Hyper**T**ext **T**ransfer **P**rotocol: *A Short Course*

John Yannakopoulos
giannak@csd.uoc.gr

Department of Computer Science
University of Crete
Heraklion, Crete, Greece

August 2003

The innovations that Berners-Lee added to the Internet to create the World Wide Web had two fundamental dimensions: connectivity and interface. He invented a new protocol for the computers to speak as they exchanged hypermedia documents. This Hypertext Transfer Protocol (HTTP) made it very easy for any computer on the Internet to safely offer up its collection of documents into the greater whole; using HTTP, a computer that asked for a file from another computer would know, when it received the file, if it was a picture, a movie, or a spoken word. With this feature of HTTP, the Internet began to reflect an important truth - retrieving a file's data is almost useless unless you know what kind of data it is. In a sea of Web documents, it's impossible to know in advance what a document is - it could be almost anything - but the Web understands "data types" and passes that information along.

– Mark Pesce, "VRML - Browsing and Building Cyberspace", New Riders Publishing, 1995.

1 Introduction

Underlying the user interface represented by browsers, is the network and the protocols that travel the wires to the servers or "engines" that process requests, and return the various media. The protocol of the web is known as HTTP, for **H**yper**T**ext **T**ransfer **P**rotocol (RFC 1945, 2616).

Tim Berners-Lee implemented the HTTP protocol in 1990-1 at CERN, the European Center for High-Energy Physics in Geneva, Switzerland. HTTP stands at the very core of the World Wide Web. According to the HTTP/1.0 specification, the Hypertext Transfer Protocol is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks, such as name servers and distributed object management systems, through extension of its request methods (commands). A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

2 HTTP/1.0 Properties

2.1 A comprehensive addressing scheme

The HTTP protocol uses the concept of reference provided by the **U**niversal **R**esource **I**dentifier (URI) as a location (URL) or name (URN), for indicating the resource on which a method is to be applied. When an HTML hyperlink is composed, the URL (**U**niform **R**esource **L**ocator) is of the general form `http://host:port-number/path/file.html`. More generally, a URL reference is of the type `service://host/file.file-extension` and in this way, the HTTP protocol can subsume the more basic Internet services.

HTTP/1.0 is also used for communication between user agents and various gateways, allowing hypermedia access to existing Internet protocols like SMTP, NNTP, FTP, Gopher, and WAIS. HTTP/1.0 is designed to allow communication with such gateways, via proxy servers, without any loss of the data conveyed by those earlier protocols.

2.2 Client-Server architecture

The HTTP protocol is based on a request/response paradigm. The communication generally takes place over a TCP/IP connection on the Internet. The default port is 80, but other ports can be used. This does not preclude the HTTP/1.0 protocol from being implemented on top of any other protocol on the Internet, so long as reliability can be guaranteed.

A requesting program (a client) establishes a connection with a receiving program (a server) and sends a request to the server in the form of a request method, URI, and protocol version, followed by a message containing request modifiers, client information, and possible body content. The server responds with a status line, including its protocol version and a success or error code, followed by a message containing server information, entity metainformation, and possible body content.

2.3 The HTTP/1.0 protocol is connectionless

Although we have just said that the client establishes a connection with a server, the protocol is called connectionless because once the single request has been satisfied, the connection is dropped. Other protocols typically keep the connection open, e.g. in an FTP session you can move around in remote directories, and the server keeps track of who you are, and where you are.

While this greatly simplifies the server construction and relieves it of the performance penalties of session housekeeping, it makes the tracking of user behaviour, e.g. navigation paths between local documents, impossible. Many, if not most, web documents consist of one or more inline images, and these must be retrieved individually, incurring the overhead of repeated connections.

2.4 The HTTP/1.0 protocol is stateless

After the server has responded to the client's request, the connection between client and server is dropped and forgotten. There is no "memory" between client connections. The pure HTTP server implementation treats every request as if it was brand-new (without context), i.e. not maintaining any connection information between transactions.

2.5 An extensible and open representation for data types (MIME Types)

HTTP uses Internet Media Types (formerly referred to as MIME Content-Types) to provide open and extensible data typing and type negotiation. For mail applications, where there is no type negotiation between sender and receiver, it's reasonable to put strict limits on the set of allowed media types. With HTTP, where the sender and recipient can communicate directly, applications are allowed more freedom in the use of non-registered types.

