Towards a Uniform Library of Common Code

A Presentation of the CERN World-Wide Web Library

Henrik Frystyk, Håkon W. Lie, CERN

Note:

This is a slightly revised version of the paper submitted for the WWW Chicago Conference.

Henrik Frystyk, CERN October 24, 1994

Abstract

This paper describes the current status of the World-Wide Web Library of Common Code and the work done in order to start the convergence process towards a uniform Library. Currently many World-Wide Web applications are using different versions of the Library with a set of added functionality that is often either incompatible with the other versions or represents an overlap of code development. A new initiative has been taken at CERN in the context of the W3O collaboration with MIT in order to converge the current versions of the Library so that existing and future World-Wide Web applications have a powerful and uniform interface to the Internet.

1. Introduction

The CERN World-Wide Web Library of Common Code is a general code base that can be used as the basis for building World-Wide Web clients and servers. It allows software suppliers and researchers to achieve the state of the art technology and derive their own resources to pushing forward that technology. As an example, the CERN Line Mode Browser, the NeXTStep Editor and the HTTP Server (including the CERN Proxy Server) are all built on top of the Library. It contains code for accessing HTTP [3], FTP[7], Gopher[9], NNTP[8], and WAIS[10] servers, perform telnet sessions and access the local file system. Furthermore it provides functionality for loading, parsing and caching graphic objects plus a wide spectrum of generic programming utilities.

The development of the World-Wide Web Library of Common Code was started by Tim Berners-Lee in 1990. Ever since the code has been subject to changes due to modifications in the architectural model, additions of new features etc. This paper describes the current architecture in the Library and some of the ongoing projects of implementing new features and changing the architecture. The Library code is public domain together with the CERN Line Mode Browser and the CERN HTTP Server and it is available from [1].

2. Implementation and Portability Concerns

The Library is written in plain C and is especially designed to be used on a large set of different platforms. Currently it supports more than 15 Unix flavors and VMS. A year ago it also supported the MS-DOS, Macintosh, VM/CMS, and other platforms but this was with a limited set of the current functionality. Recently a new initiative has been taken at CERN as a result of the "First International Conference on the World-Wide Web" held in Geneva, Switzerland, May 1994. It involves the development teams from Lynx, Spyglass, NCSA, OmniWeb and others with special interest in the development of a uniform Library. The goal is to converge current versions of the Library into a single version which can support a broad spectrum of applications. Current topics of discussion and development involve:

- Re-porting the Library to MS-DOS and Macintosh
- Enhancing the current architecture of the Library—especially on defining a client API
- Adding new features such as security, enhanced error handling, multiple threads, format converters, protocol modules.

• Whether to use ANSI C and Posix standards and the possible portability problems this might involve

Especially the last topic is a practical problem as many platforms do not fully support the standards as defined, but only implements a subset of these. It is therefore a compromise between choosing the largest possible subset of the set of standardized functions with a minimal loss in portability. Unfortunately there is no simple answer to this problem.

3. Library Architecture

The general architecture of the Library as viewed from a client is illustrated in Figure 1. Servers and proxy servers have a slightly different view of the Library, but the client view is the most descriptive.

