ON IMPLEMENTING THE ISO FILE TRANSFER, ACCESS AND MANAGEMENT PROTOCOL FOR A UNIX 4.2 BSD ENVIRONMENT

by

MEI JEAN GOH

B.Sc.(Honours), University of British Columbia, 1984

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

October 1987

4 6

Abstract

Different computer systems have their own ways of representing, storing and managing files. One approach to facilitate file transfers among systems in a heterogeneous networked environment is for each system to locally map files for transfer onto a virtual filestore (VFS). Conceptually, a virtual filestore provides a universal model for describing files and how they can be manipulated.

The ISO File Transfer, Access and Management (FTAM) protocol offers one such virtual filestore model. This thesis reports on the prototype implementation of a useful subset of the ISO FTAM protocol for the UNIX 4.2 BSD[1] file system. We call this implementation ubcFTAM. UNIX files, ordinarily regarded as unstructured, can be endowed with some internal structure thereby allowing the transfer of selective portions of a file. Furthermore, the implementation offers several file attributes not supported by UNIX.

ubcFTAM runs on several Sun Workstations[2] interconnected by a 10 Mbps Ethernet. Some performance data of ubcFTAM are also presented. This thesis also identifies

[1]UNIX is a registered trademark of American Telephone and Telegraph Bell Laboratories. BSD denotes Berkeley Standard Distribution

[2]SUN Workstation is a trademark of Sun Microsystems, Inc.

several aspects of the specifications that are ambiguous or that are inadequate, warranting further studies. Resolutions for these issues are discussed. We hope this experience will be useful to others planning to implement FTAM for UNIX systems.

Contents

List of Tables

List of Figures

Acknowledgement

I sincerely thank Dr. Samuel T. Chanson, my thesis supervisor, for his invaluable guidance, commitment and supportiveness. I am also grateful to Dr. Gerald Neufeld insightful discussions and reading the final draft. I also thank Barry Brachman for his practical and helpful suggestions as well as for proof reading the drafts. Technical assistance from Rick Sample, Frank Pronk, members of the EAN research group and the departmental office staff is much appreciated.

Financial support from the Computer Science department and from an EAN research grant is graciously acknowledged.

Many friends from campus, student residences and church deserve my gratitude for giving me moral support and various special learning experiences. I am also much indebted to my supportive family and Mr. and Mrs. Henry Gan. However, as in all things, the glory is God's.

Chapter 1

Introduction

'Tis pleasant through the loopholes of retreat
To peep at such a world; to see the stir
Of the great Babel, and not to feel the crowd.
The Winter Evening
William Cowper

Over the past decade, computer networks have become widespread linking together a diverse variety of host systems and their users. It has become common to exploit such network interconnections to distribute new software, access remote resources such as hardware and databases, operate diskless workstations in a local area network, send electronic mail and conduct computer conferences amongst other activities. Fundamental to these activities is the need for some standard means of handling information to be shared or exchanged between dissimilar systems. To meet this demand, several organisations, such as the National Bureau of Standards of the United States, the European Computer Manufacturers' Association, and vendors, such as Digital Equipment Corporation and American Telephone and Telegraph Company, have designed their own "standard" protocols for transferring units of information (known as *files*) for systems interconnected by networks. However, the protocols used by one system or network are often incompatible with those used by another, rendering interworking amongst systems more complex or less functional if not impossible. To truly achieve mutual accessibility or Open Systems Interconnection (OSI), internationally standardised protocols for networking and, in particular, for file transfer must be promulgated by some authority and be universally adopted by the various systems. This is where the International Standards Organisation (ISO) steps into the arena. The current status of ISO's work on file transfer is the File Transfer, Access and Management (FTAM) protocol as defined in the latest Draft International Standard documents [FTAM DIS 1986] put forth in July 1986[1].

Different systems have their own peculiar styles of describing the storage of data and the ways in which such data can be accessed. The ISO FTAM protocol defines a standard for transferring, accessing and managing files among open systems without having to know how file storage is implemented on different systems.

The rationale behind the concept of open systems interconnection is to minimise the amount of detailed technical information that has to be agreed upon amongst the participating systems. To this end, before protocols and procedures for file transfer, access and management can be effectively defined, a reference model to promote a universal view of files must be established. In the ISO FTAM protocol, this model, which is an abstraction of mechanisms for manipulating files, is referred to as the *Virtual Filestore* (VFS).

[1]FTAM has been passed as International Standard in June 1987 but has not been put into print yet

Using the VFS as a common model for describing files, a local mapping function can absorb differences in style and specification between different systems, thereby enabling them to interwork in mutually understood terms. Essentially, the appeal of the VFS as a common model is that this requires only N mappings instead of N $\times$ N mappings, where N is the number of distinct, disparate real systems wishing to interconnect with one another for file exchange. The VFS not only shields differences of style between similar kinds of data storage but also resolves differences of type or sophistication of these N different systems. With this approach, we also need not worry about systems protected by proprietary rights which making it difficult, if not impossible, to acquire the necessary information to do the mappings.

To realise the FTAM File Service and File Protocol, an implementation has to relate the elements of the VFS definition in the OSI environment to the resources available on a real storage system.

1.1 Objective

The principal objective of this thesis is to develop a function that maps the abstract VFS definition onto an existing file system — specifically, that in the UNIX 4.2 BSD[2] environment. It is hoped that the experience drawn from this will be valuable for implementations mapping the VFS to other kinds of host file systems.

Under our working time frame, the implementation done for this thesis is actually

[2]UNIX is a registered trademark of American Telephone and Telegraph Bell Laboratories. BSD denotes Berkeley Standard Distribution

based on the ISO FTAM second Draft Proposal [FTAM DP2 1985] rather than the later Draft International Standard. However, this is no major setback since the Draft International Standard [FTAM DIS 1986] is *not* substantially different from its predecessor, the second Draft Proposal. The differences between the two versions will be pointed out in the ensuing chapters.

1.2 Motivation

The plethora of diverse protocol standards of various organisational bodies used by different networked systems are beginning to gravitate towards a smaller, more manageable set of protocols. In fact, the trend is to adopt the ISO standards with the hope of "talking to everyone else". Since more and more systems are converging to using ISO standards for various areas of data communication, not just for file transfer, it is definitely worthwhile to develop a prototype ISO FTAM protocol implementation, especially one that can be utilised by systems running UNIX 4.2 BSD operating systems whose use have become particularly popular. Such an endeavour would interest and benefit a large community.

Moreover, although the ISO FTAM protocol was yet to be ratified at the time the project began in early 1986, the specifications were fundamentally stable judging from the kind of changes made to the second Draft Proposal to produce the current Draft International Standard over a lapse of 17 months.

Indeed, the primary motivation behind implementing and using the ISO FTAM is that the virtual filestore concept offers a common interface for storing, transferring and managing files independent of the underlying host file system. This would facilitate file transfers from our network of computer systems in the Department of Computer Science which run UNIX 4.2 BSD with other systems which already or will support the ISO FTAM protocol.

There are also secondary benefits. The local UNIX 4.2 BSD file system, through the VFS concept, stands to gain enhanced functionality.

Locally, the ISO FTAM virtual filestore can endow ordinary UNIX files (which are essentially unstructured, being simply regarded as a string of characters) with some internal structure. The granules of a structured file are commonly known as *records*. Having structured files gives the potential of supporting access, locking and protection at the record level (in addition to the file level). This, in turn, offers the possibility of using FTAM to access files structured by UNIX applications, such as files used by database applications like INGRESS.

The semantics defined by the ISO FTAM protocol for the access control file attribute is sufficiently rich for supporting an elaborate extended file access protection scheme. The protection mechanism based on this would be more comprehensive than that offered by the local UNIX file system.

1.3 Thesis Outline

The rest of the thesis is structured in the following manner :

- Chapter 2 presents the essence of the standard ISO FTAM specifications. It also cites other work related to the ISO FTAM protocol.

- Chapter 3 describes the local FTAM implementation for a system running UNIX 4.2 BSD, highlighting some of the implementation concerns and the solution techniques. The testing of the resultant implementation is also covered here.
- Chapter 4 gives a retrospective evaluation of the implementation experience. It also discusses possible future extensions.
- Chapter 5 wraps up by giving some general concluding remarks and some suggestions on areas deserving further research.

Chapter 2

The Standard Specifications

The standard specifications for FTAM are detailed in the ISO FTAM Draft International Standard documents [FTAM DIS 1986], published in July 1986, which supersedes the previous standard specifications defined by the ISO FTAM second Draft Proposal [FTAM DP2 1985] in February 1985. Apart from the clearer definitions provided by the DIS with respect to the more detailed aspects of the protocol, these two documents are fundamentally similar. In this chapter, we shall present the specifications according to the second Draft Proposal on which the implementation described in this thesis is based.[1]

The standard specifications may be described under three headings :

- the virtual filestore

  which defines the objects to be manipulated,

[1]However, the ISO FTAM terms used hereafter are those from the Draft International Standard unless there are terms belonging only to the second Draft Proposal nomenclature and not the former. Such instances and aspects where the second Draft Proposal differ from the Draft International Standard will receive special note.

- the file service

  which defines the operations that may be applied to the objects and

- the file protocol

  which defines the legal sequences in which the actions of the file service can be performed.

2.1 The Virtual Filestore

A *file* is an identifiable receptacle of information without any reference to its actual representation or physical storage and without any reference to the meaning of the data it contains, unlike a *database* where meaning is imposed on the data.

Within a file, the file contents or data are represented according to local operating system conventions (e.g., text characters can be coded in ASCII or EBCDIC). Often, the term *file* is loosely or implicitly used to refer to the data contents of the file.

Depending on the host operating system, a file may be structured into blocks of information units called *logical records*. Each logical record represents the smallest unit of information within the file that can be identified and accessed.

A *filestore* is a library or inventory of files. The files may be catalogued into nested groups (typically called *directories*).

An analogy for these terms may be drawn with terms used for books as illustrated below :

file	$\longmapsto$ book
filename	$\longmapsto$ book title
file attributes	$\longmapsto$ properties, e.g., author, number of pages, edition and date of publishing.
file contents	$\longmapsto$ manuscript
file structure	$\longmapsto$ internal organisation (logical records might correspond to chapters or words)
filestore	$\longmapsto$ library system (library catalogue + library of books + library procedures)

Along the line of this analogy, a ***virtual filestore*** corresponds to the view of a library system that includes not only books from the local library but also books from affiliated libraries available through inter-library loans. The books in the system might be identified, say, by their ISBNs (International Standard Book Numbers).

The virtual filestore provides an abstract model for describing files and filestores. Users on different systems may transfer, access and manage files on another system using some mutually understood terms in the context of this virtual filestore model. This VFS model defines the following :

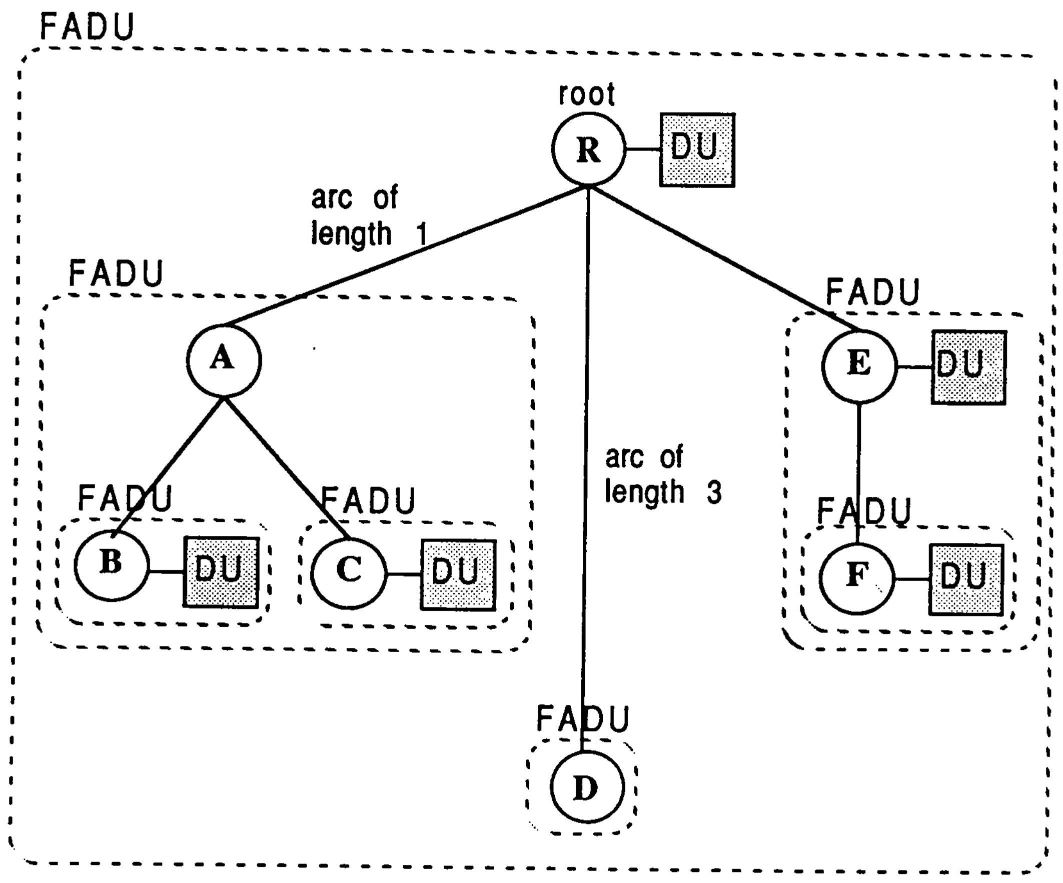
1. the internal structure of the contents of a file,
2. the properties of individual files and
3. the set of actions for manipulating the objects of the model.

Note, however, that this VFS model does not include facilities for modelling database systems although such applications could conceivably be implemented on top of it.

2.1.1 File Access Structure

The organisation of data contained within a file is described by its *file access structure* attribute. The file access structure affects the manner in which the contents of a file are accessed. A file contains one or more identifiable data units that may be related in some logical way (e.g. sequentially or hierarchically). In the virtual filestore model, complex file structures are described using a rooted tree whose nodes represent data and whose root represents the entire file. Each node in the tree corresponds to a structural unit of the real file and is assigned a level indicating its depth from the root.

Each separately accessible subtree is referred to as a *File Access Data Unit* (FADU). The FADUs may have identifiers associated with them so that they may be directly accessed. A node may or may not have data associated with it. The data associated with a node is called a *data unit* (DU). A DU may correspond to a logical record of a structured file.



FADU	=	File Access Data Unit
DU	=	Data Unit

Figure 2.1: File Access Structure – A General (*Hierarchical*) Example

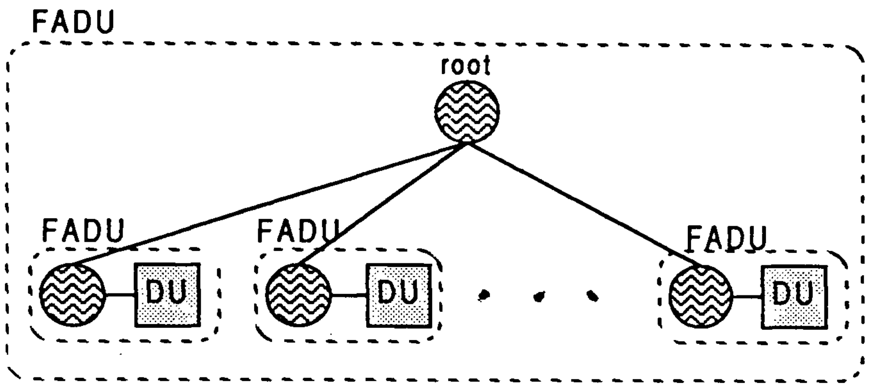
The general form of internal file organisation is described by the Hierarchical access structure. This is illustrated in Figure 2.1. Two special cases, as depicted by Figures 2.2 and 2.3 respectively, are also named :

- *Flat* access structure

  This is represented by an FADU tree with strictly two levels — the root node and its child nodes. The root node has no associated DU while each child node can have a DU.

- *Unstructured* access structure

  This is simply represented by a degenerate FADU tree, i.e., a single (root) node with which a DU may be associated.



FADU	=	File Access Data Unit
DU	=	Data Unit

Figure 2.2: Example of the *Flat* File Access Structure

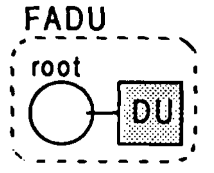


Figure 2.3: Example of the *Unstructured* File Access Structure

The ordering of the nodes in the FADU tree representing the access structure of a file is significant. Referring to Figure 2.1, (in which the nodes have been uniquely labelled purely for illustrative purposes) the ***pre-order traversal sequence*** of the nodes is :

R A B C D E F.

Access Context

For the most general case, the full hierarchical structure of a file can be transferred. This entails conveying all the structuring information and all the data in the specified FADU. In addition, the ISO FTAM protocol also permits a file to be accessed with a restricted view of its innate structure (i.e., the access structure type with which the file was originally created) by specifying different ***access contexts***. For instance, a user may wish to view a Flat file as Unstructured.

The various access contexts under which a file may be read are listed below using the terminology of the Draft International Standard. The Draft International Standard provides a larger range than the second Draft Proposal. The access contexts corresponding to those defined in the second Draft Proposal are marked by their less mnemonic names (in parentheses) as given in the second Draft Proposal. In all cases, the pre-order traversal sequence of the nodes within an FADU subtree is assumed.

- **Hierarchical All Data Units (CONTEXT 1)**

  This access context allows access to all DUs within the addressed FADU, together with the complete FADU structure description information.

- **Hierarchical No Data Units (CONTEXT 5)**

  This access context allows access to the complete FADU structure description within the addressed FADU without any DUs.

- **Flat All Data Units**

  This access context allows access to only those nodes within the addressed FADU which have DUs associated with them; both the structuring information and the DUs of such nodes are accessible.

- **Flat Single Data Unit**

  This access context allows access to both the structuring information and the DU belonging to the root node of the addressed FADU.

- **Flat One Level Data Units (CONTEXT 4)**

  This access context allows access to all the DUs in a given level of the addressed FADU, but without any FADU description information.

- **Unstructured All Data Units (CONTEXT 2)**

  This access context allows access to all DUs within the addressed FADU, but without any FADU description information.

- **Unstructured Single Data Unit (CONTEXT 3)**

  This access context allows access only to the DU associated with the root of the addressed FADU without any FADU description information.

In the second Draft Proposal, FADU is ill-defined, overlooking the transfer of structuring information necessary to support access contexts CONTEXT 1 and CONTEXT 5. However, the definition of FADU in the Draft International Standard is more precise and complete.

2.1.2 File Attributes

Various attributes are provided to describe a file. These are listed below.

- **filename**

  This attribute identifies the file, distinguishing it from the other files in the filestore.

- **permitted actions**

  This attribute indicates the range of actions that can be performed on the file.

- **access control**

  This attribute states the conditions under which access to the file would be granted. This allows specifying who (userId), upon supplying the correct passwords (passwords) can perform the requested actions(permittedActions) on the file in question. In Abstract Syntax Notation One, this attribute is represented thus[2] :

```
Condition       ::= SEQUENCE {
                      permittedAccess [0] AccessControl,
                      userId          [1] UserId OPTIONAL,
                      passwords       [2] AccessPasswords,
                      location        [3] SEAPAddress OPTIONAL}

AccessControl ::= BITSTRING {
                      read              (0),
                      insertChild       (1),
                      insertSister      (2),
                      replace           (3),
                      extend            (4),
                      erase             (5),
                      changeAttributes  (6),
                      readAttributes    (7),
                      deleteFile        (8),
                      createFile        (9)  }

AccessPasswords ::= SEQUENCE {
                      read              [0] OCTETSTRING OPTIONAL,
                      insertChild       [1] OCTETSTRING OPTIONAL,
                      insertSister      [2] OCTETSTRING OPTIONAL,
                      replace           [3] OCTETSTRING OPTIONAL,
                      extend            [4] OCTETSTRING OPTIONAL,
                      erase             [5] OCTETSTRING OPTIONAL,
                      changeAttributes  [6] OCTETSTRING OPTIONAL,
```

[2]In the Draft International Standard, the access type createFile is omitted. The reason is that these access control fields relate to a file; whereas the createFile access type describes control over FTAM usage on a user. Whether or not a user is allowed to create files or even use FTAM depends on the administrative control at that particular FTAM.

```
                         readAttributes   [7] OCTETSTRING OPTIONAL,
                         deleteFile       [8] OCTETSTRING OPTIONAL,
                         createFile       [9] OCTETSTRING OPTIONAL}

UserId          ::= GraphString

SEAPAddress     ::= EXTERNAL
```

- **storage account**

  This attribute identifies who would be responsible for accumulated file storage charges.

