09. HTTP Web Browser

WWW. Hypertext and Hypermedia

The Web is a **distributed hypermedia** system that support interactive access to Hypermedia documents (**Resources**).

Hypermedia, as opposed to Hypertext, Resources potentially contain:

- Different types of information, including text, pictures, graphics, audio, etc. Examples include: HTML files, image files, query results, etc.
- Hyperlinks to other Resources.

From a Network Programming perspective, these resources are treated as Data.

WWW and the Client-server paradigm

The distributed nature of the Web means that the Resources/Data are potentially spread across a number of computers across the Internet.

This lends itself well to the Client-server paradigm as follows:

- **On the Client-side**: Here sits the consumer of the Resources/Data, the end-users. They typically interact with a client application known as a Browser.
- **On the Server-side**: Here is where the resource repositories are located, on a remote server-class machine. Access to these resources is typically controlled by a Web server.

Problems to be addressed

This distribution of resources also introduces a number of potential problems:

- The resources may be updated, moved or removed without notification to the client applications.
- Accessing Resources on remote host machines has implications for network bandwidth usage.

These problems can affect the end-user experience and the network.

Client-server interaction. HTTP

Browsers and servers interact with each other using the HyperText Transfer Protocol (HTTP). This is a network protocol used to deliver virtually all Resources on the Web.

- The Browser client sends **HTTP Request** messages to a Web Server. These messages typically (but not always) contain requests for a Resource.
- The Web server returns **HTTP Response** messages to the clients. These messages typically contain Resources/Data (but not always).

Operation of Web Browsers and Servers

In order to appreciate the potential problems to be addressed when developing Browsers and Web servers, it is important to understand their operation.

Web servers perform a straightforward task over and over again:

- Accept connection requests from clients
- Accept and parse incoming HTTP Requests
- Retrieve and return HTTP Responses indicating success or failure
- Close the connection

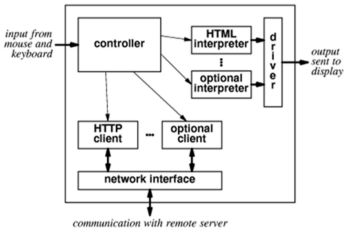
Browser Architecture

Web browsers are much more complex in their operation. The functions of a Browser include:

- Rendering and displaying disparate (different types) resources to the user
- User interaction
- Initiating interaction with Web servers to retrieve resources or in some instances to upload resources

The Browser provides these services seamlessly using a number of software components.

A browser consists of the following components:



- A set of clients for uploading/retrieving Resources.
- A set of interpreters for displaying/rendering Resources.
- A controller to manage them all. The Controller is responsible for interpreting user input via the keyboard and mouse clicks and invoking interpreter and client components at the appropriate time.

All browsers minimally contain a basic HTTP client, a basic HTML interpreter and a Controller. Modern Browsers contain much more.

Interpreter and Client components

An example interpreter is the **HTML interpreter**. It parses documents that contain standard HTML code and renders the content to the local screen. Other interpreters are needed for displaying pictures, video, audio, etc.

An example client is the **HTTP client**. It is used to interact with HTTP servers for the purpose of retrieving/uploading Resources. Other clients are needed for sending/receiving e-mail messages, file transfer using TCP, etc. The first field in the destination URL identifies the client component to invoke.

Document Transfer and HTTP

From a Network Programming perspective we are interested in the HTTP client and its interaction with HTTP servers. This interaction consists of an exchange of HTTP Requests and Responses:

- These are typically sent as plain-text encoded in ASCII (in English).
- This means that when viewed with a protocol analyzer such as Wireshark, the Application Data field can be easily read and understood.

HTTP Requests

HTTP Requests originate from the HTTP client. They support a number of operations through a set of **methods**:

- GET, HEAD, POST, OPTIONS, PUT, DELETE, TRACE and CONNECT.
- We will use GET and HEAD. These two methods should adequately demonstrate the problems to be addressed.

HTTP Responses originate from the HTTP server. Recall the problems previously outlined in relation to broken links, re-located Resources and Bandwidth limitations:

- HTTP includes a lot of functionality to address these problems.
- The server typically sends additional information with each transfer of data.
- This additional information allows the HTTP client to call an appropriate interpreter to display/render the Resource data, to infer an error condition, etc.

Structure of HTTP Messages

Recall that each layer of the Protocol Hierarchy specifies a “framing-type” structure known as a Protocol Data Unit (PDU). For example: a Data Link Frame, an IP Datagram/Packet, a TCP segment, etc.

