

HTTP

Hypertext Transfer Protocol

Refs: RFC 1945 (HTTP 1.0)

HTTP Usage

- HTTP is the protocol that supports communication between web browsers and web servers.
- A “Web Server” is a HTTP server
- We will look at HTTP Version 1.0

From the RFC

“HTTP is an application-level protocol with the lightness and speed necessary for distributed, hypermedia information systems.”

Transport Independence

- The RFC states that the HTTP protocol generally takes place over a TCP connection, but the protocol itself is not dependent on a specific transport layer.

Request - Response

- HTTP has a simple structure:
 - client sends a request
 - server returns a reply.
- HTTP can support multiple request-reply exchanges over a single TCP connection.

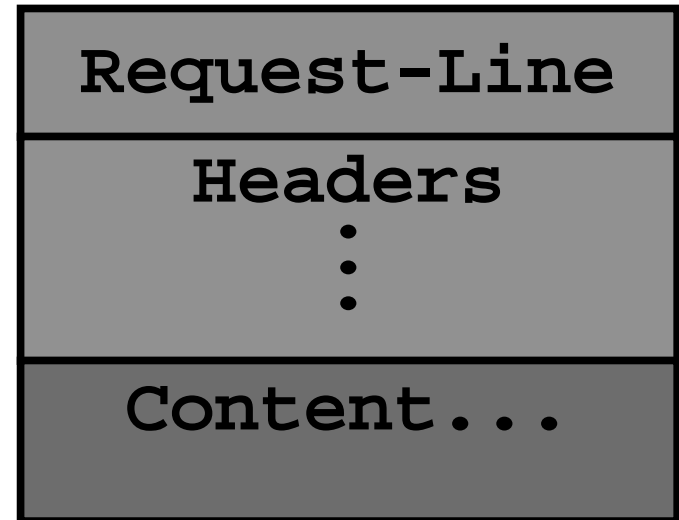
Well Known Address

- The “well known” TCP port for HTTP servers is port 80.
- Other ports can be used as well...

HTTP Versions

- The original version now goes by the name “HTTP Version 0.9”
 - HTTP 0.9 was used for many years.
- Starting with HTTP 1.0 the version number is part of every request.
- HTTP is still changing...

HTTP 1.0 Request



- Lines of text (ASCII).
- Lines end with CRLF `"\r\n"`
- First line is called "Request-Line"

Request Line

Method URI HTTP-Version \r\n

- The request line contains 3 *tokens* (words).
- space characters “ ” separate the tokens.
- Newline seems to work by itself (but the protocol requires CRLF)

Request Method

- The Request Method can be:

GET

HEAD

PUT

POST

DELETE

LINK

UNLINK

future expansion allowed

Methods

- GET: retrieve information identified by the URI.
- HEAD: retrieve meta-information about the URI.
- POST: send information to a URI and retrieve result.

Methods (cont.)

- PUT: Store information in location named by URI.
- DELETE: remove *entity* identified by URI.
- LINK, UNLINK: create/destroy a link relationship...?...?

Common Usage

- GET, HEAD and POST are supported everywhere.
- HTTP 1.1 servers often support PUT, DELETE, OPTIONS & TRACE.

URI

Universal Resource Identifier

- URIs defined in RFC 1630.

- Full URI: `proto://hostname/path`
`http://www.cs.rpi.edu:80/blah/foo`

Identifies the Server



- Partial URI: `/path`
`/blah/foo`

No server mentioned 

URI Usage

- When dealing with a HTTP server, only a partial URI is used.
- When dealing with a *proxy* HTTP server, a full URI is used.
 - client has to tell the proxy where to get the document!
 - *more on proxy servers in a bit....*

HTTP Version Number

`"HTTP/1.0"` or `"HTTP/1.1"`

HTTP 0.9 did not include a version number in a request line.

If a server gets a request line with no HTTP version number it assume 0.9

The Header Lines

- After the *Request-Line* come a number of HTTP *headers*.
- Each header line contains an attribute name followed by a “:” followed by the attribute value.

Headers

- Request Headers provide information to the server about the client
 - what kind of client
 - what kind of content will be accepted
 - who is making the request
- There can be 0 headers

Example HTTP Headers

Accept: text/html

From: neytmann@cybersurg.com

User-Agent: Netscape 4.7

Referer: http://foo.com/blah

End of the Headers

- Each header ends with a CRLF
- The end of the header section is marked with a blank line
 - `"\r\n\r\n"`
- For GET and HEAD requests the end of the headers is the end of the request!

POST

- A POST request includes some *data* after the headers (after the blank line).
- There is no format for the data (just raw bytes).
- A POST request must include a Content-Length line in the headers:
`Content-Length: 267`

Example Request

GET /~hollingd/testanswers.html HTTP/1.0

Accept: */*

User-Agent: Internet Explorer

From: cheater@cheaters.org

Referer: http://foo.com/

Example Post

POST /CGI-BIN/add_appointments HTTP/1.0

Accept: */*

User-Agent: Internet Explorer

Content-Length: 34

1220=surgery&0110=doom&0320=bypass

Typical Method Usage

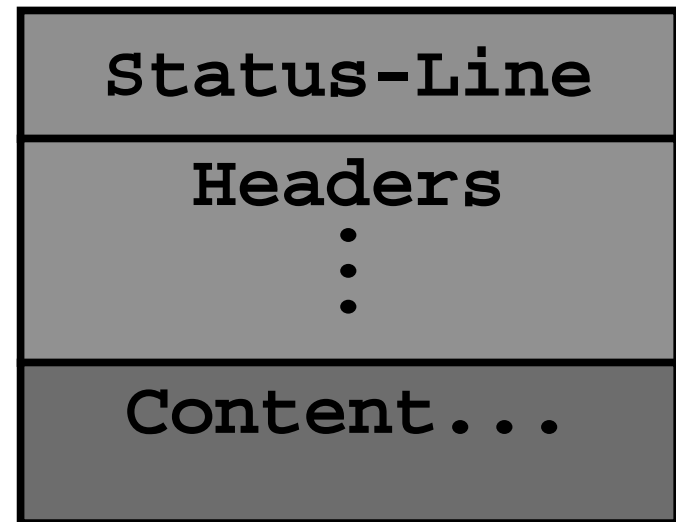
GET used to retrieve an HTML document.