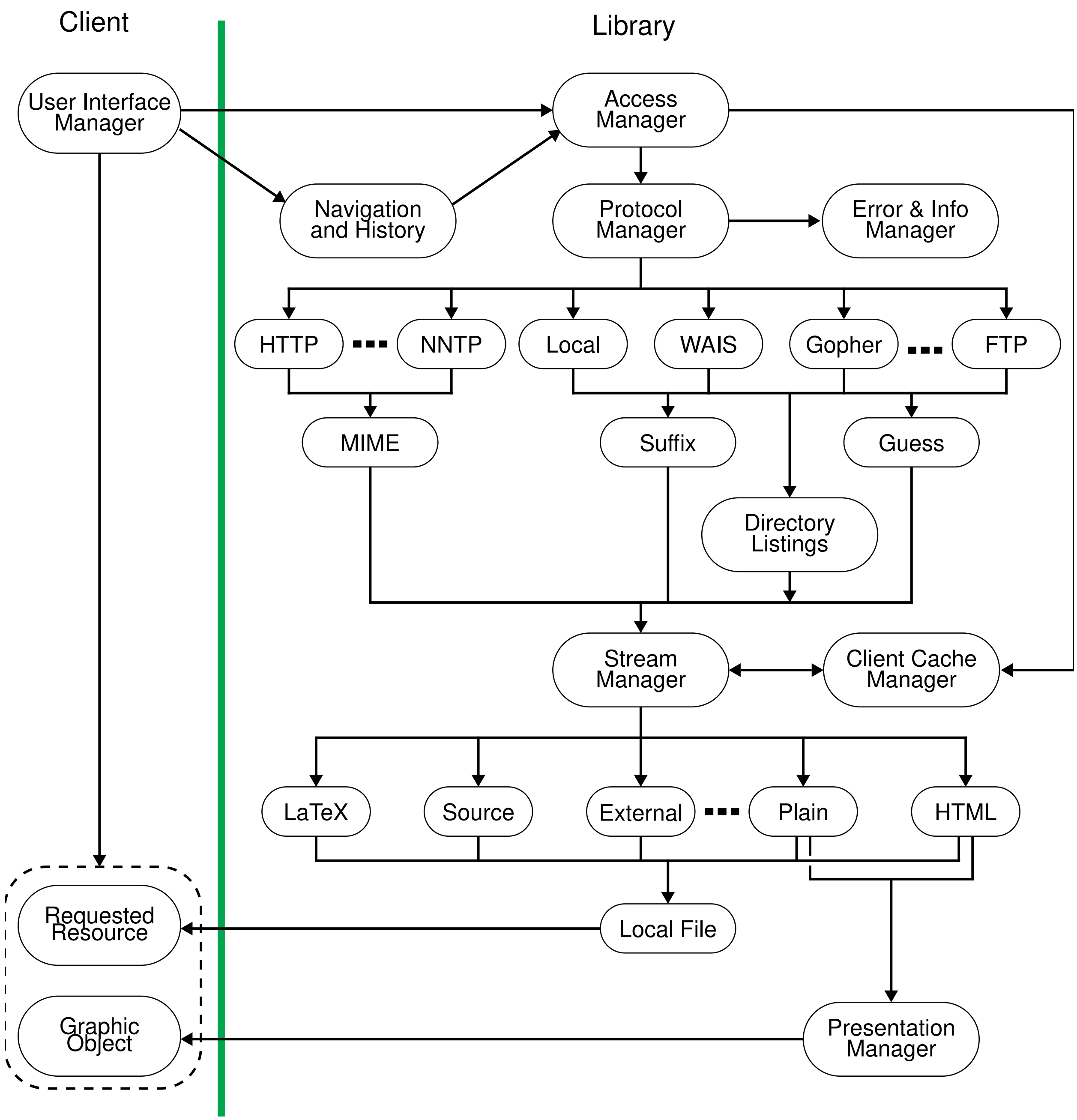

Figure 1 Control flow of the Library viewed from the Client

The flow of the Library shows that all network communication and parsing of data objects is handled internally. The client then has to present the information to the user. The main elements in the figure are explained below. A more detailed description of the implementation of the Library is given in [2].

Graphic Object: A graphic object is a displayable entity handled and maintained by the client. It is built from the data contained in a server response upon a successful request initiated by the user. The data can be of any format handled by the client. The object can either be built directly from the data, e.g., if the data object returned is an HTML[4] document, or it can be generated from a format converter within the Library. An example of the latter could be the generation of an HTML object from an FTP directory listing (7-bit ASCII). Graphic objects are in general coded differently on different GUI platforms. The graphic object is responsible for displaying itself, catching mouse clicks, and calling the Anchor Manager, either directly or via the History and Navigation Manager. Often the more common term "document" is used to describe the logical entity which a graphics object represents and displays. The client can keep the latest generated documents in memory in order to speed up back-tracking through already visited documents.

Stream Manager: The Stream Manager takes care of the transportation of streams of data from the Internet to the client and vice versa. This module is described in more detail in Section 4.

Anchor Manager: When a request is passed from the client to the Library it always has an anchor associated with it. An anchor is an object representing the URI of a either a graphic object or a fragment of a graphic object. It contains all information known about the URI. The Anchor Manager registers all requested anchors together with anchors found when parsing hypertext objects. The anchors are stored in a hash table so that multiple references to a URI all points to the same graphic object. It furthermore binds all registered anchors together corresponding to the logical binding between the related data objects. In other words, this module generates an internal model of the part of the Web the user has been in touch with. The actual bindings between anchors are described in more detail in Section 5.

Navigation and History: This module keeps track of all *visited* anchors. The set of visited anchors is always less than or equal to the total set of anchors registered in the Anchor Manager. Whereas the Anchor Manager manages a small part of the Web, this module keeps an ordered list of all visited anchors so the user can do quick navigation through a history list.

Client Cache Manager: This is a single-user disk cache module specifically for WWW Clients. It is used to save data objects once they have been downloaded from the Internet. The difference between a graphic object stored in memory and a cached data object is that the former is already parsed and can immediately be displayed to the user whereas the cached object must be parsed into a graphic object. The CERN proxy server has its own cache manager to handle a large scale cache that can serve hundreds of clients connected to it. The client cache is especially made for clients not using a proxy cache or having a very slow link but a large local temporary storage.

