Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

The workings of clients and servers in WWW

Nigel Edwards

Abstract

This document is based on the author's experience of reading the source code to understand how CERN's httpd (World Wide Web Server) and NCSA's Mosaic (World Wide Web client or browser program) work. It is aimed at those who want to understand how these programs work. It should be regarded as a supplement rather than an alternative to CERN and NCSA's own documentation.

APM.1307.01 **Approved** 28th September 1994
Briefing Note

Distribution:
Supersedes: APM.1286
Superseded by:

The workings of clients and servers in WWW

The workings of clients and servers in WWW

Nigel Edwards

APM.1307.01

28th September 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

1 The workings of clients and servers in WWW

1.1 Overview

This document is based on the author's experience of reading the source code to understand how CERN's httpd (World Wide Web Server) and NCSA's Mosaic (World Wide Web client or browser program) work. It is aimed at those who want to understand how these programs work. It should be regarded as a supplement rather than an alternative to CERN and NCSA's own documentation: [BERNERS-LEEb], [LUOTONEN], [NCSA]. The particular releases described are: Mosaic 2.4 and CERN Httpd 3.0 (pre-release 6.0) configured to run on an HP 700 running HPUX 9.01.

The remainder of this paper is structured as follows. §1.2 introduces streams, one of the fundamental concepts used in libwww — the library used by both httpd and Mosaic. §1.3 describes how Mosaic uses this library to access hypertext objects in WWW. Finally, section §1.4 describes how CERN httpd receives and responds to requests for hypertext objects.

1.2 Streams and libwww

Streams [BERNERS-LEEc] are the fundamental data structures managed and manipulated by libwww. Streams are unidirectional objects which accept characters (and strings and blocks). The library has a generic representation of a stream class so that the interface is always the same for all types of different streams (HTStream Module). Streams can be stacked one on top of the other, each stream will process data flowing through it. This idea is used very effectively in both Mosaic and CERN httpd. Output streams are often referred to as sinks or targets.

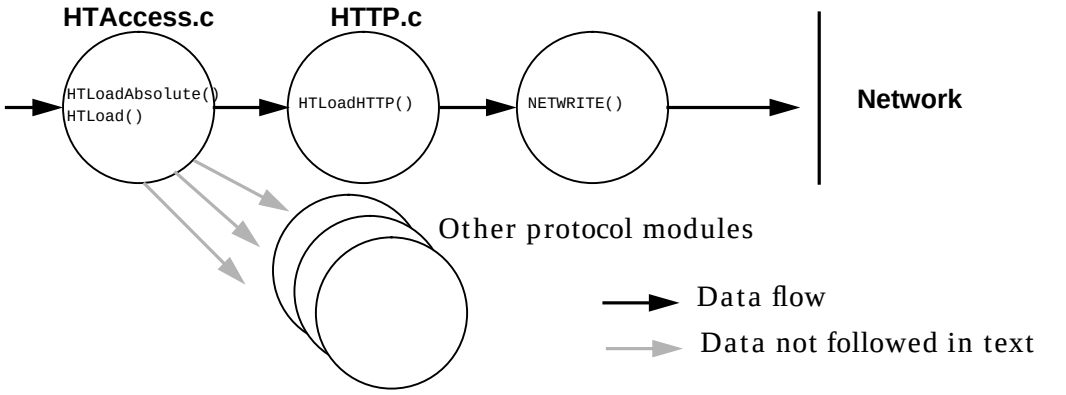
1.3 Mosaic

This section looks at how Mosaic drives libwww to access hypertext objects; it is not concerned with the architecture of the user interface. It describes the very simplest case, so it does not discuss more complicated interactions which can arise (e.g. the server redirects the request).

