(DRAFT) FILE ACCESS PROTOCOL

George D.M. Ross 03/12/85

Overview

The Computer Science Department's first network file server (The Filestore) was commissioned in 1976. Its purpose was to provide a central file storage and management capability for a collection of unintelligent clients, most of which had neither the resources nor the sophistication to perform such a task for themselves. The hardware consisted of an Interdata 70 with 32 Kbytes of core and one (latterly two) 67 Mbyte replaceable disc drive. The clients, which included a cluster of Interdata 74s, a PDP-9 and a PDP-15 which used the Filestore as a virtual DEC-tape server, and an ICL 7502 used as an editing station, communicated with the Filestore by means of point-to-point bi-directional serial links, each running at either 264 Kbaud or 1 Mbaud. In view of the restricted nature of many of the clients, the 1976 file access protocol was kept simple, with syntax being such that commands could be typed on the Filestore console on behalf of those clients which were unable to perform any filestore operations at all without assistance: the typical boot sequence for an Interdata 74, for example, consisted of keying in the so-called SO-sequence to run the AUTOLOAD instruction and then, from the console, instructing the Filestore to throw the primary bootstrap file down the link at the waiting Interdata.

Since then the nature of the client community has changed quite considerably, as has the Filestore hardware. Filestores are now based around 68000 processors; communication with clients is primarily over an ethernet-like LAN, though point-to-point RS232 connections are supported; and the storage medium is based around Winchester drives, either 19" 300 Mbyte or 5.25" 80 Mbyte. The clientele are also mostly 68000-based (around 60 personal work stations), though some Sirii use a Filestore as a virtual disc server. Although the file access protocol has changed slightly to accommodate, it remains essentially the same as was used in the original 1976 Filestore.

While there were still only a couple of Filestores on the network, with each client being primarily concerned with only one of these, the 1976 protocol remained viable, though restricting performance in a number of ways. The present situation on the Department's LAN, however, is that in addition to the Filestores there are other machines with their own file storage capability. Users typically have their files spread over several of these file systems. It is inconvenient for them to have to move their files around between file systems as they themselves move from machine to machine, and the process inevitably results in inconsistencies and wasted space as several versions of files are left on a number of file systems in turn. Ideally we would like users to be able to access their files from whichever machine they happen to be using, irrespective of the file system which holds them. This is the target of the File Access Protocol described here: machines with their own file systems may choose to export their file storage capability to the network, allowing users access to their files from whichever system they happen to be using at the time. In addition to the dedicated file servers, these would include a number of VMS, UNIX and MUSS based multi-access systems. The emphasis has moved away from providing basic facilities to a number of primitive clients towards integrating the various file systems, at the same time making each user's files accessible both to discless work stations and to larger, more conventional client systems.

General

The protocol assumes that the identity of the user is implied by the connection established between the communication entities in the client and server systems. This could be by means of the Network Authorisation Protocol, described elsewhere. The token obtained from the authorisation server would be presented to the file server by the client, thereby allowing the server to determine the identity of the user at the other end of the connection. Thereafter the identity of the user would be implied by the context of the connection. In any case, it is assumed here that the details are the concern of lower-level protocol layers, and that we are here presented with a connection to the client, with the user's identity implied by the connection. In ISO terms, the file access protocol is a level 6 protocol, with the users' identity being established at level 5.

As with the 1976 protocol, clients make a request to which the server responds. Unlike the old protocol, however, where clients were expected not to make a new request until they had received the response for the previous one, the new protocol allows clients to send several requests without waiting for responses (subject, of course, to flow control imposed by the transport medium). Each request is identified by a client-generated tag, which is not processed by the file server but is merely echoed in the response. This allows the client to match up responses with requests, and leaves the server at liberty to reorder requests at its own convenience (for example, to expidite disc head scheduling algorithms). In the case of requests which generate multiple responses each such response will receive the same tag; however each such response will be distinguishable from the rest since they will all differ in other fields (specifically the byte offset of the first item of data in each response).

Files have three aspects which must be handled by the protocol: they have names; they contain data; and they have miscellaneous attributes such as ownership and protection.

In order to gain access to a file we must know its name. The form of the name as it is known by the filing system will, of course, vary from system to system. The user of each client will expect to see the name in the same form as any other of the files accessible from that client, while the server will know the file by some equivalent name. Rather than there being a requirement that each client should know about every server's peculiarities, the protocol defines a canonical form of filename which each server is required to export and each server will expect to import. The filename, in the form that it is known by the client, is broken down into its component pieces using the client's normal rules; the component parts are transmitted as a sequence of counted strings; and the component parts are reassembled by the server in the form which its file system requires. Using counted strings means that the protocol imposes no restrictions on the valid characters which may constitute a filename. Of course, for a character to be usable it has to be acceptable both to the client and the server: for example, a filename with a '/' in it would probably not be easily accessible from a UNIX client. However, this is a problem for the user to resolve.