Protocol Manager: The Protocol Manager is invoked by the Anchor Manager in order to access a document not found in memory or in the cache (for the proxy server it is the proxy cache). Each protocol module is responsible for establishing the connection to the remote server (or the local file-system) and extract information using a specific access method. Depending on the protocol, the Protocol Module either builds a hypertext object itself from the server response, or it passes a socket descriptor to the Stream Manager for parsing by one of the parser modules.

Error Manager: This module manages an information stack which contains information of all errors occurred during the communication with a remote server or simply information about the current state. Using a stack for this kind of information provides the possibility of nested error messages like:

```
Fatal Error 500
Can't access document (ftp://info.cern.ch/foo.bar)
Reason: FTP-server replies: foo.bar: No such file or directory
```

Directory Listing Manager: This module handles long directory listings with icons for HTTP and FTP, and local file access. Long directory listings are supported for Unix, VMS, and Window NT.

3.1 Modularity of the Library

The Stream Manager and the Protocol Manager are both designed in a highly modular style in that they use pointers to functions when they decide on what protocol or parser to use respectively. For the Protocol Manager, the actual binding between an access scheme specified in the URL [5] and the protocol module used is done in a separate protocol structure which can be setup at run-time. Likewise for the Stream Manager where the binding is based on MIME-types

[6], either found directly in the response code from the remote server or by guessing. This model makes it very easy to install new stream converters and protocol modules.

4. The Stream Concept

Streams are the main method used in the Library for transporting data from the network to the client or vice versa, and they do therefore deserve a more thorough presentation.

4.1 What is a Stream

A stream is an object which accepts sequences of characters. It is a destination of data which can be thought of much like an output stream in C++ or an ANSI file stream for writing data to a disk or another pheripheral device. The Library defines a generic stream class with five methods as illustrated in Figure 2. The output is also a stream and is often referred to as the “target” or “sink”. This class is a superclass of all other stream classes in the Library and it provides a uniform interface to all stream objects regardless of what stream sub-class they originate from.

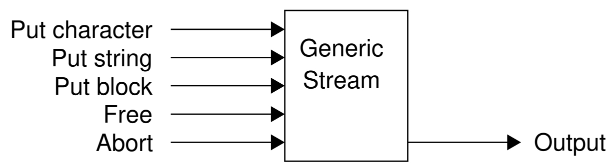

Figure 2 The generic stream class

Streams can be cascaded so that one stream writes into another using one or more of the methods shown in Figure 2. This means that a required processing of data, for example reading data from the Internet can be done as the total effect of several cascaded streams. The Library currently includes a large set of specific stream modules for writing to an ANSI C-file structure, writing to a socket, stripping Carriage Returns, splitting a stream into two (used for caching) etc.

The stream-based architecture allows the Library (and hence applications built on top of it) to be event-driven in the sense that when input arrives, it is put into a stream, and any necessary actions then cascade off this event. An event can either be data arriving from the Internet, or data arriving from the client application. The latter would be the case when the client is posting a data object to a remote server.

4.2 Structured Streams

A structured stream is a subclass of a stream, but instead of just accepting data, it also accepts the SGML “begin element”, “end element”, and “put entity” as illustrated in Figure 3

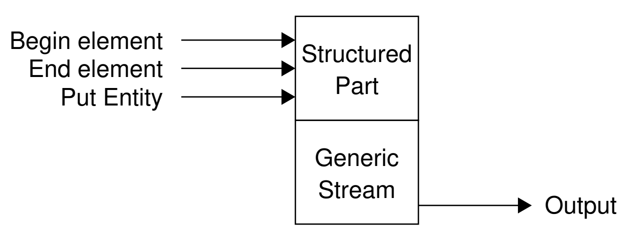

Figure 3 The generic structured stream class

A structured stream therefore represents a structured document and can be thought of as the output from an SGML parser. It is more efficient for modules which generate hypertext objects to output a structured stream than to output SGML which is then parsed. The elements and entities in the stream are referred to by numbers, rather than strings. A DTD contains the mapping between element names and numbers, so each instance of a structured stream is associated with a corresponding DTD. The only DTD which is currently in the Library is an extended version of the HTML DTD level 1, but current work is done to update this to comply with the emerging HTML level 2.

4.3 Format Conversion Using the Stream Stack

The stream stack is used to select the most appropriate converter from the input format given by the protocol modules and the desired output format specified by the client. A converter is simply a stream which is registered as a converter from a given input MIME-type to a output MIME-type. If more than one converter is capable of doing the same conversion a quality factor is used to (subjectively) distinguish which one is the best. Currently the stream stack module only manages a single converter at the time, for example from *text/plain* to *text/html*, that is, the size of the stack is always 1. However, work is being done to expand the stack so that several converters can be cascaded in order to obtain the desired conversion.