- **date and time of creation**

  This attribute indicates when the file was created according to the local time at the host machine of the filestore.

- **date and time of last modification**

  This attribute indicates when the contents of the file was last changed.

- **date and time of last read access**

  This attribute indicates when the contents of the file was last read.

- **identity of creator**

  This attribute indicates who created the file.

- **identity of last modifier**

  This attribute indicates who was the last to modify the file contents.

- **identity of last reader**

  This attribute indicates who was the last to read the file contents.

- **file availability**

  This attribute indicates whether delay should be expected before the file can be opened.

- **access structure type** †

  This attribute indicates the innate access structure type (either hierarchical, flat or unstructured) of the file. In fact, the value of this attribute represents the most complex access structure under which the file contents may be accessed. A file of access structure type hierarchical may be accessed as hierarchical, flat or unstructured while a file of access structure type flat may be accessed as flat or unstructured. Files of access structure type unstructured can only be accessed as unstructured.

- **presentation context** †

  This attribute indicates how the contents of the file are to be represented and interpreted. For instance, the presentation context of a file may be text and the file would be expected to be composed of ASCII characters.

† In the Draft International Standard, a new file attribute, *contents type* is defined. This attribute encompasses both the file access structure type and presentation context attributes of the second Draft Proposal, permitting a more precise and practical description of the file contents. The contents type attribute indicates the abstract data types as well as the structuring information of the contents of the file. It may take on values such as sequential text, sequential binary, unstructured binary and simple hierarchicaL

- **encryption**

  This attribute gives the name of the encryption algorithm to be used for data storage.

- **current filesize**

  This attribute indicates the size of the entire file.

- **future filesize**

  This attribute indicates the size to which the file may grow due to modification and extension.

- **legal qualifications**

  This attribute describes information about the legal status of the file and its usage.

- **private use**

  The ISO FTAM documents leave the meaning for this attribute open. This attribute is provided in case a local implementation has no other choice but to use this attribute to embody its own private information for local use only. Its use is discouraged by the standards documents.

2.1.3 File Operations

A variety of actions may be done either

- on a file as a whole or

- on a constituent portion of the file contents (i.e., on an FADU).

Operations on Entire Files

The following are operations that act on a file as a whole :

- **create file**

  This action creates a new file or selects an existing file. In the case of a new file, its attributes are established.

- **select file**

  This action establishes a particular file as the currently "selected" file on which subsequent file operations are performed until the file gets "deselected". Precisely one file may be selected at a time.

- **deselect file**

  This action relinquishes the "selected" file.

- **delete file**

  This action permanently removes the "selected" file from the VFS and thus the file is automatically deselected.

- **open file**

  This action opens the "selected" file for subsequent reading or writing. At this time, the current location within the file is at the root node of its FADU tree representation.

- **close file**

  This action closes the "selected" file in a normal fashion. At this time, the file attributes *time of last modification*, *time of last read access* and *time of creation* are updated.

- **read file attributes**

  This action interrogates the values of the requested file attributes of the "selected" file.

- **change file attributes**

  This action modifies the values of the requested file attributes of the "selected" file. The file attributes that are allowed to be explicitly modified by the user through the change file attributes operation are :

  - filename
  - access control
  - account
  - file availability
  - encryption name
  - future file size
  - legal qualifications
  - private use

Operations on File Contents

The following are operations that affect the file contents; in particular, the actions are on a per-FADU basis :

- **locate FADU**

  This action locates the specified FADU thereby setting the "current location" within the file. An FADU may be identified in one of the following ways :

  - by specifying either the 'first' (i.e., the root node) or the 'last' FADU according to the pre-order traversal sequence,
  - by relative position — 'current', 'previous' or 'next' FADU — according to the pre-order traversal sequence,
  - directly by the node name of the desired FADU,
  - by specifying a sequence of FADUs as the traversal path to trace to the desired FADU,
  - by the level number of the FADU tree. (This way is valid only with Access Context Flat One Level Data Units.)

- **read**

  This action locates and reads an FADU.

  Depending on the access context requested, the data units and structuring information transferred may pertain to the root node of the FADU or all component nodes of the entire FADU subtree.

- **insert**

  This action creates a new FADU and inserts it into the appropriate position in the file.

- **replace**

  This action replaces the contents in an existing FADU. Either the entire FADU subtree currently located is replaced or only the contents of the DU associated with its root node is replaced.

- **extend**

  This action appends data to the end of the data unit associated with the root-node of the current FADU. The extend action only applies to existing DUs.

- **erase**

  This action erases the current FADU (i.e., the entire subtree) and the current FADU is set to the first FADU in preorder traversal sequence after the erased FADU.

2.2 The File Service

The services provided to the FTAM users are modelled as *service elements* and their corresponding ***service primitives.*** For instance, the service primitives corresponding to the service element, F-CREATE, are : F-CREATE.request, F-CREATE.response, F-CREATE.indication and F-CREATE.confirm. FTAM information is conveyed between the FTAM user and the VFS via FTAM service primitives. The various services of the

File Service are outlined below.

1. **FTAM regime control**

   The relevant service elements are :

F-INITIALIZE	F-TERMINATE
F-U-ABORT	

   These elements are used to establish or terminate an FTAM connection. The F-INITIALIZE service element initiates the connection and negotiates the service class and functional units available. The user can effect orderly (or graceful) termination using F-TERMINATE or abrupt termination using F-U-ABORT.

2. **filestore management**

   This service allows management operations on the filestore. Although details for these operations have yet to be defined in the Draft International Standard. Filestore management operations might conceivably include, for example, ascertaining the total file space in the VFS consumed by the user or listing all the user's own files.

3. **file selection regime control**

   The relevant service elements are :

F-SELECT	F-DESELECT
F-CREATE	F-DELETE

The file selection regime control service allows the identification or creation of a unique file (to be known as the "selected" file) on which subsequent operations will apply, and the deselection or deletion of the selected file.

4. **file management**

   The relevant service elements are :

   F-READ-ATTRIB F-CHANGE-ATTRIB

   These allow reading and modifying the file attributes of the selected file.

5. **file open regime control**

   The relevant service elements are :

   F-OPEN F-CLOSE

   These allow opening and closing the selected file.

6. **access to file content**

   The relevant service elements are :

   F-LOCATE F-ERASE

   These allow location and erasure of certain portions of the file contents.

7. **bulk data transfer**

   The relevant service elements are :

F-READ	F-WRITE
F-DATA	F-DATA-END
F-TRANSFER-END	F-CANCEL

These facilitate the bulk transfer of a file.

8. **grouping control**

   The relevant service elements are :

F-BEGIN-GROUP	F-END-GROUP

   The grouping control service allows the initiator to delimit (i.e., indicate the start and the end of) a set of primitives to be processed and responded to as a group. However, the range and the sequences of primitives that are allowed to be sandwiched between the F-BEGIN-GROUP.request and F-END-GROUP.request are restricted to those specified in the standard document. (The valid sequnces are cited in Appendix D). These sequences respect the contextual changes when crossing regime boundaries (refer to section 2.2.1) during a dialogue. Using the grouping control service to submit certain frequently used sequences of request primitives together eliminates having to wait for the response to each corresponding request before sending the next request. The sequence of primitives sent together are responded to as a group. Total elapsed time can thus be reduced.

9. **recovery**

   The relevant service element is :

F-RECOVER

This allows the initiator to re-create the open file regime destroyed by failures signalled by F-P-ABORT, F-U-ABORT or F-CANCEL.

10. **checkpointing and restarting**

    The relevant service elements are :

    F-RESTART F-CHECK

    These allow the "sender" of data to plant marks (or checkpoints) in the flow of data for the purpose of subsequent recovery or restart. F-RESTART offers the possibility of interrupting a transfer in progress and to negotiate a point within the current bulk data transfer regime at which data can be re-transmitted immediately.

2.2.1 File Service Regimes

The file service definition includes a number of *regimes* in which the actual discourse is performed. The regimes are nested. Figure 2.4 illustrates the regime nesting and the service elements allowed in each regime. The service elements that invoke or terminate a particular regime are also indicated. The Filestore Management service elements belong to the application association regime. Each regime establishes a context under which implicit conditions prevail. For instance, while the file selection regime is in effect, all operations are implicitly performed on the file that was selected at the onset of the file selection regime.

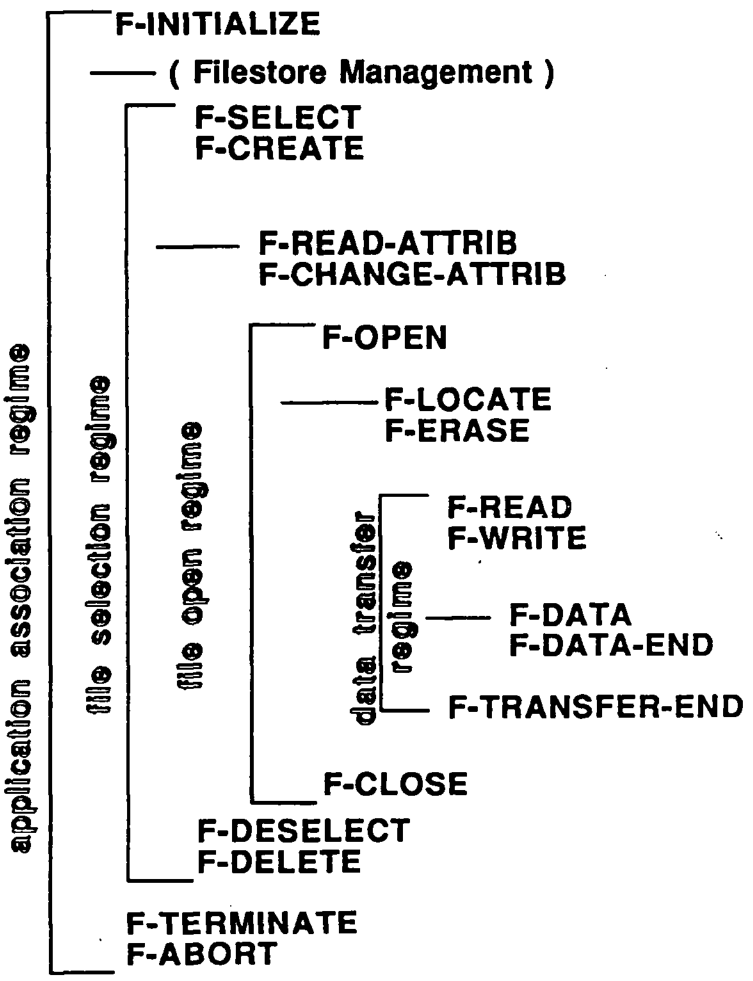


Figure 2.4: File Service Regimes

Only one instance of a regime type is allowed at any one time; in other words, regimes cannot be recursively invoked. However, for regimes within the application association regime, successive instances of a particular regime are permitted. For example, there may be a sequence of one or more data transfer regimes within a file open regime.

2.2.2 Types of File Service

Two types of file service are defined :

1. **user correctable file service**

   This allows the user, through the available file service facilities, to have direct control of error recovery and error management.

2. **reliable file service**

   For this service, the user is not informed of error detection and recovery. Instead, these functions are left to the file service provider once the user has stated its quality of service requirements.

2.3 The File Service Protocol

The file service protocol describes

- the actions to be taken when service primitives are invoked by the file service user or the underlying service provider and
- the actions to be taken as a result of events within the local system.

The FTAM protocol is connection-oriented. On a single FTAM connection, only one file may be active (or"selected") at any one time.

There are two levels of protocols corresponding to the two types of file service are defined in the preceding section. The *Basic Protocol* supports the user-correctable file service while the *Error Recovery Protocol* supports the reliable file service.

2.3.1 The Basic Protocol

The basic protocol supports the user correctable file service. It supports functions such as :

- representation of the user correctable file service primitives as a sequence of data items for transmission by the presentation service,
- concatenation, when appropriate, of the representations of logically separate service primitives and
- ensuring the progress of the protocol.

2.3.2 The Error Recovery Protocol

The error recovery protocol, using the user correctable file service, supports the reliable file service. It supports functions such as :

- management of error recovery information during the normal operation of the file service,

- restart of data transfer after interruption by errors which do not destroy the file data transfer regime,
- recovery from errors which destroy the file open or file selection regime but do not destroy the application association regime and
- recovery from errors which destroy the application association regime.

2.4 An Example of the Use of FTAM

A typical FTAM session is exemplified in Figure 2.5.

2.4.1 Transferring a Portion of a File

Viewing the internal structure of a file as an FADU tree provides the semantics that allow accessing and transferring specific portions of a file (corresponding to particular FADUs).

If a file is structured and the File Access service is supported, a selected *portion* of the file contents, rather than all the file contents, may be transferred.

If only a segment of the file is to be read, F-LOCATE is first used to specify the desired FADU before issuing the F-READ. (Refer to Figure 2.6.)

Also, if an additional "chunk" of data is to be inserted at a specific position in a file, F-LOCATE is first used to specify the desired position (corresponding to an FADU) prior to issuing the F-WRITE. (Refer to Figure 2.7.)

The specifications do not stipulate the number of F-DATA.requests that must be exchanged for each bulk transfer. Hence, this issue was locally resolved. To F-READ

