A HTTP Primer

A HTTP beginner's guide

Introduction

HTTP (short for Hyper Text Transfer Protocol) is the tongue that web servers and browsers speak. As such it is the foundation of the World Wide Web, and a fundamental piece of the Internet as we know it.

Internet standards are proposed, discussed and, eventually, specified in documents known as RFCs (Request For Comments) and HTTP is no exception: its latest incarnation, Rev. 1.1, is described in RFC 2616 (*"Hypertext Transfer Protocol -- HTTP/1.1", by Fielding et al.*)

As it happens, most RFCs are rather dry documents, that require a fair amount of technical knowledge to be fully comprehended. Besides, they usually refer to each other: the reading of RFC2616, for instance, requires a working knowledge of the specifications for HTML, MIME, and URIs, all of which are explainded in painstaking detail in one or more RFCs of their own.

Hence the reason for this document, in which I will try to give to my readers (if any) a *bird's eye view* of the basic components and internal workings of HTTP. I have, of course, no pretense to replace or supplement Request For Comments, to which everybody wanting to achieve a deeper understanding of the issues should ultimately refer. The IETF maintains a comprehensive catalog of these documents.

URLs, URIs and all that

The whole purpose of the web is to allow people to easily locate documents and information on the Internet, something that became rather hard to do at the beginning of the 1990s, as the size of the Internet began to increase. The net is a jumble of computers and protocols, and getting to a particular piece of information requires the knowledge of quite a bit of ancillary trivia (sometimes called metadata): on which computer it sits and where, the protocol that must be used to retrieve it, etc.

Until Tim Berners-Lee turned around and invented the Web, there was no agreed upon way to describe sinthetically all this (although the University of Minnesota "gopher" system was going in the same direction: but, since they kept it proprietary, and did not have HTML to go with it, they basically blew it, and gopher is today a protocol on its way to extinction).

Berners-Lee and his colleagues at CERN invented the idea of Universal Resource: a string which described an internet "thing" and could be parsed by a program that presumably knew how to retrieve it. The most general syntax of this string

(specified with a BNF grammar in RFC 1630) can be complicated, but it often looks like this:

scheme:	//host	[:port]	/path/to/resource	[?query]
Protocol identifier, e.g. http, ftp, news	*FQDN of the host: e.g www.linux.org*	*Optional port number of the service: e.g 80*	*Location of the resource, as known to the server*	*Optional context information*

These universal addresses are collectively known as Universal Resource Identifiers or URIs: it is sometimes necessary to distinguish them further in Universal Resource Locators or URLs, and Universal Resource Names or URNs - these refer to "entities" that can be located by name, but do not really have a "location". Of course, unless you spent the last six years hiding in a cave, you know what a URL is.

Berner-Lee's group then devised HTTP as a new, simple protocol for serving documents written in a new format they had invented (and which I will not describe here: go read RFC 1866 if you want to know much more about HTML). Servers speaking HTTP would directly handle URIs to be retrieved with the "http:" schema: the world wide web.

What's MIME got to do with it?

As good a thing as URIs are, they say nothing about the content of the resource that is going to be retrieved, because this is something that is (a)subject to change and (b)hard to infer from the outside. While it can be speculated - usually with reason - that a path ending in ".txt" will contain textual information, guessing what might come out of a URL pointing to, say, a database query is much harder. Besides, the idea that a Chinese has of textual information is rather different from that of a Western European.

So, HTTP clients have to trust the server to supply information on how the content has to be treated upon delivery. Born to address a separate problem, the Multipurpose Internet Mail Extension (MIME) standard, described in RFC 2045 and RFC 2046, has much - even too much - to say about character sets, data types and encodings. Since it was already there, it was (partially) adopted to describe the way in which the retrieved content was to be handled.

HTTP: the works

We are now ready to describe the general phases of what a general HTTP interaction looks like.