When the client sends a transaction to the server, headers are attached that conform to standard Internet e-mail specifications (RFC 822). Most client requests expect an answer either in plain text or HTML. When the HTTP Server transmits information back to the client, it includes a MIME-like (**M**ultipart **I**nternet **M**ail **E**xtension) header to inform the client what kind of data follows the header. Translation then depends on the client possessing the appropriate utility (image viewer, movie player, etc.) corresponding to that data type.

2.5.1 Discussion

RFC 2045, defines a number of header fields, including `Content-Type`. The `Content-Type` field is used to specify the nature of the data in the body of the MIME entity, by giving media type and subtype identifiers, and by providing auxiliary information that may be required for certain media types. After the type and the subtype names, the remainder of the header field is simply a set of parameters, specified in an attribute/value notation. The ordering of parameters is not significant.

In general, the top-level media type is used to declare the general type of data, while the subtype specifies a specific format for that type of data. Thus, a media type of ”`image/xyz`” is enough to tell a user agent that the data is an image, even if the user agent has no knowledge of the specific image format ”`xyz`”. Such information can be used, for example, to decide whether or not to show a user the raw data from an unrecognized subtype - such an action might be reasonable for unrecognized subtypes of ”`text`”, but not for unrecognized subtypes of ”`image`” or ”`audio`”. For this reason, registered subtypes of ”`text`”, ”`image`”, ”`audio`”, and ”`video`” should contain embedded information that is really of a different type. Such compound formats are represented using the ”`multipart`” or ”`application`” types as the RFC 2045 states.

2.5.2 Definition of a Top-Level Media Type

The definition of a top-level media type consists of:

1. a name and a description of the type, including criteria for whether a particular type would qualify under that type,
2. the names and definitions of parameters, if any, which are defined for all subtypes of that type (including whether such parameters are required or optional),
3. how a user agent and/or gateway should handle unknown subtypes of this type,
4. general considerations on gatewaying entities of this top-level type, if any, and
5. any restrictions on content-transfer-encodings for entities of this top-level type.

2.5.3 Overview of the Initial Top-Level Media Types

The five discrete top-level media types are:

1. text – textual information. The subtype ”`plain`” in particular indicates plain text containing no formatting commands or directives of any sort. Plain text is intended to be displayed ”as-is” and no special software is required in order to be displayed by a user agent. Other subtypes are to be used for enriched text in forms where application software may enchance the appearence of the text, but such software is not required in order to get the general idea of the content. Possible subtypes of ”`text`” thus, include any word processor format that can be read without resorting to software that understands the format. A very simple and portable subtype, ”`richtext`”, was defined in RFC 1341, with a further revision in RFC 1896 under the name ”`enriched`”.

2. image – image data. ”`image`” requires a display device (such as a graphical display) to view the information. Subtypes are defined for the most widely-used image formats such as `jpeg`, `gif`, `png`, etc.

3. audio – audio data. ”`audio`” requires an audio output device (such as a speaker) to ”display” the contents. Some of the numerous subtypes that are defined for this type, are: `basic`, `mp3`, `wav`, etc.

4. video – video data. ”`video`” requires the capability to display moving images, typically including specialized hardware and software (e.g. Microsoft Media Player, QuickTime, RealPlayer, etc.). Subtypes which belong to this type’s category are: `mpeg`, `mpg`, `ra`, `avi`, etc.

5. application – represents some other kind of data, typically either uninterpreted binary data or information to processed by an application. The subtype ”`octet-stream`” is used in the case of uninterpreted binary data, in which case the simplest recommended action is to offer to write the information into a file for the user. The ”`postscript`” and ”`pdf`” subtypes are also defined for the transport of Postscript and PDF material respectively. Other expected uses for ”`application`” include spreadsheets, data for mail-based scheduling systems, and languages for word processing formats that are not directly readable.

The two composite top-level media types are:

1. multipart – data consisting of multiple entities of independent data types. Four subtypes were initially defined, including the basic ”`mixed`” subtype specifying a generic mixed set of parts, ”`alternative`” for representing the same data in multiple formats, ”`parallel`” for parts intended to be viewed simultaneously, and ”`digest`” for multipart entities in which each part has a default type of ”`message/rfc822`”.

