Project 2
Design notes and rubric

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.

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, separated by linear whitespace. 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 (9.3, 9.4) (otherwise you can reject with a 501) (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 (5.1.2). (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 only upper limit to the length of a URI is what resources you're serving, so your request processor has to be able to handle arbitrarily long URIs OR you have to have some justification on why you have a max length (and reject the request with a 414, don't crash) (3.2.1).

The header lines all start with the label of the header, followed by a colon, some linear whitespace (2.2), 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, typically a 501 Not Implemented (10.5.2). You can ignore the

Host header (5.2) but if it is missing entirely you must respond with a 400 (19.6.1.1).

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) (2.2); 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.

Outgoing responses

Every outgoing response will start with a status line (6.1), then have header lines (6.2) 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, but immediately by the next response, which corresponds to the order the requests were received (8.1.2.2).

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: CMSC242-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 (3.7.1)—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.

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.cs.longwood.edu 80
```

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

```
GET /index.html HTTP/1.1
Host: www.cs.longwood.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, try running your prep work and pointing a browser at it! As indicated earlier, if you compile your prep work as `tcpserv`, you could run

```
./tcpserv 6543
```

on (say) kernighan, then point any browser in the lab at

```
http://kernighan.cs.longwood.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.

RUBRIC

General (30)

10 Prep work

10 Design work

10 Documentation

Simplest server (17)

10 Starts server on spec'ed port, accepts connections repeatedly (until server is cancelled with `^C` or terminated by signal), and reads from each until connection is terminated or requested by client to be closed (i.e. the prep work)

5 Sends data to client

2 Response includes header: status line, [fields,] blank line

Response format and headers (14)

2 Status line is valid

2 Response includes Server header field with appropriate value

5 Response includes correct Date header field

5 Response includes correct Content-Length header field

Request processing, filenames and files (23)

5 Sends file spec'ed in request to client

2 Detects and responds to nonexistent/unavailable file

2 File location computed relative to web root directory (command line arg)

2 Understands file location as absolute path

5 Understands file location as URI

2 ...with decoding of quoted bytes in URI

5 Permits `..` in path/URI but prevents it getting higher than root of webserver

Request processing, everything else (16)

2 Detects and responds to bad method in request

2 Detects and responds to bad http version in request

2 Distinguishes GET from HEAD, responds appropriately

2 Endlines and blank lines are produced and processed correctly

2 Handles missing `Host` field

2 Handles `Connection: close`

2 Responds to un-handled header lines appropriately

1 Closes file descriptors when done with them (i.e. don't leak resources)

1 Has at least the prep work done and avoids overflowing any buffers (note that particularly bad/frequent/large overflows may jeopardise other points as well)