1. Armed of a http: URL the client contacts the server over the network, and sends a properly formatted request describing sinthetically what needs to be done, any additional information needed to carry out the request, and the

context in which the results will be handled. All this is packaged in chunks called the request header and the request body

2. The server parses the request, if possible carries it out, and reacts by returning wether the operation was successful, any meta-information regarding the result data (if any), and the content that was retrieved as result of the transaction. This is formatted and packaged in chunks known as the server status, the response headers and the response body

3. The client-server connection is closed.

HTTP is stateless

This is rather simple stuff: ask a question, get an answer, now go away. In fact, the statelessness built into the HTTP protocol, constitutes both its strength and its weakness. By requiring that all transactions be self contained, HTTP makes for very simple clients and servers, while at the same time being very robust with respect to the unreliable nature of Internet communications.

On the other hand, the type of involved multistep transactions, that are often required when retrieving anything more complex than a single document, are difficult to achieve through HTTP alone, as programmers writing online shopping systems know all too well. Besides, multiple requests originating from the same client to the same server required separate connections anyway, which are a drain on the resources of very busy servers.

In truth, it must be said that statelessness is inherent to the the client browsing paradigm that is the foundation of the web. HTTP, as a protocol, merely certifies it. A user is free to initiate a a complex database transaction, get tired of it while she is smack in its middle, and decide that she would rather go see the latest results of the World Series. This kind of interactive and erratic behavior will happily send southwest the transaction and its painstakingly constructed and kept state, but, after all, in it rests much of the beauty of the web concept. At the same time, it is also a royal pain in the neck for many developers of transactional content. (Developers of standard document content, by contrast, are perfectly happy with it: you can leave and return to this article as many times as you want, and you will always find it in the correct "state". Unless I decide to remove it in the meantime, that is, but that's a separate issue.)

This in HTTP 1.0 . When the above limitations were understood, the HTTP group started working on the 1.1 specifications, as an effort to mend some shortcomings of the initial implementation and to add other needed functionalities. The multiple connections problem was addressed by allowing to keep the connection open across requests for the same client-server pair: still, the requests that are carried out on this persistent connection are required to be idempotent, that is, self-contained and repeatable without prejudice.

Idempotence is a concept that is often invoked, but seldom completely realized, in network protocols - witness the way in which "idempotence" in NFS got promptly

subverted by rpc.lockd. The way in which HTTP is used today is no exception, and there are many ways to keep intra-request status, including the - much debated - devices called "Cookies" which I will discuss later. Here, suffice it to say that these method exist, and that they are still far from ideal.

The Gory Details

Having said what a HTTP conversation looks like to a casual onlooker, it is time to get down and dirty and have a look at **what** the client and server actually tell to each other. The taxonomy of the possible conversations is quite wide, and it is not my objective to cover it comprehensively: what I will show is the anatomy of a rather typical GET transaction, where a web browser asks for a HTML page to be sent. This type of conversation constitutes the bulk of what happens on the Web.

The Client	
`GET /index.html HTTP/1.1`	Request type (GET), path, protocol & version.
`Connection: Keep-Alive`	Don't close the connection.
`User-Agent: Mozilla/4.7 [en] (WinNT; I)`	Identifier of the client (Netscape).
`Host: www.linux.org`	The host this request is for.

`Accept: image/gif, image/x-xbitmap, image/jpeg,`
`        image/pjpeg, image/png, */*`
`Accept-Encoding: gzip`
`Accept-Language: en`
`Accept-Charset: iso-8859-1,*,utf-8`
	Data types and encoding this client can handle (MIME types).

	An empty line terminates the client request: this particular GET request has no body, as can be inferred by the missing Content* headers.
The Server	
`HTTP/1.1 200 OK`	The result status: numeric and description.
`Date: Thu, 09 Dec 1999 12:23:29 GMT` `Server: Apache/1.3.9 (Linux Debian 2.2) ApacheJServ/1.0`	Time stamp and identifier of the server software.

`Last-Modified: Mon, 04 Oct 1999 09:33:15 GMT` `ETag: "0-374-37f8745b"`	Time stamp of the retrieved document and tag for cache validation.

`Accept-Ranges: bytes`
`Content-Length: 884`
`Content-Type: text/html`
	MIME formatted information on charset, length and type (html) of the result.