2. message – an encapsulated message. A body of media type ”`message`” is itself all or a portion of some kind of message object. Such objects may or may not in turn contain other entities. The ”`rfc822`” subtype is used when the encapsulated content is itself an RFC 822 message. The ”`partial`” subtype is defined for partial RFC 822 messages, to permit the fragmented transmission of bodies that are thought to be too large to be passed through transport facilities in one piece. Another subtype, ”`external-body`”, is defined for specifying large bodies by reference to an external data source.

It should be noted that the media types/subtypes’ discussion that was made in the previous paragraphs, it’s intended to be used *only* as a brief overview and *not* as an exhaustive guide to the subject.

For a more in-depth analysis with regard to the Internet media types, you 're encouraged to refer to the appropriate RFCs and Internet standards track protocols (RFC 2045, 2046, 2047, 2048, 2049, and 2077) as well as the references cited in these.

I would also advise and suggest you to have a look at the configuration file(s) that various commercial WWW servers use in order to control what Internet media types should send to the client for given file extensions (e.g. the `mime.types` file which is contained in the `conf` directory of the Apache 1.3/2.0 top-level distribution directory).

3 HTTP/1.0 Header Fields

An HTTP transaction consists of a header followed optionally by an empty line and some data. The header will specify such things as the action required of the server, or the type of data being returned, or a status code. The use of header fields sent in HTTP transactions gives the protocol great flexibility. These fields allow descriptive information to be sent in the transaction, enabling authentication, encryption, and/or user identification. The header is a block of data preceding the actual data, and is often referred to as meta information, because it is information about information.

The header lines received from the client, if any, are placed by the server into the CGI environment variables 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. An example of this is the `HTTP_ACCEPT` variable, another example is the header `User-Agent`.

- `HTTP_ACCEPT` (or `Accept` header)

  The MIME types which the client will accept, as given by HTTP headers. Other protocols may need to get this information from elsewhere. Each item in this list should be separated by commas as per the HTTP spec.

  Format: type/subtype, type/subtype (e.g. `Accept: text/xml,text/html,...`).

- `HTTP_USER_AGENT` (or `User-Agent` header)

  The browser the client is using to send the request.

  General format: software/version library/version.

  The server sends back to the client:

  - A status code that indicates whether the request was successful or not. Typical error codes indicate that the requested file was not found, that the request was malformed, or that authentication is required to access the file.
  - The data itself. Since HTTP is liberal about sending documents of any format, it is ideal for transmitting multimedia such as graphics, audio, and video files. This complete freedom to transmit data of any format is one of the most significant advantages of HTTP and the Web.
  - It also sends back information about the object being returned. Note that the following is not a complete list of header fields, and that some of them only make sense in one direction.

- `Content-Type`

  The Content-Type header field indicates the media type of the data sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET. This field is used by browsers to know how to deal with the data. The client uses this information to determine how to handle a video file or an inline graphic. An example:

  ```
  Content-Type: text/html
  ```

- `Date`

  The Date header represents the date and time at which the message was originated. An example is:

  ```
  Date: Fri, 08 Aug 2003 08:12:31 GMT
  ```

- `Expires`

  The Expires field gives the date after which the information in the document ceases to be valid. Caching clients, including proxies, must not cache this copy of the resource beyond the date given, unless its status has been updated by a later check of the origin server.

  ```
  Expires: Fri, 01 Sep 2003 16:00:00 GMT
  ```

- `From`

  The From header field, if given, should contain an Internet e-mail address for the human user who controls the requesting user agent. An example is:

  ```
  From: giannak@csd.uoc.gr
  ```

  This header field may be used for logging purposes and as a means for identifying the source of invalid or unwanted requests. It should not be used as an insecure form of access protection. The interpretation of this field is that the request is being performed on behalf of the person given, who accepts responsibility for the method performed. In particular, robot agents should include this header so that the person responsible for running the robot can be contacted if problems occur on the receiving end.

- `If-Modified-Since`

  The If-Modified-Since header field is used with the GET method to make it conditional: if the requested resource has not been modified since the time specified in this field, a copy of the resource will not be returned from the server; instead, a 304 (not modified) response will be returned without any data. An example of the field is:

  ```
  If-Modified-Since: Fri, 08 Aug 2003 19:43:31 GMT
  ```

- `Last-Modified`

  The Last-Modified header field indicates the date and time at which the sender believes the resource was last modified. The ”Last Modified” field is useful for clients that eliminate unnecessary transfers by using caching. The exact semantics of this field are defined in terms of how the receiver should interpret it: if the receiver has a copy of this resource which is older than the date given by the Last-Modified field, that copy should be considered stale. An example of its use is:

```
Last-Modified: Fri, 01 Sep 2003 12:45:26 GMT
```