The data flow diagram in figure 1.1 shows how Mosaic requests a document from a WWW server. Once the user has selected a hyperlink (e.g. <URL:http://www.ansa.co.uk/>) entry to libwww is via `HTLoadAbsolute()` (module HTAccess.c). This eventually drops into `HTLoad()` which decides which protocol to use (information derived from the URL). In the case of a hypertext document the protocol is HTTP, so the load routine defined in the HTTP dispatch table is invoked. This routine corresponds to `HTLoadHTTP()` in module HTTP.c

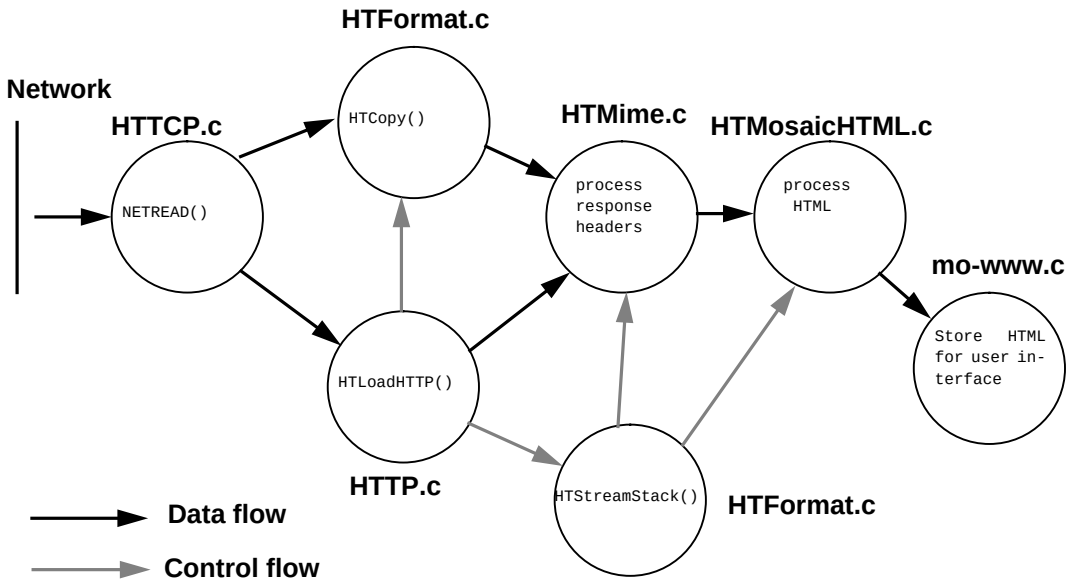
`HTLoadHHTP()` first constructs the request to send to the server. Amongst other things this indicates which method to invoke (e.g. GET) and also

Figure 1.1: Data flow diagram for a Mosaic request



typically “accept” strings to indicate the document formats the browser can deal with (see [BERNERS-LEEa] for details of these headers). The request is sent to the (remote) HTTP server by calling `NETWRITE()`. `NETWRITE()` is defined in the header file tcp.h, on most systems it corresponds to the routine write() provided in the system library (see the system manual page for socket).

Figure 1.2: Data flow diagram for server response



Mosaic is now ready for the response from the server: figure 1.2 shows the data flow for the response. First, `HTLoadHHTP()` calls `NETREAD()` to pull in the headers returned by the server[1]. Next the headers are checked to see what protocol the server is talking. Assuming it is HTTP1.0 [BERNERS-LEEa] and the server response status code is 200 (indicating success), `HTStreamStack()` (in module HTFormat.c) is called to create a stream down which to write the body of the response. This creates a MIME [BORENSTEIN 94] object which will accept a stream of bytes written to it (see HTMIME.c).

1. In Mosaic `NETREAD()` corresponds to `HTDoRead()` in HTTCP.c; in CERN httpd it defaults to the `read()` system call.

`HTLoadHTTP()` is unaware of the type of the stream object created — it is just given a pointer to a generic stream instance to which it writes the data (the remaining headers) using the `put_block()` provided by all stream objects. Finally it calls `HTCopy()` (in HTFormat.c) to pull the bulk of the data in from the HTTP server using `NETREAD()`. On returning from `HTCopy()`, `HTLoadHTTP()` cleans up and returns back up the calling stack eventually to hand control back to Mosaic's entry point.