5. Anchors

Anchors represent any references to graphic objects which may be the sources or destinations of hypertext links. There are basically two types of anchors: *parent* anchors which represent whole graphic objects and *child* anchors which represent parts of a graphic object. As mentioned in Section 3, every request and hence every graphic object has a parent anchor associated with it. Anchors exist throughout the lifetime of the client, but as this generally is not the case for graphic objects, it is possible to have a parent anchor without a graphic object. If the data object is stored in the client cache, the parent anchor contains a link to it so that the client can access it through the Cache Manager.

Both types of anchors are subclasses of a generic anchor class which defines a set of outgoing links to where the anchor points. The relationship between parent anchors and child anchors is illustrated in Figure 4.

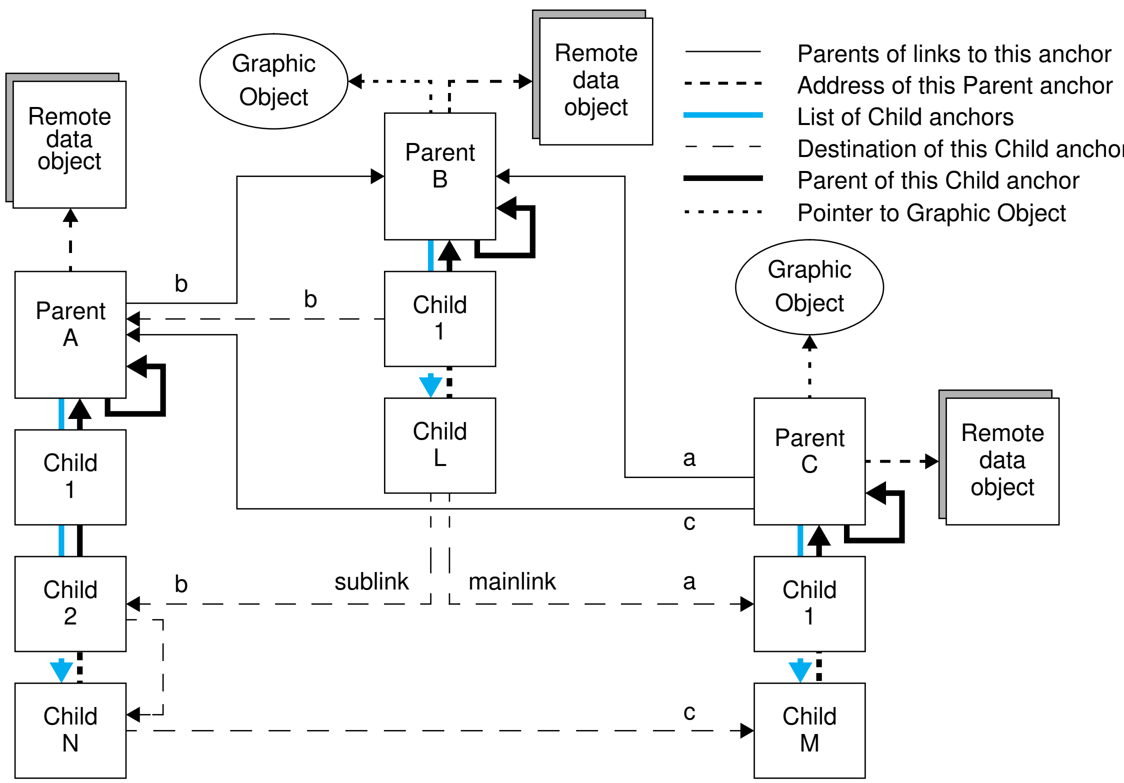

Figure 4 Relationship between *parent* and *child* anchors

Every parent anchor points to a remote data object. In the case of posting an anchor to a remote server, the data object is yet to be created. The client can assign a URI for the object but it might be overwritten (or completely denied) by the server. In Figure 4, parent **A** has no associated graphic object. This can either be because the anchor has not yet been requested by the user or the graphic object has been discarded from memory.

When child **B1** is created, pointing to parent **A**, parent **B** is registered in parent **A** as pointing to **A**. The same is the case for the link between child **BL** and **A2**, but parent **B** is only registered once in parent **A** (this is marked with **b** in the figure). The same is the case for the links marked **a** and **c**. A child can have more than one link to other anchors as indicated in child **BL**. This is often the case using the POST method, where for example the same data object is to posted to a News group, a mailing list and a HTTP server.