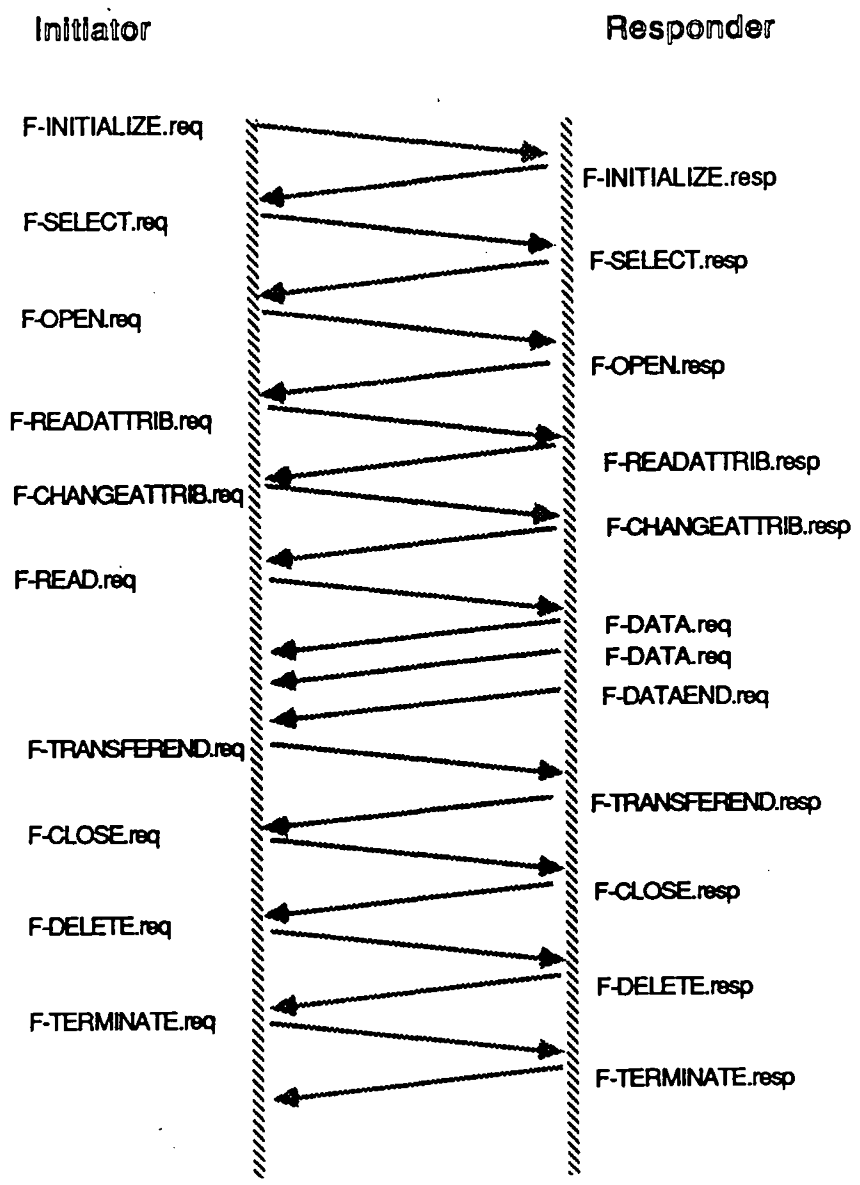


Figure 2.5: An Example of the Use of ISO FTAM Protocol

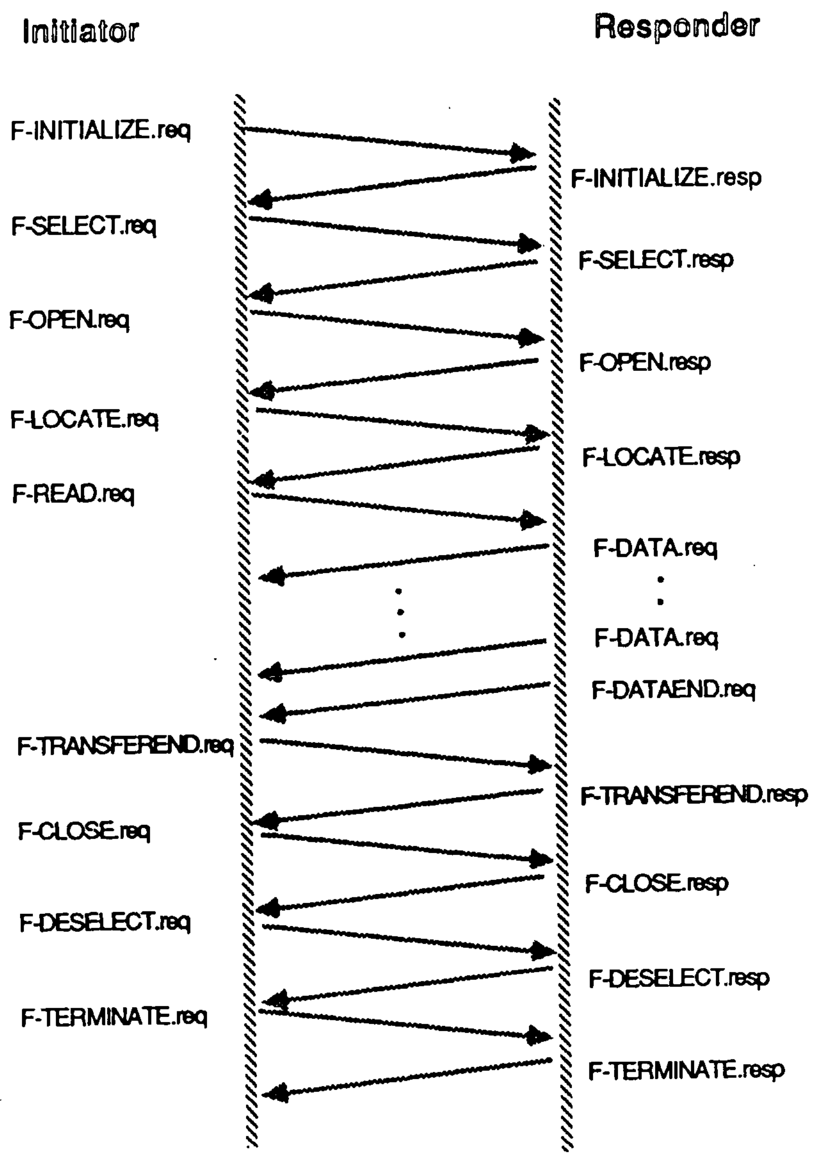


Figure 2.6: Reading a Portion of a File

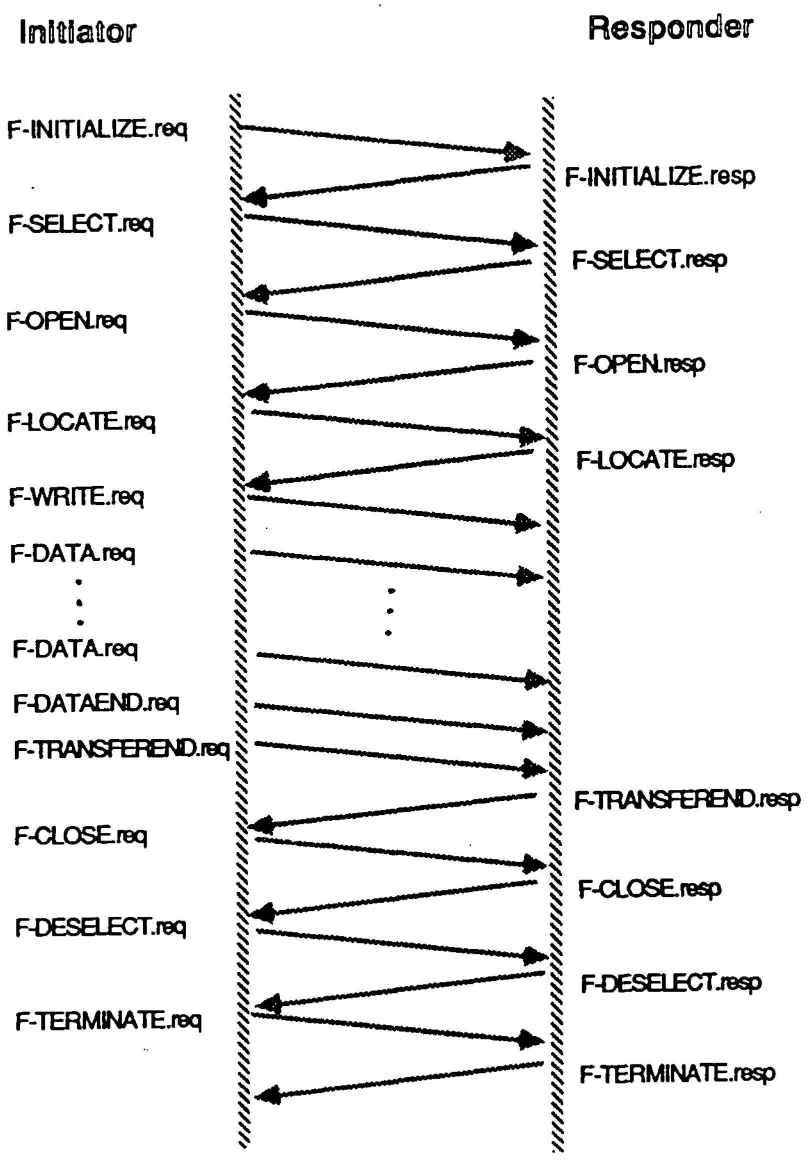


Figure 2.7: Updating a Portion of a File

an *Unstructured* file, only a single F-DATA.request is used while to F-WRITE onto an *Unstructured* file, the data of all F-DATA.requests within a bulk transfer regime constitute the same (single) FADU of the *Unstructured* file access structure. For *Flat* files, each F-DATA.request corresponds to one FADU.

2.5 Related Work

Since the release of the draft proposals for the ISO FTAM protocol, several different groups have embarked on producing the mapping functions for various existing systems onto the FTAM VFS. In lieu of a full scale implementation of the actual protocol, some have chosen to mimic the behaviour of the ISO FTAM protocol. This was the case in the Danish PAXNET project [Petersen 1985] where the semantics of the ISO FTAM service are mapped onto an existing and widely accepted interim file transfer protocol, the Network Independent File Transfer Protocol NIFTP-B(80) (otherwise dubbed the "Blue Book protocol") [NIFTP 1981]. Figure 2.8 gives an idea of how this masquerade is accomplished by PAXNET.

The advantage of this approach is that it avoids continually changing the file transfer service (and hence the application programs utilising it) during the progressive replacement of non-standard protocols with ISO standard protocols. In this respect, the PAXNET file transfer module claims to have "forward compatibility" with the ISO FTAM service standard. However, the drawback is that the ISO FTAM subset available to users may be unnecessarily limited due to the constraints imposed by the file transfer protocol being disguised, not to mention the inefficiency accruing from the

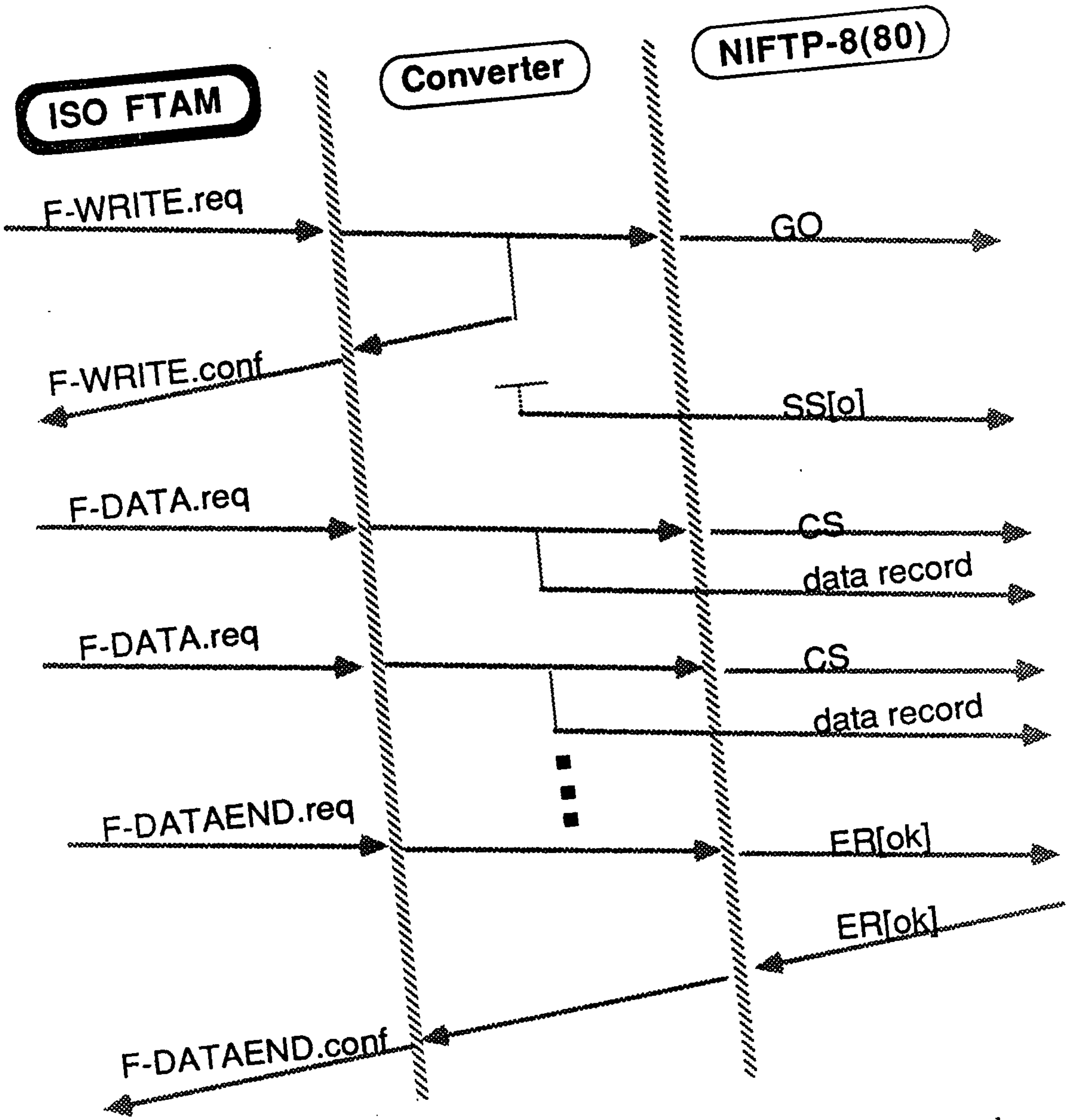


Figure 2.8: Mimicry of part of the ISO FTAM Protocol

translation process.

At the University of Montreal, a subset of FTAM has been implemented for the VMS file system on a VAX 11/750 machine [Bessette 1986]. VMS supports several different types of files including unstructured files, structured (or flat) files with records of variable or fixed length which may be indexed. Also, the record management system(RMS) on VMS can be exploited to support the flat and hierarchical FTAM access structure types. The range of file attributes supported by the VMS file system is reasonably large and covers many of those defined by the FTAM VFS.

An FTAM project for the UNIX environment is also being pursued at the University College, London as part of a larger system known as the ISO development environment (ISODE). However, we are presently unaware of any publication on this project relating to its FTAM implementation.

Chapter 3

The Implementation

At the University of British Columbia, a subset of the ISO FTAM protocol has been implemented for the Sun 3/260 and Sun 2/120 Workstations[1] running respectively Sun UNIX Release 3.2 and Sun UNIX Release 2.3, respectively, (both being compatible with UNIX 4.2 BSD). Hereafter, *ubcFTAM* would denote this local implementation whereas *FTAM* would refer to the ISO FTAM model as specified in the four-part ISO document, *Information Processing Systems – Open Systems Interconnection – File Transfer, Access and Management* [FTAM DIS 1986] and [FTAM DP2 1985]. Since ubcFTAM is a subset compliant with the ISO FTAM, unless noted otherwise, comments about FTAM would apply to ubcFTAM as well. ubcFTAM was written in C and accounts for approximately 12,000 lines of documented source code. This chapter describes the implementation details of ubcFTAM.

[1]SUN Workstation is a trademark of Sun Microsystems, Inc.

3.1 The File System under Study

It would be edifying to present a cursory overview of the UNIX file system upon which the ubcFTAM VFS is built. However, we shall focus on those features of the UNIX file system (refer to [Quarterman 1985] and [Ritchie 1978]) relevant to the ubcFTAM implementation.

From the user's point of view, three main types of files provided by UNIX :

1. *ordinary disk files*,
2. *directories* and
3. *special files.*

3.1.1 Ordinary Files

An ordinary file contains whatever information the user places into it. It may contain text or binary (object) programs. A file is simply regarded as a string of characters (or bytes) and no structuring information is expected by the system. A binary file is one that contains a binary program which is the sequence of words resembling the core memory ready for execution. Text files are usually considered to be structured in that they are composed of "lines" which, by convention, are demarcated by newline characters (the ASCII line feed characters) although the system is oblivious to such structuring. Nevertheless, a program is free to impose internal structure on files it manipulates. The structure of a file is determined by the programs utilising it and not by the system.

3.1.2 Directories

Files (be they ordinary files, directories or special files) are organised into directories giving the file system, as a whole, a tree structure. A directory, as the appellation suggests, is itself a file that contains information on how to find other files. Subdirectories may be created under a directory. A directory behaves just like an ordinary file except that it cannot be modified directly by a program but rather it must be modified through system calls. In this way, the system controls the contents of directories.

All files in the system can be located by tracing a path through a series of directories until the desired file is reached. The ultimate starting point is the *root* directory representing the root of the entire directory for the file system. One way of specifying a filename to the system is in the form of a *pathname* which is, syntactically, a sequence of directory names separated by slash ('/') symbols and ending with a filename. As an example, for the pathname **/user/jolly/project**, the system will begin to search from the root directory (denoted by the first '/') for the directory user. The directory jolly is to be searched under directory **user** and the entry project under directory jolly. From the pathname syntax alone, we cannot determine whether project is an ordinary file or a directory.

A non-directory file may also be referenced by aliases (i.e., synonyms) that may appear under possibly several different directories. This is accomplished through the directory entries for files known as *links* which behave like pointers to the actual file.

3.1.3 Special Files

An unusual feature of the UNIX file system is the special file concept. Each supported I/O (input/output) device is associated with at least one special file. To the user, a special file may be read and written just like an ordinary disk file although such read or write requests automatically activate the device associated with it. Entries for each special file conventionally reside in directory **/dev**. For instance, special files exist for each disk, each tape drive, each terminal, each printer, each communication line and for physical main memory.

At the user level, files and devices reflect uniform behaviour. The advantage of this scheme is that file and device I/O can be treated essentially in the same way. Since filenames and device names share the same syntax and meaning, a program expecting a filename as a parameter can be passed a device name as well. Furthermore, special files are subject to the same protection mechanism as ordinary files.

3.1.4 A Sample UNIX Directory Structure

Figure 3.1 shows how some directories, ordinary files and special files might be organised in a segment of a real file system.

3.1.5 File Protection

The access control scheme is relatively simple. Each user of the system is assigned a unique user identification number (UID). When a file is created, it is branded with the UID of its owner. Each file is also assigned ten protection bits. Nine of these

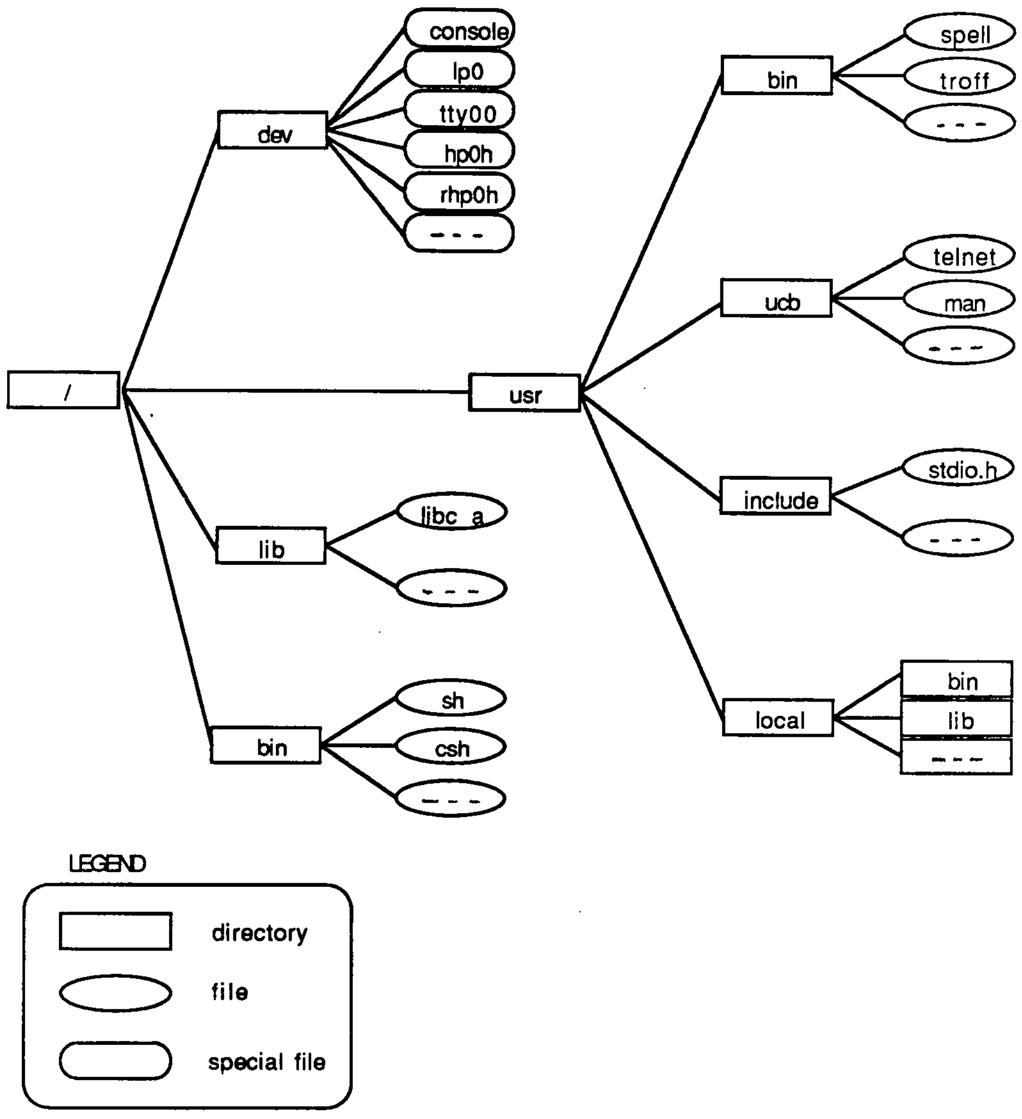


Figure 3.1: A Sample UNIX Directory Structure

independently specify read, write and execute access permissions for the owner of the file, for the user members of the file's group and for all other users ("the world at large"). Only the system administrator can create groups and specify who is to be included in each group.

One unique feature of UNIX lies in the power of the tenth bit. When this bit is on, subject to the other protection bits, the system will execute the file as a program with the privileges of the owner of the file instead of the user of the program.

3.2 The Scope of ubcFTAM

ubcFTAM is a subset of the ISO FTAM. Accordingly, the implementation of ubcFTAM, can be described in terms of the virtual filestore, the file service and the file service protocol. The subset that we have chosen to support in ubcFTAM is one that would exhibit what we consider the essential features of the ISO FTAM that are reasonably useful as a prototype implementation of the protocol. In this section, we delineate the scope of ubcFTAM, stating which aspects it does and does not support.

3.2.1 The Virtual Filestore

The VFS can take on at least two possible views, namely:

- the *"universal"* view — where the VFS knows of all files in the host's file system and
- the *"exclusive"* view — where the VFS knows only of files that have been explicitly created using the FTAM F-CREATE primitive. (These files can thus be said to

have been registered with the VFS.)

The exclusive view was chosen for ubcFTAM. It allows better control in terms of security than does the universal view. Moreover, it promotes a more uniform view of all VFS files. The choice between these two views is discussed at length in the next chapter.

The ubcFTAM VFS was developed on top of the UNIX 4.2 BSD file system. Virtual files are achieved through a mapping onto resources and services provided locally by the UNIX file system. However, UNIX notions, such as links, special files, sockets (used for inter-process communication) and directories which, albeit being referred to as "files" in UNIX terminology, do not each fit into the generic mould of a file. They bear connotations that prevent them from straightforward embodiment into the mapping since they do not have a meaningful place within the confines of the VFS model defined by the ISO FTAM protocol. For instance, special files represent I/O devices and are treated by the UNIX kernel as device interfaces; and hence their significance would be lost under the ISO FTAM VFS.

In ubcFTAM, the VFS files are placed under the VFS's own directory, /user/vfs and access to them by other users is strictly controlled.

File Attributes

Of the list of file attributes specified by the ISO FTAM documents, ubcFTAM supports the following :

- permitted actions

- access control
- last modification time
- last read access time
- creation time
- identity of creator
- identity of last modifier
- identity of last reader
- presentation context
- access structure type
- current file size

while the following file attributes are currently *not* supported:

- account
- file availability
- encryption name
- future file size
- legal qualifications
- private use

3.2.2 The File Service

ubcFTAM supports user-correctable service and offers the following the file services:

- FTAM regime control
- file selection regime control
- file management
- file open regime control
- file access
- bulk data transfer

The services currently *not* supported include:

- grouping control
- recovery
- checkpointing and restarting

3.2.3 The File Service Protocol

Presently, ubcFTAM supports a single connection at a time. It provides user-correctable service and hence supports only the user-correctable protocol. To this end, the file service would, as far as possible, supply diagnostics to indicate the source of errors but does not attempt to apply intelligent guesses to automatically rectify errors.

The task of error recovery is left entirely to the responsibility of the user or the user program.

3.3 Implementation Structure

The conceptual model for the FTAM protocol is portrayed in Figure 3.2. The model has two main entities : the *initiator* and the *responder*. These two entities may reside in two separate computers or on the same computer. The standard does, however, stipulate that an implementation claiming to support ISO FTAM be able to act as either initiator or responder.

The initiator plays the active role in an FTAM session. It allows FTAM users to submit requests to the responder. The responder's role is to service the initiator's requests. The responder accesses files through the aid of the virtual filestore.

