HTTP

1 Overview

HTTP is the most important protocol in the Internet and makes what the World Wide Web is now possible.

HTTP is short for *HyperText Transfer Protocol*, which was brought forward by Tim Berners-Lee and his colleagues at CERN (European High-Energy Particle Physics Laboratory) in 1989. To enable collaboration between physicists and other researchers in high energy physics community, they wrote a proposal and brought forward a document-sharing system. In the system, three new technologies are incorporated: HTML (HyperText Markup Language) used to write the web documents, HTTP to transmit the pages, and a web browser client program to receive and interpret data and display results. This is still how the current world-wide web works.

Tim Berners-Lee's idea was usually believed inspired by Xanadu Project. An article presenting the project's main idea has been added to the reference section on our course web page. You are required to read it.

In the TCP/IP context, as we mentioned before, there are 5 layers. HTTP is a protocol belonging to the topmost application layer and usually implemented based on socket service provided by TCP protocol.

Like most network protocols, HTTP uses the client-server model: An HTTP client opens a connection and sends a request message to an HTTP server; the server then returns a response message, usually containing the resource that was requested. After delivering the response, the server closes the connection (making HTTP a *stateless* protocol, i.e. not maintaining any connection information between transactions). Typically, an HTTP server listens on port 80, though it may use any port available.

2 Resources and URI

Everyone should have had web surfing experience so far. We typically use a graphical web browser. And by either providing explicitly an address in the address field or clicking a

link in the current web page, we tells the browser to request the corresponding *resource* on the server. We say *resource* instead of *HTML page* as we usually see because what we are supposed to request could be anything: HTML pages, JPEG images, executable programs, etc.. The resource may already be available on the server or be generated dynamically by the server upon request.

The address either provided explicitly or hidden behind the link is called the *Uniform Resource Identifier* (*URI*) of the resource. Commonly we use the term, *Uniform Resource Locator* (*URL*). Theoretically they are different. URI is more general. A URI could be a URL or a *Uniform Resource Name* (*URN*). The difference between URL and URN is that URNs are used for identification, while URLs for locating or finding resources.

A URN identifies a resource or unit of information. It may identify, for example, intellectual content, a particular presentation of intellectual content, or whatever a name assignment authority determines is a distinctly namable entity. A URL identifies the location or a container for an instance of a resource identified by a URN. The resource identified by a URN may reside in one or more locations at any given time, may move, or may not be available at all.

A URL, e.g. `http://www-cs.ccny.cuny.edu/~jniu/index.html`, typically consists of three parts: Protocol, i.e. the http scheme; Host – the server to contact, `www-cs.ccny.cuny.edu`; and path – identifies document on that host `/index.html`.

According to *RFC 1737 – Functional Requirements for Uniform Resource Names*, the requirements for URNs' functional capabilities are as follows:

- Global scope: A URN is a name with global scope which does not imply a location. It has the same meaning everywhere.
- Global uniqueness: The same URN will never be assigned to two different resources.
- Persistence: It is intended that the lifetime of a URN be permanent. That is, the URN will be globally unique forever, and may well be used as a reference to a resource well beyond the lifetime of the resource it identifies or of any naming authority involved in the assignment of its name.
- Scalability: URNs can be assigned to any resource that might conceivably be available on the network, for hundreds of years.
- Legacy support: The scheme must permit the support of existing legacy naming systems, insofar as they satisfy the other requirements described here. For example, ISBN numbers, ISO public identifiers, and UPC product codes seem to satisfy the functional requirements, and allow an embedding that satisfies the syntactic requirements described here.

- Extensibility: Any scheme for URNs must permit future extensions to the scheme.
- Independence: It is solely the responsibility of a name issuing authority to determine the conditions under which it will issue a name.
- Resolution: A URN will not impede resolution (translation into a URL, q.v.). To be more specific, for URNs that have corresponding URLs, there must be some feasible mechanism to translate a URN to a URL.