HTTP is also a protocol which exists in the Application Layer. There are many other protocols that exist in the Application layer.

When talking about Application layer protocols the term PDU has no real meaning. This is because Application layer protocols typically follow a **request-response** or **command-response** model of interaction:

- Clients typically request something from the server or issue a command to the server.
- Servers typically respond to the Client with an indication of success or failure.
- However, sometime servers issues requests and commands.

Application layer protocols are more usefully described in terms of **Syntax** and **Semantics**.

Syntax and Semantics

Syntax describes the structure of the request-response messages.

Semantics describes the interaction between the client and the server. Essentially, this relates to the sequence of request/response messages. This

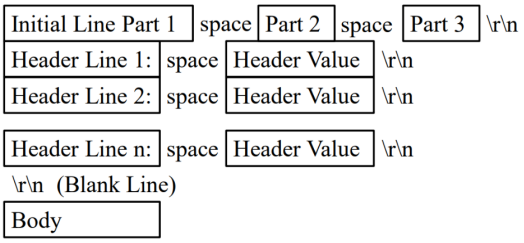
can be described as “who talks first”, which side issues the first message.

Syntax of HTTP Messages

HTTP Messages have a particular structure or format. The format of the **Requests** and **Responses** messages are similar. Both consists of:

- An initial line
- Zero or more Header Lines
- A blank line
- An optional Message Body, typically containing Resource data from a file or a query output

```
<inital line, different for request and
Header1: value1
Header2: value2
Header3: value3
<Blank line>
<optional message body, file contents or
```



Initial lines and header should end in CRLF (”\r\n”).

Initial Request Line

The initial line for a Request line has three parts separated by spaces:

```
GET /path/to/file/index.html
```

- Method name
- Local path of the requested resource
- HTTP version being used

GET is the most common HTTP method. It is used to fetch resources.

- Method names are always uppercase.
- The path is the part of the URL after the host name.
- The HTTP version always takes the form “HTTP/x.x”, uppercase.

Initial Response Line

The initial response line also has three parts separated by spaces:

```
HTTP/1.0 200 OK
HTTP/1.0 404 Not
```

- HTTP version
- Response status code that gives the result of the request
- English reason phrase describing status code

- The HTTP version is of format “HTTP/x.x”.
- The status code is meant to be computer-readable. It comprises a three-digit integer, and the first digit identifies the general category of response.
- The reason phrase is meant to be human-readable and may vary.

Header Lines

Header lines provide information about the request or response, or about the Resources sent in the message body.

The header lines are in a particular format:

- One line per header, of the form “Header-Name: value”, ending with CRLF.
- Similar format used for email, defined in RFC 822.
- The header name is not case-sensitive, but the value may be.

Versions of HTTP

HTTP 1.0	Older	Defines 16 headers, none are required
HTTP 1.1	Newer	Defines 46 headers, one (Host:) is required in requests

HTTP/2.0	Latest version	Addresses inefficiencies of HTTP 1.1

We will focus on HTTP/1.0 and HTTP/1.1

Header Field Name	Description	Example
Accept	Content-Types that are acceptable for the response	`Accept: text/plain`
Cache-Control	Used to specify directives that *must* be obeyed by all caching mechanisms along the request-response chain	`Cache-Control: no-cache`
Connection	What type of connection the user-agent would prefer	`Connection: keep-alive`
Cookie	An HTTP cookie previously sent by the server with `Set-Cookie` (below)	`Cookie: $Version=1; Skin=new;`
Content-Length	The length of the request body in octets (8-bit bytes)	`Content-Length: 348`
Content-Type	The MIME type of the body of the request (used with POST and PUT requests)	`Content-Type: application/x-www-form-urlencoded`
Date	The date and time that the message was sent (in "HTTP-date" format as defined by RFC 7231)	`Date: Tue, 15 Nov 1994 08:12:31 GMT`
From	The email address of the user making the request	`From: user@example.com`
If-Modified-Since	Allows a *304 Not Modified* to be returned if content is unchanged	`If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT`
If-None-Match	Allows a 304 Not Modified to be returned if content is unchanged, see HTTP ETag	`If-None-Match: "737060cd8c284d8af7ad3082f209582d"`
User-Agent	The user agent string of the user agent	`User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:12.0) Gecko/20100101 Firefox/21.0`