	An empty line separates the response headers from the response body.
`<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">` `[...]`	Now the document (HTML as expected) is sent.

As stated above, HTTP makes provisions for different types of conversations, serving purposes different from GET. The POST request, for instance, allows the client to send data to the server for further processing, and is the foundation upon which fill-out web forms are built. All these requests, however, follow the general schema outlined above, give or take a few MIME headers.

The GET request does, however, fulfill the basic need of serving a URL - presumably stored into a HTML documents - to a client, and this is the core idea of the web.

The dynamic web

The first problem with all this is, of course, that every time you do not know the URL of the document you are looking for, you're out of luck. Besides, URLs are not easily inferred from, say, the knowledge of the content of the document. And don't even think about more general type of interaction, such as the ones involved in - for instance - e-commerce, this year's hot meme.

At the very least, schemes must exist to send queries to "document indices", repository of URLs that can be selected based on custom criteria. (I am using the "index" word with considerable latitude here: while Yahoo clearly fits the bill as far as indices are concerned, so would do a database listing the availability of parts in a mechanical shop.)

In a not too far past, people would refer to "What's new" pages, where people of good will used to collect the new offerings of the World Wide Web. Visitors would insert the new interesting stuff in their bookmarks, and come back periodically to check. The historical, and long gone, "What's new" page at NCSA - updated weekly - used to take all of five or ten minutes to read circa 1994.

This approach quickly broke down when the web exploded, both because of the sheer quantity of the material, and because of the request for more sophisticated services. (Though manual indexing is alive and well: the last time I looked - not too long ago - Yahoo was still being maintained that way). Clearly a different, programmatic approach was needed. As it is often the case, fulfilling this need places different demands on the protocol, server and the client, and we will examine them in what follows.

The dynamic URL

First of all, we need to change the usual mental picture of a URL as a a file sitting somewhere and waiting to be served. If we are to interact with a dynamic document base, we must reinterpret a URL as something that means, roughly, *"a network resource that, when contacted through HTTP, will* ***eventually****, after suitable processing, use HTTP to return a content displayable by a web browser*".

Since we are expecting to be able to perform queries, we also have to provide means to supplement a URL request with parameters describing the context of our request. At the most basic level, a standard way to parametrize a given URL is to provide a query string, which, sintactically, is everything that follows a question mark in the URL itself.

As it turns out, the query string is not enough for our purposes. Since the requests we are anticipating are of a very general nature, it would be unwise to place unnecessary restrictions on the nature of the parameters that may be needed to complete them.

An example would be the request to insert a local document in a document management system: such a request could send the document itself as a "parameter". This hardly fits in the query string scheme (often called the "GET" method), because the client and the server place limits on the length of the URLs they can handle, thus subjecting very long URLs to arbitrary truncations.

For this reason HTTP specifies means to package (long) parameters in the body of a request, using the "POST" request and (in more recent versions) the mime "multipart/..." document type. Since the length of the body of a POST request can in principle be unlimited, these provisions complement the query string creating a general system for supplying parameters to a URL.

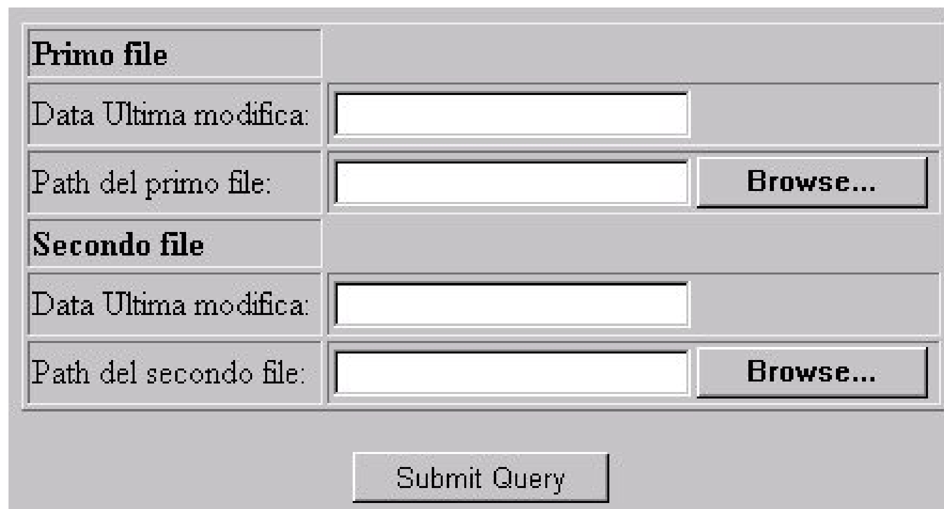
I will not describe the details of either the query string or of multipart documents (which is somewhat complex) because this kind of information can be obtained from several other sources. For our purpose, it is sufficient to know that a URL can be given an unlimited number of parameters, as long as it knows what to do with them.