At each of the initiator and the responder, the global behaviour of the protocol may be described by a *finite state machine* whose states correspond to distinct phases or *regimes* (as detailed in the previous chapter) during the progress of the protocol. (Refer to Appendix A for the state transition diagrams for the initiator and the responder.) The *protocol entity*[2] is responsible for keeping track of the protocol state transitions at the initiator or responder and ensuing the correct progress of the protocol.

With respect to the ISO Open Systems Interconnection seven layered reference model [OSI 1983], the ISO FTAM protocol belongs to the Application Layer (see Figure 3.3). At the lower boundary of this layer lies what is collectively called the service

[2]The term *protocol entity* is also referred to as the *protocol state machine* or *protocol machine*.

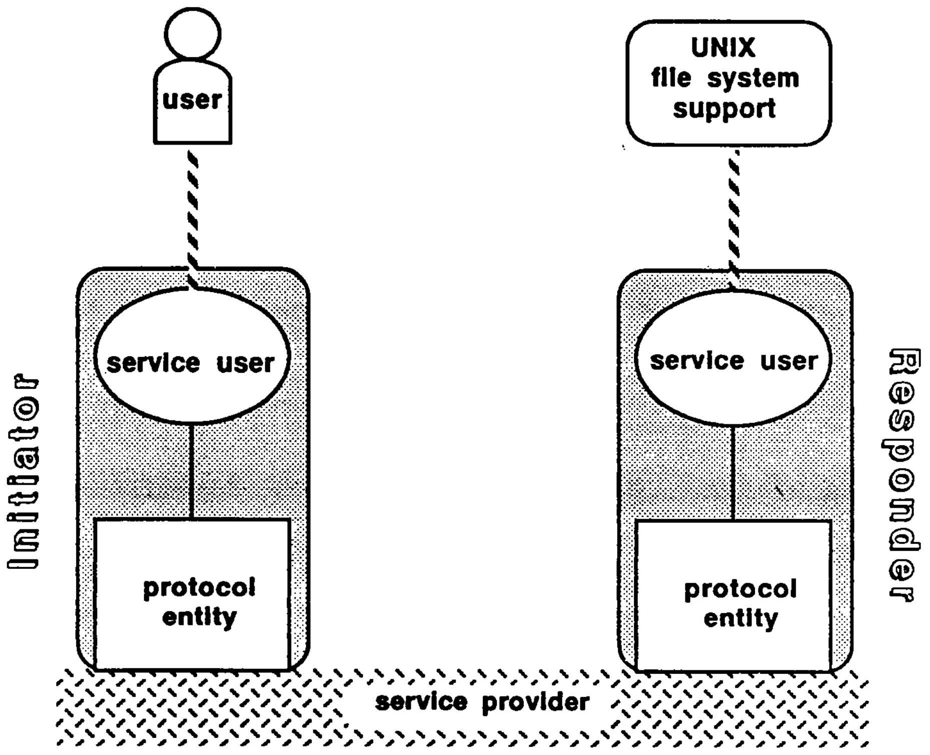


Figure 3.2: The Conceptual FTAM Model

provider; and at the upper boundary, the application user.

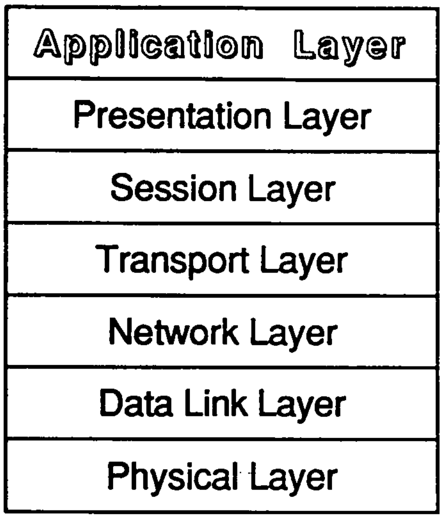


Figure 3.3: Open Systems Interconnection Reference Model

3.3.1 The Service Provider

The service protocol straddles upon what is called the *service provider* in OSI terminology. The ISO OSI reference model dictates that the immediate underlying layers supporting an application service, such as FTAM, be the Session and Presentation layers. For the Application layer protocol entities to interface with the lower support layers, the FTAM document suggests using the yet to be standardised Common Application Service Elements (CASE) [CASE 1986].

However, in the ubcFTAM implementation, the Presentation and Session layers are not distinct and are not ISO-standard. Presentation layer functions exist only insofar as the encoding and decoding of protocol data units to or from the ASN.1 notation is concerned. In fact, the topmost layer of the service provider is the Transport layer, specifically the ARPANET Transmission Control Protocol / Internet Protocol (TCP/IP). The provisional interface to this service provider makes use of TCP/IP stream sockets supported by UNIX 4.2 BSD. This Transport layer offers reliable connection service, facilitating the establishment of a connection over which the *initiator* entity and the *responder* entity communicate. (The responder may be physically resident on a different computer than the initiator.) At this stage of development, the implementation does not promise much in terms of error recovery; it merely goes as far as reporting that something has gone amiss.

3.3.2 The User Interface

An ubcFTAM user has the choice of issuing FTAM service primitives either using an interactive interface or from within a C program.

Interactive Mode

A simple interactive user interface is available. The interface prompts the interactive user with menus. Menu displays are particularly handy for the user because of the large number of FTAM service primitives along with the variety of ways of specifying their parameters.

One may wish to use the UNIX feature of redirecting input from a file where the requests are recorded rather than from the terminal. However, the present menu style user interface inevitably makes such non-interactive input rather cryptic.

Program Mode

Within a C program, the user can simply embed routine calls from a given library that invokes FTAM service primitives. An example of how to use ubcFTAM within a C program is furnished in Appendix C.

3.3.3 A Walkthrough

Let us trace what normally happens when an FTAM user makes a file service request.

1. An FTAM user (a human user or an application program) executes the FTAM protocol, submitting FTAM service request primitives.

2. The initiating service user entity then forwards the FTAM request service primitive to the initiating protocol entity.

3. The initiating protocol entity, while maintaining the protocol state for the initiator, encodes the service primitive, based on its ASN.1 definition, to build a protocol data unit (pdu) which gets transmitted to the destination responder via the service provider.

4. The service provider carries the pdu across to the destination responder (possibly across a network to another computer system).

5. The responding protocol entity, attentive to incoming pdus from the service provider, accepts the pdu, decodes it to rebuild the request service primitive duly updating the protocol state for the responder.

6. The request service primitive is forwarded to the VFS. The VFS user executes the command, possibly invoking system calls to the host file system. Following this, the VFS user conveys the result of the request by forwarding the response service primitive back to the responding protocol entity.

7. The responding protocol entity encodes the response service primitive, building a pdu to be sent via the service provider back to the initiator.

8. The initiating protocol entity, listening to the service provider, accepts the incoming pdu, decodes it and reconstructs the response primitive which it forwards to the initiating service user entity.

9. The response primitive is finally returned to the FTAM user.

Both protocol entities are continuously listening both to the service provider and their respective service user entities. They must be ready to abruptly sever the FTAM association upon receiving request to do so as signalled by an F-U-ABORT.request or F-P-ABORT.request. The connection can also be gracefully terminated after an F-TERMINATE request from the initiator has been met.

3.3.4 Functional Modules

The implementation structure is tiered according to the interfaces between the entities of the FTAM model depicted in Figure 3.2. Its organisation is captured by the functional modules listed below.

- user interface

  interfaces between the initiating entity and the FTAM user.

- FTAM service elements

  interface between the initiating protocol entity and the initiating user and also between the responding protocol entity and the VFS user.

- ASN.1 encoding-decoding module

  for both protocol entities.

- protocol state transition maintenance

  for both protocol entities. (The protocol state machine is realised through using a C language "switch" statement. Depending on the prevailing protocol state and the current event, control branches conditionally into different routines for appropriate actions and finally the protocol state is updated accordingly.)

- (lower boundary) communication interface module

  interfaces between the (initiating or responding) protocol entities and the service provider.

- host system calls

  interface between the VFS user and the UNIX file system.

- VFS housekeeping module

  includes modules for maintaining file structure, file attributes and bulk transfer.

3.4 Local Implementation Decisions

It is not the intention of the ISO FTAM to dictate the inner workings of an implementation of the protocol. Indeed, design decisions made for ubcFTAM were often shaped by the resources available, the administrative policies and the peculiarities or features within the local system. Under normal circumstances, such decisions should be transparent to the application users of ubcFTAM. The ensuing sections raise some issues encountered and qualify why certain decisions were made.

3.4.1 File Attributes

Several of the file attributes listed in the FTAM specifications are not maintained by the host UNIX filesystem. To provide for these, the attributes must be initialised and maintained by ubcFTAM.

Each file created in the virtual filestore has an affiliated "attribute file" whose existence is hidden from the FTAM user. This attribute file is used to store values of file attributes for which support provided by UNIX 4.2 BSD is unavailable, inadequate or inconsistent. On the other hand, ubcFTAM draws on the resources of the host UNIX operating system to automatically maintain certain file attributes, such as *current* *filesize.*

3.4.2 File Access Structure

The access structure types currently supported are the *unstructured* and *flat* types. The *hierarchical* access structure type is not supported.

The FADU tree structure of a file is maintained via information kept in an affiliated "structure file" that contains indices to the file (or the file contents, to be precise). To support operations pertaining to the access structure of files, a rooted binary tree, reflecting the FADU tree structure of the file, is dynamically created and maintained.

3.4.3 Semantics of Filename

According to the ISO FTAM documents, syntactically, a *filename* is composed of a sequence of character strings. As the semantics are not specified by the standard, this syntax offers a potentially rich means of interpretation.

An FTAM user may choose to use the filename components to reflect the nature of his local host file naming convention. For instance, an avid UNIX user of FTAM might use the sequence of FTAM filename components to correspond with the pathname components of a file. Alternatively, one may prefer to ignore the local filename naming convention altogether and to use one's own customised scheme. This can easily be accommodated by the flexible FTAM definition for filename. As an example, a user might use the first component to represent file ownership; the second, filename; the third, file type and the fourth, version number of the file.

To allow such latitude from the FTAM user's point of view, the VFS has to decide how to interpret and handle FTAM filenames within its capabilities.

For our implementation on UNIX, one general view would be a flat one where the VFS recognises each file simply by a single name — the concatenation of the filename components and a special character delimiting filename component boundaries. This scheme, however, is constrained by the maximum number of characters for a filename imposed by the local file system. Although this limit on UNIX 4.2BSD is very large, this scheme tends to lead to "overpopulation" under one directory and reduced efficiency in searching for a file.

Alternatively, to exploit the hierarchical directory organisation of files offered by the UNIX file system, each filename component may be mapped as a subdirectory name of the pathname representing the filename. Consider the following scheme :

> Under the VFS's private directory (**/user/vfs**) are subdirectories for each FTAM user. The files created by a user are placed under his designated directory. Suppose the filename given by the user `adam@ubc.cdn` consists of the following three filename components :
>
> `oceanography`,
>
> `project` and
>
> `version2`.
>
> Then, the VFS's directory organisation might resemble the set up shown in Figure 3.4

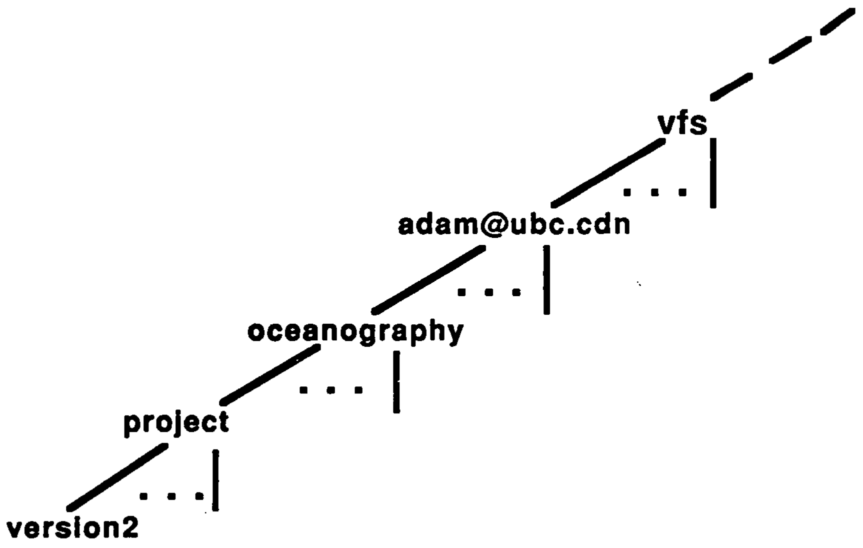


Figure 3.4: One Way to Interpret and Treat the Filename Attribute

In our implementation, a two level tree structure was adopted. The first level contains the identities of the creators. Under each of the (creator's) directory are files whose pathnames are the concatenations of the filename components. Classifying files according to their creators can be helpful for accounting purposes; it also allows different users to create files with the same names.

3.4.4 Semantics of User Identifier

In the ISO FTAM documents, *UserId* is simply a string of alphanumeric characters without any semantic directions ascribed to it, as in the case with *Filename*. *UserId* is an application-specific type defined for attributes like the identity of initiator and the identity of creator.

One possible useful interpretation for UserId is to regard it as an *Originator/Recipient Address (O/R Address)* type of the CCITT X.400 Recommendation [X400 1984]. An O/R Address is a symbolic name designed to express various electronic addresses and can be used for electronic messaging systems. An example of an O/R Address is `adam@ubc.cdn` where `cdn` is referred to as the message domain identifier; `ubc`, the sub-domain name (e.g., the name of an organisation) and `adam`, the domain-specific string (e.g., the local user identifier).

3.4.5 File Access Control and Protection

The full-fledged version of the access control scheme defined by FTAM is very comprehensive and elaborate. The access control offered by the host UNIX system is simplistic by comparison. One limitation is that the users themselves cannot freely create groups of persons and specify precisely the kind of access each person may be given for each file. Also, the types of protection in UNIX offered are restricted only to read, write and execute accesses (and their combinations), in contrast to the array of nine elements (read, insert as child, insert as sister, replace, erase, extend, change attributes, read attributes, delete file, create file[3]) ISO FTAM VFS offers. For instance, we can incorporate the access control mechanism proposed by Brachman and Chanson [Brachman 1987].

In ISO FTAM, access control is provided only on a per-file basis and not on a per-FADU (or per-record) basis. The latter case can be useful especially for database management if commitment and concurrency control were also enforced on a per-FADU basis.

3.4.6 ASN.1 Encoding

Communication between the initiator and the responder occurs through the exchange of protocol data units (pdu's). The syntax for the FTAM pdu's are defined in ISO FTAM documents in Part IV of [FTAM DP2 1985] in terms of Abstract Syntax Notation One (ASN.1) (refer to Appendix B). Detailed specification of ISO's ASN.1 notation may be found in [ASN1 1985a] and [ASN1 1985b].

A pdu has a hierarchical tree structure. During an FTAM dialogue, pdu trees are constructed with dynamically allocated storage. The pdu tree can be represented as a binary tree where each node of a pdu tree corresponds to the C structure displayed in

[3]In contrast to the 2nd Draft Proposal, the access control for create file is omitted in the Draft International Standard.

Figure 3.5. This scheme is borrowed from the software developed for the EAN X.400 messaging system [Neufeld 1985]. Here, *id* holds the protocol code for this node. To

```
typedef struct ENODE {
      long                id;
      long                length;
      unsigned char *     primitive;
      struct ENODE *      constructor;
      struct ENODE *      next;
}
```

Figure 3.5: Representing a pdu tree node in C

send a pdu out on a transmission line, the pdu tree must be "flattened" into a stream of octets (or bytes). The number of octets needed to represent this pdu is stored in *length* at the root node of the pdu tree. This *length* value is used in the reconstruction of the pdu tree from the pdu received. The *constructor* field points to the pdu subtree of this node. If this node is a leaf node in the pdu tree, then *primitive* will point to the value of the data contained in this leaf node. The *next* field points to the sibling of this node in the pdu tree (in contrast to the "subtree" of a node which represents the "descendants" of the node).

3.5 Specification Issues and Resolutions

Incompleteness and ambiguities in the ISO FTAM specification necessitated exploring possible avenues of interpretation and picking out the "most reasonable" path to follow for the local system under prevailing circumstances.

3.5.1 Defining the F-DATA Protocol Data Unit

The representation for all the protocol data units in ubcFTAM abides by the abstract syntax definitions for the ISO FTAM as given in Part 4 of [FTAM DP2 1985]. However, an adaptation had to be resorted to.

For each FTAM request service primitive, the specification defines a corresponding pdu. An exception is the F-DATA request service primitive. According to the ISO FTAM document, the F-DATA.request service primitive corresponds to the Presentation layer P-DATA service element and no F-DATA request pdu is defined. As a result of this direct correspondence between the F-DATA.request service primitive encoding and the P-DATA.request, a series of F-DATA.requests may be conveyed on a single P-DATA.request. On the other hand, unlike F-DATA, the other service primitives with their variety of more complex, application specific parameters, result in pdu's that are complex data types each of which requires description by a *compound* set of data elements, each to be transmitted by separate P-DATA.requests.

Since ubcFTAM does not interface directly with a clear Presentation layer (whose function is to support the P-DATA service and data type representation transparently), an explicit F-DATA pdu has to be defined. The parameter of the F-DATA pdu is simply a stream of octets (or bytes) without any notion of data types. In ASN.1 notation, this implementation-dependent definition is as follows :

```
        BulkdataPDU ::= CHOICE {

                        . . . .
```

```
                                        [55] IMPLICIT F-DATArequest }
        ....
        FDATArequest ::= OCTETSTRING
```

Incidentally, the context specific tag was chosen to be 55 to conform to an FTAM implementation specification by the National Bureau of Standards [NBS 1986]. Refer to Appendix B for full details of the abstract syntax definitions ubcFTAM uses.

This decision could affect agreement between two different FTAM implementations unless the underlying layers implicitly assume that the pdu following a F-READ pdu or F-WRITE pdu has to be a F-DATA pdu.

3.5.2 Presentation Context

Currently, the only values for the presentation context file attribute that ubcFTAM recognises are *text* and *binary*.

In a practical implementation, it is neither efficient nor feasible to scan the contents of the entire file to determine the presentation context of the file. Since a text file may also be considered as binary, it is up to the user of the file to "declare" the appropriate presentation context according to how the user wishes to view the file contents. Hence, verification that the presentation context of a file *does* indeed match what the initiator has stated is not performed and the presentation context value supplied by the initiator is simply taken at face value.

Conversion of the contents of a given file to a common presentation context before storing the file is not advisable. Firstly, the conversion function may not be reflexive and the exact, original file may not be recoverable. A prime example is the problematic conversion of data types like floating point numbers. Secondly, the chances are high that a user subsequently requests the file with the *same* presentation context as that with which the file was created.

3.5.3 Authentication

Authentication refers to the ability to verify that someone (the initiator, in our case) is indeed whoever is claimed. Robust authentication is not easy and is beyond the scope of this thesis. Thus, we choose to rely on the login authentication and security provided by the host UNIX operating system.

The credentials of the FTAM user can be locally verified at the system the initiator runs on before the user is permitted to initiate an FTAM association. Then, the responder has to authenticate the identity of the initiator machine. One approach would be to require that the FTAM user or the initiator machine possess the rights to log on to the UNIX system at the responder machine. In other words, the responder will only accept FTAM association requests from a predetermined clique of trusted initiators.

3.5.4 Commitment Control

A file is said to be "committed" the moment all operations requested are "committed", i.e., from this time on, all changes are guaranteed to be effected and are made permanent. Prior to the point of commitment, the changes can only be thought of as tentative, since they are subject to possible rollback (i.e., restoration to the state before the changes).

If several file service activities are to be considered as atomic (i.e., as an indivisible action), or if a file service activity is to be combined with other Application layer activities to form an atomic action, the FTAM file service can serve as a cooperating main service. In other words, the file service can cooperate with the other activities to recognise the point where the atomic action begins or ends (i.e., the point of commitment) or both points. An atomic action begun with one file service activity may end on a different activity. An FTAM responder never offers commitment if any activity is unsuccessful.