HEAD used to find out if a document has changed.

POST used to submit a form.

HTTP Response

- ASCII Status Line
- Headers Section
- Content can be anything (not just text)
 - typically is HTML document



Response Status Line

HTTP-Version Status-Code Message

- *Status Code* is 3 digit number (for computers)
- Message is text (for humans)

Status Codes

1xx Informational

2xx Success

3xx Redirection

4xx Client Error

5xx Server Error

Example Status Lines

HTTP/1.0 200 OK

HTTP/1.0 301 Moved Permanently

HTTP/1.0 400 Bad Request

HTTP/1.0 500 Internal Server Error

Response Headers

- Provide the client with information about the returned *entity* (document).
 - what kind of document
 - how big the document is
 - how the document is encoded
 - when the document was last modified
- Response headers end with blank line

Response Header Examples

Date: Thu, 27 Jan 2000 12:48:17 EST

Server: Apache/1.17

Content-Type: text/html

Content-Length: 1756

Content-Encoding: gzip

Content

- Content can be anything (sequence of raw bytes).
- Content-Length header is required for any response that includes content.
- Content-Type header also required

Try it with telnet

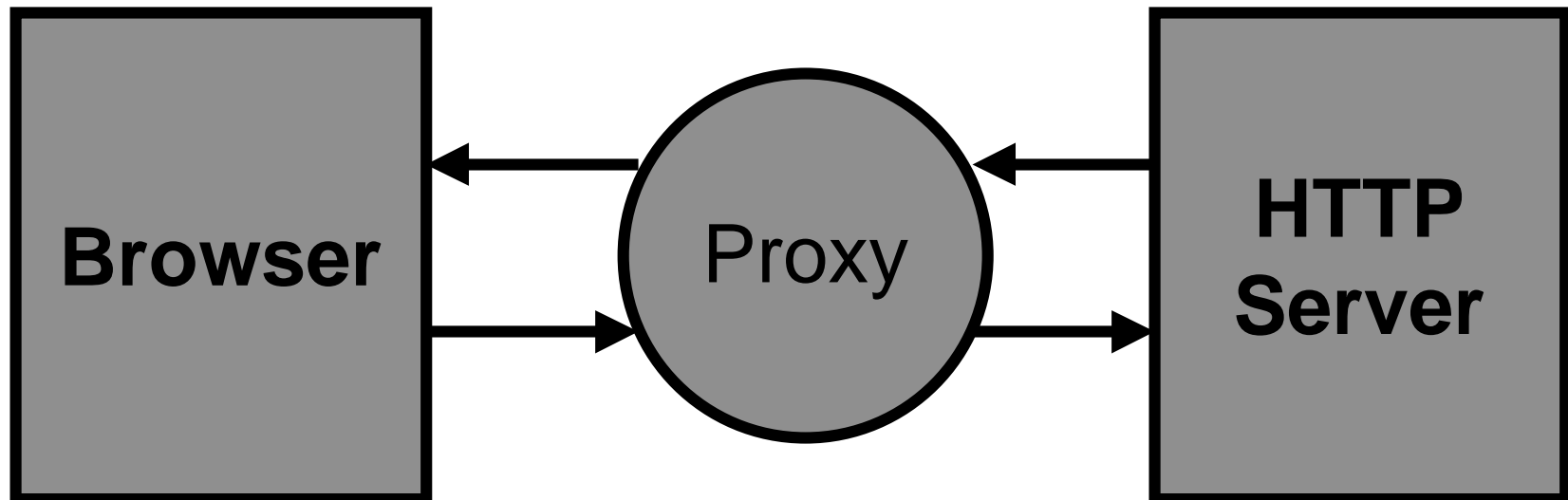
```
> telnet www.cs.rpi.edu  
GET / HTTP/1.0  
HTTP/1.0 200 OK  
Server: Apache  
...
```

Request ←

← *Blank Line
(end of headers)*

← *Response*

HTTP Proxy Server



Project #2 HTTP Proxy

- You need to write a proxy server.
- Test it with a browser.
- Must be able to handle GET, HEAD and POST requests.

What you need to know

- *You* need to understand HTTP
- You *need* to understand HTTP
- You need to *understand* HTTP

The code you need

- Proxy is both a client and a server
- Parsing the HTTP request is much of the code.
 - feel free to find some existing code, but make sure you understand it!

Testing

- Tell your browser to use a proxy
 - Edit preferences/options.
- Interrupt a long transfer (press stop).
- Fill out a form (probably uses POST).

What is expected

- We should be able to surf through your proxy!
- We should not be able to kill your proxy by sending a bad request.
- Proxy should print some info about each request (print the request line).

Stuff you need that we have not yet covered!

- Converting hostnames to IP addresses
- Handling signals (SIGPIPE)
- Providing Concurrency (not required, but not hard either).