The dynamic client

Supplying parameters to a URL is the job of the client, and ultimately of the person

who is using it. So an interface must be in place to allow the specification of the required parameters, and this interface must be able to change for any non-static URL that is referenced. This is needed because each and every dynamic URL will have its idea of what parameters it needs.

This is addressed by a specific set of HTML tags - <FORM>, <INPUT> and few others - which are interpreted by browsers as directives to display an appropriately laid out form, like this one:



The form tags also convey, as attributes, all the details about the parameters that the queried URL expects to receive and about the HTTP method (typically GET or POST) that must be used for submission.

In this way, the browser knows everything it needs to package a fully functional request to the target URL. But, what is the source of the form? In many cases, providing an adequate form structure and layout is part of what the dynamic URL must be able to do in order to get its invocation parameters from the client: after all, nobody else can be expected to know the internal details of this particular URL. Less often, the form is stored in a separate document, which refers the parameter data to the target dynamic URI when the user hits the submit button. This way of separating the form code from the processing part, while still common, is less appealing, as it implies maintaining two separate documents (form and handler code) rather than just one.

It may be noted that the standard says nothing about the additional client side processing that can happen between the moment the form layout is sent to the client and the moment it is submitted. So a particular URL can send very complicated forms, including parts of client-side scripting languages (e.g. JavaScript), Java Applets, Dynamic HTML, XML, and so on, in the expectation that a sophisticated client-side processing will take place. This is obviously fraught with perils, as this type of interaction tends to be highly dependent on the particular browser being used, and often results in trading functionality and client independence with visual appeal (or with window dressing, depending on how you look at it). The final result may well be exactly the kind of incompatibility that the web was designed to defeat.

By contrast, plain HTML forms are implemented by all browsers, and effectively cover the core functionalities of interacting with dynamics URL, even if in a somewhat plain fashion.

The dynamic server

All this is nice, as far as semantics go, but it still does not say just **what** a HTTP server has to do when servicing requests directed to a dynamic URL.

The short answer to this question is, quite obviously, that the server can do anything it well pleases. It can pre/postprocess the request till the cows come home, hand it off to a different process or even a different server and it all will be all right as long as someone, mindful of the waiting client, eventually returns the awaited HTTP reply.

While this is pretty cool, standards and guidelines are still needed if we are to avoid hopeless fragmentation of server side programming: if every server is allowed to process URLs in its own custom way, there will be no compatibility at all, and we'll all be back to the bad old days when changing platform meant - to a programmer - undergoing a complete retrain cycle.

In many ways, proprietary fragmentation sadly depicts the current situation: most servers have developed private and rigorously incompatible extensions with an uneven split among the (largest) Apache/PHP, (large) IIS/ASP, (emerging) Java servlet API, (smaller) Netscape Server API - to name just a few of the major players.

The one server-side technology that has evaded the proprietary struggle and enjoys across the board support is the most ancient, which emerged shortly after the inception of the web, and is known as the Common Gateway Interface, or, for short, CGI. (This shouldn't surprise anyone: the people who invented the web actually believed in interoperable applications. A lot of the modern day pundits don't, even as they endlessly speak of "Open standards".)