If a distributed commitment control scheme is desired, the activity performed via the file service can be integrated into it. The file service can support this integration by acting as a cooperating main service conveying the parameters and semantics of the Commitment, Concurrency and Recovery (CCR) primitives within the parameters of certain of the file service primitives. However, FTAM users may dynamically choose not to have commitment and rollback mechanisms in effect.

The following FTAM service elements can carry commitment primitives :

F-OPEN	F-CLOSE
F-SELECT	F-CREATE
F-TRANSFER-END	F-DESELECT
F-CANCEL	F-U-ABORT

The commitment elements borne by the above FTAM service elements may be one of the following :

- C-BEGIN
- C-READY
- C-PREPARE
- C-ROLLBACK
- C-RESTART
- C-REFUSE
- C-COMMIT

Refer to Annex C of Part III of [FTAM DP2 1985] for details.

In our implementation, Commitment Control is currently "not supported" in the sense that if the commitment control parameters are supplied in the service request primitives, they are ignored and the following minimal commitment control scheme is effected.

Insofar as bulk data transfer is concerned, the responder will commit a file whenever F-TRANSFER-END is executed at the end of each write phase and whenever F-ERASE is executed. This ensures that an F-READ issued after an F-WRITE within the same data transfer regime will get the up-to-date view of the file.

Commiting a file in as a single atomic action is accomplished through the aid of the UNIX 4.2 BSD system call *rename* which guarantees that the act of renaming a file is atomic. At the onset of a bulk transfer regime, if the selected file already exists, its data contents are copied to a temporary file which is updated during the data transfer. At the end of a data transfer (signified by F-TRANSFER-END), the temporary file is renamed to the original filename atomically. While this scheme tends to be inefficient, it is a simple way of ensuring atomic action.

3.5.5 Concurrency Control

Multiple, simultaneous write access to a file can cause inconsistent views of the file. To prevent this, concurrency control is necessary. A simple, non-optimal scheme is one that allows "one write, many reads". Under such a scheme, before a user is granted write access (i.e., insert, replace, erase, extend and change attributes, in the case of ISO FTAM) to a file, the user must first acquire a "lock" for it. A file is said to be "locked" when only the holder of the lock has exclusive access to the file at that time; other requests to access the locked file are not granted during this time. For ISO FTAM, read accesses encompass read and read attributes operations. Unlike write accesses, read accesses to a file do not update the file and thus would not affect the view of the file; hence, more than one simultaneous read accesses can be permitted.

Actually, the ISO FTAM leaves it to the user to specify different kinds of locks for different operations. The forms of locking for the operations — read, insert, replace, erase, extend, read attributes and change attributes — available are :

- *shared* — "I may perform the operation; so may others";
- *exclusive* — "I may perform the operation; others may not";
- *not required* — "I will not perform the operation; others may";
- *no access* — "No one may perform the operation".

Concurrency control problems are presently circumvented by restricting the number of FTAM associations the responding protocol machine will accept to one. In ubcFTAM, all files under the jurisdiction of the VFS belong to the VFS's own directory (and so can be appropriately protected from other users even with the regular UNIX file protection scheme). Also, for each connection, at any time, there can be exactly one "selected" file on which file operations may be performed. Therefore, it is possible for the VFS to orchestrate concurrent accesses to the "selected" file.

Chapter 4

Evaluation

ubcFTAM which is a subset of the ISO FTAM protocol has been implemented. The implementation can be run as either the initiator or the responder. Currently, ubcFTAM runs on two departmental computers, a Sun 2/120 and a Sun 3/260 (which is approximately five times as fast as the former). Both computers support UNIX file systems compatible with UNIX 4.2 BSD and are linked by a 10 Mbps Ethernet.

In this chapter, we give some general retrospective remarks on the implementation of ubcFTAM as well as on the standard specifications on which ubcFTAM is based. Some empirical results of its performance is also presented.

4.1 Remarks on the Implementation

4.1.1 The VFS Views

To recapitulate, the VFS can assume at least two views, namely,

- the **universal** view — where the VFS knows of all files in the host's file system and

- the **exclusive** view — where the VFS knows only of files that have been explicitly created using the FTAM file creation service.

Here, we present the issues and arguments for these two views. In general, an argument that supports one view disputes the other view, and vice versa.

- **accessibility of files**

  Exclusive View

  Where the VFS takes on the exclusive view, an FTAM user cannot normally or easily access a remote file that is resident on the VFS's host computer but not recognised by the VFS (i.e., the requested file was not previously created by the FTAM file creation service).

  Universal View

  Any file on the VFS's host file system is known to the VFS regardless of whether or not it had been explicitly created by the FTAM file creation service. Hence, any file can be directly accessed. When a file is selected to be read, the universal view VFS dynamically maps the requested file onto a virtual file and transfers it to the host computer of the FTAM user and then later removes the virtual file. These actions at the VFS are transparent to the FTAM user.

- **isolated FTAM effects and autonomous control**

  Exclusive View

  Actions on files initiated by FTAM are confined to those files under the dominion of the VFS and hence other (non-FTAM) users of the system are insulated from any inexplicable side-effects due to the FTAM program.

Also, accounting of space usage is simplified since all files are owned by the VFS.

Universal View

Since the universal view VFS knows of all files on its host system, contention can occur due to a file being requested through FTAM and also by some non-FTAM process on the host computer at the same time. Careful concurrency control and priority policy that might affect even non-FTAM users on the host computer become necessary.

- **user authentication**

  Exclusive View

  An advantage of the VFS owning the files known to FTAM is that users without login accounts on the host machine can create files. Such users who are known to FTAM can utilise FTAM primitives to create (subject to the VFS access control, of course) and manipulate files. The VFS host machine need not create new login accounts for remote users.

  Universal View

  Remote FTAM users must have login accounts on the host machine so that the local operating system can identify the FTAM user before allowing local files not created by FTAM to be accessed.

- **support features not provided by local file system**

  Exclusive View

When augmenting the host UNIX file system with features (such as file access structure and additional file attributes) that it does not normally support, the exclusive view VFS can promote a uniform view of all files since all the VFS files would have similar support.

Universal View

With the universal view VFS, to support those features not automatically supported by the host UNIX file system, files that cannot be fully described in terms of the host file system require affiliated support files (see sections 3.4.1 and 3.4.2) for the VFS to maintain the extra features. Two problems may arise. First, only some files (namely, those intended for VFS use) would have the affiliated support files; and so, in this respect, the view of all files known to the universal VFS is not consistent. Second, when a file is deleted by a non-FTAM process, these VFS-specific affiliated files may be left "dangling" without any files referencing them. This necessitates periodic garbage collection on affiliated support files unless the UNIX kernel is modified to delete any affiliated support files when a file is destroyed.

- **extended file access control and security**

  Exclusive View

  Since all files known to the VFS under the exclusive view are within its direct control, the VFS can exert its own level of access control (possibly more refined than what the host file system can offer) over all the VFS files in the best interests of the FTAM users.

Universal View

In contrast, on a file system such as UNIX that offers only a simple form of access control, the universal view VFS may have to directly use whatever file access control the host file system can offer. However, this can seriously stifle the sophistication of the access control mechanism that an ISO FTAM VFS can be expected to support. This also aggravates the problem of protecting files on the host computer from unscrupulous or unauthorised users. It is unfair to the users of the host computer who are unwary of the existence of FTAM activities if an FTAM user (possibly from a remote system) is permitted to access all the local files whose access control mode have been set to allow access to everyone. Authentication of FTAM users becomes a real issue. It would have to be stringently enforced and this is a non-trivial task.

The View Adopted

Both the universal and the exclusive views are acceptable since both conform to the specifications. The implementor has the liberty to decide which view to adopt. From one standpoint, the clash between these two views essentially hovers around the conflicting demands on time and space. In other respects, the resolution of this polemical contest involves weighing the costs and benefits of achieving certain goals like ease of use for the FTAM user, minimising the complexity (and thus the possibility of erroneous interpretation) of implementation and conceptual consistency. Another criterion is to consider the main application of the FTAM users — whether FTAM is to be used primarily for archival purposes or for frequent exchange and updates of files. The exclusive view tends to favour the former whereas the universal view, the latter. In light of the above considerations, our decision was to adopt the exclusive view for the ubcFTAM VFS. The key reasons include the following — conceptual consistency in the support for features not supported by the host system, security of the host system and the insulation between FTAM and non-FTAM users on the local system. Clearly, tradeoffs in this decision are inevitable and justification cannot be easily quantified. Perhaps, a compromise between these two views can be worked out.

4.1.2 Structured Files

The philosophy behind the design of UNIX was simplicity and modularity. The premise is that when more complex structures or programs are required, these can be developed from exploiting the simple, basic building blocks UNIX provides. This philosophy is reflected by the fact that UNIX only supports one type of file — all UNIX files are simply regarded as a string of characters without any internal structure.

As a result, mapping UNIX files onto the VFS file attributes is trivial. All UNIX files can simply be regarded as having an 'Unstructured' Access Structure.

However, most UNIX users tend to regard a file containing only text characters to be structured as a sequence of "lines" of text (i.e., a sequential file with variable-length records of text characters). In such files, the linefeed character may have the special role of marking the end of a line (or a record). In order to convey this commonly accepted implicit structuring information, such files should be mapped onto a VFS file as having Access Structure 'Flat' where the DU of each child node of the FADU tree corresponds to a line (or a record).

On the other hand, the simplicity of UNIX takes its toll on the overhead necessary to support structured files. Since UNIX does not provide any file record management facilities, these had to be implemented.

4.2 Remarks on the Standard Specifications

4.2.1 File Identification

The standard does not attach any semantics to the filename components. If a file is to be selected solely on the basis of the string of filename components, qualifying the file with its owner is not straightforward.

Taking the UNIX file system as an example, if the pathname identifying a file begins with a string with syntax, /user/xxx, then xxx would by convention indicate that the owner of the file is someone whose user identity is xxx. Thus, the file naming convention on UNIX can be said to have semantic bearings since it conveys implicit information about the file.

In support of universality of the VFS as a common model, the filename should not implicitly reveal properties of the file. If desired, such information can be conveyed by the other file attributes. Since the FTAM documents do not impose any semantics on *filename*, we are compelled to assume that a filename identifies a file belonging to the FTAM user requesting it. It is not unusual to allow different users to own or create files with the same syntactic names and yet allow such files to be uniquely identified network-wide. Hence, under this file identification scheme, there must be a way for an FTAM user to qualify the filename with the ownership of the file. This can be done by having the VFS use both *filename* and *fileAttributes* to identify a file. The *identityOfCreator* field of *fileAttributes* can be used to specify the owner of the file.

The ASN.1 definition in the second Draft Proposal for F-SELECT.request allows both filename and fileAttributes to be passed as parameters and this file identification scheme can be facilitated. However, in the Draft International Standard, although the ASN.1 definition for F-SELECT.request syntactically allows both *filename* and other *fileAttributes* to be specified, there is an additional clause stating that only the filename field is used.

Greater flexibility in file identification may be desired. For instance, a user might wish to be able to pinpoint a file, not merely by its filename, but rather by supplying a list of file attribute values describing and characterising a particular file. Hence, to accommodate generality, the standard specifications should allow the possibility of selecting a file based also on the given values of the attributes of the file apart from the filename.

4.2.2 Semantics for Structured Files

The internal structure of a file can be described in terms of the ordering of records within the file and how these records can be directly accessed. The common file organisation styles include :

- *sequential* : sequentially ordered records, sequentially accessible;
- *relative* : sequentially ordered records, accessible by position;
- *random* : sequentially ordered records, accessible by key;
- *indexed* : key-ordered records, accessible by key.

The hierarchical FADU tree representation for the file access structure is sufficient to describe all the organisation styles mentioned above except for indexed organisation. When mapping indexed files to the virtual file format, the primary key (or index) can correspond to the FADU identifier; however, the FADU structure does not offer facilities to represent the other (secondary) keys, thus deterring traversal of the file based on these keys.

Furthermore, the protocol should allow the negotiation of whether or not direct accessibility of records by their keys is supported. It is unrealistic to assume that all systems support access techniques other than sequential.

4.3 Performance

To give some idea of the performance of ubcFTAM, a number of test runs and time measurements were made. The following apply to the test runs.

- The Responder was run on the Sun 3/260 and the Initiator on the Sun 2/120.
- To observe the behaviour of the implementation, the current value of the system clock was recorded at certain strategic points of interest. Time measurements

using the system clock are only accurate to the second.

- The total elapsed time of interest is measured at the Initiator's side from the time the F-INITIALIZE.request primitive is sent until the time the F-TERMINATE.response primitive is received. In other words, the total elapsed time corresponds to the length of time for one Association regime.

In each run, one main task is performed. The task may be the reading or writing of a file with access structure *Unstructured* or *Flat*. In general, each run involves the following steps : establishing an FTAM association, creating or selecting a file, opening the file, transferring the file (by reading or writing), ending the transfer, closing the file, deselecting the file and finally relinquishing the association. The test runs may be categorised according to the main task performed as follows :

- *write to Unstructured file* :

  The main task is to create a file with access structure *Unstructured* and write data into the file.

- *write to Flat file* :

  The main task is to create a file with access structure *Flat* and sequentially write records to the file. Actually, in our case, a utility routine is used to read lines (terminated by newline characters) as records from a local UNIX file to be written to the VFS file via FTAM.

- *read entire file* :

The main task is to select a file already existent at the VFS and read the entire file with access context *Unstructured All Data Units.* Note that the requested file can be one which had been created with access structure *Unstructured* or *Flat.* In the case of *Flat* files, the records (leaf FADU nodes) are returned one by one in pre-order traversal sequence.

The time measurements from several runs of each case were averaged. The empirical results are tabulated in Table 4.1 and Table 4.2. While these statistics cannot be taken as absolute, they provide some indication of the relative performance of the runs. We observed that the performance varied with the load on the system and the network. So, attempts were made to execute the different runs under approximately similar light system workload conditions.

# data bytes transferred	Total Elapsed Time (seconds)	
	Write Unstructured file	Read entire Unstructured file
3,000	8.3	8.0
5,000	9.3	9.0
10,000	12.7	13.0
30,000	25.0	43.3
50,000	37.7	90.5

Table 4.1: Transfer of *Unstructured* files

# records transferred	# data bytes transferred	Total Elapsed Time (seconds)	
		Write Flat file	Read entire Flat file
60	3,000	62.3	22.5
106	5,000	109.0	33.0
300	14,175	419.0	83.5
400	18,896	659.5	108.5
1600	50,000	6517.5	415.0

Table 4.2: Transfer of *Flat* files

4.3.1 Observations

Transferring an entire *Flat* file evidently takes longer than transferring an entire *Unstructured* file containing the same number of data bytes. The main factor influencing the total elapsed time in the case of *Flat* files is the number of records (i.e., FADU nodes) whereas that for *Unstructured* files is the number of data bytes.

Writing to *Flat* files takes significantly longer than reading them. This is partly attributed to the larger number of service primitives that have to be exchanged in the case of writing. For each record, the following sequence of five service primitives must be exchanged — F-WRITE.request, F-DATA.request, F-DATAEND.request, F-TRANSFEREND.request and F-TRANSFEREND.response. For reading the entire *Flat* file, the sequence of service primitives involved are — F-READ.request, F-DATA.request, . . . [as many as there are records] . . ., F-DATAEND.request, F-TRANSFEREND.request and F-TRANSFEREND.response. Furthermore, when a *Flat* file is being written, the present simple commitment control scheme requires that the VFS commit the file each time after a record is serviced (at the time of F-TRANSFEREND.request). This certainly lengthens the overhead processing time at the VFS. A major factor contributing to this overhead is the scheme used to commit files in an atomic fashion, as described in Section 3.5.4.

The experiment *Write to Flat file* was repeated such that the file is committed only after *all* the records have been transferred (i.e., committed only at F-CLOSE) rather than after every record. As anticipated, the time elapsed were shorter and closer to being proportionate to the number of records transferred. The results are shown in Table 4.3. Of course, a price must be paid for committing a file only after all records have been transferred — should the transfer be prematurely halted during an open regime, whatever records transferred thus far would be lost.

As pointed out in Section 2.4.1, when an Unstructured file is read, the initiator receives the contents in a single F-DATA.request whereas the contents of an Unstructured file for an F-WRITE.request are transmitted in fragments (of size 1K in the above test runs). From Table 4.1, for an *Unstructured* file larger than a certain size (5K in our test runs), reading the entire file takes longer than writing it. This occurs partly because the F-DATA.request primitive carrying the single DU of an *Unstructured* file gets so large that it adversely affects the time for fragmenting and re-assembling the large pdu before sending and on receiving it. In fact, according to the table, the turning point occurs between transferring 5K and 10K bytes; and this reflects the 8K optimal

# records transferred	# data bytes transferred	Write Flat File	
		Total Elapsed Time (seconds)	
		Commit after each record transferred	Commit after all records transferred
60	3,000	62.3	32.2
106	5,000	109.0	50.4
300	14,175	419.0	130.0
400	18,896	659.5	179.0
1600	50,000	6517.5	649.3

Table 4.3: Write *Flat* files – comparing different commit schemes

packet size used by the underlying TCP/IP software. When the file to be transferred is smaller than the turning point size, the exchange of several F-DATA.requests takes longer than that of a single F-DATA.request (of the same total size). This can be attributed to the fact that the lower layer software waits to accumulate the smaller units until some threshold size is reached before sending them as a whole. Correspondingly, the receiving entity has to appropriately break down the packet received.

Furthermore, the data flow between the initiator and the responder is changed more frequently during a read data transfer regime than during a write data transfer regime. For writing, data flow starts with the initiator sending an F-WRITE.request, followed by a series of F-DATA.requests, an F-DATAEND.request and finally a F-TRANSFEREND.request before the data flow switches when the responder sends the

F-TRANSFEREND.response. On the other hand, for reading, data flow starts with the initiator sending an F-READ.request, switches to the responder which sends F-DATA.requests and an F.DATAEND.request, then switches again to the initiator which sends an F-TRANSFEREND.request and finally switches back to the responder which sends the F-TRANSFEREND.response. In other words, data flow alternates four times for reading in contrast to two times for writing. This can become a contributing factor to longer time delays for reading especially when the network is particularly slow.

In general, using FTAM takes longer than using the UNIX remote copy (*rcp*) utility. On an average, *rcp* takes only up to 6 seconds to transfer a 50,000 byte (unstructured) file between the two Sun machines. This is expected since FTAM attempts to accommodate a variety of file systems and so more parameters have to be exchanged and negotiated. On the other hand, *rcp* is meant to be used for file transfers only among UNIX file systems. Moreover, part of the delay was due to using FTAM without the grouping control service support and the "stop and wait" nature of the protocol. Most of the time, the Initiator has to wait to receive a response primitive before it can send the next request primitive. Also, the FTAM protocol must proceed strictly in stages as dictated by the regimes. However, if the grouping control service were supported, a series of request primitives may be submitted together to be serviced and responded to as a group.

4.4 Improvements

Reflecting on the present implementation, there is room for improvement in several aspects.

The current scheme to support structured files is simple and requires the entire FADU to be resident in memory for dynamic manipulation. This demand for memory is exacerbated when several concurrent FTAM associations are entertaining files represented by large, complex hierarchical FADU trees. A better and more efficient algorithm for record management requiring less dynamic memory space can be sought.

The algorithm used to maintain the data contents of a file is simple. Every time a file is committed, if any FADU node of the structure tree has been modified, the entire file containing the file contents is re-written. This method can be expensive especially when the frequency of file commitment is high. Nonetheless, it has the advantage of not creating "holes" in the file that stores the file contents per se due to records being replaced by shorter ones. Better algorithms for maintaining the file contents can be sought and tested for efficiency under general FTAM usage.

With the aid of the timing results reflecting the behaviour of the implementation under common usage, modules can be identified for code optimisation.

4.5 Extensions

The current ubcFTAM implementation is a subset of the ISO FTAM protocol. This subset can be extended to offer more functionality and provide more services. As the next phase of development, the implementation can be progressively enhanced to support the following useful features as well.

