record can subsequently be read, modified (provided its size is not changed) or deleted on presentation of the token. A size change requires that the record be deleted and reinserted, probably resulting in a different token being returned.

Both the key management and the data storage sections are self-contained units, and could easily be packaged up in an I/O section suitable for a user application. Indeed, the only changes required to the current I/O section would be to open the database file by name rather than by file-ID, and to adapt the block-read, block-write and close calls appropriately.

## 2.4 User Requests and the Interface Layer

The user interface layer is responsible for taking user requests in the standard form and performing the appropriate sequence of file and directory operations. These operations may map directly, may involve several more primitive operations, may be a no-op or may be rejected. Requests are serviced by a pool of procecsses, sharing access to common tables and all waiting on a single request queue. Each request is dealt with in its entirety by whichever process happens to have picked it up before the process returns to the tail of the queue to wait its turn for the next request. The request message format is given in appendix B:

- Each 32-bit request code consists of two parts. The least significant 16 bits specify the operation, while the most significant 16 bits indicate whether the operation is common to all file systems or whether it is a specific extension supported by one particular file system.
- For specific extension codes, bit 15, if set, indicates that there is a path in the "usual place." File systems which are otherwise unable to understand the request are nevertheless expected to part-translate the path, in this case probably returning a redirector pointing to another file system. This mechanism allows non-standard operations to take place through the normal global directory structure.
- User identification is as described in section 4.1.
- Each request contains a 32-bit identification tag, allowing the client to distinguish between responses to several concurrent requests.
- Responses contain a standard status value, a file system specific status code, and a textual message containing either an error message or the data corresponding to a redirector.

Standard request codes include the following: open file; read data; write data; close file; truncate file; make accessible (to unlock files to which the file system has blocked access because they were not closed cleanly); create directory; remove file; rename file. Non-standard codes recognised by the user interface layer of the local file system include: insert local redirector; insert external redirector.

Note that the user interface layer does not interpret redirectors; instead it passes the resulting text back to the user-level run-time support together with an indication as to the type of the redirector. A co-resident client would probably want to interpret both local and external redirectors, the former resulting in a new request to the local file system, with the latter resulting in a request to a different file system. A protocol interpreter acting on behalf of an external client, however, would only interpret local redirectors, returning external redirectors to the client for further processing. This means that the remote client can obtain the same view of the global directory structure as would a co-resident client without the need for the protocol interpreter to act as a proxy.

The interface layer is also responsible for obtaining the usernames corresponding to the supplied user tokens, and for mapping usernames to file system access records. The former is done by making requests to the identification manager (see section 4.1 below),

while the basis for the latter is the text file `$UserIDs`, described in section 5.4. This file is loaded into an internal data structure at system initialisation time, and can be reloaded on request. There is, therefore, no need to trust the client to identify itself correctly, thus avoiding a major security loophole.

# Chapter 3

# Protocols

Having described the construction and features of the file system, we now consider how the client and server communicate so as to make a (remote) file system available to the end user.

The file servers are not constrained to export only one protocol. Instead, several protocol interpreters can coexist, using a variety of transport media and protocols.[1] There is, therefore, no reason why one protocol should be chosen, to the exclusion of all others, and indeed it is likely that a number of protocols will be operated.

The interpreters enjoy no special status with respect to the local file system; instead they are all accorded equal treatment with any co-resident clients which the particular machine may have chosen to run. Indeed, they may even be less privileged than would be a co-resident client, for example by disabling "local" access authority.

## 3.1 "1976" Protocol

The protocol currently used by the majority of the APM clients is essentially identical with that of the original INTERDATA-based Filestore [4], with a few small changes to support some of the extended functionality of the new-style servers. The original Filestore was intended to support clients of a very limited intelligence; functionality was therefore kept deliberately minimal, while the balance of labour was tilted towards simplifying the clients. Although this is no longer a requirement, the need for upward compatibility between versions of the protocol has meant that many original protocol restrictions are still in force.

Except for the "pass" request the protocol offered by the new servers is fully upwards-compatible with the protocol of the old-style filestores.

Figure 3.1 on page 19 summarises the command letters and corresponding parameters, though the reader should refer to [4] for a detailed discussion of the protocol. The primary transport medium is that of ELAN "reliable" virtual circuits [6], though direct point-to-point RS-232 links are also supported. The protocol interepreters are required to maintain some state in order to deal with the notions of "current directory" and "sequential access" which should really be the client's concern.

