Web Protocols

Contents

[Record of Issue 1](#_Toc382391633)

[Copyright Notice 1](file:///V:/Projects/L2G/Research/L2G%20Research%20Audio%20Streaming%20Possibilities%20draft01.docx#_Toc382391634)

[Contents 2](#_Toc382391635)

[1 Introduction 3](#_Toc382391636)

[1.1 Scope 3](#_Toc382391637)

[1.2 Intended Audience 3](#_Toc382391638)

[1.3 References 3](#_Toc382391639)

[2 Streaming Audio over IP – Feature Definition 4](#_Toc382391640)

[2.1 General 4](#_Toc382391641)

[2.2 Supported Scenarios 4](#_Toc382391642)

[2.3 Bit Rate and Latency 4](#_Toc382391643)

[3 System Design: 5](#_Toc382391644)

[3.1 On the Device: 5](#_Toc382391645)

[3.2 On the Server: 5](#_Toc382391646)

[4 System analysis 6](#_Toc382391647)

[4.1 L2-G HW and SW capabilities 6](#_Toc382391648)

[4.2 Network Capabilities 6](#_Toc382391649)

[4.3 Transport Protocols 6](#_Toc382391650)

[4.4 Streaming Protocols 6](#_Toc382391651)

[4.4.1 RTP (Real-time Transport Protocol) 6](#_Toc382391652)

[4.4.2 Additional protocols to look at: 7](#_Toc382391653)

[4.4.3 Streaming Protocol – Conclusion 7](#_Toc382391654)

[4.5 Codecs 7](#_Toc382391655)

[4.5.1 Requirements 7](#_Toc382391656)

[4.5.2 Opus 7](#_Toc382391657)

[5 Conclusions 9](#_Toc382391658)

[6 Open Issues 10](#_Toc382391659)

[6.1 General 10](#_Toc382391660)

[6.2 On Device 10](#_Toc382391661)

[6.3 On Server 10](#_Toc382391662)

[7 POC 11](#_Toc382391663)

# Introduction

## Scope

This document contains

## Intended Audience

The intended audience for this document includes managers, engineers.   
  
A good technical knowledge is suggested to best understand all content in this document.

## References

|  |  |  |
| --- | --- | --- |
| Document Title | Document Path | Comments |
| HTTP/1.1 Protocol | <http://www.w3.org/Protocols/rfc2616/rfc2616> |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

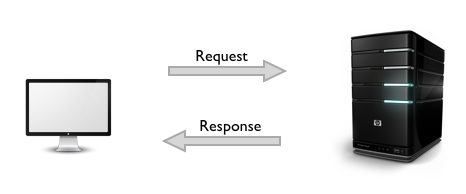
# HTTP – Hypertext Transfer Protocol

Latest version: HTTP/1.1

HTTP allows for communication between a variety of hosts and clients, and supports a mixture of network configurations.

To make this possible, it assumes very little about a particular system, and does not keep state between different message exchanges.

This makes HTTP a **stateless** protocol. The communication usually takes place over TCP/IP, but any reliable transport can be used. The default port for TCP/IP is **80**, but other ports can also be used.



Communication between a host and a client occurs, via a **request/response pair**. The client initiates an HTTP request message, which is serviced through a HTTP response message in return.

HTTP is based on the client-server architecture model and a stateless request/response protocol that operates by exchanging messages across a reliable TCP/IP connection.

An HTTP "client" is a program (Web browser or any other client) that establishes a connection to a server for the purpose of sending one or more HTTP request messages. An HTTP "server" is a program ( generally a web server like Apache Web Server or Internet Information Services IIS, etc. ) that accepts connections in order to serve HTTP requests by sending HTTP response messages.

HTTP makes use of the Uniform Resource Identifier (URI) to identify a given resource and to establish a connection. Once the connection is established, **HTTP messages** are passed in a format similar to that used by the Internet mail [RFC5322] and the Multipurpose Internet Mail Extensions (MIME) [RFC2045]. These messages include **requests** from client to server and **responses** from server to client which will have the following format:

HTTP-message = <Request> | <Response> ; HTTP/1.1 messages

HTTP requests and HTTP responses use a generic message format of RFC 822 for transferring the required data. This generic message format consists of the following four items.