The Common Gateway Interface: history

While the world of server-specific dynamic extensions is interesting in its own rights, I will not discuss it here (I am planning to write a technology review on this subject). What I will do, albeit briefly, is give an overview of CGI programming: this is because, as of January 2000, it is still, by far, the most prevalent way of serving dynamic content on the web, even if its dominance is receding.

CGI is a sort of word-of-mouth standard. It is not enshrined in any RFC, and, in fact, in no officially published standards. At present, there is an ongoing effort to convert the existing *"coding practices"* into a RFC: it has so far produced a number of Internet Drafts. Work on the standardization issue is, however, far from frantic: the group working page was last updated in June, 1999, and the "CGI 1.1 Internet Draft" (version 11.03) expired on December 31, 1999. So the only firmly

established (by tradition, if by nothing else) common ground appears to be set by the original 1.0 document.

While historical details are lacking, the 1.0 specs were invented by the NCSA software development team (which also developed the first popular web browser, Mosaic, thus helping to set the premises for the web and Marc Andressen's riches) as a quick hack to connect already existing programs to their web server.

The specifications themselves are quite simple. They basically dictate how a server prepares, executes, and postprocesses the outcomes of an external program, and, in the process, also define what the behavior of a CGI compliant program must be. Without entering the details, which can be found in the referenced documents, I will now summarize the most important of these provisions. (Note: CGI is often called cgi-bin, from the name of the directory, or folder, in which CGI programs are traditionally kept. The official name, however, is CGI, and these days, cgi-bin directories - if they even exist - often contain at least some executables which have little to do with the original CGI.)

The Common Gateway Interface: mechanics

Upon recognizing the requested URI as handled by a CGI program, the server must prepare the execution environment by setting the value of a number of variables. The 1.1 draft specifies several such variables, however, the 1.0 specs mandates the following list (excerpted from the original document):

SERVER_SOFTWARE	The name and version of the information server software answering the request (and running the gateway). Format: name/version
SERVER_NAME	The server's hostname, DNS alias, or IP address as it would appear in self-referencing URLs.
GATEWAY_INTERFACE	The revision of the CGI specification to which this server complies. Format: CGI/revision
SERVER_PROTOCOL	The name and revision of the information protocol this request came in with. Format: protocol/revision
SERVER_PORT	The port number to which the request was sent.
REQUEST_METHOD	The method with which the request was made. For HTTP, this is "GET", "HEAD", "POST", etc.
PATH_INFO	The extra path information, as given by the client. In other words, scripts can be accessed by their virtual pathname, followed by extra information at the end of this path. The extra information is sent as PATH_INFO. This information should be decoded by the server if it comes from a URL before it is passed to the CGI script.
PATH_TRANSLATED	The server provides a translated version of PATH_INFO, which takes the path and does any virtual-to-physical mapping to it.

SCRIPT_NAME	A virtual path to the script being executed, used for self-referencing URLs.
QUERY_STRING	The information which follows the ? in the URL which referenced this script. This is the query information. It should not be decoded in any fashion. This variable should always be set when there is query information, regardless of command line decoding.
REMOTE_HOST	The hostname making the request. If the server does not have this information, it should set REMOTE_ADDR and leave this unset.
REMOTE_ADDR	The IP address of the remote host making the request.
AUTH_TYPE	If the server supports user authentication, and the script is protects, this is the protocol-specific authentication method used to validate the user.
REMOTE_USER	If the server supports user authentication, and the script is protected, this is the username they have authenticated as.
REMOTE_IDENT	If the HTTP server supports RFC 931 identification, then this variable will be set to the remote user name retrieved from the server. Usage of this variable should be limited to logging only.
CONTENT_TYPE	For queries which have attached information, such as HTTP POST and PUT, this is the content type of the data.
CONTENT_LENGTH	The length of the said content as given by the client.
HTTP_*	The header lines received from the client, if any, are placed into the environment with the prefix HTTP_ followed by the header name. Any - characters in the header name are changed to _ characters. The server may exclude any headers which it has already processed, such as Authorization, Content-type, and Content-length. If necessary, the server may choose to exclude any or all of these headers if including them would exceed any system environment limits

After having prepared the environment, the server must execute the handler (program). The way in which this is done is different according to the request type (GET or POST/PUT): a further distinction is made according to the nature of the query string.