- `Location`

  The Location response header field defines the exact location of the resource that was identified by the request URI. If the value is a full URL, the server returns a ”redirect” to the client to retrieve the specified object directly.

  ```
  Location: http://www.csd.uoc.gr/~hy556/notes/HTTP/index.html
  ```

  If you want to reference another file on your own server, you should output a partial URL, such as the following:

  ```
  Location: /~hy556/notes/HTTP/index.html
  ```

  The server will act as if the client had not requested your script, but instead requested `http://yourserver/~hy556/notes/HTTP/index.html`. It will take care of all access control, determining the file’s type, etc.. In this case clients don’t do the redirection, but the server does it ”on the fly”. Important: Only full URLs in Location field can contain the #label part of URL (i.e. fragment), because that is meant only for the client-side, and the server cannot possibly handle it in any way.

  As an example of actual use, suppose that the ”Ask Dr. Marazakis” form has a Yes/No toggle after the question ”Did you search the site and read the HTTP tutorial?”. The default is No, so if the user doesn’t reset this to Yes they will simply be redirected to the HTTP tutorial and their question will not be sent. A simple example in Perl (inside a CGI script), is the following:

  ```
  if ($input{'YN'} eq "No") {
    print "Location: http://www.csd.uoc.gr/~hy556/notes/HTTP/index.html\r\n";
  }
  else {
    print "Content-Type: text/html\r\n";
    &Feedback;
  }
  ```

- `Referer`

  The Referer request header field allows the client to specify, for the server’s benefit, the address (URI) of the resource from which the request URI was obtained. This allows a server to generate lists of back-links to resources for interest, logging, optimized caching, etc. It also allows obsolete or mistyped links to be traced for maintenance. Example:

  ```
  Referer: http://www.csd.uoc.gr/~hy556/index.html
  ```

If a partial URI is given, it should be interpreted relative to the request URI. The URI must not include a fragment (#label within a document).

- `Server`

  The Server response header field contains information about the software used by the origin server to handle the request. The field can contain multiple product tokens and comments identifying the server and any significant subproducts. By convention, the product tokens are listed in order of their significance for identifying the application. Example:

  ```
  Server:  Apache/1.3.27 (Unix)
  ```

- `User-Agent`

  The User-Agent field contains information about the user agent originating the request. This is for statistical purposes, the tracing of protocol violations, and automated recognition of user agents for the sake of tailoring responses to avoid particular user agent limitations - such as inability to support HTML tables. By convention, the product tokens are listed in order of their significance for identifying the application. Example:

  ```
  User-Agent:  Mozilla/5.0 (X11; U; Linux i686) Gecko/20020830
  ```

4 HTTP/1.0 Methods

HTTP/1.0 allows an open-ended set of methods to be used to indicate the purpose of a request. The three most often used methods are GET, HEAD, and POST.

4.1 The GET method

The GET method is used to ask for a specific document - when you click on a hyperlink, GET is being used. GET should probably be used when a URL access will not change the state of a database (by, for example, adding or deleting information) and POST should be used when an access will cause a change. The semantics of the GET method changes to a "conditional GET" if the request message includes an "If-Modified-Since:" header field. A conditional GET method requests that the identified resource be transferred only if it has been modified since the date given by the "If-Modified-Since:" header. The conditional GET method is intended to reduce network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring unnecessary data.

4.2 The HEAD method

The HEAD method is used to ask only for information about a document, not for the document itself. HEAD is much faster than GET, as a much smaller amount of data is transferred. It's often used by clients who use caching, to see if the document has changed since it was last accessed. If it was not, then the local copy can be reused, otherwise the updated version must be retrieved with a GET. The metainformation contained in the HTTP headers in response to a HEAD request should be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the resource identified by the request URI without transferring the data itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

4.3 The POST method

The POST method is used to transfer data from the client to the server; it's designed to allow a uniform method to cover functions like: annotation of existing resources; posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles; providing a block of data (usually a form) to a data-handling process; extending a database through an append operation.

```
POST /~hy556/cgi-bin/post-query HTTP/1.0
Accept:  text/html,video/mpeg,image/gif,application/postscript
User-Agent:  Lynx/2.8.4 libwww/5.4.0
From:  giannak@csd.uoc.gr
Content-Type:  application/x-www-form-urlencoded
Content-Length:  150
* a blank line *
org=Distributed%20Systems&professor=Marazakis&browsers=lynx
```

This is a "POST" query addressed for the program residing in the file at "/~hy556/cgi-bin/post-query", that simply echoes the values it receives. The client lists the MIME-types it is capable of accepting, and identifies itself and the version of the WWW library it is using. Finally, it indicates the MIME-type it has used to encode the data it is sending, the number of character included, and the list of variables and their values it has collected from the user.