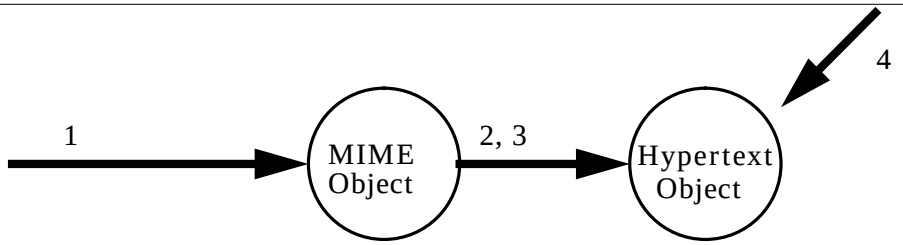
`HTCopy()` calls `NETREAD()` until there is no more data to be read from the socket connected to the server. After each call of `NETREAD()` it pushes the data into the output stream using `put_block()`.

The stream object processes its input stream to leave Mosaic an "html" object which it can display. The MIME object is responsible for parsing the (MIME) headers returned by the HTTP server.

The end of headers is denoted the receipt of a blank line. At this point the MIME object calls `HTStreamStack()` again creating another stream object passing it parameters about the incoming stream which have been obtained from the header files. In the case of an html file, an hypertext object is created (HTMosaicHTML.c). The MIME object the enters "transparent" mode and passes all data it receives unprocessed into the hypertext object, calling `HTMLMosaic_put_character()` via the object's dispatch table. This in turn calls routines which are defined in HText.h.

The routines defined in HText.h are callbacks: they are all implemented in the browser (Mosaic). So the MIME object parses the headers, initialises an internal hypertext object in the browser, and then pumps the html stream into this object. Hence when control is passed back to the browser an internal hypertext object has been created which it can display. The arrangement of stream objects is shown in figure 1.3

Figure 1.3: Cascading of stream objects in the browser



1. Stream pumped into MIME object which parses headers
2. MIME object creates hypertext object
3. MIME object becomes transparent, pumping unprocessed data into hypertext object
4. Hypertext object displayed by user interface

How does `HTStreamStack()` know what kind of stream object to create? When the browser is initialised routines in HTInit.c are called which tell `HTStreamStack()` what objects to create for a given type of input stream. The type of an input stream is determined by the MIME object when it parses the headers of the response. In addition the browser can set flags to force it to create a file object.

For example, if the html object contains a gif image, the browser may decide to retrieve this from the HTTP server. It goes through exactly the same steps as it did to retrieve the html file, until `HTStreamStack()` is called for the second

time. Then instead of creating a hypertext object, `HTStreamStack()` calls `HTSaveAndExecute()` to create a file writer stream (HTFWriter.c). Thus when the MIME object enters transparent mode it will pump data into a stream object which writes the data into a (temporary) file. The file is accessible to Mosaic when control is handed back to the user interface.

1.4 CERN httpd — a WWW server

This section attempts to explain how CERN httpd serves an html file. It assumes that the daemon is acting as a regular http server and not in either proxy or caching mode. It also assumes the simplest kind of interaction: no errors, no redirection etc. Finally it is assumed that the server is configured for unix in "standalone mode" (see HTDaemon.c). Figure 1.4 shows the flow of data in the server when it responds to a request.

The main loop of the server listens on its socket (procedure `stand_alone_server_loop()` in HTDaemon.c); each time a new request comes in on the socket, the server uses the accept() routine in the sockets library to create a new socket to handle the request. The server then forks a process: the child handles the request and the parent goes back to listen on the main socket again. The child calls `HTHandle(soc_number)` to handle the request; on return from `HTHandle()` the child exits.