For compatibility with the old-style filestores, the following conventions have been adopted: if a filename contains no path-separator characters (colons) the reference is

[1] Similarly, individual clients may use a number of protocols and transports to speak to their servers.

```
%constinteger FC openmod   = 'A'    { Uno: filename                : Xno
%constinteger FC rename    = 'B'    { Uno: filename, filename      :
!constinteger FC dchange   = 'C'    { Uno: filename, date          :
%constinteger FC delete    = 'D'    { Uno: filename                :
%constinteger FC permit    = 'E'    { Uno: filename, permissions   :
%constinteger FC finfo     = 'F'    { Uno: ownername, file-number  : packet
%constinteger FC general   = 'G'    { Uno: <option-dependent>      : packet
%constinteger FC uclose    = 'H'    { Xno:                         :
%constinteger FC readback  = 'I'    { Xno:                         : packet
%constinteger FC setdir    = 'J'    { Uno: ownername               :
%constinteger FC close     = 'K'    { Xno:                         :
%constinteger FC logon     = 'L'    { 0  : ownername, password     : Uno
%constinteger FC logoff    = 'M'    { Uno:                         :
%constinteger FC ninfo     = 'N'    { Uno: filename                : packet
%constinteger FC copyfile  = 'O'    { Uno: filename, filename      :
%constinteger FC pass      = 'P'    { Uno: password, old-pass      :
%constinteger FC quote     = 'Q'    { Uno: password                :
%constinteger FC readda    = 'R'    { Xno: block-number, blocks    : packets
%constinteger FC openr     = 'S'    { Uno: filename                : Xno
%constinteger FC openw     = 'T'    { Uno: filename                : Xno
%constinteger FC reset     = 'U'    { Xno: block-number            :
%constinteger FC credir    = 'V'    { Uno: new-directory-name      :
%constinteger FC writeda   = 'W'    { Xno: block-number, ...packet :
%constinteger FC readsq    = 'X'    { Xno: blocks                  : packets
%constinteger FC writesq   = 'Y'    { Xno: ...packet               :
%constinteger FC readfile  = 'Z'    { Uno: filename                : ...file
```

Figure 3.1: "1976" Filestore Protocol, as adapted

regarded as relative to the user's "current directory"; if the filename begins with a colon it is again taken as relative to the user's current directory; and if the filename contains one or more colons but does not begin with a colon then the reference is taken to be absolute (relative to the root). This approach allows the client APMs to use the same system and utilities irrespective of whether they are speaking to an old-style filestore or a new-style server.

There are a number of problems with this protocol, primarily in the areas of throughput and user identification, which make it unattractive for general long term use. In the short term it will continue to be used by old-style APM client systems, and in the medium term as a boot protocol for the APMs.

The protocol is basically half-duplex. The model of the client's operation is that it issues a request to the server and then must wait for the server to respond. Should the client attempt to send a second request before the first has been processed, the server is at liberty to defend itself, in the absence of any flow control in the transport, by throwing such a request away. Even if the request were accepted there is no guarantee that the server would respond to the two requests in the order in which they were issued—indeed disc scheduling would quite likely result in requests being reordered. Because requests do not carry any form of tag which could be echoed back by the server, the client has no way to disentangle the responses and match them up with the corresponding requests.

User identification is done by the client making a "logon" request to the server, quoting a username and password, in return for which the server issues a user identification token ("Uno"). This token, and the corresponding "Xno" issued in response to an open-file request, are relative to the transport connection path (virtual circuit or RS-232 line)—given that the tokens are small numbers, and would be easy to forge, there is really no other option. This does mean, however, that the client is unable to transfer authority to some other agent merely by telling it which Uno or Xno it should use.[2] Transparency of file location is also inhibited by the requirement to log on to each server individually in order to assert authority.

### 3.1.1 The 1976-Protocol Interpreter

The task of the "1976" protocol interpreter is generally straightforward, though rendered non-trivial by the need to maintain state information about its clients and their open files. This information is held in three tables, indexed by the ether context number of the client,[3] by "Uno" and by "Xno." A connection manager handles connect requests forwarded to it by the Port0 manager, while a pool of interpreter processes wait at a common mailbox for client requests to arrive from the ELAN driver.

## 3.2 NFS

Sun's NFS protocol [20,21] is already in use on the Department's network, linking together a number of proprietary workstations and servers. However it too has some

[2]The compute server mentioned in section 1.1 acted as a communications multiplexor for its clients, and hence was able to use any tokens issued to them. Of course, it also had to validate any tokens passed through it, as the filestore would have been unable to distinguish between the individual clients asserting them.

[3]Local "port" numbers are now allocated on a per-peer basis, rather than from a common pool.

deficiencies which suggest that its rôle should be limited rather than universal.

NFS is a stateless protocol. The most obvious consequence of this is the requirement that a file be opened and closed separately for each data transfer operation, with the attendant overheads of access rights checking (filename path-following is done by the client rather than the server, giving rise to unnecessary network traffic, with the resultant "file-handle" being saved for future direct reference to the file). There is no way to enforce file-grained locking without resorting to a separate lock manager and protocol, and even then no guarantee that other clients will respect it. There is no file token, and hence no way to delegate authority with respect to a particular file. There is not even any guarantee that a file which was accessible for one operation will still be accessible for a second: some other user may have changed its access permissions in the interim, for example, or even deleted it. Workarounds are possible for some of these problems but any solutions are unlikely to be particularly elegant.

The only generally-supported transport medium for NFS is UDP/IP [15,13], though others are possible in principle. Because this is an unreliable transport, NFS must perform its own timeout and retry operations. Network variability is confounded with the variability inherent in performing file access tasks of differing complexity, with the result that it is impossible for NFS to make any reasonable estimate of the appropriate network delay and retransmission parameters. The result is less than optimal behaviour.