* A Start-line : request/status line   
  e.g. GET /hello.htm HTTP/1.1 or HTTP/1.1 200 OK
* Zero or more header fields followed by CRLF  
  in the format: field-name ":" [ field-value ]  
  e.g. User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3

Host: www.example.com

Accept-Language: en, mi

Etc

* An empty line (i.e., a line with nothing preceding the CRLF)
* indicating the end of the header fields
* Optionally a message-body. If entity body is associated, then usually **Content-Type** and **Content-Length** headers lines specify the nature of the body associated.

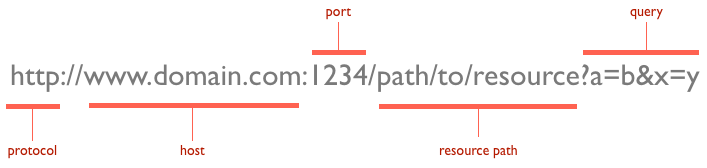
### HTTP 2.0

Is already supported in all browsers.

Download all the resources (scripts etc) in parallel and not serially like http 1.1 did.

### URLs

At the heart of web communications is the request message, which are sent via Uniform Resource Locators (URLs). I'm sure you are already familiar with URLs, but for completeness sake, I'll include it here. URLs have a simple structure that consists of the following components:



The protocol is typically http, but it can also be https for secure communications. The default port is 80, but one can be set explicitly, as illustrated in the above image. The resource path is the local path to the resource on the server.

### Verbs

URLs reveal the identity of the particular host with which we want to communicate, but the action that should be performed on the host is specified via HTTP verbs. Of course, there are several actions that a client would like the host to perform. HTTP has formalized on a few that capture the essentials that are universally applicable for all kinds of applications.

These request verbs are:

* **GET**: fetch an existing resource. The URL contains all the necessary information the server needs to locate and return the resource.
* **POST:** create a new resource. POST requests usually carry a payload that specifies the data for the new resource.
* **PUT**: update an existing resource. The payload may contain the updated data for the resource.
* **DELETE**: delete an existing resource.

The above four verbs are the most popular, and most tools and frameworks explicitly expose these request verbs. PUT and DELETE are sometimes considered specialized versions of the POST verb, and they may be packaged as POST requests with the payload containing the exact action: create, update or delete.

There are some lesser used verbs that HTTP also supports:

* **HEAD**: this is similar to GET, but without the message body. It's used to retrieve the server headers for a particular resource, generally to check if the resource has changed, via timestamps.
* **TRACE**: used to retrieve the hops that a request takes to round trip from the server. Each intermediate proxy or gateway would inject its IP or DNS name into the Via header field. This can be used for diagnostic purposes.
* **OPTIONS**: used to retrieve the server capabilities. On the client-side, it can be used to modify the request based on what the server can support.

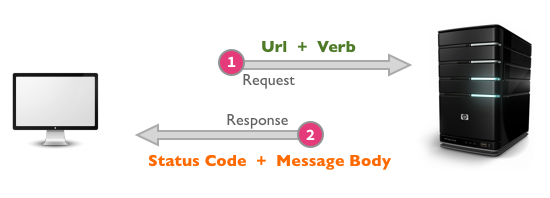
### Status Codes

With URLs and verbs, the client can initiate requests to the server. In return, the server responds with status codes and message payloads. The status code is important and tells the client how to interpret the server response. The HTTP spec defines certain number ranges for specific types of responses:

* + **1xx: Informational**It means the request was received and the process is continuing.  
    **100-continue message**, telling the client to continue sending the remainder of the request, or ignore if it has already sent it.
  + **2xx: Success**It means the action was successfully received, understood, and accepted.  
    **200 – OK**
  + **3xx: Redirection**It means further action must be taken in order to complete the request.  
    **301** Moved permanently: the resource is now located at a new URL.
  + 301 a successful POST request.  
    **303** See other: the resource is temporarily located at a new URL. The Location response header contains the temporary URL.  
    **304** Not Modified: the server has determined that the resource has not changed and the client should use its cached copy. This relies on the fact that the client is sending ETag (Enttity Tag) information that is a hash of the content. The server compares this with its own computed ETag to check for modifications.
  + **4xx: Client Error**It means the request contains incorrect syntax or cannot be fulfilled.  
    **404 Not Found** - indicates that the resource is invalid and does not exist on the server
  + **5xx: Server Error**It means the server failed to fulfill an apparently valid request.  
    **500 Internal Server Error**

### Request and Response Message Formats

So far, we've seen that URLs, verbs and status codes make up the fundamental pieces of an HTTP request/response pair.