- standard Presentation layer interface

The interface to the service provider can be modified to utilise the standard ISO Presentation layer services using Presentation layer service primitives or CASE.

- hierarchical file access structure

  The VFS can be extended to support hierarchically structured files. The present representation of the internal structure of files is adequate for supporting the hierarchical file access structure. However, some special treatment is needed for the reading or writing of a hierarchically structured file, especially when it is accessed with access contexts *Hierarchical All Data Units* or *Hierarchical No Data Units.*

- grouping control service

  This would permit the concatenation of frequently used sequences of service primitives together to be processed and responded to as a group. The FTAM user may find this feature attractive since delays can be reduced, especially when the time lapse due to communication between the initiator's host and the responder's host is long. For service elements such as F-TRANSFEREND, F-CLOSE and F-DESELECT for which failure is unlikely or failure would not cause serious repercussions, the FTAM user probably would not care to wait to receive every single response primitive. State transition maintenance would have to be modified to take into consideration the fact that sequences of (grouped) primitives are regarded as a unit.

- multiple associations

The responder can be upgraded to accept more than one FTAM associations at a time. Consequently, more attention must be paid to concurrency control since a particular file in the VFS may be accessed simultaneously by separate FTAM associations. Users on different, concurrent FTAM associations must have a consistent view of the same file. Depending on the concurrency control algorithm, some delays may arise due to files being temporarily unavailable if they are locked.

- checkpointing, recovery and restarting services

  For a more reliable file service, the implementation should also allow checkpoints to be interspersed within the data being transferred to facilitate restarting the transfer of data as well as recovery from failure during data transfer. To support recovery and restarting, buffer management schemes are needed.

- commitment, concurrency and recovery support

  This feature is necessary if the file service activity is to be used in conjunction with other Application layer activities to form an atomic action. Checkpointing, recovery and restarting services are necessary to support commitment and rollback mechanisms.

Chapter 5

Conclusion

This thesis aimed at developing a prototype system for a UNIX 4.2 BSD system to allow files to be transferred, accessed and managed in a manner conforming to the ISO FTAM protocol. This goal has been realised. Furthermore, through developing the ISO FTAM virtual filestore mapping for UNIX, we have also succeeded in enduing the local file system with the ability to store and handle structured files, to access and transfer segments of file contents as well as to support several file attributes not supported by UNIX.

We have also drawn attention to several areas which the standard specifications have left open to question and that warrant further standardisation. In the meantime, practical development work can only be interim since different implementors would resort to their own interpretation rendering conformance to the standard difficult and thus defeating the noble goals of Open Systems Interconnection.

In addition, the experience gained from studying and implementing a protocol such as the ISO FTAM has also prompted us to identify the following issues meriting further research and work.

- name resolution

  In order to provide network wide access control, there must be a scheme to resolve how to represent a user's identity, *UserId*, since the syntax may differ from machine to machine. When building up an access control list, how would a user on machine A know how to identify another user on another machine B that may use a different naming convention? A common user identification scheme is needed for authentication, access control and accounting purposes.

- third party file transfers

  In its most general configuration, a file transfer can involve three host computers (or parties):

  - the initiating or controlling host which specifies and coordinates the transfer,
  - the sending host at which the file originally resides and
  - the receiving host(s) where the file is to be sent.

  A ***third party file transfer*** is one that allows a user at host A to initiate the transfer of a file residing on another host B to a third host C.

  The definition of the FTAM service primitives do not make provision for specifying third party file transfers. Third party transfers using a connection-oriented protocol necessitates more intricate synchronisation control since two connections (between A and B as well as between B and C) have to be maintained. It might be useful to provide service primitives to support the specification and coordination of third party file transfers.

- locking at a per-FADU basis

  This would be particularly useful for use of the file service by database applications where files are usually accessed record by record, necessitating locking at the record level. Often, a record maps onto a component FADU of the FADU tree. On many operating systems, the locking mechanism takes a file as the smallest object that can be locked; that is, individual component records within a file cannot be locked. In such cases, to support locking on a per-FADU basis is definitely non-trivial.

- compression of data to be transferred

  To minimise the traffic due to the volume of data transferred, it might be desirable to condense the data. Then, an additional file attribute to specify the data compression algorithm used is needed. If the data transferred is to be encrypted, there must be some agreement on whether the compression algorithm is to be applied before or after the encryption algorithm. Encoding the contents of files (for purposes such as encryption and data compression) really should be handled by the Presentation Layer service which, according to the OSI reference model, is responsible for the representation of data and their data types.

Bibliography

[Aggarwal 1985]	Aggarwal, S., Sabnani K. and Gopinath, B., "A New File Transfer Protocol", AT&T Technical Journal, 64 (10), December 1985.
[Aggarwal 1986]	Aggarwal, S. and Sabnani K., "Formal Specification of a File Transfer Protocol", *Proceedings of the IEEE INFOCOM '86*, 1986, pp. 44–57.
[ASN1 1985a]	International Standards Organisation, "Information Processing Systems – Open Systems Interconnection – Specification of Abstract Syntax Notation One", *ISO DIS 8824*, June 1985.
[ASN1 1985b]	International Standards Organisation, "Information Processing Systems – Open Systems Interconnection – Basic Encoding Rules for Abstract Syntax Notation One", *ISO DIS 8825*, June 1985.
[Bessette 1986]	Bessette, Sylvie, "Implantation d'un protocole de transfert, d'accés et de gestion de fichiers", *M.Sc. thesis, Unversité de Montreal*, July 1986.
[Brachman 1987]	Brachman, B.J. and Chanson, S.T., "An Access Control Mechanism based on the ISO FTAM recommendation", *Canadian Information Processing Society Conference Proceedings*, May 1987, pp.161-166.
[CASE 1986]	ISO DIS 8649/2, "Information Processing Systems - OSI - Service Definition for Common Application Service Elements Part 2: Association Control" June 1986.

[ECMA 1982]	European Computer Manufacturers' Association, "Virtual File Protocol", *ECMA-85*, September 1982.
[FTAM DIS 1986]	ISO DIS 8571/1, 2, 3, 4, "Information Processing Systems - OSI - File Transfer, Access and Management" *ISO/TC 97/SC 21 N1215, N1216, N1217, N1218*, July 1986.
[FTAM DP2 1985]	ISO DIS 8571/1, 2, 3, 4, "Information Processing Systems - OSI - File Transfer, Access and Management" *ISO/TC 97/SC 16 N1190, N1222, N 1223, N 1224* , February 1985.
[Lewan 1983]	Lewan, D. and Long, H. G., "The OSI File Service", *Proceedings of the IEEE*, 71 (12), December 1983, pp. 1414–1419.
[Linington 1984]	Linington, P. F., "The Virtual Filestore Concept", *Computer Networks*, 8 (1), January 1984, pp. 13–16.
[Neufeld 1985]	Neufeld, G., Demco, J., Hilpert B., and Sample, R., "EAN: An X.400 Message System", *Computer Message Systems – 85*, Proceedings of the IFIP TC 6 International Symposium on Computer Message Systems, Uhlig, R. P. (ed.), September 1985, pp. 3–15.
[NBS 1986]	National Bureau of Standards, Edited by Heafner, J., "Implementation Agreements Among Implementors of OSI Protocols", *NBSIR 86-3385-2*, October 1986.
[NIFTP 1981]	File Transfer Protocol Implementors Group, "A Network Independent File Transfer Protocol", *NIFTP-B(80)*, February 5, 1981.
[OSI 1983]	International Standards Organisation, "Information Processing Systems – Open Systems Interconnection – Basic Reference Model", 1983.
[Petersen 1985]	Petersen, N. K. and Skovgaard T., "Anticipating the ISO File Transfer Standards in an Open Systems Implementation", *Computer Networks and ISDN Systems*, 9 (4), April 1985, pp. 267–280.

[PRES 1986a] International Standards Organisation, "Information Processing Systems – OSI – Connection oriented presentation service definition", *ISO DIS 8822*, June 1986.

[PRES 1986b] International Standards Organisation, "Information Processing Systems – OSI – Connection oriented presentation protocol specification", *ISO DIS 8823*, June 1986.

[Quarterman 1985] Quarterman, J. S., Siberschatz A. and Peterson, J. L., "4.2BSD and 4.3BSD as Examples of the UNIX System", *ACM Computing Surveys*, 17 (4), December 1985, pp. 371–401.

[Ritchie 1978] Ritchie, D. M. and Thompson K., "The UNIX Time-Sharing System", *The Bell System Technical Journal*, 57 (6), July/August 1978, pp. 1905–1930.

[X400 1984] Comité Consultatif Internationale de Télégraphique et Téléphonique, "Fascicle VIII.7 Recommendation X.400, Data Communication Networks : Message Handling Systems", *Recommendation X.400*, October 1984.

[X409 1983] Comité Consultatif Internationale de Télégraphique et Téléphonique, "Recommendation X.409, Message Handling Systems: Presentation Transfer Syntax and Notation", *Recommendation X.409*, December 1983.

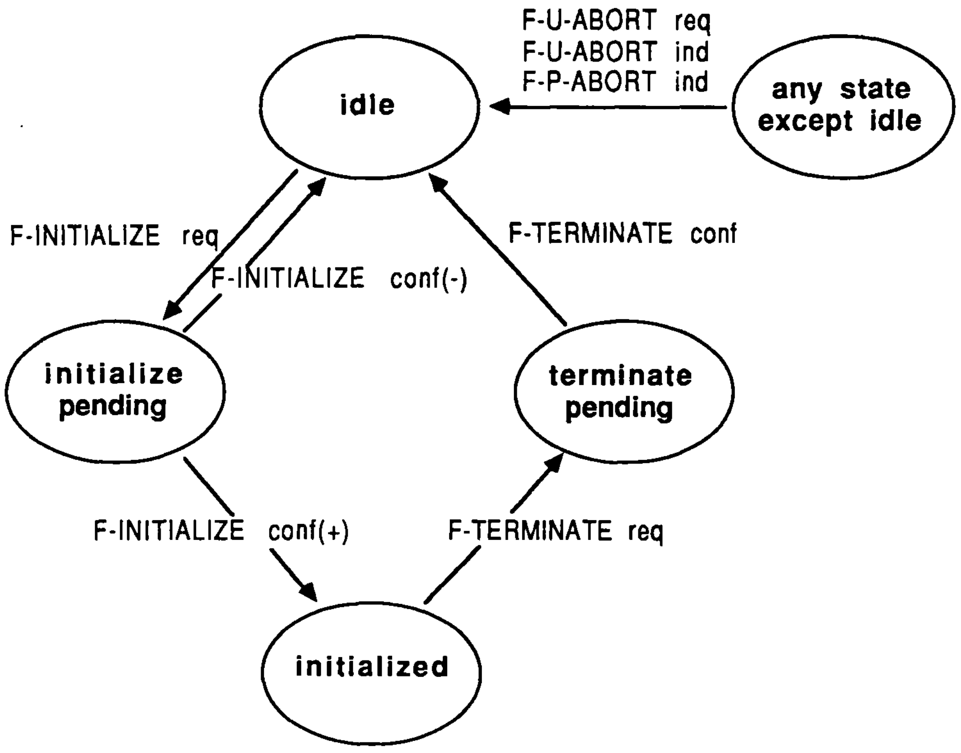
Appendix A

State Transition Diagrams

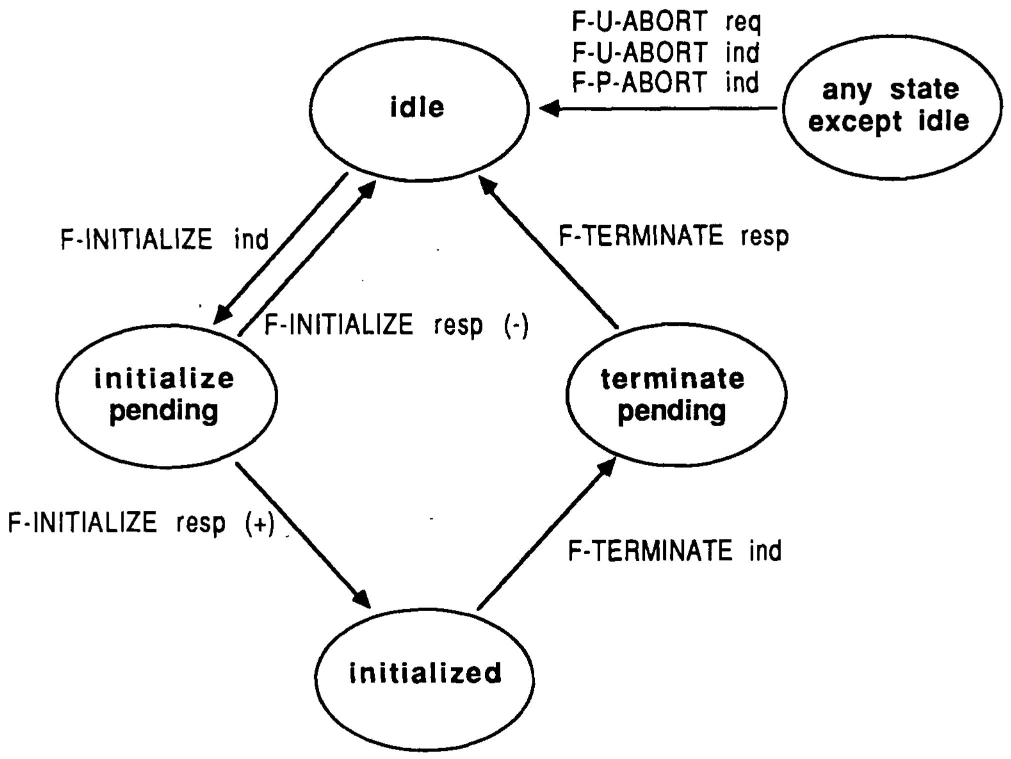
Legend for the service primitives :

req	=	request
ind	=	indication
res	=	response
conf	=	confirm

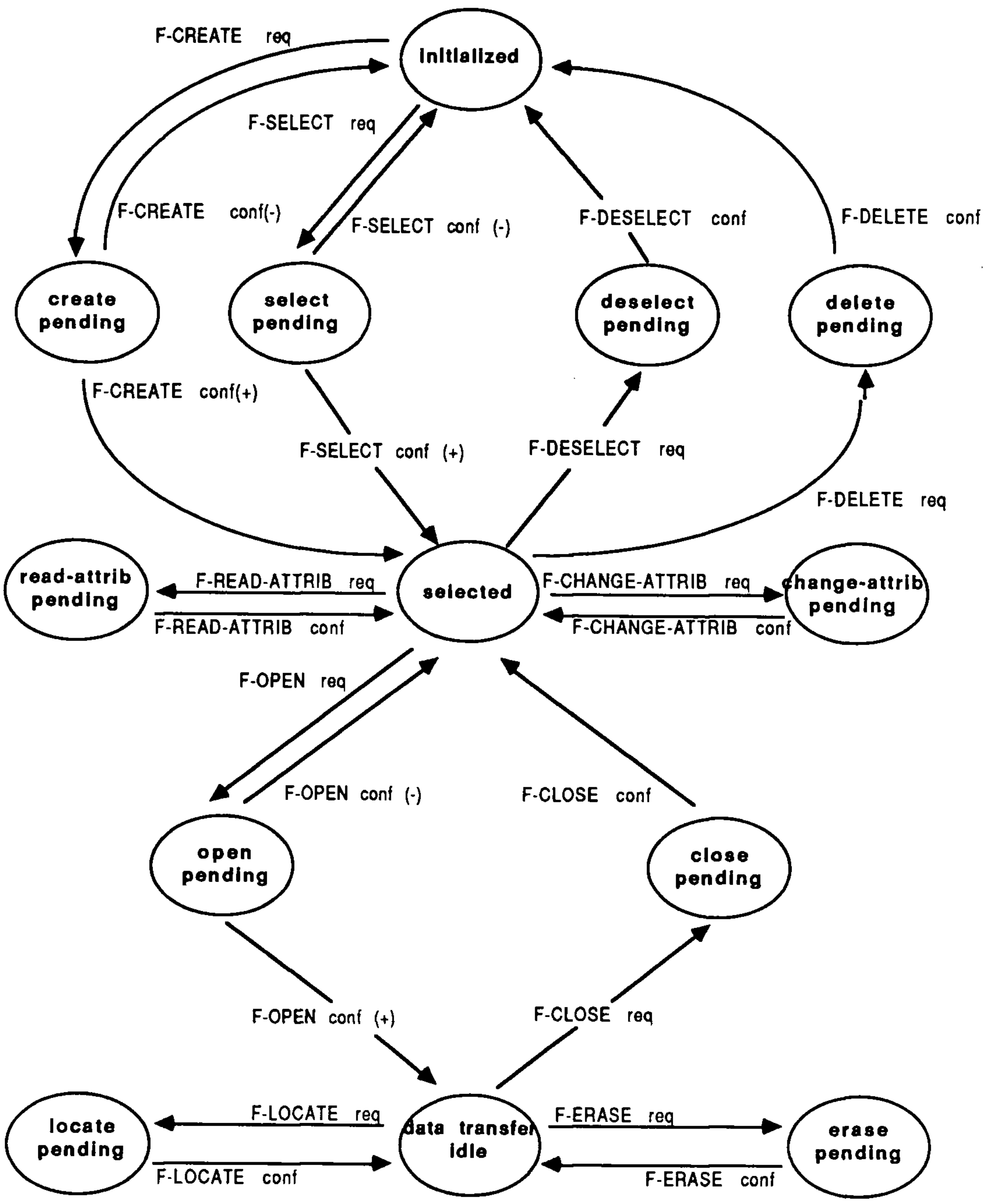
State Transition Diagram
for Association Establishment (Initiator)



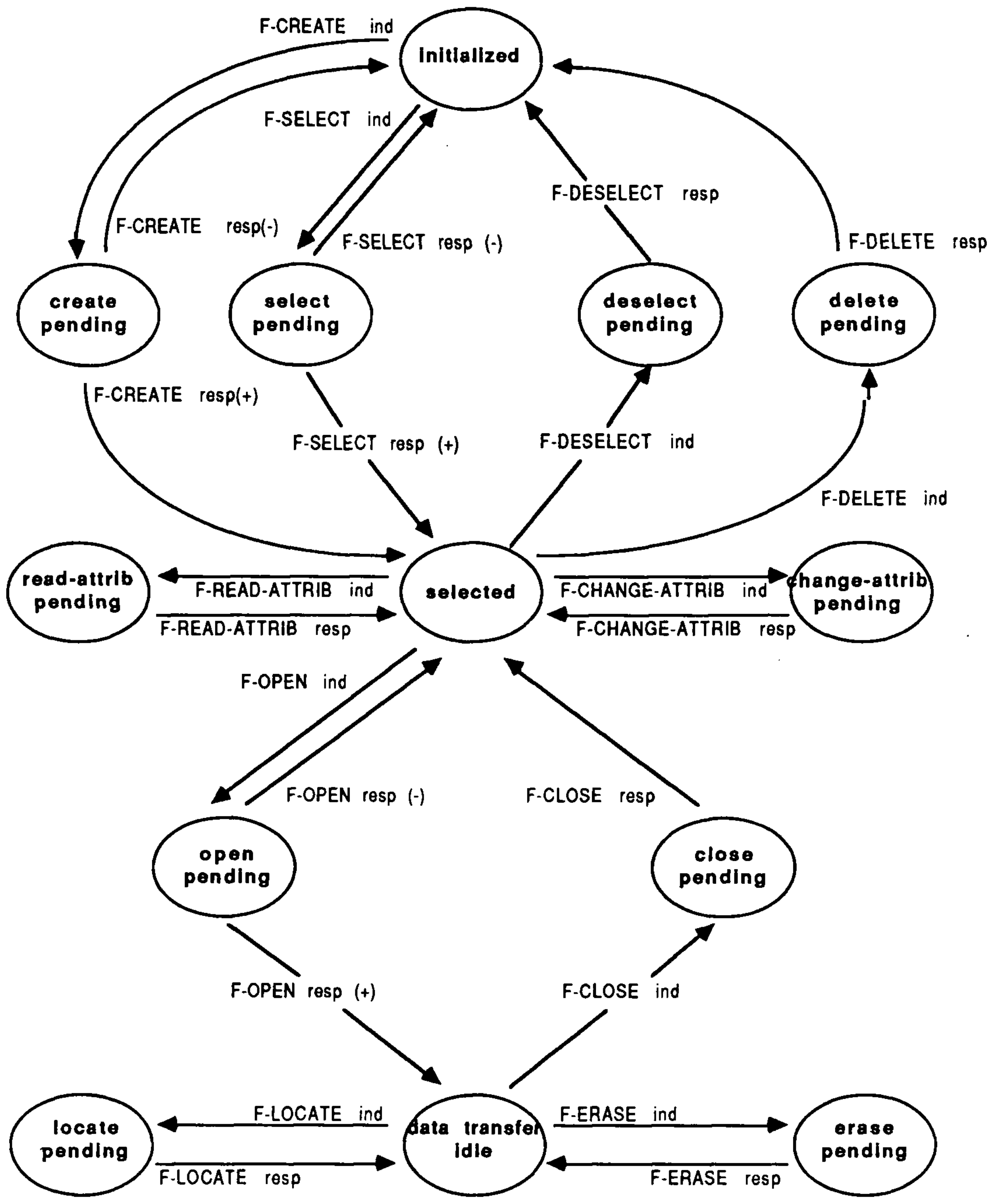
State Transition Diagram
for Association Establishment (Responder)