MIME-type "application/x-www-form-urlencoded" means that the variable name-value pairs will be encoded the same way a URL is encoded. Any special characters, including puctuation, will be encoded as `nn` where `nn` is the ASCII value for the character in hex.

5 HTTP/1.0 Response

Here is an example of an HTTP response from a server to a client request:

```
HTTP/1.0 200 OK
Date:  Fri, 08 Aug 2003 08:12:31 GMT
Server:  Apache/1.3.27 (Unix)
MIME-version:  1.0
Last-Modified:  Fri, 01 Aug 2003 12:45:26 GMT
Content-Type:  text/html
Content-Length:  2345
* a blank line *
<HTML> ...
```

The server agrees to use HTTP version 1.0 for communication and sends the status 200 indicating it has successfully processed the client's request. It then sends the date and identifies itself as an Apache HTTP server. It also indicates it is using MIME version 1.0 to describe the information it is sending, and includes the MIME-type of the information about to be sent in the "Content-Type:" header. Finally, it sends the number of characters it is going to send, followed by a blank line and the data itself.

Client and server headers are RFC 822 compliant mail headers. A client may send any number of (comma-separated or multiple) "Accept:" headers and the server is expected to convert the data into a form the client can accept.

6 HTTP/1.1 - The Next Generation

The essential simplicity of HTTP was a major factor in its rapid adoption, but this very simplicity became its main drawback; HTTP/1.0 does not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, or virtual hosts. In addition, the proliferation of incompletely-implemented applications calling themselves "HTTP/1.0" has necessitated a protocol version change in order for two communicating applications to determine each other's true capabilities.

The HTTP/1.1 (RFC 2616), is a replacement for HTTP/1.0 with much higher performance and adding some extra features needed for use in commercial applications. It's designed to make it easy to implement the basic functionality needed by all browsers, whilst making the addition of more powerful features such as security and authentication much simpler.

6.1 Persistent Connections

A significant difference between HTTP/1.1 and earlier versions of HTTP is that persistent connections are the default behavior of any HTTP connection. That is, unless otherwise indicated, the client should assume that the server will maintain a persistent connection, even after error responses from the server.

Persistent connections provide a mechanism by which a client and a server can signal the close of a TCP connection. This signaling takes place using the `Connection`[1] header field. Once a close has been signaled, the client must not send any more requests on that connection.

Prior to persistent connections, a separate TCP connection was established to fetch each URL, increasing the load on HTTP servers and causing congestion on the Internet. The use of inline images and other associated data often require a client to make multiple requests of the same server in a short amount of time.

Persistent HTTP connections have a number of advantages:

- By opening and closing fewer TCP connections, CPU time is saved in routers and hosts (clients, servers, proxies, gateways, tunnels, or caches), and memory used for TCP protocol control blocks can be saved in hosts.
- HTTP requests and responses can be pipelined in a connection. Pipelining allows a client to make multiple requests without waiting for each response, allowing a single TCP connection to be used much more efficiently, with much lower elapsed time.
- Network congestion is reduced by reducing the number of packets caused by TCP opens (three-way handshakes' cost), and by allowing TCP sufficient time to determine the congestion state of the network.
- Latency on subsequent requests is reduced since there is no time spent in TCP's connection opening handshake.
- HTTP can evolve more gracefully, since errors can be reported without the penalty of closing the TCP connection. Clients using future versions of HTTP might optimistically try a new feature, but if communicating an older server, retry with old semantics after an error is reported.

[1]Setting the `Connection` header field to `close`, the persistent connections are disabled in a client-server communication over HTTP/1.1 (thus, HTTP/1.0 behavior).