NFS makes use of XDR and RPC [20] to pass requests from client to server, with the NFS definition specifying that either `AUTH_UNIX` or `AUTH_DES` authentication be used. `AUTH_UNIX` authentication is the original "flavour" of authentication used for NFS, and indeed is the only "flavour" which most servers currently support. In essence what this form of authentication means is that the client system is trusted to tell the server the user's (UNIX-style) UID. Hence, if a client can be persuaded to lie about a user's identity, which is often not difficult, then the user has effectively unlimited access to all files held by servers which trust that client system. Worse, many servers even trust potential clients to identify themselves correctly, rather than attempt any validation such as by network address. Some clients might be considered reasonably secure and hence trustworthy; many personal machines and workstations, including Sun's own workstations and the APMs, can not be. The latter, indeed, are ***required*** to run untrustworthy systems as part of many student practical exercises. This need to trust the client is a fundamental flaw which makes `AUTH_UNIX`-based NFS unsuitable for general Departmental use. Most `AUTH_UNIX` server implementations will also expect that one common set of UIDs exist across all systems; this problem could be worked around for locally-written clients and servers by the expedient of quoting and interpreting UIDs differently depending on the identities of the server and client respectively.

`AUTH_DES` is intended to block `AUTH_UNIX`'s security holes. Essentially, the client uses a public key encryption scheme to identify the user to the server; in return the server issues the client a token by which to identify the user in future transactions. Along with this user token, the client and server send DES-encrypted timestamps in order to assure each other that their conversation has not been broken into. *Sun*'s estimate is that it takes a couple of seconds to perform the public key exchange, followed by a 25% performance hit over `AUTH_UNIX` on each subsequent transaction. Although this protocol would appear to achieve the desired effect in the case of a user who uses a private workstation exclusively, impersonation is still possible in the case where there are two or more concurrent users of a system. In addition, it is now difficult to transfer

authority between client systems. UNIX-style setUID still cannot be made to work properly without trusting the client.

Additional overheads are imposed by the XDR layer.

## 3.3 Other File Access Protocols

Two other file access protocols are in use on the Department's local area networks, or could be made available:

- FTP [16] is a file transfer protocol, available as standard on BSD-based UNIX systems (amongst others). It is intended as a means of transferring entire files between systems rather than to allow access to small parts of files. Authorisation is by means of an explicit logon command, a username and password being supplied to the server for validation.

- DECnet file access [10,1] exists on the Department's VAX/VMS systems, and would be available for a number of UNIX systems. Although it allows access to remote files (almost) exactly as though they were local, clients and servers are both required to know about internal Files-11 structure. Authentication is either by means of an explicit username/password pair or by the server trusting the client to identify the user correctly. Lower level DECnet protocol modules would also need to be written for the APM systems.

NIFTP [3] ("blue book") is a widely-supported WAN protocol which has also been implemented for LANs on top of an ISO transport. It is a file transfer protocol rather than a file access protocol, however, and so it not suitable for the purpose required here.

The distributed filesystem of clustered VAX/VMS systems [9,8,19] is actually based around disc servers rather than file servers, with the activities of the local file system managers co-ordinated by a distributed lock manager. The whole edifice is, of course, tied much too closely to VMS for it to be considered for general use as a file access protocol.

A number of other standard and proprietary protocols could be considered for implementation at a later date. Of these, OSI FTAM appears the most promising and could be considered once the relevant standards are firmed up. There is, in any case, no reason why only one protocol need be used exclusively.

## 3.4 A New Protocol

# Chapter 4

# File Access Protocol

## 4.1 User Identification Protocol

## 4.2 File Access Protocol

# Chapter 5

# Administration

There are a number of files involved in the administration of the new file servers, generally manipulated by means of special utilities. A standard text editor can sometimes be used, though this is not recommended, as the correct format must be observed and inconsistencies avoided. Some form of privilege is required to modify the files.

## 5.1 $Authority

This file stores the mapping between username and encrypted password. The file itself is held as a B-tree, keyed by username. It is protected against all except its owner (`$System`) and should only be manipulated through the identification manager's message interface. Usernames in the file may be marked as privileged, giving those users the ability to modify entries in the database. Utilities exist to add and delete usernames, to modify passwords, and to modify privilege; these will shortly be merged into the user administration utility described in section 5.6. Unprivileged users can modify their own passwords, provided they correctly quote their old password in the request. The absence of this file implies that no local user identification is to be performed, effectively blocking all but "anonymous" access via the 1976-protocol interpreter.

## 5.2 $BootArea

This is the on-disc bootstrap file system, as known to the ROM loader. Files can be added, renamed and deleted using the `SASHOE` utility, which is "self-documenting." File system `bootarea` privilege is required to access this file.

## 5.3 $GroupIDs

This file is[1] used by the file system to establish textual names for group IDs. It is, essentially, a supplement to `$UserIDs` described below. The format of the file is a sequence of lines containing textual names and corresponding group IDs, one per line. Comments are introduced by a '!' character; the final line of the file is a comment containing the next group ID to be assigned.