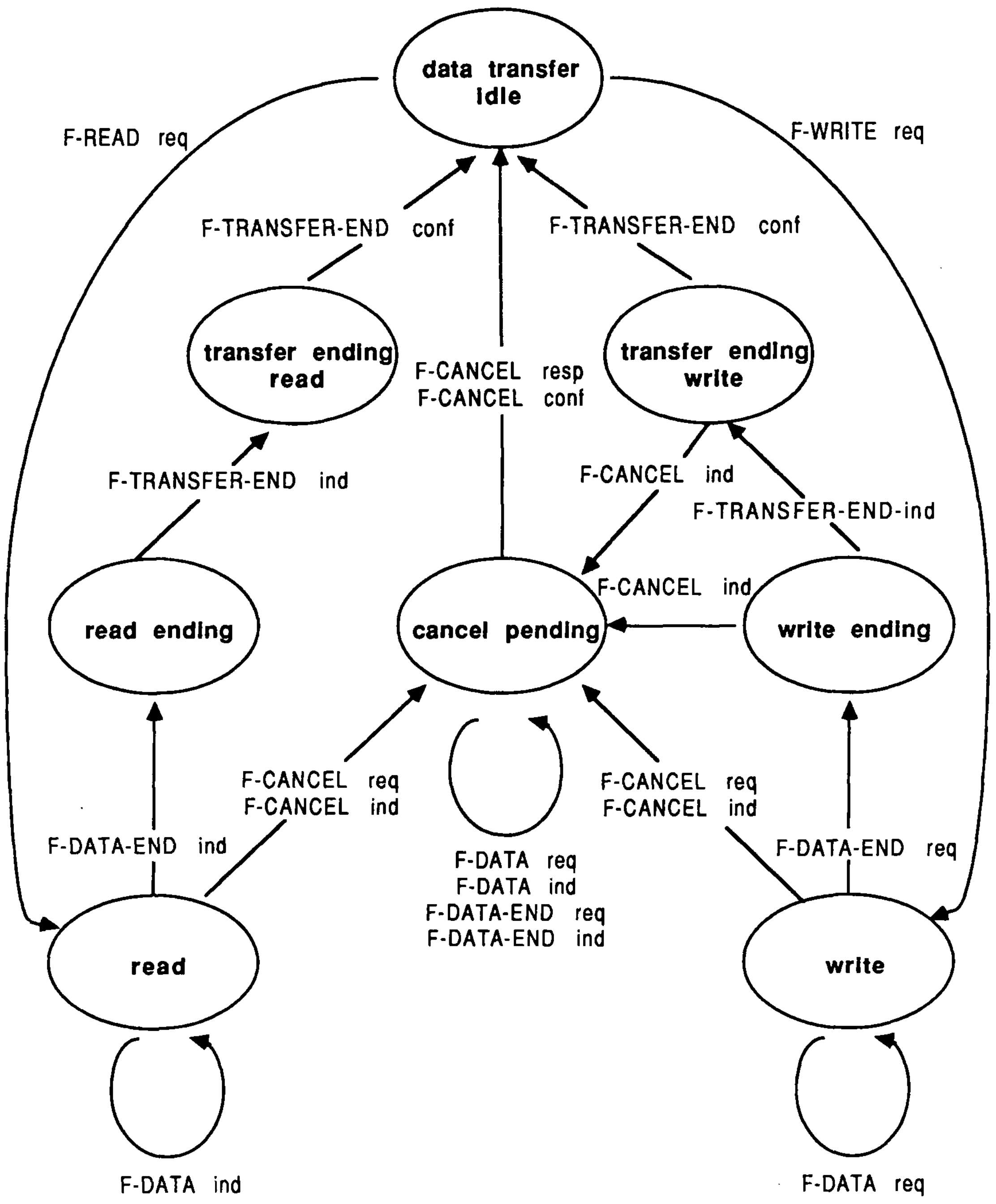
State Transition Diagram
for the File Selection Establishment service (Initiator)



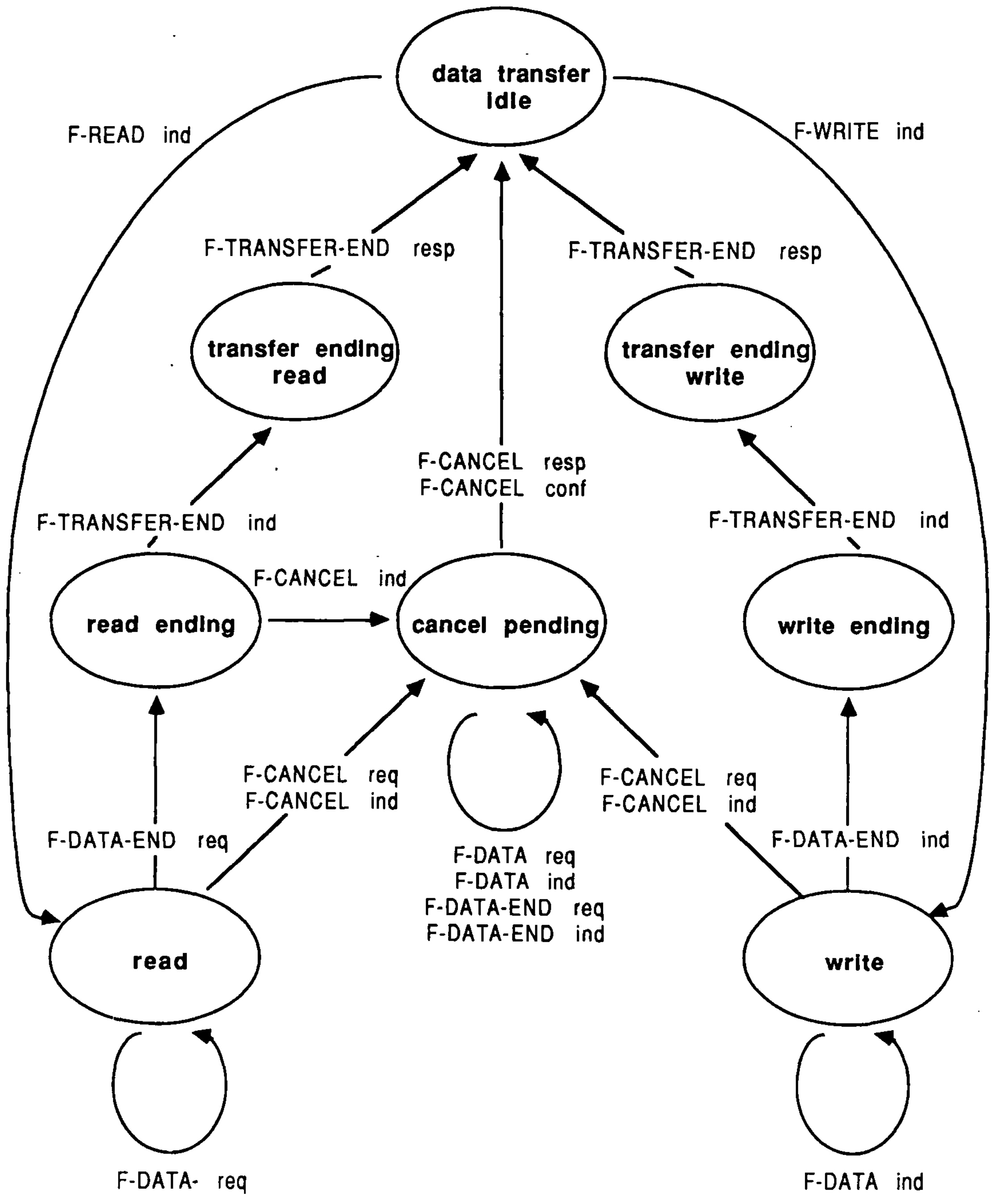
State Transition Diagram
for the File Selection Establishment service (Responder)



State Transition Diagram
for the Bulk Data Transfer Service (Initiator)



State Transition Diagram
for the Bulk Data Transfer Service (Responder)



Appendix B

ASN.1 definitions for FTAM

```
IS08571-FTAM definitions ::=

BEGIN

  PDU ::= CHOICE{InitializePDU, FilePDU, BulkdataPDU}

  InitializePDU ::= CHOICE {
                          [APPLICATION 1] IMPLICIT FINITIALIZErequest,
    [1] IMPLICIT FINITIALIZEresponse,
    [2] IMPLICIT FTERMINATErequest,
    [3] IMPLICIT FTERMINATEresponse,
    [4] IMPLICIT FUABORTrequest,
    [5] IMPLICIT FUABORTresponse }

  FilePDU := CHOICE {
            [6]  IMPLICIT FSELECTrequest,
            [7]  IMPLICIT FSELECTresponse,
            [8]  IMPLICIT FDESELECTrequest,
            [9]  IMPLICIT FDESELECTresponse,
            [10] IMPLICIT FCREATErequest,
            [11] IMPLICIT FCREATEresponse,
            [12] IMPLICIT FDELETErequest,
            [13] IMPLICIT FDELETEresponse,
            [14] IMPLICIT FREADATTRIBrequest,
            [15] IMPLICIT FREADATTRIBresponse,
            [16] IMPLICIT FCHANGEATTRIBrequest,
            [17] IMPLICIT FCHANGEATTRIBresponse,
            [18] IMPLICIT FOPENrequest,
            [19] IMPLICIT FOPENresponse,
            [20] IMPLICIT FCLOSErequest,
            [21] IMPLICIT FCLOSEresponse,
            [22] IMPLICIT FBEGINGROUPrequest,
            [23] IMPLICIT FBEGINGROUPresponse,
            [24] IMPLICIT FENDGROUPrequest,
            [25] IMPLICIT FENDGROUPresponse,
            [26] IMPLICIT FRECOVERrequest,
            [27] IMPLICIT FRECOVERresponse,
            [28] IMPLICIT FLOCATErequest,
            [29] IMPLICIT FLOCATEresponse,
            [30] IMPLICIT FERASErequest,
            [31] IMPLICIT FERASEresponse
            }


BulkdataPDU := CHOICE {
            [32] IMPLICIT FREADrequest,
            [33] IMPLICIT FWRITErequest,
            [34] IMPLICIT FDATAENDrequest,
            [35] IMPLICIT FTRANSFERENDrequest,
            [36] IMPLICIT FTRANSFERENDresponse,
            [37] IMPLICIT FCANCELrequest,
            [38] IMPLICIT FCANCELresponse,
            [39] IMPLICIT FRESTARTrequest,
            [40] IMPLICIT FRESTARTresponse,
            [55] IMPLICIT FDATArequest
            }


[APPLICATION 1] IMPLICIT FINITIALIZErequest ::=
               SEQUENCE {
                          protocolId    [0] INTEGER { isoFTAM(0) },
                          versionNumber [1] IMPLICIT SEQUENCE {
                                            major INTEGER,
                                            minor INTEGER }
```

```
                                        -- initially {major 0,
                                                      minor 0}
                          serviceType   [2] INTEGER
                                        { reliable        (0),
                                          user correctable(1) },
                          serviceClass  [3] INTEGER
                                        { transfer(0),
                                          access  (1),
                                          management (2) },
                          functionalUnits [4] BITSTRING {
                                          read                 (0),
                                          write                (1),
                                          fileAccess           (2),
                                          limitedManagement    (3),
                                          enhancedManagement   (4),
                                          grouping             (5),
                                          recovery             (6),
                                          restartDataTransfer  (7) }
                          attributeGroups [5] BITSTRING {
                                          storage   (0),
                                          security  (1) }
                          rollbackAvailability [6] BOOLEAN DEFAULT FALSE,
                          PresentationContextName,
                          identityOfInitiator  [7] GraphString OPTIONAL,
                          CurrentAccount           OPTIONAL,
                          filestorePassword    [8] OCTETSTRING OPTIONAL,
                          checkpointWindow     [9] INTEGER OPTIONAL,
             }


 [1] IMPLICIT FINITIALIZEresponse ::=
             SEQUENCE {
                          Diagnostic,
                          protocolId [0] INTEGER {isoFTAM(0) },
                          versionNumber [1] IMPLICIT SEQUENCE {
                                        major INTEGER,
                                        minor INTEGER }
                          serviceType   [2] INTEGER
                                        { reliable        (0),
                                            user correctable(1) },
                          serviceClass   [3] INTEGER
                                            { transfer(0),
                                              access   (1),
                                              management (2) },
                          functionalUnits [4] BITSTRING {
                                              read                 (0),
                                              write                (1),
                                              fileAccess           (2),
                                              limitedManagement    (3),
                                              enhancedManagement   (4),
                                              grouping             (5),
                                              recovery             (6),
                                              restartDataTransfer  (7) }
                          attributeGroups [5] BITSTRING {
                                              storage   (0),
                                              security  (1) }
                          rollbackAvailability [6] BOOLEAN DEFAULT FALSE,
                          PresentationContextName,
                          checkpointWindow     [7] INTEGER OPTIONAL,
                  }


[2] IMPLICIT FTERMINATErequest ::=
             SEQUENCE {-- no members defined for now -- }

[3] IMPLICIT FTERMINATEresponse ::= SEQUENCE { Charging OPTIONAL }

[4] IMPLICIT FUABORTrequest ::= SEQUENCE { Diagnostic }

[5] IMPLICIT FPABORTrequest ::= SEQUENCE { Diagnostic }

[6] IMPLICIT FSELECTrequest ::=
             SEQUENCE {
                         Filename,
                         OtherAttributes                  OPTIONAL,
                         AccessControl,
                         AccessPasswords                  OPTIONAL,
                         ConcurrencyControl               OPTIONAL,
                              CommitmentControl           OPTIONAL,
                              currentAccessStructureType [0]
                                    AccessStructureType   OPTIONAL,
                              CurrentAccount              OPTIONAL }

[7] IMPLICIT FSELECTresponse ::=
             SEQUENCE {
                              Diagnostic,
                              Filename                    OPTIONAL,
                              OtherAttributes             OPTIONAL,
                              AccessControl }

[8] IMPLICIT FDESELECTrequest ::= SEQUENCE { }

[9] IMPLICIT FDESELECTresponse ::=
             SEQUENCE {
                              Diagnostic,
                              Charging    OPTIONAL }

[10] IMPLICIT FCREATErequest ::=
              SEQUENCE {
                              Filename,
                              OtherAttributes             OPTIONAL,
                              AccessControl,
                              AccessPasswords             OPTIONAL,
                              ConcurrencyControl          OPTIONAL,
                              CommitmentControl           OPTIONAL,
                              override            INTEGER OPTIONAL,
                              CurrentAccount              OPTIONAL }

[11] IMPLICIT FCREATEresponse ::= FSELECTresponse

[12] IMPLICIT FDELETErequest ::=
              SEQUENCE { deletePassword [0] OCTETSTRING OPTIONAL }

[13] IMPLICIT FDELETEresponse ::= FDESELECTresponse

[14] IMPLICIT FREADATTRIBrequest ::= SEQUENCE { AttributeNames }
```

```
[15] IMPLICIT FREADATTRIBresponse ::=
            SEQUENCE {
                         Diagnostic,
                         Filename OPTIONAL,
                         OtherAttributes  OPTIONAL }
                         --- At least one of the OPTIONALs

[16] IMPLICIT FCHANGEATTRIBrequest ::=
            SEQUENCE {
                         Filename                     OPTIONAL,
                         OtherAttributes              OPTIONAL }
                         --- At least one the OPTIONALs

<< NOTE >> :
  -- That the 2DP requires at least one OPTIONAL is not consistent
     because if none of the fields in OtherAttributes were successfully
     changed, the OtherAttributes would be absent.
  -- So, it is reasonable to remove this requirement that
     "at least one of the OPTIONALs" especially since this is
     consistent with the DIS.


[17] IMPLICIT FCHANGEATTRIBresponse ::= FREADATTRIBresponse

[18] IMPLICIT FOPENrequest ::= SEQUENCE {
                      processingMode [0] BITSTRING {
                                              read         (0),
                                              insertChild  (1),
                                              insertSister (2),
                                              replace      (3),
                                              extend       (4),
                                              erase        (5)  },
                      PresentationContextName  OPTIONAL,
                      ConcurrencyControl       OPTIONAL,
                      CommitmentControl        OPTIONAL,
                      ActivityIdentifier       OPTIONAL,
                      [1] RecoveryMode         OPTIONAL }

RecoveryMode ::= INTEGER { none(0),
                                  atStartOfFile(1),
                                  atAnyActiveCheckpoint(2) }


[19] IMPLICIT FOPENresponse ::=
              SEQUENCE {
                     Diagnostic,
                     PresentationContextName  OPTIONAL,
                     ConcurrencyControl       OPTIONAL,
                     [0] RecoveryMode         OPTIONAL
                     [1] PresentationActions  OPTIONAL }

PresentationActions ::= BITSTRING { pDefine(0), pDelete(1) }


[20] IMPLICIT FCLOSErequest ::=
              SEQUENCE { CommitmentControl OPTIONAL }

[21] IMPLICIT FCLOSEresponse ::=
              SEQUENCE {
                        Diagnostic,
                        CommitmentControl  OPTIONAL }

[22] IMPLICIT FBEGINGROUPrequest  ::= SEQUENCE { threshold [0] INTEGER }

[23] IMPLICIT FBEGINGROUPresponse ::= SEQUENCE { -- no members -- }

[24] IMPLICIT FENDGROUPrequest    ::= SEQUENCE { -- no members -- }

[25] IMPLICIT FENDGROUPresponse   ::= SEQUENCE { -- no members -- }

[26] IMPLICIT FRECOVERrequest ::=
              SEQUENCE {
                        ActivityIdentifier,
                        bulkTransferNumber  [0] INTEGER,
                        AccessControl               OPTIONAL,
                        AccessPasswords             OPTIONAL,
                        recoveryPoint       [1] INTEGER OPTIONAL }
```

```
[27] IMPLICIT FRECOVERresponse ::=
           SEQUENCE {
                      Diagnostic,
                      recoveryPoint       [0] INTEGER OPTIONAL }

[28] IMPLICIT FLOCATErequest ::=
           SEQUENCE {
                      FTAM_FADUIdentity,
                      ConcurrencyControl  OPTIONAL }

[29] IMPLICIT FERASErequest ::=
           SEQUENCE {
                      Diagnostic,
                      FTAM_FADUIdentity   OPTIONAL }

[30] IMPLICIT FERASErequest ::=
           SEQUENCE {
                      FTAM_FADUIdentity,
                      ConcurrencyControl  OPTIONAL }

[31] IMPLICIT FERASErequest ::= SEQUENCE { Diagnostic }

[32] IMPLICIT FREADrequest ::=
           SEQUENCE {
                      FTAM_FADUIdentity,
                      AccessContext      OPTIONAL,
                      ConcurrencyControl OPTIONAL}

[33] IMPLICIT FWRITErequest ::=
           SEQUENCE {
                      faduOperation [0] INTEGER {
                                        insertChild  (0),
                                        insertSister (1),
                                        replace      (2),
                                        extend       (3) },
                      FTAM_FADUIdentity,
                      AccessContext       OPTIONAL,
                      ConcurrencyControl  OPTIONAL}
```

```
[34] IMPLICIT FDATAENDrequest ::=
              SEQUENCE { Diagnostic }

[35] IMPLICIT FTRANSFERENDrequest ::=
              SEQUENCE { CommitmentControl OPTIONAL }

[36] IMPLICIT FTRANSFERENDresponse ::=
              SEQUENCE {
                         Diagnostic,
                         CommitmentControl OPTIONAL }

[37] IMPLICIT FCANCELrequest ::= SEQUENCE { Diagnostic }

[38] IMPLICIT FCANCELresponse ::= FCANCELrequest

[39] IMPLICIT FRESTARTrequest ::=
              SEQUENCE {
                         checkPointId [0] INTEGER,
                         Diagnostic       OPTIONAL }

[40] IMPLICIT FRESTARTresponse ::=
              SEQUENCE {
                         checkPointId [0] INTEGER,
                         Diagnostic       OPTIONAL }

[55] IMPLICIT FDATArequest ::= OCTETSTRING



  Application-wide Types
--------------------------

Diagnostic ::= [APPLICATION 0] IMPLICIT SEQUENCE OF
                    SEQUENCE {
                      errortypeidentifier[0]  ErrorTypeIdentifier,
                      erroridentifier[1]      ErrorIdentifier,
                      errorObserver[2]    INTEGER,
                      errorSource[3]   INTEGER,
                         suggestedDelay[4] INTEGER OPTIONAL,
                         furtherDetails CHOICE {
                                        [5] GraphString,
                                        [6] OCTETSTRING } OPTIONAL

ErrorTypeIdentifier ::= INTEGER {
                            success            (0),
                            warning            (1),
                            recoverableError   (2),
                            unrecoverableError (3)  }

ErrorIdentifier     ::= INTEGER --- as defined in 2DP Part III

CurrentAccount ::= [APPLICATION 2] GraphString


Charging ::= [APPLICATION 3] IMPLICIT SEQUENCE OF
                         SEQUENCE { resoureceId        GraphString,
                                    chargingUnit       GraphString,
                                    chargingValue      INTEGER }

Filename   ::= [APPLICATION 4] IMPLICIT SEQUENCE OF GraphString

OtherAttributes ::=
  [APPLICATION 5] IMPLICIT SEQUENCE OF
                             CHOICE {
                              permittedActions [1] BITSTRING  {
                                                     read(0),
                                                     insertChild(1),
                                                     insertSister(2),
                                                     replace(3),
                                                     extend(4),
                                                     erase(5),
                                                     readforwards(6),
                                                     readbackwards(7),
                                                     readAnyOrder(8),
                                                     writeforwards(9),
                                                     writebackwards(10),
                                                     writeAnyOrder(11),
                                                writeEOF(12)
                                              }
                              accessControl[2] IMPLICIT SEQUENCE OF Condition,
                              account [3] CurrentAccount,
                              creation[4] GeneralisedTime,
                              lastModification [5] GeneralisedTime,
                              lastReadAccess [6] GeneralisedTime,
                              identityOfCreator [7] UserId
                              identityOfLastModifier [8] UserId,
                              identityOfLastReader [9] UserId,
                              fileAvailability [10] INTEGER {
                                                    immediate (0),
                                                    deferred  (1)  }
                              presentationContext [11]
                                 IMPLICIT SEQUENCE OF PresentationContextName,
                              encryptionName [12] GraphString,
                              accessStructureType [13] AccessStructureType,
                              currentFileSize [14] Filesize,
                              futureFileSize  [15] Filesize,
                              legalQualifications [16] GraphString,
                              privateUse [17] OCTETSTRING  }

AccessControl  ::= [APPLICATION 6]  BITSTRING {
                                     read             (0),
                                     insertChild      (1),
                                     insertSister     (2),
                                     replace          (3),
                                     extend           (4),
                                     erase            (5),
                                     changeAttributes (6),
                                     readAttributes   (7),
                                     deleteFile       (8),
                                     createFile       (9) }

AttributeNames  ::=  BITSTRING {
                        filename          (0),
                        permittedActions  (1),
                        accessControl     (2),
                        account           (3),
                      creation            (4),
                      lastModification    (5),
                      lastReadAccess      (6),
                      identityOfCreator   (7),
                      identityOfLastModifier (8),
                      identityOfLastReader   (9),
                      fileAvailability       (10),
                      presentationContext    (11),
                      encryptionName         (12),
                      accessStructureType    (13),
                      currentFileSize        (14),
                      futureFileSize         (15),
                      legalQualifications    (16),
                      privateUse             (17)
                    }

Condition ::= SEQUENCE { permittedAccess     [0] AccessControl,
                         identity            [1] UserId         OPTIONAL,
                         passwords           [2] AccessPasswords,
                         locationOfInitiator[3] SEAPAddress    OPTIONAL }


AccessPasswords ::= [APPLICATION 7] IMPLICIT SEQUENCE {
                             read             [0] OCTETSTRING OPTIONAL,
                             insertChild      [1] OCTETSTRING OPTIONAL,
                             insertSister     [2] OCTETSTRING OPTIONAL,
                             replace          [3] OCTETSTRING OPTIONAL,
                             extend           [4] OCTETSTRING OPTIONAL,
                             erase            [5] OCTETSTRING OPTIONAL,
                             changeAttributes [6] OCTETSTRING OPTIONAL,
                             readAttributes   [7] OCTETSTRING OPTIONAL,
                             deleteFile       [8] OCTETSTRING OPTIONAL,
                             createFile       [9] OCTETSTRING OPTIONAL }

UserId ::= GraphString

AccessStructureType  ::= SEQUENCE {
                                     [0] INTEGER {
                                          unstructured (0),
                                               flat         (1),
                                               hierarchical (2) },
                                  maxDepth [1] INTEGER OPTIONAL,
                                   -- maxDepth only permitted with hierarchical
                                }

 Filesize ::= SEQUENCE {  units        INTEGER,
                          sizeInUnits INTEGER,
                          residue     INTEGER}


 PresentationContextName ::= IS08822-PRES.ref
 --- ubcFTAM used this :
 PresentationContextName ::= BITSTRING { binary(1), ascii(2) }


 ConcurrencyControl  ::= [APPLICATION 8] AccessControl
   -- set for exclusive,
      unset for shared

 CommitmentControl  ::= [APPLICATION 9] ISO-8649/3.ref
-- as defined in the CCR standards

 ActivityIdentifier  ::= [APPLICATION 10] INTEGER


 FADUIdentifier ::= CHOICE { GraphString, INTEGER }

 FTAM_FADUIdentity ::=
          [APPLICATION 11] CHOICE
                      { faduIdentity  [0] IS08571 - FSTR.FADUIdentity,
                        identity      [1] EXTERNAL }
                -- FTAM Structure Module definition: PART II, Clause 5.3.1

 FADUIdentity ::= SEQUENCE{
                       CHOICE{
                              [0] INTEGER {first(0), last(1)},
                              [1] INTEGER {current  (0),
                                           next     (1),
                                          previous (2)},
                                [2] FADUIdentifier,
                                [3] IMPLICIT SEQUENCE OF SEQUENCE {
                                     faduIdentifier  FADUIdentifier,
                                     arclength [1] INTEGER DEFAULT 1 },
                                levelnumber[4] INTEGER
                                -- levelnumber only in access context 4
                          }
                      }

FADUStructure ::= SEQUENCE {
                        faduIdentifier [0] IMPLICIT FADUIdentifier,
                        dataExists     [1] BOOLEAN,
                        subtree        SEQUENCE OF SEQUENCE {
                                         arclength [0] INTEGER DEFAULT 1,
                                         FADUStructure
                                       }
                      }

FADU  ::= SEQUENCE {
               faduIdentifier [0] IMPLICIT FADUIdentifier,
               dataUnit        DU OPTIONAL,
               subtree        SEQUENCE OF SEQUENCE {
                                    arclength [0] INTEGER DEFAULT 1,
                                    FADU  }
           }

FileTransferStructure ::=  CHOICE {
                          accessContext1 [1] IMPLICIT FADU,
                          accessContext2 [2] IMPLICIT SEQUENCE OF DU,
                          accessContext3 [3] DU,
                          accessContext4 [4] IMPLICIT SEQUENCE OF DU,
                            -- Same level number
                          accessContext5 [5] IMPLICIT FADUStructure   }

DU  ::= dataUnit [0] EXTERNAL

AccessContext        ::= [APPLICATION 12] INTEGER {
                                accessContext1 (1),
                                        accessContext2 (2),
                                        accessContext3 (3),
                                        accessContext4 (4),
                                        accessContext5 (5) }

 SEAPAddress  ::= EXTERNAL -- outside scope of this standard
 --- ubcFTAM used this :
 SEAPAddress  ::= GraphString;

END.
```