6.2 Caching

The HTTP/1.1 protocol includes a number of elements intended to make caching work as well as possible. Caching would be useless if it did not significantly improve performance. The goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, and to eliminate the need to send full responses in many other cases. That is, there are two main reasons that web caching is used:

- To **reduce latency** – Because the request is satisfied from the cache (which is closer to the client) instead of the origin server, it takes less time for the client to get the object and display it. This makes Web sites seem more responsive.
- To **reduce traffic** – Because each object is only gotten from the server once, it reduces the amount of bandwidth used by a client. This saves money if the client is paying by traffic, and keeps their bandwidth requirements lower and more manageable.

Before proceeding to the description of how we can control caches in HTTP/1.1, at this point we 'll summarize shortly how web caches work. All caches have a set of rules that they determine when to serve an object from the cache, if it's available. Some of these rules are set in the protocols (HTTP 1.0 and 1.1), and some are set by the administrator of the cache (either the user of the browser cache, or the proxy administrator).

Generally speaking, these are the most common rules that are followed for a particular request (don't worry if you don't understand the details, it will be explained below):

1. If the object's headers tell the cache not to keep the object, it won't. Also, if no validator is present, most caches will mark the object as uncacheable.
2. If the object is authenticated or secure, it won't be cached.
3. A cached object is considered fresh (that is, able to be sent to a client without checking the origin server) if:
   - It has an expiry time or other age-controlling directive set, and is still within the fresh period.
   - If a browser cache has already seen the object, and has been set to check once a session.
   - If a proxy cache has seen the object recently, and it was modified relatively long ago.

   Fresh documents are served directly from the cache, without checking with the origin server.
4. If an object is stale, the origin server will be asked to validate the object, or tell the cache whether the copy that it has is still good.

Together, freshness and validation are the most important ways that a cache works with content. A fresh object will be available instantly from the cache, while a validated object will avoid sending the entire object over again if it hasn't changed.

At this point, we will analyze the different ways which someone can control caches with, in general. We 'll begin our discussion with methods used prior HTTP/1.1, ending with the description of the methods that HTTP/1.1 provides.

6.2.1 HTML Meta Tags vs. HTTP Headers

HTML authors can put tags in a document's `<HEAD>` section that describes its attributes. These *Meta tags* are often used in the belief that they can mark a document as uncacheable, or expire it at a certain time.

Meta tags are easy to use, but they aren't very effective. That's because they 're usually honored by browser caches (which actually read the HTML), not proxy caches (which almost never read the HTML markup in the document). While it may be tempting to slap a `Pragma: no-cache` meta tag on a home page, it won't necessarily cause it to be kept fresh, if it goes through a shared cache.

On the other hand, true *HTTP headers* give you a lot of control over how both browser caches and proxies handle your objects. They can't be seen in the HTML, and are usually automatically generated by the Web server. However, they can be controlled to some degree, depending on the server it's used.

6.2.2 Pragma HTTP Headers

Many people believe that assigning a `Pragma: no-cache` HTTP header to an object will make it uncacheable. This is not necessarily true; the HTTP specification does not set any guidelines for Pragma response headers; instead, Pragma request headers (the headers that the browser sends to a server) are discussed. Although a few caches may honor this header, the majority won't, and it won't have any effect. The following headers must be used instead.

6.2.3 Expires HTTP header

The `Expires` HTTP header is the basic means of controlling caches; it tells all caches how long the object is fresh for; after that time, caches will always check back with the origin server to see if a document is changed. Expires headers are supported by practically every client.

Most Web servers allow someone to set Expires response headers in a number of ways. Commonly, they will allow setting an absolute time to expire, a time based on the last time that the client saw the object (last *access time*), or a time based on the last time the document changed on the server (last *modification time*).

Expires headers are especially good for making static images (like navigation bars and buttons) cacheable. Because they don't change much, an extremely long expiry time on them can be set, making a web site appear much more responsive to its users. They are also useful for controlling caching of a page that is regularly changed. For instance, if a news page is updated once a day at 6 a.m., the object it can be set to expire at that time, so caches will know when to get a fresh copy, without users having to hit the 'reload' button.

As we stated in section 3, the only value in an Expires header is a HTTP date; enything else will most likely be interpreted as 'in the past', so that the object is uncacheable. Note that the time in a HTTP date is Greenwich Mean Time (GMT), not local time.

6.2.4 Warnings

In HTTP/1.1, whenever a cache returns a response that is neither first-hand or "fresh enough", it attaches a warning to that effect, using a `Warning` general header field. The warning allows clients to take appropriate action. Warnings may be used for other purposes, both cache-related and otherwise. The use of warning, rather than an error status code, distinguish these responses from true failures. For more information about warnings, refer to the HTTP/1.1 draft specification (RFC 2616, Sections 13.1.2, 14.46).

