Project 2

Due: 4 April 2022

20220315-1130

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 Chrome or some other web browser.

Objectives

In the course of this project, the successful student will:

- write a networked program using the TCP/IP network libraries;
- interact with the network and filesystem while defending against buffer overflow and access to non-permitted files; and
- read RFCs and implement their specifications.

Spec

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

EDIT: added "in that order"

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://datatracker.ietf.org/doc/rfc2616/
```

Prep work

Your initial prep work is to wrangle existing code into the form we'll need for our project; much of this is already written, in rough form, in the shared directory.

Your prep work program should:

- accept at least one command line argument, which will represent the port number it should run on (e.g. 8000 or 8080 or 12345), and fail nicely if it does not get any arguments

- start a TCP server listening on that port
- check *all* return values for error codes and gracefully error out of the program if any of the network stuff fails
- when a connection is made or terminated, print info to that effect (and where the connection is coming from)
- print everything transmitted over that connection, until the connection is closed
- keep awaiting connections after the first one closes (i.e. don't quit after the first client disconnects)
- when it receives a SIGTERM, calls `exit(0)` to exit gracefully (this also flushes any pending print buffers).

If your program is working, and you run it on port 12345, you should be able to test it by (on the same machine) typing

```
telnet localhost 12345
```

or (if it's on, say, knuth), going to any other machine in the lab and running

```
telnet knuth 12345
```

or

```
telnet knuth.cs.longwood.edu 12345
```

or even, foreshadowing the rest of the project,

```
links http://knuth:12345/arbitrary/stuff/here/
```

Hand in the prep work (as `proj2`) when you've got that working, no later than 4pm on Wednesday the 16th. Keep a copy of it around, as this bare-bones server will be useful later on to see what an actual browser sends when it's requesting things.

Design work

RFC 2616 is, as RFCs go, not outrageously long, but there's a lot there. The good news is, a *lot* of it is about cache and proxy behaviour, and clients; and more importantly, a lot of the features it mentions for servers 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.

For the design work, you will run through the document—not a deep dive, not yet anyway—to identify the things you need to worry about. Write down any feature that the RFC says you actually have to implement (plus persistent connections, which I'm requiring even though the RFC makes them optional), along with the section number in the RFC that describes its requirements. Have that list with you in class on Monday, the 21st.

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,
- and that `read` returns the full number of bytes requested.
- 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.

Final version

A full-credit final version will be a complete, non-buggy, working implementation of the RFC TOGETHER WITH convincing proof that it is correct. The program should be able to run with arbitrary input without crashing even if the user or client gives bad input (a graceful exit with an error message is not a crash, nor is terminating the connection of an ill-behaved client). The "proof" will take the form of a clean and complete set of test cases, including both input/running instructions and expected results.

Note that I am not able to spend a ton of time with your program (and in fact may not read it at all, and definitely won't do your debugging for you), so your documentation will need to tell me anything I need to know to run and test your program. There need to be clear instructions on how to run it in general as well as how to run each/all of the tests and quickly verify that they ran correctly (and which rubric items each one corresponds to). Having complete and correct documentation is an easy 10 points, but if your documentation omits important info or tells me the wrong thing, you'll get less than full credit there.

After prep work (10 points), design work (10 points), and documentation (10 points), there remain 70 points in the rubric, which will be awarded according to the table below.

NOTE: if your code doesn't compile, or immediately crashes when it's run, you will get zero of these points. Don't let this happen to you!

The detailed rubric will be published after we have our design discussions on the 21st, but here are the general outlines with just a few details sketched in:

RUBRIC

General (30)
10 Prep work
10 Design work
10 Documentation
Simplest server (17)
10 Starts a server on spec'ed port, accepts connections repeatedly (until server is cancelled with `^C` or terminated by signal), and reads from each connection until connection is terminated or requested by client to be closed (i.e. the prep work)
5 Sends data to client
2 Other
Response format (14)
14 Various things
Request processing, filenames and files (23)
5 Reads full line through `\r\n`, extracts URI/filename, and sends that spec'ed file to client
18 Stuff
Request processing, everything else (16)
14 Other stuff
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)

Handing in

For both the prep work and the final version, hand it in as `proj2` using the handin script. The final version is due at 4pm on Monday, 4 April.