[1] will be

```
! Username          ID      P    S    Groups
GDMR                10002 13    0    -1 -2 -3
RWT                 10003  0    0
AJS                 10004  0    0    -5
! 10010
```

Figure 5.1: Example $UserIDs file

Conventionally, user IDs are positive and group IDs negative (zero is reserved), with group names beginning with a '$' character. Note that these conventions are not enforced by the file system, but are solely for the convenience of users and the system manager.

## 5.4 $UserIDs

This file contains the file system's mapping between username and userID, privilege, supervisor, and group membership. It is loaded into an internal data structure when the file system initialises itself, and can be reloaded on request by a user with `admin` privilege. Each line of the file contains information on one user: the username is followed by the user's ID, privilege mask, supervisor and zero or more groups, and is terminated by the end of line. Lines which begin with the '!' character are treated as comments and are ignored. An example is shown in figure 5.1. Note that the first line of the file is *always ignored*; it is conventionally a comment describing the columns of the succeeding lines. Usernames can in principle contain more-or-less any printable character; in practice, however, it is a good idea to stick to alphanumerics. Conventionally, a '$' character indicates that a name is a group name rather than a username.

The file can be edited using a standard text editor, though it is best to use the utility described in section 5.6. The number in the comment on the final line is interpreted by this utility as the next ID to be allocated; as IDs are chosen from a large space there is no reason to reuse them—indeed this should be avoided as it could lead to confusion as to files' ownership.

The file is owned by `$System`, allows read and modify access to local users with `admin` privilege, and is protected against the world in general. If it is absent the file system will default to a user ID of $-1$ (`$System`) with `admin` privilege.

## 5.5 $UserData

This file does not confer any rights on users; rather it is provided for the convenience of client systems and protocol interpreters, and contains "useful" information about the users listed therein. An example is given in figure 5.2. Each line of the file contains data on one user. There are at present three columns of data: the first is the username, the second the user's home directory, and the third the user's name. The entries in this final column are enclosed in double quotes, as the number of words in a user's name is

```
! Note that this file must be sorted by Username
! Username   Home directory        User's "real" name
AJS          Staff:AJS             "Alastair Scobie"
GDMR         Staff:GDMR            "George Ross"
RWT          Staff:RWT             "Rainer Thonnes"
```

Figure 5.2: Example $UserData file

variable. Lines beginning with the '!' character are once again treated as comments. The file must be readable by the world in general, as protocol interpreters disable "local" access, and can be modified by local users with `admin` privilege. The user administration utility described in section 5.6 should be used to manipulate this file, though a text editor can also be used if required.

## 5.6 The User Administration Utility

This utility hasn't been written yet. The intention is that it will be able to manipulate `$Authority`, `$GroupIDs`, `$UserIDs` and `$UserData` in a consistent manner. Operations supported will include: adding a user or a collection of users, deleting a user, changing a user's password, creating a new group and granting a user the right to assert a group ID.

## 5.7 Bootstrapping a New File System

Bootstrapping a new file system is generally straightforward, though complicated slightly by the fact that the standard run-time support sends requests in the first instance to a local file system if one exists, in preference to a remote file system. Once the disc has been formatted, the partition and directory structure must be established before the standard file system can be used. This is done using a special stand-alone utility, `InitDisc`, booted from another server, as follows:

1. Create a cut-down boot file containing only the following: the appropriate disc process, the file system module, the B-tree module, the directory module and the `InitDisc` utility, together with those of the run-time support libraries as are absolutely necessary. Note that the user interface module should *not* be included at this point. Likewise the CLI should not be included, as the `InitDisc` utility runs in its place. Boot the minimal system.

2. You will be asked whether you want to initialise the disc or define directory entries. The only sensible reply at this point is 'I' to initialise the disc. You will then be asked for the size of the index file to be used for the file-structured partitions; this should be a reasonable estimate, as although the index file can be stretched at a later date, this will inevitably be done non-contiguously. The `InitDisc` utility is currently configured to define one large file-structured partition in addition to the disc header and primary boot areas.

3. The utility will write the disc "header," describing the layout of the partitions, and zero the index files on each of the file structured partitions. When this has completed you will be asked to reboot the machine: this is necessary in order that the internal tables are initialised properly from scratch.

4. Having rebooted the machine, you should decide whether you are happy with the disc partitioning. If you are not, return to step 2. If you are, you should now select 'D' to set up the initial directory entries.

5. The file system will initialise itself, followed by the directory module. The system will discover that there is, as yet, no root directory, and ask if you want to create a new one; you should answer "yes" to this question. It is also convenient at this point to insert the directory entry for the primary boot area: answer "yes" to the next question to do this.

6. You are now in a position to define the initial directory structure. The minimum required is to create redirectors for all the necessary system directories, so that when you do come to boot up a standard system the libraries and utilities will be found. By redirecting these references to a remote server it is possible to operate quite happily with no files on the disc whatsoever. You may also want to create external redirectors for the root directories of the remote file servers, and indeed any other redirectors you might find useful. If you intend running the server-statistics file system you should also create the redirector for "$."

7. It is convenient at this point to create the system directories which will be needed later. You will, of course, have to give them dummy names for the moment, as the "real" names will have already been defined by the redirectors created in step 6 above. They can be renamed later.

The file system should now be useable. Create a boot file listing all the "standard" files and reboot the machine. When it comes up you should find that it behaves in the "usual way."