6.2.5 Cache-Control HTTP Headers

Although the Expires header is useful, it's still somewhat limited; there are many situations where content is cacheable, but the HTTP/1.0 protocol lacks methods of telling caches what it is, or how to work with it.

HTTP/1.1 introduces a new class of headers, the `Cache-Control` response headers, which allow web publishers to define how pages should be handled by caches. They can be broken down into the following general categories:

- Restrictions on what are cacheable; these may only be imposed by the origin server.
- Restrictions on what may be stored by a cache; these may be imposed by either the origin server or the user agent.
- Modifications of the basic expiration mechanism; these may be imposed by either the origin server or the user agent.
- Controls over cache revalidation and reload; these may only be imposed by a user agent.
- Control over transformation of entities.
- Extensions to the caching system.

What is Cacheable?

By default, a response is cacheable if the requirements of the request method, request header fields, and the response status indicate that it is cacheable. The following `Cache-Control` directives allow an origin server to override the default cacheability of a response:

`public`

Indicates that the response may be cached by any cache, even if it would normally be non-cacheable or cacheable only within a non-shared cache. For instance, if your pages are authenticated, this directive makes them cacheable.

`private`

Indicates that all or part of the response message is intended for a single user and must not be cached by a shared cache. This allows an origin server to state that the specified parts of the response are intended for only one user and are not a valid response for requests by other users. A private (non-shared) cache may cache the response.

`no-cache`

In general, this directive forces caches (both proxy and browser) to submit the request to the origin server for validation before releasing a cached copy, every time. This is useful to assure that authentication is respected (in combination with `public`), or to maintain rigid object freshness, without sacrificing all the benefits of caching.

Technically speaking, if the no-cache directive does not specify a field-name, then a cache can not use the response to satisfy a subsequent request without successful revalidation with the origin server. This allows an origin server to prevent caching even by caches that have been configured to return stale responses to client requests.

If the no-cache directive does specify one or more field-names, then a cache may use a response to satisfy a subsequent request, subject to any other restrictions on caching. However, the specified field-names must not be sent in the response to a subsequent request without revalidation with the origin server. This allows an origin server to prevent the re-use of certain header fields in a response, while still allowing caching of the rest of the response.

Note: Most HTTP/1.0 caches does not recognize or obey this directive.

What May be Stored by Caches?

`no-store`

The purpose of this directive is to prevent the inadvertent release or retention of sensitive information. This directive applies to the entire message, and may be sent either in a response or in a request. If sent in a request, a cache must not store any part of either this response or the request that elicited it. "Must not store" in this context means that the cache must not intentionally store the information in non-volatile storage, and must make a best-effort attempt to remove the information from volatile storage as promptly as possible after forwarding it.

In general, the purpose of this directive is to meet the stated requirements of certain users and service authors who are concerned about accidental releases of information via unanticipated accesses to cache data structures.

Modifications of the Basic Expiration Mechanism

The expiration time of an entity may be specified by the origin server using the `Expires` header field. Alternatively, it may be specified using the `max-age` directive in a response. When the max-age cache-control directive is present in a cached response, the response is stale if its current age is greater than the age value given at the time of a new request for that resource. The max-age directive on a response implies that the response is cacheable (i.e. "public") unless some other, more restrictive cache directive is also present.

If a response includes both an Expires and a max-age directive, the latter overrides the former header, even if the Expires header is more restrictive. This rule allows an origin server to provide, for a given response, a longer expiration time to an HTTP/1.1 cache than to an HTTP/1.0 cache. This might be useful if certain HTTP/1.0 caches improperly calculate ages or expiration times, perhaps due to desynchronized clocks.

`max-age` = [seconds]

Specifies the maximum amount of time that an object will be considered fresh. Similar to `Expires`, this directive allows more flexibility. [seconds] is the number of seconds from the time of the request you wish the object to be fresh for.

`s-maxage` = [seconds]

Similar to `max-age`, except that it only applies to proxy (shared) caches.

Cache Revalidation and Reload Controls

Sometimes a user agent might want or need to insist that a cache revalidate its cache entry with the origin server (and not just with the next cache along the path to the origin server), or to reload its cache entry from the origin server. End-to-end revalidation might be necessary if either the cache or the origin server has overestimated the expiration time of the cached response. End-to-end reload may be necessary if the cache entry has become corrupted for some reason.