Example Request Header Lines

Field name	Description	Example
Age	The age the object has been in a proxy cache in seconds	`Age: 12`
Cache-Control	Tells all caching mechanisms from server to client whether they may cache this object. It is measured in seconds	`Cache-Control: max-age=3600`
Connection	Options that are desired for the connection[20]	`Connection: close`
Content-Encoding	The type of encoding used on the data. See HTTP compression.	`Content-Encoding: gzip`
Content-Length	The length of the response body in octets (8-bit bytes)	`Content-Length: 348`
Content-Location	An alternate location for the returned data	`Content-Location: /index.htm`
Content-Range	Where in a full body message this partial message belongs	`Content-Range: bytes 21010-47021/47022`
Content-Type	The MIME type of this content	`Content-Type: text/html; charset=utf-8`
Date	The date and time that the message was sent (in "HTTP-date" format as defined by RFC 7231)	`Date: Tue, 15 Nov 1994 08:12:31 GMT`
ETag	An identifier for a specific version of a resource, often a message digest	`ETag: "737060cd8c284d8af7ad3082f209582d"`
Expires	Gives the date/time after which the response is considered stale	`Expires: Thu, 01 Dec 1994 16:00:00 GMT`
Last-Modified	The last modified date for the requested object (in "HTTP-date" format as defined by RFC 7231)	`Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT`
Location	Used in redirection, or when a new resource has been created.	`Location: http://www.w3.org/pub/WWW/People.html`
Server	A name for the server	`Server: Apache/2.4.1 (Unix)`
Set-Cookie	An HTTP cookie	`Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1`

Example Response Header Lines

Header Lines. Net-politeness

Consider including the following headers in **client requests**:

- A **From** header gives the email address of the person making the request or running the program. This must be user-configurable for privacy concerns.
- A **User-Agent** header identifies the program that is making the request, in the form “Program-name/x.xx”, where x.xx is the alphanumeric version of the program.
- Example: Netscape 3.0 sends “User-agent: Mozilla/3.0Gold”

Consider including the following headers in **server responses**:

- A **Server** header, similar to the User-Agent header, it identifies the server software in the form “Program-name/x.xx”
- Example: An Apache server might return “Server: Apache/1.2b3-dev”.
- The **Last-Modified** header gives the modification date (in GMT) of the resource that’s being returned. It is used in caching.
- Example: Last-Modified: Fri, 31 Dec 1999 23:59:59 GMT

The Message Body

An HTTP message may have a body of data sent after the header lines.

In a **response** this is where the requested resource is returned to the client (most common use of the message body) or some explanatory text for an error

condition.

In a **request** this is where form data or uploaded files are sent to the server.

If a HTTP message includes a body, there are usually header lines in the message that describe the body.

- The **Content-Type** header gives the MIME-type of the data in the body, such as text//html or image/gif.
- The **Content-Length** header gives the number of bytes in the body.

Other HTTP Methods. HEAD and POST

Two other commonly used methods are HEAD and POST.

The HEAD Method

Similar to a GET request, except it asks the server to return the response headers only, not the actual resource, no message body.

It is useful for checking characteristics of a resource without actually downloading it.

The response to a HEAD request must never contain a message body.

The POST Method

A POST request is used to send data to the server to be processed in some way, like by a CGI script.

It differs from a GET request in the following ways:

- There’s a block of data sent with the request in the message body. There are usually extra headers to describe this message body, like **Content-Type:** and **Content-Length:**.
- The request URI is not a resource to retrieve, it’s usually a program to handle the data you’re sending.

HTTP Versions 1.0 and 1.1

There are some key differences between them in the following areas: Persistent connections, multi-hosting and more efficient cache control.

HTTP/1.0 Connections

With HTTP/1.0 the connection between the server and the client is closed by the server immediately after the HTTP Response is transmitted.

Consider a simple HTML page containing five image tags:

- Downloading and rendering this page requires six connection establishments and cessations. One for the HTML page and one for each of the images, which are downloaded separately on a separate connection.

This behaviour has implications for:

- **Network bandwidth**: due to the amount of extra segments required to establish and terminate connections.
- **Internal TCP resources**: each connection consumes resources within the TCP memory space.

HTTP/1.1 Persistent Connections

With HTTP/1.1, the connection between the server and the client remains open by default. This allows for multiple HTTP Requests to be submitted across a single connection.

This default behaviour is not always required and it can be overridden using the **Connection:** header as follows: `Connection: close\r\n` . This is an instruction to the server to close the connection after the resource has been returned.

HTTP/1.1 facilitates **Multi-hosting**, multiple web servers sharing a single IP address. The problem is that each server is listening on Port 80 and this causes problems for TCP determining which server should receive an incoming HTTP request.

The problem is solved using the **Host:** header as follows: `Host: www.tudublin.ie` . Its inclusion is mandatory in all HTTP/1.1 requests.