In the absence of any administrative files, the file system defaults to a privileged state. If you are configuring a personal machine and not intending exporting any files then this is probably reasonable. If you are configuring a general-use machine or a server you should now create the necessary files. Note that once the files exist the file system will apply unprivileged defaults, so you should make sure that at least one line in each correctly identifies a privileged username. Note also that the first entry you insert in the `$Authority` database will have privilege with respect to that database, but that ***you will not be able to insert any others*** unless you quote a user token for a privileged username. Reboot the system to bring the administrative files into force.

The system is now fully configured, but utilities are still loaded from the remote server you defined in step 6 above. If you want to load libraries and utilities from the local disc you should copy them into the appropriate shadow directories, and then delete the redirectors and rename the directories. Note that the utilities you use to do this may become inaccessible under their standard names during this process. In any case, if you intend acting as a boot server you will need to have disc copies of the relevant files.

If the system is to be self-booting you will need to write the relevant files into the primary boot area, using the SASHOE utility.

# Bibliography


[1] P. R. Beck and J. A. Krycka. The DECnet-VAX product—an integrated approach to networking. *Digital Technical Journal*, 3:88–99, 1986.

[2] G. Brebner and F. King. *The Evolution of the Fred Machine.* Technical Report CSR-246-87, University of Edinburgh, Computer Science Department, 1987.

[3] Data Communication Protocols Unit. *A Network Independent File Transfer Protocol.* National Physical Laboratory, Teddington, Middlesex, February 1981.

[4] H. Dewar, V. Eachus, K. Humphry, and P. McLellan. *The Filestore.* Technical Report, University of Edinburgh, Computer Science Department, 1977. Second Revision: August 1983.

[5] H. M. Dewar and M. R. King *et al. APM Reference Manual.* Technical Report, University of Edinburgh, Computer Science Department, 1983.

[6] W. P. Enos, I. B. Hansen, and R. W. Thönnes. *Edinburgh Local Area Network.* Technical Report, University of Edinburgh, Computer Science Department, 1981.

[7] W. P. S. Enos. *Ethernet Protocols: Design and Implementation.* Master's thesis, University of Edinburgh, Computer Science Department, 1981.

[8] A. C. Goldstein. The design and implementation of a distributed file system. *Digital Technical Journal*, 5:45–55, 1987.

[9] N. P. Kronenberg, H. M. Levy, W. D. Strecker, and R. J. Merewood. The VAXcluster concept: an overview of a distributed system. *Digital Technical Journal*, 5:7–21, 1987.

[10] A. G. Lauck, D. R. Oran, and R. J. Perlman. A digital network architecture overview. *Digital Technical Journal*, 3:10–24, 1986.

[11] P. M. McLellan. *The Design of a Network Filing System.* PhD thesis, University of Edinburgh, Computer Science Department, 1981. Available as Technical Report CST-12-81.

[12] P. M. McLellan. *Shrines.* Technical Report, University of Edinburgh, Computer Science Department, 1982.

[13] J. Postel. *Internet Protocol.* RFC 791, University of Southern California, Information Sciences Institute, September 1981.

[14] J. Postel. *Transmission Control Protocol.* RFC 793, University of Southern California, Information Sciences Institute, September 1981.

[15] J. Postel. *User Datagram Protocol.* RFC 768, University of Southern California, Information Sciences Institute, August 1980.

[16] J. Postel and J. Reynolds. *File Transfer Protocol.* RFC 959, University of Southern California, Information Sciences Institute, October 1985.

[17] G. D. M. Ross. *The New Filestores.* Technical Report, University of Edinburgh, Computer Science Department, 1984.

[18] G. D. M. Ross. *Virtual Files: a Framework for Experimental Design.* PhD thesis, University of Edinburgh, Computer Science Department, 1983. Available as Technical Report CST-26-83.

[19] W. E. Snaman, Jr. and D. W. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, 5:29–44, 1987.

[20] Sun Microsystems, Inc. *Network Programming.* May 1988. Part Number: 800-1779-10.

[21] Sun Microsystems, Inc. *Security Features Guide.* May 1988. Part Number: 800-1735-10.

# Appendix A

# Internal Interfaces

## A.1 Exported File System Operations

```
%integerfnspec fsys open file  (%record(fsys access fm)%name access,
                                %integer ID, mode, compatible, req flags,
                                %integername token, size, file flags)
%integerfnspec fsys flush header %c
                               (%record(fsys access fm)%name access,
                                %integer token)
%integerfnspec fsys close file (%record(fsys access fm)%name access,
                                %integer token, flags)
%integerfnspec fsys read file block %c
                               (%record(fsys access fm)%name access,
                                %integer token, block,
                                %integername bytes,
                                %record(*)%name buffer)
%integerfnspec fsys write file block %c
                               (%record(fsys access fm)%name access,
                                %integer token, block, bytes,
                                %record(*)%name buffer)
%integerfnspec fsys truncate open file %c
                               (%record(fsys access fm)%name access,
                                %integer token, bytes)
%integerfnspec fsys create file(%record(fsys access fm)%name access,
                                %string(255) creation name,
                                %integer pn, benefactor ID, flags,
                                %integer initial allocation,
                                %integername ID)
%integerfnspec fsys bump refcount %c
                               (%record(fsys access fm)%name access,
                                %integer ID, flags, increment)
%integerfnspec fsys obtain attributes %c
                               (%record(fsys access fm)%name access,
                                %integer file ID, flags,
                                %record(attributes list fm)%name a)
%integerfnspec fsys modify attributes %c
                               (%record(fsys access fm)%name access,
                                %integer file ID, flags,
                                %record(attributes list fm)%name a)
```

```
%integerfnspec fsys exchange   (%record(fsys access fm)%name access,
                                %integer ID1, ID2, flags)
```