Appendix C

Program Mode

The following illustrates the use of ubcFTAM from within a program written in C.

```
#include "../h/defn.h"
#include "../h/param.h"
#include "../h/prim.h"
#include "../h/access.h"
#include "../h/service.h"

#define MAP_F_INITIALIZE  "0000000000000001111101111101100000000000"
#define MAP_F_TERMINATE   "0000000000000000000000000000000000000000"
#define MAP_F_READATTRIB  "0010000000000000000000000000000000000000"
#define MAP_F_SELECT      "0100010010000000000000000000000000000000"
#define MAP_F_DESELECT    "0000000000000000000000000000000000000000"
#define MAP_F_OPEN        "0000000000110000000000000000000000000000"
#define MAP_F_CLOSE       "0000000000000000000000000000000000000000"
#define MAP_F_READ        "0000000000001010000000000000000000000000"
#define MAP_F_WRITE       "0000000000001110000000000000000000000000"
#define MAP_F_DATA        "0000000000000001000000000000000000000000"
#define MAP_F_DATAEND     "1000000000000000000000000000000000000000"
#define MAP_F_TRANSFEREND "0000000000000000000000000000000000000000"
/*                         0123456789 123456789 123456789 1234567890
*                                   1         2         3          4
*/
#define  exiting( rcval )    { rc = rcval; goto out; }
#define  PROCESSING_MODE     "100111"
```

```
                                            /* processing mode set for:     */
                                            /* READ, REPLACE, EXTEND, ERASE */
#define  NEW_DATA               "This is the sample data for replacement."
#define  WAIT                   1

/*
* ------------------------------------------------------------------
* The routines used here for sending and receiving
* FTAM service primitives are stateSend() and stateRecv().
* These routines are invoked thus :
*   stateSend( param, &_state );
*   stateRecv( &param, &_state );
* and automatically maintain the protocol state in _state.
*
* If the protocol state is not to be maintained,
* invoke routines
*   sendFTAMprim( param ) and recvFTAMprim( &param ).
* ------------------------------------------------------------------
*/

main()
{
  ParamListType      *param = NULL;
  int                rc, _state;

  rc = 0;

  START_FTAM();      /* -- initialises _state            */
                     /* -- invokes the initiator process */

  /* --- initialising an FTAM association ----------------------- */

  initParamList( &param );
  param->service      = F_INITIALIZE_RQ;
  (void) strcpy(param->paramMap, MAP_F_INITIALIZE);
  param->protocolId   = 0;
  param->versionMajor = 0;
  param->versionMinor = 0;
  param->calledAddr   = "ftamtest@csgrads";
  param->callingAddr   = "ftamtest@dsrg";
  param->serviceType   = ST_USER_CORRECTABLE;
  param->serviceClass  = SC_TRANSFER;
  param->funcUnits     = "11111000";  /* READ, WRITE, FILE_ACCESS,   */
                                      /* LIMITED_FMGT, ENHANCED_FMGT */
  param->attrGrps      = "11";        /* STORAGE, SECURITY           */
  param->rollback      = FALSE;
  param->idInitiator   = "goh";
  setPasswd( &param->filestorePasswd, "secret" );
  param->charging      = NULL;

  if ( stateSend( param, &_state )  < 0 )        exiting( -1 );
  if ( stateRecv( &param, &_state, WAIT ) < 0 )  exiting( -1 );


  /* --- selecting a file ---------------------------------------- */

  initParamList( &param );
  param->service     = F_SELECT_RQ;
  (void) strcpy(param->paramMap, MAP_F_SELECT);
  param->filename              = ALLOCREC( FilenameNode );
    param->filename->next      = NULL;
    (void) strcpy(param->filename->filename, "myfile" );
  param->accessCtrl = ALLOCSTR( MAXACCESSCTRL );
  param->accessCtrl = "1001111110";      /* turns on bits for */
                                         /* READ,   REPLACE,  */
                                         /* EXTEND, ERASE,    */
                                         /* CHANGEATTRIB,     */
                                         /* READATTRIB,       */
/* DELETEFILE         */
  param->accessStr  = (int) AS_UNSTRUCTURED;

  if ( stateSend( param, &_state )  < 0 )        exiting( -1 );
  if ( stateRecv( &param, &_state, WAIT ) < 0 )  exiting( -1 );


  /* --- read attributes of the selected file -------------------- */

  initParamList( &param );
```

```
param->service    = F_READATTRIB_RQ;
(void) strcpy(param->paramMap, MAP_F_READATTRIB);
param->attrNames   = "0110110000001011000";
                     /* requesting to read the following attributes: */
                     /*   permitted actions, access control,         */
                     /*   time of creation,                          */
                     /*   time of last modification,                 */
                     /*   presentation context,  access structure,   */
                     /*   current filesize                           */

if ( stateSend( param, &_state )  < 0 )        exiting( -1 );
if ( stateRecv( &param, &_state, WAIT ) < 0 )  exiting( -1 );

showParamList( param );     /* to display values of attributes read */

/* --- open the selected file ---------------------------------- */

initParamList( &param );
param->service     = F_OPEN_RQ;
(void) strcpy(param->paramMap, MAP_F_OPEN);
param->processMode = ALLOCSTR( strlen(PROCESSING_MODE) );
  (void) strcpy( param->processMode, PROCESSING_MODE );
param->presContext  = ALLOCSTR( strlen("01") );
  (void) strcpy(param->presContext, "01" );           /* ascii */

if ( stateSend( param, &_state )  < 0 )        exiting( -1 );
if ( stateRecv( &param, &_state, WAIT ) < 0 )  exiting( -1 );

/* --- read the selected file ---------------------------------- */

initParamList( &param );
param->service     = F_READ_RQ;
(void) strcpy(param->paramMap, MAP_F_READ);
setFaduId( &param->faduId, FADU_REF_FIRST );
param->accessContext = CONTEXT2;

/* send F_READ_RQ */
if ( stateSend( param, &_state )  < 0 )   exiting( -1 );
```

```
/* --- to receive incoming F_DATA_RQ --- */

if ( stateRecv( &param, &_state, WAIT ) < 0 ) exiting( -1 );
while ( param->service == F_DATA_RQ ) {
   collect( param->data );
   /* to receive F_DATA_RQ */
   if ( stateRecv( &param, &_state, WAIT ) < 0 )
      exiting( -1 );
}

if ( param->service == F_DATAEND_RQ ) {
   initParamList( &param );
   param->service    = F_TRANSFEREND_RQ;
   (void) strcpy(param->paramMap, MAP_F_TRANSFEREND);

   if ( stateSend( param, &_state )  < 0 )        exiting( -1 );

   /* to receive F_TRANSFEREND_RP */
   if ( stateRecv( &param, &_state, WAIT ) < 0 )  exiting( -1 );
}
else
   exiting( -1 );                  /* unexpected primitive received */

/* --- replace the Data Unit ----------------------------------- */

initParamList( &param );
param->service    = F_WRITE_RQ;
(void) strcpy(param->paramMap, MAP_F_WRITE);
param->faduOp = AC_REPLACE;
setFaduId( &param->faduId, FADU_REF_FIRST );
param->accessContext = CONTEXT2;

if ( stateSend( param, &_state )  < 0 )       exiting( -1 );
if ( stateRecv( &param, &_state, WAIT ) < 0 ) exiting( -1 );

   /* --- send data --------------- */
initParamList( &param );
param->service    = F_DATA_RQ;
```

```
(void) strcpy(param->paramMap, MAP_F_DATA);
param->data = ALLOCREC( OCTETSTRING );
  param->data->length  = strlen( NEW_DATA );
  param->data->content = ALLOCSTR( param->data->length );
  (void) strcpy ( param->data->content, NEW_DATA );
if ( stateSend( param, &_state )  < 0 )  exiting( -1 );

   /* --- send dataend ------------ */
initParamList( &param );
param->service    = F_DATAEND_RQ;
(void) strcpy(param->paramMap, MAP_F_DATAEND);
if ( stateSend( param, &_state )  < 0 )  exiting( -1 );

   /* --- send transferend --------- */
initParamList( &param );
param->service    = F_TRANSFEREND_RQ;
(void) strcpy(param->paramMap, MAP_F_TRANSFEREND);
/* to send F_TRANSFEREND_RQ */
if ( stateSend( param, &_state )  < 0 )        exiting( -1 );
/* to receive F_TRANSFEREND_RP */
if ( stateRecv( &param, &_state, WAIT ) < 0 )  exiting( -1 );

/* --- close the selected file -------------------------------- */

initParamList( &param );
param->service    = F_CLOSE_RQ;
(void) strcpy(param->paramMap, MAP_F_CLOSE);
if ( stateSend( param, &_state )  < 0 )        exiting( -1 );
if ( stateRecv( &param, &_state, WAIT ) < 0 )  exiting( -1 );

/* --- deselect the file -------------------------------------- */

initParamList( &param );
param->service    = F_DESELECT_RQ;
(void) strcpy(param->paramMap, MAP_F_DESELECT);
if ( stateSend( param, &_state )  < 0 )        exiting( -1 );
if ( stateRecv( &param, &_state, WAIT ) < 0 )  exiting( -1 );

/* --- terminate the FTAM association ------------------------- */
```

```
  initParamList( &param );
  param->service     = F_TERMINATE_RQ;
  (void) strcpy(param->paramMap, MAP_F_TERMINATE);
  if ( stateSend( param, &_state )  < 0 )        exiting( -1 );
  if ( stateRecv( &param, &_state, WAIT ) < 0 )  exiting( -1 );

out :

  END_FTAM();     /* -- kills the initiator process */
  exit( rc );

}
```

Appendix D

Grouping

The grouping control service allows certain sequences of service primitive requests to be concatenated. A complete grouped sequence enclosed by the F-BEGIN-GROUP and the F-END-GROUP primitives constitute a single protocol state transition. The primitives in a grouped sequence are syntactic segments of a single message communicated.

The permissible grouped sequences are defined below, using the following notation:

```
Notation
  (1) Square brackets, "[" and "]" --- indicate optional primitives
                                        within a sequence.

  (2) Vertical bar, "|"            --- indicate alternatives.

  (3) Round brackets, "(" and ")"  --- have normal algebraic significance.


 The valid groups are

   Group A  : F-BEGIN-GROUP
              (F-SELECT | F-CREATE)
              [F-READ-ATTRIB]
              [F-CHANGE-ATTRIB]
              F-OPEN
              F-END-GROUP
```

```
Group B  : F-BEGIN-GROUP
           F-CLOSE
           [F-READ-ATTRIB]
           [F-CHANGE-ATTRIB]
           (F-DESELECT | F-DELETE)
           F-END-GROUP

Group C  : F-BEGIN-GROUP
           (F-SELECT | F-CREATE)
           [F-READ-ATTRIB]
           [F-CHANGE-ATTRIB]
           (F-DESELECT | F-DELETE)
           F-END-GROUP

Group D  : F-BEGIN-GROUP
           (F-SELECT | F-CREATE)
           [F-READ-ATTRIB]
           [F-CHANGE-ATTRIB]
           F-END-GROUP

Group E  : F-BEGIN-GROUP
           [F-READ-ATTRIB]
           [F-CHANGE-ATTRIB]
           (F-DESELECT | F-DELETE)
           F-END-GROUP
```