If the query string includes no "=" sign, in fact, it is interpreted as a command line argument to be passed to the handler: this usage is reserved to the ISINDEX tag, and is obsolescent.

With respect to the request type, GET requests require no other setup beyond the above outlined environment preparation. POST (and the rarer PUT) requests, on the other hand, have to be handled in such a way that the request content can be read by the handler on its standard input, in the "application/x-www-form-urlencoded" encoding.

After having thusly set the stage, the server will actually execute the program, wait for its completion, and postprocess its output. The handler itself will process the request, gleaning information from the environment and (for POST/PUT requests) from its standard input. When the computation is over, the handler must write on its standard output channel the results, suitably preceded by a set of simplified HTTP headers, known as the **parsed headers**, and separated with a blank line from the content (forgetting to output the blank line is the most infamous and frequent error committed by beginning CGI coders).

The most used parsed headers are "Content-type:" for standard results, "Location:" for redirected requests, and "Status:", mostly for failed requests. The server takes it upon itself to supply the missing headers: for instance, a "Content-Length:" header will be added where appropriate. Additionally, servers may make provisions for handlers which want to supply the full set of headers by themselves. This "non-parsed-headers" feature, however, is server-specific.

The Common Gateway Interface: strengths and weaknesses

As can be seen, the CGI specifications are simplicity incarnated. According to the **KISS** (Keep It Simple, Stupid) guidelines, this is a strength in itself.

And in fact, it can be argued that the small effort that needs to be done to convert an already running console program in a working CGI handler is at the root of the wild success of this uncomplicated specification. Another reason is that there is a large number of transactions that can be satisfied by sending tabular output as the outcome of the processing of a suitably filled form (all one-step searches, by way of example, fall in this category).

Besides, a large number of software packages (libraries as well as full blown applications) supporting this programming technique are readily available for all the environments and servers. The well known CGI.pm module for the web-ubiquitous Perl language is an example among thousands.

What about the weaknesses, then? Probably the most important are statelessness (again) and overhead. I have already mentioned statelessness in the general HTTP context, and underlined its inherence in the client side. CGI is no worse, in this respect, of HTTP, but also no better. The gateway specs make no provision for multi-step transactions, and the implementation of persistence rests squarely on the shoulders of the programmers.

Overhead, on the other side, is inherent to CGI. A program invocation is in fact expected for any request to the handled URI. This paradigm is fine for relatively quiet servers, but, as the number of requests increases, the entailed process spawning becomes a resource drain. When (as it is most customary) the execution of the handler also requires the initialization of a special purpose interpreter (Visual Basic or Perl, by instance), the resource demand is even worse - and this is to be repeated **for every request.**

Several ways have been proposed, over the years, to cope with these limitations.

They span the entire gamut of options from server-embedded interpreters (FastCgi, mod-perl, Windows Scripting Host) to a complete overhaul of the CGI communication setup (most server-specific dynamic extension). None has so far succeded in becoming the next generation CGI: all of them, however, have inherited from CGI the idea of an execution environment from which the URI-handler (program) can retrieve informations about the transaction details.

The dynamic client/server environment and keeping state

I have mentioned statelessness and its shortcomings several times, and the time has come to give a short overview of the ways that the web architects have devised to alleviate this problem.

The existing solutions are based on hidden fields and cookies, and I will examine them separately. Both build on the idea of storing state information in the client itself, so that it can be suitably retrieved at the various stages of the transaction.

The dynamic client/server environment: hidden fields

A program who wants to store information about an ongoing transaction can do so by creating special form fields (tagged with the HIDDEN attribute) which are sent to the client along with the response content, which **must** be of type text/html. Successive form submissions to the target URI will embed the hidden fields content as part of the request, and, since the client is instructed not to show these fields, the embedded (state) information can be safely shuttled back and forth.