## A.2 File Attribute Item Codes

```
%recordformat attributes list fm(%record(attributes list fm)%name next,
                                 %integer code, status,
                                 %integer numeric,
                                 (%integer numeric2 %c
                                  %or %string(*)%name textual))

%constinteger file ID            attribute =  1
%constinteger file owner         attribute =  2  {Can be modified}
%constinteger file supervisor    attribute =  3  {Can be modified}
%constinteger owner access       attribute =  4  {Can be modified}
%constinteger local access       attribute =  5  {Can be modified}
%constinteger world access       attribute =  6  {Can be modified}
%constinteger group access       attribute =  7  {Can be modified}
%constinteger defined groups     attribute =  8
%constinteger file creator       attribute =  9
%constinteger static ID          attribute = 10  {Can be modified}
%constinteger audit ID           attribute = 11  {Can be modified}
%constinteger date created       attribute = 12
%constinteger date modified      attribute = 13
%constinteger date accessed      attribute = 14
%constinteger creation name      attribute = 15
%constinteger file size          attribute = 16
%constinteger file extents       attribute = 17
%constinteger file flags         attribute = 18  {Can be modified}

%constinteger first              attribute = file ID attribute
%constinteger last               attribute = file flags attribute

%constinteger attribute OK              =  0
%constinteger attribute unavailable     = -1
%constinteger attribute list overflow   = -2
%constinteger unrecognised attribute    = -3
```

## A.3 File Header Format

```
%constinteger no         access =   0   { File is inaccessible
%constinteger read       access =   1   { File can be read
%constinteger modify     access =   2   { File can be modified
%constinteger append     access =   4   { File can be appended to
%constinteger exchange   access =   8   { File can be (extent) exchanged
%constinteger link       access =  16   { File can be (un)linked
%constinteger control    access =  32   { File attributes can be modified
%constinteger deny       access =  64   { Invert sense of access bits

%recordformat header access fm(%integer ID, access)

%recordformat extent fm(%integer start, size)
```

```
%constinteger non extent size = 8 + 2 + 2 + 12 + 8 + %c
                                12 + 12 + 16 + 4 + 2 + 2

%constinteger extent limit = (512 - non extent size) // 8
%constinteger access table size = extent limit

%recordformat file header fm((%integer checksum, ID,
                              %short header refcount,
                              %short flags,
                              %integer owner, owner access, supervisor,
                              %integer world access, local access,
                              %integer creator, static ID, audit ID,
                              %integer created, modified, accessed,
                              %string(15) creation name,
                              %integer blocks used,
                              %short bytes in last block,
                              %short extent limit,
                              ( %record(header access fm)%array %c
                                    access(1 : access table size) %c
                             %or %record(extent fm)%array %c
                                    extent(1 : extent limit)) %c
                              ) %or %integerarray x(1 : 128))
```

## A.4 The Interface To The Directory Module

```
%recordformat path fm(%record(path fm)%name next,
                      %integer version, %string(*)%name key,
                      %string(255) text)

%integerfnspec directory lookup one %c
                              (%record(fsys access fm)%name access,
                               %integer request flags,
                               %integer directory ID,
                               %string(*)%name key,
                               %integer version,
                               %integername resulting ID,
                               %string(*)%name textual translation)
%record(path fm)%mapspec directory penultimate %c
                              (%record(fsys access fm)%name access,
                               %integer request flags,
                               %record(path fm)%name path,
                               %integername components translated,
                               %integername resulting ID,
                               %string(*)%name textual translation,
                               %integername status)
%integerfnspec directory lookup %c
                              (%record(fsys access fm)%name access,
                               %integer request flags,
                               %record(path fm)%name path,
                               %integername components translated,
                               %integername file ID, penultimate ID,
```

```
                                %string(*)%name textual translation)
%integerfnspec directory insert ID %c
                               (%record(fsys access fm)%name access,
                                %integer request flags,
                                %integer directory ID,
                                %string(*)%name inserting key,
                                %integer inserting ID)
%integerfnspec directory insert local %c
                               (%record(fsys access fm)%name access,
                                %integer request flags,
                                %integer directory ID,
                                %string(*)%name inserting key,
                                %string(*)%name inserting text)
%integerfnspec directory insert external %c
                               (%record(fsys access fm)%name access,
                                %integer request flags,
                                %integer directory ID,
                                %string(*)%name inserting key,
                                %string(*)%name inserting text)
%integerfnspec directory delete entry %c
                               (%record(fsys access fm)%name access,
                                %integer request flags,
                                %integer directory ID,
                                %string(255) key, %integer version)
%integerfnspec create directory %c
                               (%record(fsys access fm)%name access,
                                %integer request flags,
                                %record(path fm)%name path,
                                %integer desired partition,
                                %integer no inherit,
                                %integername components translated,
                                %integername parent ID,
                                %string(*)%name textual translation)
%integerfnspec directory check empty %c
                               (%record(fsys access fm)%name access,
                                %integer request flags,
                                %integer directory ID)
%record(*)%mapspec directory contents %c
                               (%record(fsys access fm)%name access,
                                %integer request flags,
                                %integer directory ID,
                                %integername status, flags)
```

