THOUGHTS ON A FILE ACCESS PROTOCOL

George D.M. Ross 30/10/85

This document contains some preliminary thoughts on a new file access protocol to replace the current (1976) filestore protocol. Although it has proved to be reasonably flexible, the current protocol is deficient in functionality in a number of ways which would be difficult to make up due to its structure.

The intention of the new protocol is to provide a mechanism whereby systems attached to a network which have a local file storage capability are able to export that capability to other systems on the network. Exporting systems (servers) might include, for example, both dedicated file servers, whose sole purpose is to maintain a file system on behalf of their clients, and multi-access systems, for which file serving is a convenient way of making their users' work accessible to the rest of the network. There may, of course, be any number of each of these types of server on the network. Personal machines with their own discs might also choose to allow access by means of these protocols.

There is a 1976-protocol interpreter running on ECSVAX, for example, which provides a substantial subset of the facilities of the full protocol to its clients. The most commonly exercised of these are used by the EFTP program, which can transfer files between any 1976-spec filestores; in this case ECSVAX is treated as just another filestore. In the past the interpreter proved extremely useful when the current generation of filestores was being developed since the Interdata filestore was extremely poor at providing a reliable service to its ether-based clients (link-based clients were, as always, extremely well served); the solution adopted was to dispense with the services of the Interdata filestore and rely solely on ECSVAX as the file server until such time as the development became self-supporting.

The new protocol should be sufficiently flexible that clients and servers are not unduely inhibited in their dialogue, while at the same time hiding from the clients the more idiosyncratic parts of their servers' file systems.

The protocol should use a canonical way of referring to files, with each server being responsible for translating its own forms into canonical form and each client being responsible for translating the canonical form into the form most convenient to it. If I am using a VMS client I expect to find that filenames conform to the VMS file naming conventions, while if I am using a UNIX client I would expect to import files using UNIX-style file naming conventions, for example. Each filename referred to by a client is broken down into its constituent parts by that client and sent to the server as a list of these constituent parts. The server takes the constituent parts and reassembles them to suit its file system. In all cases the full filename is required, any notion of current default directory being maintained by the client.

Suppose I am using a UNIX system and I want to refer to a file on a VMS machine. On the UNIX system I would refer to the file as

```
<routing?>/u0/gdmr/el/eldriver.mar
```

which would be broken down and transmitted as


```
u0
gdmr
el
eldriver.mar
```

This would be reassembled at the VMS end as

```
U0:[GDMR.EL]ELDRIVER.MAR
```

As another example, suppose I am using VMS and I want to refer to a file on one of the dedicated filestores ("new new" variety). On the VMS system I might refer to the file as

```
DF:[STAFF.GDMR.TEXT]FAP.LAY
```

where "DF" (strictly, "DFA0") is the name of a VMS pseudo-device which would allow the interception of "normal" VMS file system QIOs and their redirection via a network access process. This request would be broken down into its component parts and sent to one of the file servers. The file server would then either access the file on behalf of the VMS process or it would redirect the file access to another file server.

As well as illustrating how filenames are broken down and reassembled, these examples also illustrate how different servers may reply in different ways to the same request. The new new filestores would have the ability to redirect a client if they knew where the request should really have been sent, while some other file servers might not have that ability. A file access request would thus have four possible replies: success; I don't know anything about that file; why don't you ask so-and-so for that file; and I don't know anything about that file and I don't even know whom you might ask. A non-redirecting server would only ever reply with one of the first two, while a redirecting server would always reply with success or one of the last two.

Since the protocol should be sufficiently flexible to refer to any file by its canonical name, we must consider the features of those file systems we expect to be exporting file storage capability. In the first instance these will be VMS, UNIX, the 1976-type filestores and the new new filestores.

VMS filenames consist of a sequence of alphabetic and/or numeric characters and/or '_' and/or '$'. Case is not significant. The filename is separated from the file "type" by a '.', with a limit on the length of each part. Files may exist in a number of versions, the version number being separated from the filename by a ';'.

UNIX filenames can consist of pretty well any ASCII code, combined in any order, though NUL could not normally be used as it is the string terminator, while '/' is the path component separator. Case is significant. There are no versions.

The 1976-style filestore's filenames consist of an owner part separated from the filename part by a ':'. The owner part can be from 1 to 6 characters long, while the filename part can be from 1 to 12 characters long. Valid characters are alphabetic, numeric, '$', '_', '.', '#'. Case is not significant. Files are created as "transient" and only become "ordinary" when they are closed successfuly, at which time they replace any previously-existing ordinary file of the same name. Files whose name begins with '$' are regarded as temporary, and are deleted when there are no references remaining to the directory which contains them (usually when their

owner logs off).

The intention with the new new filestores is that there should be as little restriction on filenames as possible. All printing characters will be legal in filenames. Case significance will be selected on a per-directory basis. Versions will be implemented in some form yet to be decided, as might be 1976-style transience.

In adddition to the filename components, there will also have to be a canonical way of specifying versions. The file server might refer to these by means of textual tags, or they might be by number, either absolute or relative to the most recent version. The protocol should allow for a way to express either form in a way that both clients and servers can use.

In order that the client and server can agree as to the interpretation to be placed on the filename in instances where there may be some ambiguity, some flags will require to be transmitted with each request by a client and each response by a server. Access request flags will include such things as automatic case-folding. Response flags will indicate such things as whether the request was succecssful or not, whether the server supports file access redirection, whether a higher version exists, whether a lower version exists, whether case-folding took place, etc.

When a file server opens a file on behalf of a client it returns a token to the client which can be used to refer to the file for future operations. Initially, this token is valid only if used from the same clients as was the network authorisation which the client quoted in order to identify itself to the server. The client should be able to request the server to restrict or enhance the scope of use of the token. Possible subsequent requests would include: read block, write block, close file, Uclose file (1976 terminology), mark for deletion.

Clients should have some means of enquiring about files' attributes and changing these attributes. Unfortunately these vary so much between different file systems that it may not be possible to express more than a few of these in canonical form. For the remainder, forwarding a textual command, as supplied by the user to suit the particular file system, would seem to be the only solution. Likewise, for file enquiry, there would at least be a client request which would return a list of all the files in a directory in canonical form, together with one which returned some of the information available about one particular (named) file in canonical form.

Appendix: Global Authorisation

Authorisation can't be combined with the storage of users' attributes unless we want to replicate the latter function. There will be more than one point of authorisation on the network, since there will be several "conventional" multi-access systems each of which is likely to want to do its own (terminal) authorisation. Any server on the network must be prepared to interrogate any of these authorisation points in order to find out who someone really is. (A "conventional" user of one of these multi-access systems is already identified and shouldn't have to "log in" again.) In exchange for identifying him/herself to an authorisation server the user receives a token which can be presented to any network server, thereby allowing the server to (a) decide which authorisation server issued the token, (b) decide if the network server trusts the authorisation server, and (c) ask the authorisation server who the token was issued to and where it may be used from. This scheme has the virtues that the user is only required to log on once in order to be acceptable to the entire network (assuming they logged on to a trusted host), and that clients aren't required to be trusted (most of them aren't!), only the authorisation servers.

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' { D   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        :
```