Let's now look at the content of these messages. The HTTP specification states that a request or response message has the following generic structure:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | message = <start-line>            \*(<message-header>)            CRLF            [<message-body>]    <start-line> = Request-Line | Status-Line  <message-header> = Field-Name ':' Field-Value |
|  |  |

It's mandatory to place a new line between the message headers and body. The message can contain one or more headers, of which are broadly classified into:

* [*general headers*](http://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html#sec4.5): that are applicable for both request and response messages.
* [*request specific headers*](http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.3)*.*
* [*response specific headers*](http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.2)*.*
* [*entity headers*](http://www.w3.org/Protocols/rfc2616/rfc2616-sec7.html#sec7.1)*.*

The message body may contain the complete entity data, or it may be piecemeal if the chunked encoding (Transfer-Encoding: chunked) is used. All HTTP/1.1 clients are required to accept the Transfer-Encoding header.

#### General Headers

There are a few headers (general headers) that are shared by both request and response messages:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | general-header = Cache-Control                 | Connection                 | Date                 | Pragma                 | Trailer                 | Transfer-Encoding                 | Upgrade                 | Via                 | Warning |

We have already seen some of these headers, specifically Via and Transfer-Encoding. We will cover Cache-Control and Connection in part two.

The status code is important and tells the client how to interpret the server response.

* Via header is used in a TRACE message and updated by all intermittent proxies and gateways
* Pragma is considered a custom header and may be used to include implementation-specific headers. The most commonly used pragma-directive is Pragma: no-cache, which really is Cache-Control: no-cache under HTTP/1.1. This will be covered in Part 2 of the article.
* The Date header field is used to timestamp the request/response message
* Upgrade is used to switch protocols and allow a smooth transition to a newer protocol.
* Transfer-Encoding is generally used to break the response into smaller parts with the Transfer-Encoding: chunked value. This is a new header in HTTP/1.1 and allows for streaming of response to the client instead of one big payload.

#### Entity headers

Request and Response messages may also include entity headers to provide meta-information about the content (aka Message Body or Entity). These headers include:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | entity-header  = Allow                 | Content-Encoding                 | Content-Language                 | Content-Length                 | Content-Location                 | Content-MD5                 | Content-Range                 | Content-Type                 | Expires                 | Last-Modified |

All of the Content- prefixed headers provide information about the structure, encoding and size of the message body. Some of these headers need to be present if the entity is part of the message.

The Expires header indicates a timestamp of when the entity expires. Interestingly, a "never expires" entity is sent with a timestamp of one year into the future. The Last-Modified header indicates the last modification timestamp for the entity.

Custom headers can also be created and sent by the client; they will be treated as entity headers by the HTTP protocol.

This is really an extension mechanism, and some client-server implementations may choose to communicate specifically over these extension headers. Although HTTP supports custom headers, what it really looks for are the request and response headers, which we cover next.

#### Request Format

The request message has the same generic structure as above, except for the request line which looks like:

|  |  |
| --- | --- |
|  | Request-Line = **Method** SP **URI** SP **HTTP-Version** CRLF  Method =  | "GET"  - The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.  | "HEAD" - Same as GET, but it transfers the status line and the header section only.  | "POST" - A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.  | "PUT"  - Replaces all the current representations of the target resource with the uploaded content.  | "DELETE" - Removes all the current representations of the target resource given by URI.  | "TRACE" - Performs a message loop back test along with the path to the target resource.  | "OPTIONS" - Describe the communication options for the target resource.  | "CONNECT" - Establishes a tunnel to the server identified by a given URI |

**Request-URI**

The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request. Following are the most commonly used forms to specify an URI:

Request-URI = "\*" | absoluteURI | abs\_path | authority

* **The asterisk \*** - is used when an HTTP request does not apply to a particular resource, but to the server itself, and is only allowed when the method used does not necessarily apply to a resource. For example:   
   **OPTIONS \* HTTP/1.1**
* The **absolute URI -**  is used when an HTTP request is being made to a proxy. The proxy is requested to forward the request or service from a valid cache, and return the response. For example:  
   **GET** [**http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1**](http://www.w3.org/pub/WWW/TheProject.html%20HTTP/1.1)
* The most common form of Request-URI is that used to identify a resource on an origin server or gateway. For example,   
  a client wishing to retrieve a resource directly from the origin server would create a TCP connection to port 80 of the host "www.w3.org" and send the following lines:  
   **GET /pub/WWW/TheProject.html HTTP/1.1  
   Host:** [**www.w3.org**](http://www.w3.org)

Note that the absolute path cannot be empty; if none is present in the original URI, it MUST be given as "/" (the server root).