Having agreed on the file's name, the client and server must agree on the file's data content. Ideally the server would not need to interfere in any way with the data, but merely present them to the client on demand. However, in order that the data should be meaningful to users accessing the file system directly as well as to users accessing it via the File Access Protocol, the server may have to perform certain transformations on inbound and outbound data

to convert files into a canonical form. Two forms are defined here, though other forms could be defined as and when required:

> Raw: no interpretation is performed on the data. Bytes are read and written exactly as presented.
>
> Text: textual data is defined to consist of a sequence of (ASCII) characters, with line-breaks being indicated by the presence of a NewLine (Line Feed) character.

Note that for the current Filestores, their proposed replacements (see "The (Revised) (New) New Filestores"), UNIX, and VMS Stream-LF files the transformation required for textual files is, in fact, a no-op. Not all clients and servers will necessarily be able to access all types of files in all access modes: for example, servers which require to transform text files from some complicated internal form into canonical form might restrict clients to sequential access only to these files. In general, dedicated servers will probably not have to transform the data, while non-dedicated servers may need to.

The format and function of the various additional file attributes will vary greatly between file systems, with the result that the protocol will be able to do little beyond forwarding the user's request to the server for interpretation there. Attribute enquiry will be similarly constrained, though there may be common attributes whose values can be returned to the client in canonical form.

In order to access the data in a file the client must first open it in a suitable mode. Modes defined by the protocol are: read, modify, append, execute, though not all of these will necessarily be acceptable to the server when requested from an arbitrary client. The client can also specify the modes which it will regard as acceptable for other open requests for the same file, thereby enforcing a crude measure of locking; whether a new file should be created, or whether any existing file should be replaced; whether the file is going to be accessed sequentially or randomly; and the type of data conversion required (raw, text, etc.). In response the server will issue the client with a file token which can be used to refer to the file in future operations. The file token is valid universally, provided it has been used by the user to whom it has been issued. Thus one client can open a file on behalf of a user, with a different client performing operations subsequently on behalf of that same user. When a file is accessed from multiple clients in this way it is the clients' responsibility to arrange any synchronisation which may be required. The file token will be issued from a sufficiently large space that it is unlikely that clients will accidentally interfere, and in any case separate users are completely isolated from each other. When an already-existing file is opened the server's response will also include the size of the file in bytes, and an indication as to whether this is an exact size or merely an upper bound on the size.

Once the client has opened the file it is able to read from and write to it by issuing the appropriate request and quoting the file access token issued by the server in response to the open request. In the case of a read request the client indicates how many bytes it would like to receive and the offset of the first desired byte in the file (counting from zero). The server will usually attempt to satisfy a small request in a single response, but large requests may elicit multiple responses. Each response will contain the number of data bytes and the offset of the first of these data bytes in addition to the data themselves. In the case of a write request the client supplies the server with the offset of the first byte of data and the number of data bytes, together with the actual data to be written. Each write requests elicits a single response.

For a file opened for sequential access only, the client may also request that reading or writing should begin again at the first byte: in the case of writing this implies that any data which the file previously contained will be lost. Files are automatically extended by writing beyond the end of the already-existing data.

When the client has finished with the file it requests that the server should close it. It can request that the file be closed successfully, in which case the data in it become available for further access. Alternatively, the client can request that the file be closed unsuccessfully, in which case the data might not become available without the client issuing a subsequent request, or that the file should be deleted. The exact details of the way that multiple versions of files are handled are for the server's file system to determine.

In addition to opening the file and reading from and writing to it, there are several other operations which the client may request: the file may be deleted or renamed; it may be copied (by the server) to another file within the same file system; its data content may be (atomically) exchanged with that of another file; its other attributes may be queried; its protection may be modified; for a directory, its contents may be queried.

As well as the operations described above which will be common to all implementations of the file access protocol, though possibly in a restricted form in some cases, there are a number of operations which are specific to the particular file system. These might include: accrediting and discrediting users, and modifying their quotas; and mounting and dismounting components of the file system.