On entry to `HTHandle()` a few new data structures are created to hold information about the request. Then `HTParseRequest()` in HTRequest.c is called; this parses the request and places the parameters in a newly allocated HTRequest object.

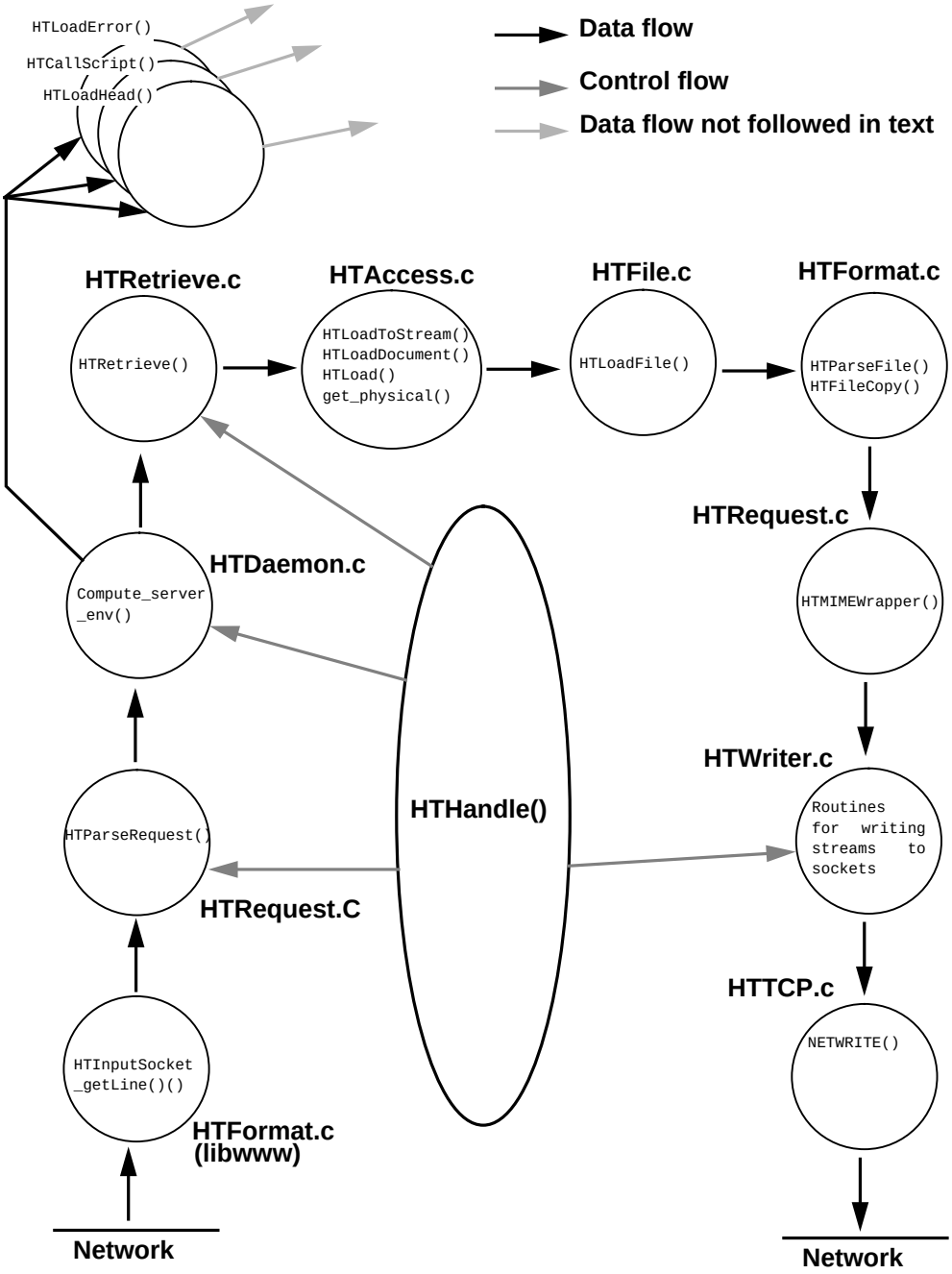
On entry to `HTParseRequest`() nothing has been read from the socket, the first thing it does is to read from the first line of the request from the socket. This is done by calling routines in the HTFormat.c module of libwww which provide input buffering for sockets, allowing input to be read line by line; in this case the routine called is `HTInputSocket_getLine()`.