SP is the space separator between the tokens. HTTP-Version is specified as "HTTP/1.1" and then followed by a new line. Thus, a typical request message might look like:

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6 |  | GET /articles/http-basics HTTP/1.1  Host: www.articles.com  Connection: keep-alive  Cache-Control: no-cache  Pragma: no-cache  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8 |

Note the request line followed by many request headers. The **Host** header is mandatory for HTTP/1.1 clients. **GET** requests do not have a message body, but **POST** requests can contain the post data in the body.

The request headers act as modifiers of the request message. The complete list of known request headers is not too long, and is provided below. Unknown headers are treated as entity-header fields.

You can introduce your custom fields in case you are going to write your own custom Client and Web Server.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19 | request-header = Accept                 | Accept-Charset                 | Accept-Encoding                 | Accept-Language                 | Authorization                 | Expect                 | From                 | Host                 | If-Match                 | If-Modified-Since                 | If-None-Match                 | If-Range                 | If-Unmodified-Since                 | Max-Forwards                 | Proxy-Authorization                 | Range                 | Referer                 | TE                 | User-Agent |

The Accept prefixed headers indicate the acceptable media-types, languages and character sets on the client. From, Host, Referer and User-Agent identify details about the client that initiated the request. The If- prefixed headers are used to make a request more conditional, and the server returns the resource only if the condition matches. Otherwise, it returns a 304 Not Modified. The condition can be based on a timestamp or an ETag (a hash of the entity).

**Examples of Request Message**

Now let's put it all together to form an HTTP request to fetch **hello.htm** page from the web server running on tutorialspoint.com

GET /hello.htm HTTP/1.1

User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

Host: www.tutorialspoint.com

Accept-Language: en-us

Accept-Encoding: gzip, deflate

Connection: Keep-Alive

Here we are not sending any request data to the server because we are fetching a plain HTML page from the server. Connection is a general-header, and the rest of the headers are request headers. The following example shows how to send form data to the server using request message body:

POST /cgi-bin/process.cgi HTTP/1.1

User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

Host: www.tutorialspoint.com

Content-Type: application/x-www-form-urlencoded

Content-Length: **length**

Accept-Language: en-us

Accept-Encoding: gzip, deflate

Connection: Keep-Alive

licenseID=string&content=string&/paramsXML=string

Here the given URL */cgi-bin/process.cgi* will be used to process the passed data and accordingly, a response will be returned. Here **content-type** tells the server that the passed data is a simple web form data and **length** will be the actual length of the data put in the message body. The following example shows how you can pass plain XML to your web server:

POST /cgi-bin/process.cgi HTTP/1.1

User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

Host: www.tutorialspoint.com

Content-Type: text/xml; charset=utf-8

Content-Length: **length**

Accept-Language: en-us

Accept-Encoding: gzip, deflate

Connection: Keep-Alive

<?xml version="1.0" encoding="utf-8"?>

<string xmlns="http://clearforest.com/">string</string>

#### Response Format

The response format is similar to the request message, except for the status line and headers. The status line has the following structure:

|  |  |
| --- | --- |
| 1 | Status-Line = **HTTP-Version** SP **Status-Code** SP **Reason-Phrase** CRLF |

* HTTP-Version is sent as HTTP/1.1
* The Status-Code is one of the many statuses discussed earlier.
* The Reason-Phrase is a human-readable version of the status code.

A typical status line for a successful response might look like so:

|  |  |
| --- | --- |
| 1 | HTTP/1.1 200 OK |

The response headers are also fairly limited, and the full set is given below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | response-header = Accept-Ranges                  | Age                  | ETag                  | Location                  | Proxy-Authenticate                  | Retry-After                  | Server                  | Vary                  | WWW-Authenticate |

* Age is the time in seconds since the message was generated on the server.
* ETag is the MD5 hash of the entity and used to check for modifications.
* Location is used when sending a redirection and contains the new URL.
* Server identifies the server generating the message.

It's been a lot of theory upto this point, so I won't blame you for drowsy eyes. In the next sections, we will get more practical and take a survey of the tools, frameworks and libraries.