6. Implementation of the HTTP Module

The HTTP client is based on the HTTP 1.0 specification but is backwards compatible with version 0.9. The major difference between this and previous implementations is that this version is a state machine based on the state diagram illustrated in Figure 5. As will be discussed in Section 8, the state-machine design has some inherent advantages which makes it suitable in a multi-threaded environment, even though the HTTP protocol is stateless by nature.

.

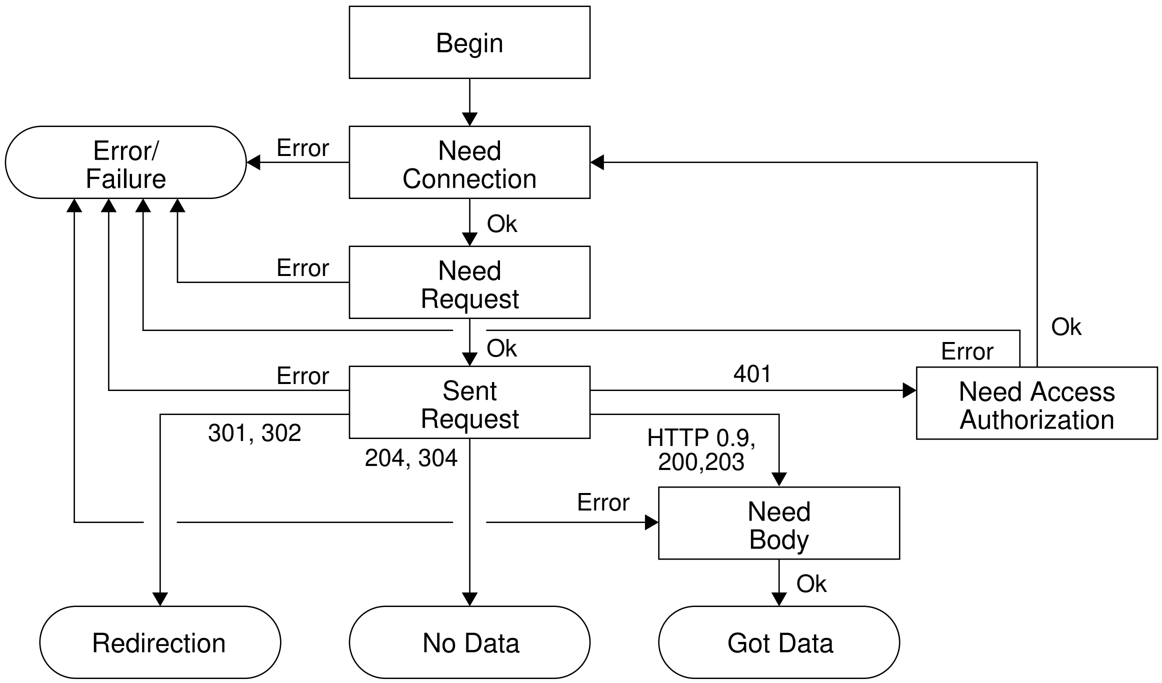

Figure 5 State Diagram of the Client part of the HTTP Protocol

The individual states and the transitions between them are explained below.

Begin: This state is the initial state where the HTTP module sets up the required data structures needed for handling the request.

Need Connection: The HTTP client is ready for setting up a connection to the remote host. The connection is always initiated by a *connect* system call. This procedure has been optimized by using a cache of visited host names as will be explained in Section 7.

Need Request: After connection establishment, the module sends the HTTP request to the remote server. The request consists of an HTTP header line, a set of MIME Headers, and possibly a data object to be posted to the server. Current methods supported in the module are GET and HEAD, but support for PUT and POST is under development. The supported MIME-headers are:

- *Accept:* The current implementation uses one accept line for each MIME-type supported by the client. For advanced clients this means that the “Accept:” sequence is repeated 20-30 times which gives a overhead of 200-300 bytes per request (including the CRLF telnet EOL-sequence). This should be changed so that either a comma separated list is transmitted instead or only the MIME content types without any subtypes.
- *Referer:* If any parent anchor is known to the requested URI it is sent in the referer field. This is to let the server know what link has led to the current request. Nothing is sent if the parent anchor is unknown or does not exist.
- *From:* The full email address is sent along the request. It is meant as an informative service to the recipient and can be changed to any value the user wishes to sign the request with. As it is possible to manipulate the email address, this field can not be used as identity verification for access authentication.

- *User-Agent:* The user agent is by many clients currently generated in a somewhat verbose format. The goal is to make this field machine readable so it can be used on the server side to perform individual actions as a function of the client implementation. As a side effect it can also be used for statistics etc.
- *Authorization:* If the user already has typed in a user ID and a password for access authorization in a previous HTTP request the HTTP client also transmits the authorization MIME-header. However, this does not guarantee that the user ID and password are sufficient for actually accessing the requested resource.

