

Project 2: httpd

Due: 29 May 2009

In this project, you'll implement a daemon that communicates over TCP/IP and speaks HTTP, the hypertext transport protocol. You're only writing the server end here—the client end of the communication is just any old copy of Firefox or Explorer or some other web browser.

1 Spec

Your server should take two command line arguments: a port number and a directory, which will be the root of the web filesystem.

Implement all the portions of RFC 2616 that are REQUIRED of an HTTP server, plus persistence of connection. RFC 2616 can be found at

`http://www.ietf.org/rfc/rfc2616.txt`

Your server should be able to handle up to five simultaneous connections (using either threads or processes, your pick; TSIC §5.4 gives a lot of help with this).

2 About RFC 2616...

Don't worry, I won't actually make you read all 176 pages of the RFC; a lot of it is about cache and proxy behaviour, and there's a fair amount about clients, too. Furthermore, lots of features are either optional or recommended, but not required; these are indicated by the all-caps keywords MAY and SHOULD, respectively, and except for persistent connections you can ignore those too (although several of them will be potential project extensions).

Below is a breakdown of the parts you need to work on; dotted numbers in parens refer to the section of the RFC where a topic is explained in more detail.

2.1 Incoming requests

Every incoming request (5) will start with a request line, then have header lines, and end in a blank line. The request line (5.1) has three parts: a method, an address, and a protocol version. For instance, one possible request line would be

```
GET /index.html HTTP/1.1
```

The only methods you need to support are GET and HEAD (5.1.1), and you may reject any request that is not according to the current version of the protocol (HTTP/1.1) with a 505 (10.5.6).

The requested address may be given either as an absolute path (which is only ‘absolute’ relative to the root of the webserver, not the whole filesystem) or as a complete URL (5.1.2), but in either case you need to decode any quoted bytes: a quoted byte is a percent sign followed by two hex digits. (One common one is `%20`, which represents a space (0x20, or 32, in ASCII).) Furthermore, you must disallow the use of “..” to go any higher than the root of the webserver, in order to prevent privacy attacks (15.2).

The header lines all start with the label of the header, followed by a colon, a space, and the value of that header. You can ignore almost all request headers; the only ones you must somehow process are Connection (14.10), Expect (14.20), and several headers that start with “If-” (14.24 ff), but your response might well be an “I don’t handle that” error message.

The next request, assuming there is one, will normally come immediately after the blank line from the previous one.

Every line, including the blank line at the end of the headers, is terminated by both a CR (carriage return, ASCII 13) and an LF (linefeed, ASCII 10); this is distinct from usual UNIX practice, where newlines are indicated only by an LF. In C, the LF we are used to is represented with `'\n'` as usual; to represent a CR you can use `'\r'` instead. NB: if you get messed up and end up printing a CR to the terminal without a subsequent LF, this often manifests as the cursor going to the start of the current line and overwriting what’s already there. This can make `fprintf`-based debugging a surreal experience if you don’t know to expect it.

2.2 Outgoing responses

Every outgoing response will start with a status line, then have header lines that end in a blank line; if the response has a message body (e.g. a web page being served) this will follow the blank line that ends the headers. The message body, if present, is *not* followed by an extra blank line.

The status line contains the protocol version, then a space, then a three-digit status code, then a space, then a human-readable ‘reason phrase’ of a few words that correspond to the status code (10). The two primary status codes for you will be 200 OK and 404 Not Found, but you’ll also have to deal with 100 Continue, 400 Bad Request, and 501 Not Implemented; there are one or two others that you might be required to send depending on other programming choices you make (e.g. 414, 505).

The response headers must include Date (14.18 and 3.3.1), must include “Connection: close” if this connection is not persisting (14.10), and must include an appropriate Content-Length in bytes (14.13) if the connection is persisting (but can, should, and might as well include a Content-Length in any case). You should also include a Server header to identify this product (3.8) as *your* server; use the format

```
Server: CS226-yourlogin-1.0
```

replacing “yourlogin” as appropriate and a version number that reflects your own versioning.

As in the request headers, every line of the response headers must be terminated with the two-character newline sequence CR+LF.

After the blank line comes the message body. This is a straightforward bytestream, and does not have to have CR+LF newlines—it just dumps whatever was in a file onto the TCP stream. Remember that even most error codes have associated message bodies; the only times there are no message bodies are in response to HEAD requests and on responses whose status code is 204, 304, or anything starting with 1xx.