Formats

Unlike the 1976 protocol, the new file access protocol tries to keep fixed-size fields in a fixed place in requests and responses wherever this is reasonable, thereby lessening the work required to generate and interpret them. Fields are aligned on natural boundaries in order to accommodate those systems which are unable to perform unaligned transfers, either efficiently or at all. Numeric data are transmitted most significant byte first, in accordance with the network order of the lower protocol layers, with the result that processors which address things DEC-style may have to byte-swap these fields for transmission and after reception. Tokens and fields which are merely echoed need not be treated in this way, of course, while flag fields will require only a new set of actual values for the relevant symbolic constants.

The following definitions apply to the packet formats below: the basic unit is the byte (octet); a word consists of two bytes; a long consists of 4 bytes (2 words); a string is a length byte together with the appropriate quantity of data, plus a pad byte as required to make the total length of the string even; a filename is a sequence of strings, terminated by a null string (one of length zero).

All requests start with the same 4 byte preamble: a word for the client's request tag; a byte for the request type; and a byte for the request sub-type. Likewise, all responses have a 4 byte preamble: a word for the client's request tag (echoed from the request); and a word containing a success or failure indicator. This latter will be negative for failure, with the exact value being chosen by the server from a list of canonical errors as being closest to the actual error reported by the file system. Failure responses will include a textual message (a string) immediately following the status code. Success is indicated by a non-negative response; this will generally be zero, but some

"non-standard" cases will return a positive success code (for example, a redirect reponse from a (revised) (new) new filestore in reply to an open request). In any case, the exact format of the success response depends on the request which prompted it.

Request type 0: data access and transfer. Requests of this type enable reading of data from or writing of data to a file on the server's file system. The following sub-request types are defined:

Type 0: open file. Following the preamble comes a word to indicate the required access modes; then comes a word to indicate the access modes which will be regarded as compatible if used for other opens of the file; then a long to indicate, in the case of a file being created, the client's estimate of the file's eventual size (in bytes) when it is closed; then there is a string to indicate the data translation mode (one of "raw" or "text"); and finally the name of the file being accessed (a filename). Two forms of successful response are defined: normal success (status 0) -- the file has been opened in the requested mode, the file accesss token is returned to the client as a long immediately following the response preamble; while the size of the file (if positive, or an upper bound thereto if negative) follows that (another long); or, if the filename translates as the symbolic name of some object, this will be returned as a string following the response preamble, together with a status code of 1. Note that when checking compatibility of the request with other requests the server is at liberty to take a pessimistic view of the likely interaction of the client's request with the wishes of any previous client. The mode bits have the following meaning:

16_0001 execute mode
16_0002 read mode
16_0004 append mode
16_0008 modify mode
16_0080 sequential access only
16_0100 create this file if it does not exist
16_0200 create a new file (not allowed with the previous flag)
16_0400 temporary file -- no directory entry required (valid for file creation) -- delete on close
16_0800 generate a unique name for this file (not allowed with the previous flag)

Type 1: read data. The preamble is followed by a long containing the file access token issued when the file was opened; a long containing the byte offset of the first byte required; and a word containing the number of bytes requested. In response the server will send one or more packets as follows: the preamble will contain the client's tag and indicate success or failure; then, in the success case, a long containing the byte offset of the first byte of data; a word containing the number of bytes being returned in this response; and finally the data.

Type 2: write data. Following the preamble and the file access token (a long) the packet format is identical to a read response (offset, bytes, data). A successful response contains only the tag and status. Note that although the server may send multiple responses to a single read request, the client should not send multiple write requests to the server under the same tag as there will be no way to distinguish the responses.

Type 3: close file. Following the preamble comes the file access token (a long) issued when the file was opened; and an indication as to whether the close should be regarded as successful (a word containing a positive value), as unsuccessful but keeping the file (a word containing zero), or as indicating that the file should be deleted (a word containing a negative value).

Type 4: truncate. The preamble is followed by a file access token; then there is a long containing the new size of the file (in bytes).

Type 5: make accessible. An unsuccessful close may result in a file's contents being inaccessible. This sub-request allows subsequent requests to access the contents. Following the preamble comes the name of the file to be freed (a filename); then, optionally, a filename indicating what the freed up file should be called (or, alternatively, a null filename).

Request type 1: file attribute access. One of the sub-requests allows the client to enquire about a file's attributes; because different file systems have different ideas about file attributes, only those which are generally available can be queried using this canonical mechanism. In addition, one sub-request allows the client to send textual commands to the server and to receive textual responses, thereby allowing the client to set the file's attributes in the style to which the server is accustomed, as well as allowing for querying of the more exotic attributes.