The first line is assumed to contain the method, arguments and protocol of the incoming request (not protocol for HTTP0). If it is an HTTP1, request the server has to read more header information: the accept fields [BERNERS-LEEa]. These are parsed and dumped into the appropriate part of the HTRequest data structure. Immediately before return from `HTParseRequest`(), `compute_server_env()` is called (HTDaemon.c). This does further processing of the HTRequest object data structures. In particular, the argument(s) are parsed; the result is placed in HTReqArgPath if it is a pathname or HTReqArgKeyWords if it is an input to a script. Control is then returned to `HTHandle()`.

HTHandle() creates an output stream object in which to write the results of the request. The stream object is implements by HTWriter.c in libwww; it encapsulates the socket which the server will use to return a response to the client.

HTHandle() then checks for various errors such as attempts to invoke invalid or disabled methods; `HTLoadError()` is called to write an appropriate error into the output stream before the server exits. If no errors are detected, the server checks to see if it is authorised to serve the file and in the process determines the file's absolute pathname; again errors result in a call to `HTLoadError()` followed by `exit()`.

Figure 1.4: Data flow for incoming request in CERN httpd



Assuming that the daemon is not acting as a gateway or doing redirection, `HTHandle()` checks to see if the HTReqScript or HTReqArgKeywords variables have been set. If they have been set, it invokes `HTCallScript()` to execute the script. Otherwise it looks at the method field of the HTRequest object to see which method is being invoked. Invocation of `PUT`, `CHECKIN`, `POST` and `DELETE` methods are all treated as variants of scripts. `Invocation of the HEAD method` results in a call to `HTLoadHead()` (in HTLoad.c) which generates the reply headers (`HTReplyHeader()` in HTRequest.C) and writes them into the output stream before closing it.

If the method is `CHECKIN` or `GET`, `HTHandle()` first checks to see if the "HTIfModifiedSince" field has been set. If it has, and the document has not been modified since the given time, `HTLoadHead()` will be called to return the

headers only. Otherwise the whole document needs to be returned and `HTRetrieve()` is called.

Once the requested method has been executed `HTHandle()` returns control to the main loop which exits (recall the server was forked only to service a particular request).

1.4.1 HTRetrieve()

`HTRetrieve()` (see HTRetrieve.c) is called if a whole file needs to be returned by the server. First, it sets the global variable `HTImServer` to the absolute pathname of the file to be served. It then calls HTGetAttributes() (in module HTAAServ.c) which sets the content_length field by looking at the length of the file using `stat()`. `HTRetrieve()` then enters libwww calling `HTLoadToStream()` (in module HTAccess.c in libwww) on return from `HTLoadToStream()` the file has been written into the output stream and the document has been sent.