3 A couple of exploration techniques

Something to remember is that there are real HTTP clients (“web browsers”) and real HTTP servers (“web sites”) out there, and you have tools to interact

with them directly and see how they react to various stimuli.

For seeing the responses a real web server gives, use `telnet`. If you type

```
telnet www.knox.edu 80
```

it gives you a fairly unfiltered TCP stream to the HTTP port of our web-server; if you speak HTTP to it, e.g.

```
GET /index.html HTTP/1.1
Host: www.knox.edu
```

(don't forget the blank line), it will spew a response at you, headers and all. To terminate a telnet connection, press `^]` and then type `quit`.

For seeing what kinds of requests a web browser makes, you can use a modified version of `chatlogin`: I've written `tcpserv` to permit arbitrary back-and-forth flow (much like telnet), so that you can run it on some arbitrary port and point a browser at it. If you run

```
tcpserv 6543
```

on `dijkstra`, then point any browser at

```
http://dijkstra.lab.knet.edu:6543/foo.html
```

you'll see just what that browser is sending in its request. You can even manually type a response. In this case, a simple `^D` will end the communication.

4 The first pass

I strongly recommend you sprint towards the earliest, simplest version of the program that lets a browser request and receive *something*, however fragile that might be. From there, it becomes a lot easier to debug and use iterative development, where you start with a (more or less) working system, change one thing, and see what happens.

To that end, here's my thoughts on what a super-duper-bare-bones system would look like:

- It assumes all requests are GET,
- and all addresses are paths (not URLs),
- that don't have any hex-quoted bytes in them.
- It further assumes that all headers are completely irrelevant, and just throws them out.
- It only accepts one connection at a time.
- It assumes the specified file exists,
- and blindly sends it over the connection.

Your mileage may vary; feel free to chart your own course.

5 Extensions

Some are harder than others. Some are a *lot* harder than others. Check with me before you do any of these. As with the previous project, the base 100% of the points are in getting the details of the spec right, not in any of the extensions. Unlike the previous project, there are details in the spec that I don't care as much about—they're in there because RFC 2616 requires them for a compliant server. Specifically, the If- headers are more of a pain than they're probably worth, and you might get more bang for your buck with a couple of the extensions.

- Send the Content-Type (14.17 and 3.7) with every response. At a minimum, handle text/html and text/plain; everything else can be application/octet-stream (the generic bytestream media type).
- Send the Last-Modified (13.3.4, 14.29) header with every successful response.
- Handle If-Modified-Since (14.25) and If-Unmodified-Since (14.28) as they “SHOULD” be handled, not just as they “MUST” be handled.
- Handle If-Match (14.24) and If-None-Match (14.26) as they “SHOULD” be handled, not just as they “MUST” be handled.

- When an URL corresponds to a directory, serve the file `index.html` within that directory.
- When an URL corresponds to a directory that has no `index.html` file, serve a constructed-on-the-fly HTML page with links to each file in the directory.
- When the server is killed at the console (using `^C`) it should stop accepting new connections but complete the already-opened transmissions, and then gracefully clean up after itself. (Involves signal handling; see §5.2 in TSIC.)
- Cleanly handle the SIGPIPE that results from a client abandoning the socket. (Involves signal handling; see §5.2 in TSIC.)
- Each time the server is run, create a log that tracks all requests made: the source, the requested resource, and disposition thereof.
- On startup, read a config file that includes URL-file aliases; that is, when the given URL is requested, the given file is served. (The config file must be somehow protected from being loaded via the webserver, to prevent attacks (15.2).)
- On startup, read a config file that includes URL-URL redirects; that is, when the given URL is requested, a 301 or 307 is sent to redirect the request to the other one.
- Implement some other SHOULD or MAY behaviour from the RFC.

6 Final notes

There are a few places in this project (especially in the extensions) where a clean solution would seem to require implementing some complicated data structure or algorithm that doesn't come in the C library. There's no need to reinvent the wheel, though: far better to reuse the work of others and focus on the specifics of this project. If you decide to include library code or functions that you have found elsewhere, just make sure you clearly delineate what's yours and what's not yours (and mention where you got it in a comment or a README or something). If you're not sure, you can always ask.

Hand in all relevant `.c` and `.h` files as `proj2`.