The relationship between URL and URN is similar to that between DNS names, e.g. `www.cuny.edu`, and IP addresses, e.g. `134.74.192.6`. For the present, we have a lot of URLs but few URNs.

2.1 Document Organization on Server

An HTTP server typically listens on port 80, gets HTTP requests, and sends back responses. In the simplest case, the connection is closed at the end of a single request/response pair.

Usually, the HTTP server has a *document root* that points to the directory in the filesystem that contains the documents to serve up. Suppose `/etc/htdocs/` is the document root for an HTTP server (`htdocs` is short for hyper-text documents). There are subdirectories `a`, `b`, and `c`: (`/etc/htdocs/a/`, `/etc/htdocs/b/`, `/etc/htdocs/c/`), and each of those contains an `x.html` file. The server gets request paths like `/a/x.html` and `/b/x.html`, and maps them into the filesystem. That is the request path `/a/x.html` maps into the filesystem document `/etc/htdocs/a/x.html`.

3 HTTP Requests and Responses

The format of the request and response messages are similar, and English-oriented. Both kinds of messages consist of:

- an initial line,
- zero or more header lines,
- a blank line (i.e. a `CRLF` by itself), and
- an optional message body (e.g. a file, or query data, or query output).

If we put it in another way, the format of an HTTP message is:

- <initial line, different for request vs. response>

- Header1: value1
- Header2: value2
- Header3: value3
- <optional message body goes here, like file contents or query data; it can be many lines long, or even binary data `&*%@!^@#`>

Initial lines and headers should end in `CRLF`, though you should gracefully handle lines ending in just LF. (More exactly, `CR` and `LF` here mean ASCII values 13 and 10, even though some platforms may use different characters.)

3.1 HTTP Requests

The initial line is different for the request than for the response. A request line has three parts, separated by spaces: a method name, the local path of the requested resource, and the version of HTTP being used. A typical request line is:

```
GET /path/to/file/index.html HTTP/1.0
```

Note that

- *GET* is the most common HTTP method; it says "give me this resource". Other methods include *POST*, *HEAD*, *PUT*, and *DELETE* (more on those later). 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.

3.2 HTTP Responses

The initial response line, called the *status line*, also has three parts separated by spaces: the HTTP version, a response status code that gives the result of the request, and an English reason phrase describing the status code. Typical status lines are:

```
HTTP/1.0 200 OK
```

or

```
HTTP/1.0 404 Not Found
```

Notes:

- The HTTP version is in the same format as in the request line, `HTTP/x.x`.
- The status code is meant to be computer-readable; the reason phrase is meant to be human-readable, and may vary.
- The status code is a three-digit integer, and the first digit identifies the general category of response:
  - **1xx** indicates an informational message only.
  - **2xx** indicates success of some kind.

    200 OK The request succeeded, and the resulting resource (e.g. file or script output) is returned in the message body.
  - **3xx** redirects the client to another URL.

    301 Moved Permanently The `Location:` field of the header gives the correct URL.

    302 Moved Temporarily The server may suggest the correct URL using a `Location:` in the header.

    303 See Other (HTTP 1.1 only) The resource has moved to another URL (given by the Location: response header), and should be automatically retrieved by the client. This is often used by a CGI script to redirect the browser to an existing file.

    304 Not Modified The request had `If-Modified-Since:` field, but the document has not been modified, so the client should use its cached copy.
  - **4xx** indicates an error on the client's part.

    401 = Unauthorized

    403 = Forbidden The access to the specified resource is denied.

    404 Not Found The requested resource doesn't exist.
  - **5xx** indicates an error on the server's part.

    500 Server Error An unexpected server error. The most common cause is a server-side script that has bad syntax, fails, or otherwise can't run correctly.

    503 Service Unavailable The server is temporarily not able to provide the service. The header may contain a `Retry-After:` field to indicate when the client might give it another shot.

A complete list of status codes with more details is in the HTTP specification – RFC 2616.