Sent Request: The client waits until a response is given from the server or the connection is timed out in case of an error situation. The client is capable of differentiating HTTP 0.9 responses from HTTP 1.0 responses.

Need Access Authorization: If a 401 Unauthorized status code is returned, the module prompts the user for a user ID and a password. The connection is closed before the user is actually asked, so any new request initiated upon a 401 status code causes a new connection to be established.

Redirection: The module supports both a temporarily (302) and a permanent (301) redirection code returned from the server. In both cases, the connection is closed and the Protocol Manager is called recursively until a document is found, an error has occurred, or the maximum number of redirections has been reached. The URI returned in the redirection response from the remote server is parsed back to the client via the Error and Information Manager. If it is a 301 code, an intelligent client can use this information to change the document in which the redirected URI originates.

Need Body: If a body is included in the response from the server, the module reads the data from the network and direct it to the Stream Manager which sets up a stream stack to handle the desired format conversion.

Got Data: When the data object has been loaded from the Internet, the module terminates the request and handles control back to the client.

Error or Failure: If at any point in the request handling an error occurs the request is aborted and the connection closed. All information about the error is parsed back to the client via the Error and Information Manager. As the HTTP protocol is stateless, all errors are fatal between the server and the client. If the erroneous request is to be corrected, the state machine jumps back to the initial state.

7. Cache of Host Names

As excessive communication with Domain Name Servers (DNS) can produce a significant time-overhead, a new memory cache of host names have been implemented to limit the amount of requests to DNS. Once a host name has been resolved into an IP-address, it is stored in the cache. The entry stays in the cache until an error occurs when connecting to the remote host or it is removed during garbage collection.

Multi-homed hosts are treated specially as all available IP-addresses returned from DNS are stored in the cache. Every time a request is made to the host, the time-to-connect is measured and a weight function is calculated to indicate how fast the IP-address was. The weight function used is

$$W(n+1) = \begin{cases} W(n) + (1-\alpha) \cdot \Delta & \text{if active IP-address} \\ 0.9 \cdot W(n) & \text{else} \end{cases}$$

where α indicates the sensitivity of the function and Δ is the connect time. If one IP-address is not reachable a penalty of x seconds is added to the weight where the penalty is a function of the error returned from the “connect” call. The next time a request is initiated to the remote host, the IP-address with the smallest weight is used.

A problem with both the host cache and the document cache (either on server side or client side) is to detect when two URLs are equivalent. The only way this can be done internally in the Library is to canonicalize the URLs before they are compared. This has for some time been done by looking at the path segment of the URLs and remove redundant information by converting URLs like

foo/./bar/ = foo/redundant/../bar/ = foo/bar/

The method is now optimized and expanded so that also host names are canonicalized. Hence the following URLs are all recognized to be identical:

http://info/ = http://info.cern.ch:80/ = http://INFO.CeRn.CH/ = http://info.cern.ch./ = http://info.cern.ch/

However, the canonicalization does not recognize alias host names which would require that this information is stored in the cache.

8. Support for Multi-threaded I/O Operation

In a single-process, single-threaded environment all requests to, e.g., the I/O interface traditionally block any further processing. However, a combination of a multi-process or multi-threaded implementation makes provision for the user to request several independent URIs at the same time without getting blocked by slow I/O operations. As a World-Wide Web client is expected to use much of the execution time doing I/O operations such as "connect" and "read", a high degree of optimization can be obtained if multiple threads can run at the same time. This section describes the current implementation of multiple threads in the HTTP Module. Later it is expected that other protocol modules are added. The NNTP, Gopher, and FTP module have all been rewritten as state machines so they can with minor changes be included in the event-loop.

8.1 Platform Independent Implementation

The major concern in the design of the multi-threaded Library has been to make a platform independent implementation which excludes use of traditional thread packages like DECthreads. IEEE has publicized the POSIX standard 1003.4 for multi-threaded programming but even this will eventually limit the portability referring to the discussion in Section 2. Instead, the multi-threaded functionality of the HTTP client has been designed to be used in a single-processor, single-threaded environment as illustrated in Figure 6