Some problems arise with this schema, mainly, that it only applies to text/html interaction types, and that the content of the hidden fields is in plain view of the sophisticated user (via the "view source" browser capability). The content issue is an important limitation when non-textual content (e.g. videos, images) has to be served as a request outcome; the end-user visibility can be of concern as it could expose the internal workings of the application, and someone could use this knowledge to subvert the application's intended use. (Of course, web applications **should** be written in a way that such subversions are impossible, but we all know how these things go.)

A third problem is related to how the state information (kept in hidden fields) is set up at the transaction's beginning. This presumably happens when the user first contacts the URI and fills a form. While this is acceptable for casual "customers" it quickly becomes a bore for frequent visitors of a given site, which are required to fill a form with the same data over and over again.

Even taking in accounts this sort of limitations, hidden fields are a simple device that allow the conduction of meaningful multi-step transactions and, as such, are widely used.

The dynamic client/server environment: Cookies

We have just seen how (volatile) information can be preserved by shuttling it back and forth, invisibly embedded in a HTML form.

Cookies - originally devised by Netscape and enshrined as a proposed standard in RFC2109 - take a different approach, and store state information in the HTTP headers themselves.

RFC2109 specifies two new HTTP headers ("Cookie:" and "Set-cookie:") containing information on the origin and the content - under the species of attribute-value pair - of a piece of persistent information, which is called - surprise - "cookie".

When a browser receives a Set-cookie request, it may - and usually does - choose to store it persistently. When the originating URI is again visited, the browser may choose to return the cookie - presumably stored during a previous visit - in a suitable "Cookie:" header, provided that the cookie - which is time-stamped - has not expired in the meantime.

This solves brilliantly the content limitation that is inherent in hidden fields: since cookies are sent as a part of the headers (which are always included in all HTTP transactions), the body of the response can be of any type, even non-textual.

The problems related to the exposed content of the translation details are not solved (cookies are stored on the user machine, and can be easily examined). On the other hand, and most importantly, the cookie mechanism does in fact provide a way to memorize persistent user information across visits to the same URL. This has made cookies wildly successful: for instance, they are at the core of Amazon's "one click shopping" and of the recent legal spate between Amazon and Barnes & Noble.

This technique also opens up considerable privacy concerns. Clients trade cookies with servers in a behind the scene fashion that is invisible to most users: while the information contained in the cookie has at some time been willingly supplied by the user, he is usually not notified when it is sent again. This is convenient, but it can also allow a silent tracking of the user's browsing patterns. Also, the user may not want to send this information again, the site owner may have changed after its last visit, and, finally, cookies open up backdoors for possible information stealing by malicious sites. All this is compounded by the fact that most people on the web are not even aware of the cookies' existence - except in their common, snack-oriented meaning.

The debate on these themes is ongoing - in the meantime, cookies are rampant. Just turn on the appropriate warning feature in your browser to have a feeling of their pervasive presence.

What now?

This is just the "nickel tour" of HTTP. It is hopefully worth a dime, but much remains to be covered. I hope to be able to come back to this document from time to time and flesh it out adding more information.

But even if I don't, inquisitive minds should be able to find much more on the web on this subject. Beyond the usual search engines (but don't query Altavista for "HTTP", if you can't handle zillions of hits), the W3 Consortium home page and Netscape's developer site are two good starting points on the subjects covered in this document.

And, of course, the often quoted RFC are the ultimate source of wisdom.

Links and resources

Resources, links and sites used during the writing of this document.

- The W3 Consortium
- Netscape's developer site
- The RFC catalog at IETF
- RFC 2616: HTTP specifications
- RFC 1630: URI definition.
- RFC 1866: HTML specifications
- RFC 2045: MIME extensions.
- RFC 2046: MIME types.
- RFC2109: Cookie standard draft.
- The CGI 1.0 specifications.
- The CGI standardization working group.
- The CGI 1.1 Internet Draft, version 11.03 (expired December 31, 1999
- NCSA Mosaic home page
- CGI.pm: documentation of the CGI perl library.
- The Perl language

Alessandro Forghieri,Tue Jan 4 2000