`must-revalidate`

Tells caches that they must obey any freshness information you give them about an object. The HTTP allows caches to take liberties with the freshness of objects; by specifying this header, you 're telling the cache that you want it to strictly follow your rules.

`proxy-revalidate`

Similar to `must-revalidate`, except that it only applies to proxy caches.

For example:

```
Cache-Control:  max-age=3600, must-revalidate
```

For a more in-depth analysis of the `Cache-Control` directive's options, refer to the excellent documentation provided in the HTTP/1.1 draft specification (RFC 2616, Sections 13.1.3, 14.9).

6.2.6 Validation and Validators

In the previous paragraphs, we claimed that validation is used by servers and caches to communicate when an object has been changed. By using it, caches avoid having to download the entire object when they already have a copy locally, but they are not sure if it's still fresh. Validators are very important; if one isn't present, and there isn't any freshness information (`Expires` or `Cache-Control`) available, most caches will not store an object at all.

The most common validator is the time that the document last changes, the `Last-Modified` time. When a cache has an object stored that includes a `Last-Modified` header, it can use it to ask the server if the object has changed since the last time it was seen, with an `If-Modified-Since` request (Conditional GET).

HTTP/1.1 introduces a new kind of validator called the `ETag`. ETags are unique identifiers that are generated by the server and changed every time the object does. Because the server controls how the ETag is generated, caches can be sure that if the ETag matches when they make a `If-None-Match` request, the object really is the same.

Almost all caches use the `Last-Modified` times in determining if an object is fresh; as more HTTP/1.1 caches come online, `ETag` headers are also used. Most modern web servers (e.g. Apache) generate both `ETag` and `Last-Modified` validators for static content automatically; the maintainer of a site need not do anything. However, servers don't know enough about dynamic content (like CGI, ASP, JSP or database sites) to generated them.

7 References and Further Information

HTTP/1.1 Specification
`http://www.w3.org/Protocols/`

The HTTP/1.1 spec is the authoritative guide to implementing the protocol. At least, see the references reported in this text (Sections 6.2.4, 6.2.5).

Key Differences between HTTP/1.0 and HTTP/1.1, Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol.
`http://citeseer.nj.nec.com/7642.html`

This paper discusses the differences between the two protocol versions, as well as some of the rationale behind those changes. A MUST read document.

What's Wrong with HTTP (and why it doesn't matter), Jeffrey C. Mogul.
In *Proceedings of the USENIX Technical Conference (invited talk)*, June 1999.
`http://www.usenix.org/publications/library/proceedings/usenix99/invited_talks/mogul.pdf`

Another paper that it does worth it to be read. Jeffrey Mogul is one of the principal architects of the *persistent connection* concept.

Internet Requests For Comments (RFCs)
`http://www.faqs.org/rfcs/`

All the Internet RFC/STD/FYI/BCP archives can be searched either by name or by number in this address.

RFC-Editor
`http://www.rfc-editor.org/rfcsearch.html`

This is another location where you may search the RFCs you 're interested for. The RFC Editor is the publisher of the RFCs and is responsible for the final editorial review of the documents. The RFC Editor also maintains a master file of RFCs called the "RFC Index", which can be searched online at the specified address.

Web Caching Overview
`http://www.web-caching.com/`

An excellent introduction to caching concepts, with links to other online resources.

Known HTTP Proxy/Caching Problems (RFC 3143)
`ftp://ftp.rfc-editor.org/in-notes/rfc3143.txt`

This is an informational RFC which catalogs a number of known problems associated with the Web (caching) proxies and cache servers.

Cache Now! Campaign
`http://vancouver-webpages.com/CacheNow/`

Cache Now! is a campaign to raise awareness of caching, from all perspectives.

SQUID Web Proxy Cache
`http://www.squid-cache.org/`

Squid is the most widely used WWW proxy server with caching capabilities. It's an open source project which implements the latest industrial standards (e.g. HTTP 1.1, HTTPS, SOCKS etc.), as well as various innovative ideas proposed by universities and research institutes across the world (e.g. Internet Cache Protocol (ICP)).

HTTP Made Really Easy, James Marshall.
`http://www.jmarshall.com/easy/http/`

A practical guide to writing HTTP 1.0/1.1 clients and servers. A resource for developers who intend to write HTTP applications.