**Examples of Response Message**

Now let's put it all together to form an HTTP response for a request to fetch the **hello.htm** page from the web server running on tutorialspoint.com

HTTP/1.1 200 OK

Date: Mon, 27 Jul 2009 12:28:53 GMT

Server: Apache/2.2.14 (Win32)

Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT

Content-Length: 88

Content-Type: text/html

Connection: Closed

<html>

<body>

<h1>Hello, World!</h1>

</body>

</html>

The following example shows an HTTP response message displaying error condition when the web server could not find the requested page:

HTTP/1.1 404 Not Found

Date: Sun, 18 Oct 2012 10:36:20 GMT

Server: Apache/2.2.14 (Win32)

Content-Length: 230

Connection: Closed  
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">

<html>

<head>

<title>404 Not Found</title>

</head>

<body>

<h1>Not Found</h1>

<p>The requested URL /t.html was not found on this server.</p>

</body>

</html>

## Tools to View HTTP Traffic

[**Chrome/Webkit inspector**](https://developers.google.com/chrome-developer-tools/)

There are also web debugging proxies, like [**Fiddler**](http://www.fiddler2.com/fiddler2/) on Windows and [**Charles Proxy**](http://www.charlesproxy.com/) for OSX

For the command line, we have utilities like [curl](http://curl.haxx.se/), [tcpdump](http://www.tcpdump.org/) and [tshark](http://www.wireshark.org/docs/wsug_html_chunked/AppToolstshark.html) for monitoring HTTP traffic.

# SSL

SSL (Secure Socket Layer) is a protocol layer that exists between the Network Layer and Application layer. As the name suggest SSL provides a mechanism for encrypting all kinds of traffic - LDAP, POP, IMAP and most importantly HTTP.

The following is a over-simplified structure of the layers involved in SSL.

|  |
| --- |
| +-------------------------------------------+  | LDAP | HTTP | POP | IMAP |  +-------------------------------------------+  | SSL |  +-------------------------------------------+  | Network Layer |  +-------------------------------------------+ |

# WebDAV

WebDAV (Web-based Distributed Authoring and Versioning) is a file sharing and management protocol based on the Hypertext Transfer Protocol (HTTP). The WebDAV protocol allows you to edit documents directly over the Internet, making the Web a readable and writable medium. The beauty is – **since it’s an extension from HTTP, most firewalls won’t block it.**

**Most operating systems have built-in support for WebDAV**. The setting is similar to a network drive or a network shared folder. With no updates required, applications can access files stored in WebDAV folder over the Internet seamlessly. Many cloud storage services already employ WebDAV as a standard to access their online storages. One of the examples is Apple’s MobileMe iDisk service, which appears in your Mac Desktop just like a local disk.

And for those who seek having an extra layer of protection, WebDAV supports secure connection as well. By enabling HTTPS over all WebDAV connections, security is fortified. You can also install SSL certificates to increase security measure.

The protocol consists of a set of new methods and headers for use in HTTP. The added methods include:

* **PROPFIND** — used to retrieve properties, stored as [XML](https://en.wikipedia.org/wiki/XML), from a [web resource](https://en.wikipedia.org/wiki/Web_resource). It is also [overloaded](https://en.wikipedia.org/wiki/Method_overloading) to allow one to retrieve the collection structure (a.k.a. directory hierarchy) of a remote system.
* PROPPATCH — used to change and delete multiple properties on a resource in a single [atomic act](https://en.wikipedia.org/wiki/Atomic_commit)
* **MKCOL** — used to create collections (a.k.a. a [directory](https://en.wikipedia.org/wiki/Folder_%28computing%29))
* **COPY** — used to copy a resource from one [URI](https://en.wikipedia.org/wiki/Uniform_resource_identifier) to another
* **MOVE** — used to move a resource from one [URI](https://en.wikipedia.org/wiki/Uniform_resource_identifier) to another
* LOCK — used to put a [lock](https://en.wikipedia.org/wiki/Lock_%28computer_science%29) on a resource. WebDAV supports both shared and exclusive locks.
* UNLOCK — used to remove a lock from a resource

##### [Terms](http://ezcomponents.org/docs/api/trunk/introduction_Webdav.html#id7)

There are some terms used in a WebDAV environment whose meanings differ slightly from the usage in similar environments.

* **Collection** - When it comes to WebDAV, a collection means a set of files and other collections, which may be compared with directories in a normal file system.
* **Resource** - A resource equals a file, but we use a different term here to differentiate between real files on the hard disk and the virtual resources (files) in a WebDAV share.
* **Properties** - There are several default properties, like the modification time or file size of WebDAV resources, but you can also store and modify custom properties on all resources.