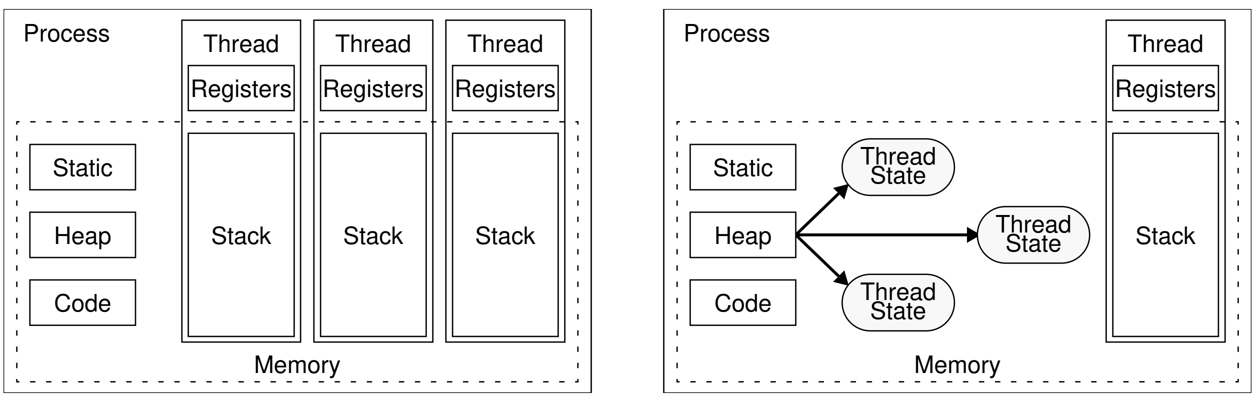

Figure 6 *Left:* Traditional threads and *right:* Model used in the Library

The difference between this technique and "traditional" threads is that all information about a thread is stored in a data object which exists throughout the lifetime of the thread. This implies that the following rules must be kept regarding memory management:

- Global variables can be used only if they at all time are independent of the current state of the active thread.
- Automatic variables can be used only if they are initialized on every entry to the function and stay state independent of the current thread throughout their lifetime.
- All information necessary for completing a thread must be kept in an autonomous data object that is passed round the control flow via the stack.

These rules makes it possible to imply a multi-threaded data model using only one stack without causing portability problems as it is all done in plain C.

8.2 Modes of Operation

In order to keep the functionality of the Library as general as possible, three different modes of operation are implemented:

Base Mode: This mode is strictly single-threaded. The difference between this mode and the other two is that all sockets are made blocking instead of non-blocking. The mode preserves compatibility with World-Wide Web applications with a single-threaded approach. Currently this mode does not provide interruptible I/O as this is a integral part of the event-loop.

Active Mode: In this mode the event-loop is placed in the Library. This mode is for dumb terminal clients which can interrupt the execution through the keyboard. The client can, however, still be multi-threaded in the sense that it can actively pre-fetch documents not yet requested by the user. If a key is hit, the Library calls a call-back function in the client so the client can decide whether the current operation should be interrupted or not. If so, the Library stops all I/O activity and handles the execution back to the client. The active mode should cause only minor changes to the client in order to obtain a simple form of multiple threads and interruptible I/O.

Passive mode: This mode requires the most advanced clients, for example GUI clients. The event-loop is now placed outside the Library. On every URI request from the client, the Library initiates the connection and as soon as it normally would block, it returns a list of active socket descriptors to the client. When the client is notified about an event on a socket, it calls a Library socket-handler passing the socket number. Then the socket handler finds the corresponding request and starts the thread. As soon as the thread would perform a blocking I/O operation, the socket handler stops the thread and returns the execution to the client event-loop.

8.3 Control Flow

A consequence of having multiple threads in the Library is that the control flow changes to be event driven where any action is initiated by an event either caused by the client or the network interface. However, as the current implementation of multiple threads is valid for HTTP access only, the control flow of the Library from Figure 1 has been preserved but with the addition of an event-loop in the HTTP module. All other access schemes still use blocking I/O and the user will not notice any difference from the current implementation. The result of this is that full multi-threaded functionality is enabled only if the client uses consecutive HTTP requests. The internal event-loop is based on call-back functions and events on a set of registered socket descriptors as illustrated in Figure 7.