Type 0: query canonical attributes. The preamble is followed by a long, the bits of which are set to indicate which attributes are required. These are returned as strings in the response in the order in which they appear in the request, from the low-order bit to the high-order. The request bits are as follows:

16_0001 file creation datestamp
16_0002 file modification datestamp
16_0004 file's owner
16_0008 owner's access
16_0010 world access
16_0020 this user's access
16_0040 file size (+ for actual, - for upper bound)

Type 128: exotic. Following the preamble comes the filename; then a sequence of strings, null terminated, each containing a textual command to the server. The format of the commands, and their interpretation, is entirely up to the server to define. In response the server will send a sequence of strings following the preamble, one per command. Note that a success indication implies only that the file was accessible; the response strings may contain error indications pertaining to the particular commands issued by the client.

Request type 2: directory access. Note that not all file servers will support the insertion and removal of arbitrary directory entries.

Type 0: list contents. The request preamble is followed by a filename indicating the directory of interest. In reply the server will send one or more responses containing a sequence of strings to indicate the names of the files in the directory.

Type 1: insert entry. Following the preamble comes the name of the entry to be inserted (a filename) and the translation which should

correspond to it (a string).

Type 2: remove entry. The preamble is followed by the name of the entry to be removed (a filename).

Request type 3: miscellaneous file operations.

Type 0: rename file. Following the preamble comes a flag word to indicate how the server should react if the file already exists; then follow two filenames, the first being the name by which the file is currently known, and the second the new name of the file. The only flag bit defined is 16_0001, which, when set, indicates that the server should return an error response if the target file exists; otherwise the target file will be superseded.

Type 1: copy file. The parameters are the same as for renaming. The file is copied internally by the server. In addition, flag bit 16_0002, when set, indicates that the server should not respond until the copy has been completed; otherwise the response will be generated as soon as the copy is found to be permissible.

Type 2: exchange files. The contents of the files, and any data-relevant attributes, are atomically exchanged. This allows database managers to commit transactions, for example. The exact mechanism whereby this is achieved is determined by the server, though the files' protection attributes would typically have to be set appropriately for both files.

Type 3: delete file. The only parameter is the name of the file to be deleted (a filename).

Request type 4: miscellaneous other operations.

Type 0: unique name. There are no parameters. The successful response is a string which the server guarantees will never be issued to another client, and which would be valid if used as a filename component.

Type 1: timestamp enquiry. There are no parameters. The value of the timestamp clock is returned as a string immediately following the response preamble.

Request types 128 to 255: reserved for server-specific functions. This is to allow servers to define request codes for operations which are only meaningful in the context of their own file system. In the case of dedicated servers, for example, a mechanism would be necessary to allow user accreditation, quota adjustment and so on, which are all functions which could best be performed by means of the standard mechanisms provided with, say, time-sharing systems.

Appendix: 1976 protocol request codes

The following are the commands defined by the "1976" filestore protocol, giving the code and parameters for each, and the response generated.

```
constinteger logon     = 'L' { 0   ownername, password     : Uno
constinteger logoff    = 'M' { Uno                         :
constinteger delete    = 'D' { Uno filename                :
constinteger rename    = 'B' { Uno filename, filename      :
constinteger permit    = 'E' { Uno filename, permissions   :
constinteger finfo     = 'F' { Uno ownername, file-number  : packet
constinteger ninfo     = 'N' { Uno filename                : packet
constinteger general   = 'G' { Uno                         : packet
constinteger pass      = 'P' { Uno password, username      :
constinteger quote     = 'Q' { Uno password                :
constinteger setdir    = 'J' { Uno ownername               :
constinteger copyfile  = 'O' { Uno filename, filename      :
constinteger readfile  = 'Z' { Uno filename                : ...file
constinteger openr     = 'S' { Uno filename                : Xno
constinteger openw     = 'T' { Uno filename                : Xno
constinteger openmod   = 'A' { Uno filename                : Xno
constinteger reset     = 'U' { Xno block-number            :
constinteger close     = 'K' { Xno                         :
constinteger uclose    = 'H' { Xno                         :
constinteger readsq    = 'X' { Xno                         : packet
constinteger writesq   = 'Y' { Xno ...packet               :
constinteger readda    = 'R' { Xno block-number            : packet
constinteger writeda   = 'W' { Xno block-number, ...packet :
constinteger readback  = 'I' { Xno                         : packet
constinteger dchange   = 'C' { Uno filename, date          :
constinteger fcomm     = ']' { Uno system command          : packet
constinteger new owner = '[' { Uno <p>ownername, quota     :
constinteger owners    = '\' { Uno partition number        : packet
constinteger new quota = '^' { Uno ownername, delta        :
```