3.3 Header Lines

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

The header lines are in the usual text header format, which is: one line per header, of the form *Header-Name: value*, ending with `CRLF`. It's the same format used for email and news postings, defined in RFC 822, section 3. Details about RFC 822 header lines:

- The header name is not case-sensitive (though the value may be).
- Any number of spaces or tabs may be between the ":" and the value.
- Header lines beginning with space or tab are actually part of the previous header line, folded into multiple lines for easy reading.

  Thus, the following two headers are equivalent:

```
Header1: some-long-value-1a, some-long-value-1b

HEADER1:    some-long-value-1a,
            some-long-value-1b
```

HTTP 1.0 defines 16 headers, though none are required. HTTP 1.1 defines 46 headers, and one (`Host:`) is required in requests. For Net-politeness, consider including these headers in your requests:

- The `From:` header gives the email address of whoever's making the request, or running the program doing so. (This must be user-configurable, for privacy concerns.)
- The `User-Agent:` header identifies the program that's making the request, in the form `Program-name/x.xx`, where `x.xx` is the (mostly) alphanumeric version of the program. For example, Netscape 3.0 sends the header `User-agent: Mozilla/3.0Gold`.

These headers help web masters troubleshoot problems. They also reveal information about the user. When you decide which headers to include, you must balance the web masters' logging needs against your users' needs for privacy.

If you're writing servers, consider including these headers in your responses:

- The `Server:` header is analogous to the `User-Agent:` header: it identifies the server software in the form `Program-name/x.xx`. For example, one version of Apache returns `Server: Apache/1.3.6`.
- The `Last-Modified:` header gives the modification date of the resource that's being returned. It's used in caching and other bandwidth-saving activities. And Greenwich Mean Time is used, in the format

```
Last-Modified: Fri, 31 Dec 1999 23:59:59 GMT
```

3.4 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 (the most common use of the message body), or perhaps explanatory text if there's an error. In a request, this is where user-entered data or uploaded files are sent to the server.

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

- The `Content-Type` header gives the MIME-type of the data in the body, such as `text/html` or `image/gif`, see section 3.5.
- The `Content-Length` header gives the number of bytes in the body, i.e. the number of bytes to read after the blank line following the header lines. Some HTTP/1.1 variants get rid of this field, since it means the server cannot send any data until it knows how many bytes there are.

3.5 MIME Types

MIME, short for *Multipurpose Internet Mail Extensions*, is a standard which predates HTTP for identifying different types of data for inclusion in email messages.

Each MIME type is specified in the form of

```
content-type/sub-type [;aux-info]
```

For example,

```
text/html
text/plain
text/plain ; charset = us-ascii
multipart/mixed ; boundary = SpecialBoundaryString
application/pdf
application/postscript
audio/basic -- .au audio
image/jpeg
video/quicktime
```

When a resource is requested, the server determines its MIME type. It may use the file extension (`.html`, `.pdf`) or some other scheme. The client sees the type in the HTTP response header, and so knows what to look for after the blank line. The date may be binary instead of text, for example GIF, JPEG. The browser may use plug-ins to handle some MIME types.

3.6 HTTP Interaction Example

```
bash-2.03$ telnet www.cs.gc.cuny.edu 80
Trying 146.96.245.5...
Connected to www.cs.gc.cuny.edu.
Escape character is '^]'.
GET /index.html HTTP/1.0

HTTP/1.1 200 OK
Date: Wed, 18 Feb 2004 11:30:13 GMT
Server: Apache/2.0.44 (Unix)
Last-Modified: Mon, 03 Dec 2001 14:29:14 GMT
ETag: "de02-106-34700280"
Accept-Ranges: bytes
Content-Length: 262
Connection: close
Content-Type: text/html

<HTML>
<HEAD>
<META HTTP-EQUIV="refresh" content="0";
  URL=http://web.gc.cuny.edu/ComputerScience">
</HEAD>
<BODY>

<a href="http://web.gc.cuny.edu/ComputerScience">
  Click here to visit the website of CUNY Computer Science Ph.D. program.
</a>
</BODY>
</HTML>
Connection closed by foreign host.
```