## A.5 The B-Tree Package Interface

```
%integerfnspec B tree open by ID  (%record(fsys access fm)%name access,
                                   %integer request flags,
                                   %integer ID, mode,
                                   %integername access token, flags)
%integerfnspec B tree close       (%integer access token, abandon)
%integerfnspec B tree create      (%record(fsys access fm)%name access,
                                   %integer request flags,
```

```
                                    %string(31) name,
                                    %integer partition, benefactor ID,
                                    %integername ID)

%integerfnspec B tree add entry   (%integer access token,
                                    %string(255) key, %integer data)
%integerfnspec B tree find entry  (%integer access token,
                                    %string(255) key, %integername data)
%integerfnspec B tree delete entry(%integer access token,
                                    %string(255) key)
%integerfnspec B tree modify entry(%integer access token,
                                    %string(255) key, %integer data)

%predicatespec B tree empty       (%integer access token)

%integerfnspec B tree data value  (%integer access token, site,
                                    %integername size,
                                    %record(*)%name target)
%integerfnspec B tree data replace(%integer access token, site,
                                    %record(*)%name source)
%integerfnspec B tree data insert (%integer access token, size,
                                    %record(*)%name source,
                                    %integername site)
%integerfnspec B tree data delete (%integer access token, site)

%recordformat key list fm(%record(key list fm)%name next,
                          %integer value, %string(255) key)
%record(key list fm)%mapspec B tree key list(%integer access token,
                                             %integername status)
```