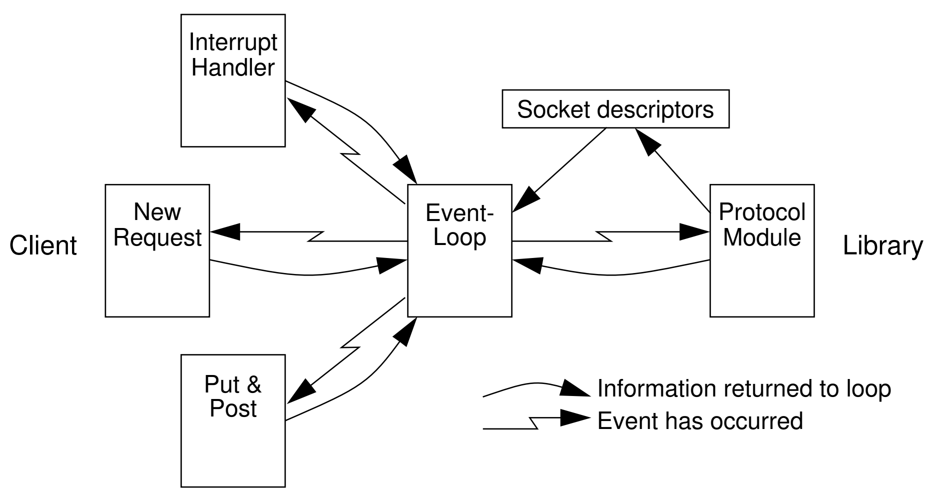

Figure 7 Event-loop based on call-back functions

The event-loop handles two kinds of call back functions: Ones that are internal Library functions such as the specific protocol modules, and the ones that require an action taken by the client application.

8.4 Interrupting a HTTP Request

The interrupt handler implemented for *active mode* is non-eager as it is a part of the select function in the socket event-loop. That is, an interrupt through standard input is caught when the executing thread is about to execute a blocking I/O operation such as *read* from the Internet and execution is handled back to the event-loop. The reason for this is that the user is not blocked even though the interrupt does not get caught right away so it is not as critical as in a single-threaded environment. In *passive mode* the client has complete control over when to catch interrupts from the user and also how and when to handle them.

9. Conclusions and Future Developments

The Library of Common Code has recently gone through a phase of heavy code development and set of new features are either being developed or is already available in the current version[1]. However, in order to have any effect on the general evolution of the World-Wide Web project, much work must still be put into the development and maintenance of the code. Furthermore it is vital that World-Wide Web developers see the Library as a powerful tool that offers a wide spectrum of functionality relevant to World-Wide Web applications. The functionality must span fundamental World-Wide Web features as well as experimental implementations such as gateways and format converters. We believe that the only way this can be achieved is by cooperation between the developers and the WWW-team at CERN which has the responsibility of organizing and synchronizing development on the Library. Therefore, the goal of this paper is to show the current state of the Library and at the same time to invite interested parties to join the working group for future developments.

Some of the plans for new features are

- Security encryption and authentication
- New MIME-parser which replaces a large number of small sub-parsers
- Implementing general support for Put and Post methods for HTTP, NNTP, and SMTP
- Implementation of forms support
- Integrating a WhoIs++ module as a new URI access scheme
- Gateway to a Hyper-G server as a new access scheme

Acknowledgments

We would like to thank Tim Berners-Lee and the large group of contributors for having started the work on the World-Wide Web Library of Common Code. We would also like to thank the current working group for inspiration and attributing code so that new features are integrated into the Library.

This work has been partly sponsored by the Norwegian research Council.

References

[1] http://info.cern.ch/hypertext/WWW/Library/Status.html, *Status of the Library of Common Code*
[2] http://info.cern.ch/hypertext/WWW/Library/Implementation/Overview.html, *Library Internals and Programmer's Guide*
[3] http://info.cern.ch/hypertext/WWW/Protocols/HTTP/HTTP2.html, *HTTP: A Protocol for Networked Information*
[4] http://info.cern.ch/hypertext/WWW/MarkUp/MarkUp.html, *Hypertext Markup Language (HTML): Working and background Material*
[5] http://info.cern.ch/hypertext/WWW/Addressing/Addressing.html, *UR* and the Names and Addresses of WWW Objects*
[6] http://info.internet.isi.edu/0/in-notes/rfc/files/rfc1521.txt
[7] http://info.internet.isi.edu/0/in-notes/rfc/files/rfc765.txt
[8] http://info.internet.isi.edu/0/in-notes/rfc/files/rfc977.txt
[9] http://info.internet.isi.edu/0/in-notes/rfc/files/rfc1436.txt
[10] http://server.wais.com/z3950.html, *WAIS Network Publishing using Z3950*

Authors

Henrik Frystyk Nielsen, frystyk@info.cern.ch: Joined the World-Wide Web Team at CERN, February 1994. He completed his MSc degree as Engineer of Telecommunications at Aalborg University, Denmark, in august 1994. Henrik is working in the CN division as a code developer in the World-Wide Web team. His research interests are enhanced network protocols and communications systems. Henrik is currently responsible for the World-Wide Web Library of Common Code and the Line Mode Browser.

Håkon W. Lie, howcome@info.cern.ch: Håkon is currently a Scientific Associate with the WWW project at CERN while on leave from Norwegian Telecom Research where he holds a Research Scientist Position. At CERN he works on the client side of the Library. He holds an MS from Massachusetts Institute of Technology where he worked in the Electronic Publishing group of the MIT Media Lab.

Authors' present Address: CERN, 1211 Geneva 23, Switzerland