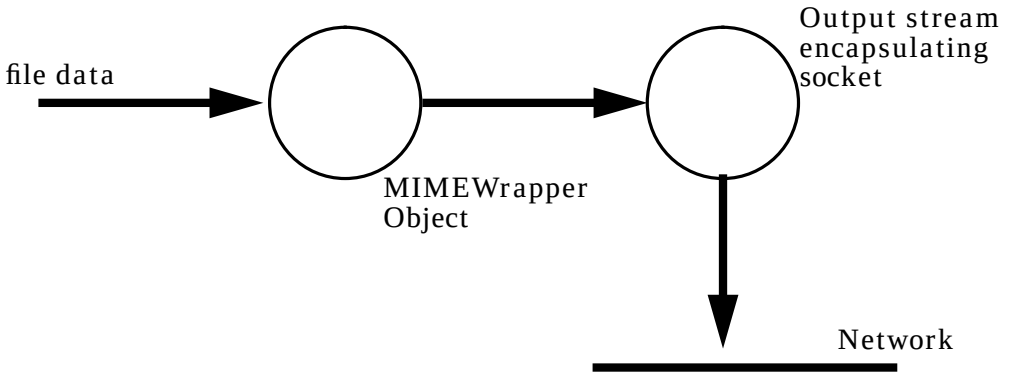
`HTLoadToStream()` first makes various calls to module HTAnchor.c. This is to locate the parent anchor object if the request is for a child anchor object. In the case of an html file the parent anchor object is the whole file, whereas the child anchor object is some point within that file: the server needs to return the whole file and allow the browser to choose the appropriate section. (More details about Anchors can be found in [BERNERS-LEE d].) `HTLoadToStream()` then calls `HTLoadDocument()` which in turn calls `HTLoad()`, assuming the server has not been set up to cache (both routines are in module HTAccess.c).

`HTLoad()` calls `get_physical()` (in HTAccess.c). This routine first checks to see if the document should be loaded via a gateway; assuming it is not `get_physical()` finds the protocol which will be used to load the returned object — in this case protocol “file”. `HTLoad()` then calls the load routine from the “file” dispatch table. This corresponds to `HTLoadfile()` in HTFile.c which opens the file and calls `HTParseFile()`.

`HTParseFile()` is in module HTFormat.c. On entry it calls `HTStreamStack()` to create another stream object on top of the existing output stream object. The object created is HTMIMEWrapper (`HTMIMEWrapper()` routine in HTRequest.c). This has been registered previously as a “converter” for the requested object type by `HTRequest()` when it parsed the client’s request. When `HTMIMEWrapper()` is called, it writes the HTTP headers into the output stream and returns a pointer to that stream. The architecture is shown in fig 1.5

Finally `HTParseFile()` calls `HTFileCopy()` in module HTFormat.c to copy the file into the output stream using the `put_block()` routine provided in HTWriter’s dispatch table. The request having been service, the daemon unwinds back through the calling hierarchy into `HTRetrieve()` then into `HTHandle()` and exits.

Figure 1.5: Cascading of streams in CERN httpd



References

[BERNERS-LEEa]

Tim Berners-Lee, “HTTP: A protocol for networked information”, internet draft, <URL:http://info.cern.ch/hypertext/WWW/Protocols/HTTP/HTTP2.html>

[BERNERS-LEEb]

Tim Berners-Lee, Jean-Francois Groff, Henrik Frystyk,“Common WWW Code Library”, <URL:http://info.cern.ch/hypertext/WWW/Library/Status.html>

[BERNERS-LEEc]

Tim Berners-Lee, Henrik Frystyk, “W3 Library Internals Overview”, <URL:http://info.cern.ch/hypertext/WWW/Library/Implementation/Overview.html>

[BERNERS-LEE d]

Tim Berners-Lee, “Anchors in the WWW architecture”, <URL:http://info.cern.ch/hypertext/WWW/Architecture/Anchors.html>

[BORENSTEIN 94]

N. Borenstein, N. Freed, “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies”, <URL:ftp://ds.internic.net/rfc/rfc822.txt>

[LUOTONEN]

Ari Luotonen, Henrik Frystyk, Tim Berners-Lee, “CERN httpd”, <URL:http://info.cern.ch/hypertext/WWW/Daemon/Status.html>

[NCSA]

National Center for Supercomputing Applications, “About NCSA Mosaic for the X Window System”, <URL:http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/help-about.html>