# Appendix B

# Internal Standard Form for File System Messages

```
! (New) File system communication message formats and codes.  This is the
! internal standard way of transmitting requests between internal & external
! file system clients and internal & external file servers.

! Filesystem mailbox names.

%conststring(31) local      file system mailbox = "FILESYSTEM:LOCAL"
%conststring(31) special    file system mailbox = "FILESYSTEM:SPECIAL"
%conststring(31) FAP        file system mailbox = "FILESYSTEM:FAP"
%conststring(31) H2         file system mailbox = "FILESYSTEM:H2"
%conststring(31) NFS        file system mailbox = "FILESYSTEM:NFS"


! Standard message format:

%recordformat fs message fm(%record(message fm) system header,
                            %integer fsys work,
                            %record(mailbox fm)%name followup mailbox,
                            %record(*)%name access token,
                            %integer tag,
                            %integer request,
                            %integer request flags, response flags,
                            %integer status, error code,
                            %string(*)%name textual response,
                            %record(*)%name filename,
                            %integer components translated,
                            %record(*)%name filename 2,
                            %integer components translated 2,
                            %integer file token,
                            %integer mode, compatible mode,
                            %integer block size,
                            %integer byte offset, byte count,
                            %bytename data buffer,
                            %record(*)%name file attributes,
                            %string(*)%name textual data,
```

```
                        %integer request specific,
                        %string(*)%name textual data2)


! Request codes are subdivided into two 16-bit parts: the high-order bits
! identify the particular filesystem for which the code is meaningful, while
! the low-order bits identify the particular operation.  Bit-15, if set,
! indicates that the request contains a filename in the "usual place" and that
! the file system module should attempt to translate it as far as possible
! even thought the actual operation is itself not understood -- this allows
! non-standard operations to name files through the standard filesystem
! mechanisms.  Note that interpreters should not expect that this bit will
! necessarily be set for standard request codes.  Not all fields need be
! used for any particular request; variant records were deliberately avoided
! in order not to pre-judge the formats of non-standard requests.

%constinteger filesystem   mask = 16_0FFF0000
%constinteger request      mask = 16_00000FFF

%constinteger interpret filename = 16_00008000

! The request flags field should always be valid: zero is the preferred default,
! meaning that there are no flags specified.  Most flag values are request-
! specific; the following, however,  are universal:
%constinteger non local flag = 16_00010000;  ! User is not co-resident

! Standard requests (filesystem 0)

%constinteger interpret filename request = 16_00008000
! This request asks that the path be examined to check the validity of the
! filename and the existence (if any) of the file.  The only guaranteed
! result is success or an appropriate error status. There is one request
! flag, viz to ask that translation stop at the penultimate filename.
%constinteger stop at penultimate = 1

%constinteger open file request = 16_00008001
! Open takes a filename, together with the requested mode and the compatible
! mode, and returns a file token and block size.  The file system module is
! expected to supply the address of a followup mailbox, to which subsequent
! reads, writes and closes will be directed, and a blocksize for the user's
! runtime support buffer management.

! Open/compatible modes are coded as follows:
%constinteger read       file mode = 16_0001
%constinteger modify     file mode = 16_0002
%constinteger append  to file mode = 16_0004
! The following additional compatible modes can be specified:
%constinteger exchange   file mode = 16_0008
%constinteger link       file mode = 16_0010
! The filesystem is not obliged to honour any compatible mode request exactly,
! but will guarantee that the compatibility constraints will be at least as
! strong as those requested.  Any unsatisfiable requests will be rejected.
```

```
! The following request flags can be specified:
%constinteger create if       flag = 16_0001
%constinteger create          flag = 16_0002
%constinteger temporary       flag = 16_0004
%constinteger unique name     flag = 16_0008
%constinteger no inherit      flag = 16_0010


%constinteger close file request = 16_00000002
! Close requires that the token obtained from the open request be supplied.
! Requests will be directed to the followup mailbox supplied in the open
! response.  There will eventually be at least two request flags, viz truncate
! and improper close.
%constinteger auto truncate flag  = 1
%constinteger improper close flag = 2


%constinteger read data request = 16_00000003
%constinteger write data request = 16_00000004
! These two require that the file token be specified, together with the byte
! offset and byte count and the address of the first byte of the buffer.
! The maximal amount to be transferred is specified for the call, the actual
! amount being returned.  Note that the two need not necessarily be the same.
! Filesystems are not obliged to accept transfers on other than the block
! boundary they specified in response to the open request.

%constinteger truncate file request = 16_00000005
! Truncate operates on open files, specified by token.  The new size is
! specified in the byte count field.  Requests will be directed to the followup
! mailbox.


%constinteger make accessible request = 16_00008006
! The file to be made accessible is specified by filename.  This operation is
! provided to "unlock" files which are inaccessible as a result of having been
! improperly closed.


%constinteger create directory request = 16_00008007
! The directory is specified by name.  There is also a local specific version
! of this request which takes a partition number in the request specific field.


%constinteger remove file request = 16_00008008
! The file to be removed is specified by filename.


%constinteger rename file request = 16_00008009
! The two files to be renamed are specified by filename and filename2.  The
! filesystem module should assume that both filenames have been verified to
! "belong to" it -- cross-filesystem rename attempts will be rejected by the
! user's runtime support at a previous stage.  The filesystem may, of course,
! apply stricter constraints at its discretion.
```

```
! The next two are in until such time as we think up a standard way to
! specify the information (i.e. probably permanently).  As different
! filesystems have different protection conventions, it is easiest to use
! a free textual form and put the onus on the file system module to parse it
! and reject anything it doesn't understand.

%constinteger textual file attributes request = 16_0000800A
! This takes a filename, and returns protection, datestamp and file size
! information as textual data, separated by one or more spaces.

%constinteger textual permit file request = 16_0000800B
! This takes a filename, and protection in textual data.  The exact form of the
! protection specifier is file system specific (unfortunately).

%constinteger last standard request = 16_000B


! Local filesystem requests (filesystem 1)

%constinteger local request = 16_00010000

%constinteger local insert local translation request = 16_00018001
%constinteger local insert external translation request = 16_00018002
! Both these specify the name by filename and the translation by textual data.
! For translations which redirect to a different filesystem, the exact form
! of the translation data depends on that filesystem.


%constinteger local get file header request = 16_00018003
! The file is specified by filename, with the buffer specified by data buffer.


%constinteger local create directory request = 16_00018004
! This is much the same as the standard create directory request, except
! that the partition number is specified in the request specific field.
! The local filesystem will default the partition to that of the parent
! directory for the standard request.


%constinteger local obtain attributes request = 16_00018005
%constinteger local modify attributes request = 16_00018006
! These two take a filename and an attributes list.  They allow access to
! attributes which are not available via the standard textual requests.


%constinteger local reload admin data request = 16_00010007
! This request asks the local filesystem manager to reload its username/user-ID
! translation database, allowing users to be added/removed and their various
! privileges and groups modified.  The request is likely to be removed again
! at some later date.  Privilege is required to issue it.
```

```
%constinteger local enquire nth entry request = 16_00018008
! This is a nasty hack, and is only in for the benefit of the old-style
! filestore-protocol interpreter.  The request asks that the name of the
! nth entry (specified in request specific) in the directory is returned
! in textual data.

%constinteger last local request = 16_0008


! New-style file access protocol remote filesystem (filesystem 2)

%constinteger fap request = 16_00020000


! Old-style 1976-protocol remote filesystem (filesystem 3)

%constinteger old style request = 16_00030000

%constinteger old logon request = 16_00030001
%constinteger old logoff request = 16_00030002
%constinteger old quote request = 16_00030003
! The old protocol requires each user explicitly to log on and off the
! filestore.  These requests allow the appropriate context to be established
! modified, and broken.

%constinteger old copy file request = 16_00038004
! This request asks that the remote filestore perform a file copy on behalf
! of the local client.  The files are specified in filename and filename2.

%constinteger old set pass request = 16_00030005
! Change the user's password, specified in textual data.

%constinteger last old request = 16_0005


! NFS filesystem (filesystem 4)

%constinteger NFS request = 16_00040000

%end %of %file
```