CSCI 330 Assignment 9 (Spring 2021)

TCP Programming with Sockets - Simplified HTTP Server (100 pts)

Purpose

The purpose of this assignment is to practice programming TCP/IP programs using sockets, and bring together the other programming skills learned throughout the course.

The Task

Write a C++ program that implements a basic HTTP server. If this is done properly, you should be able to connect to it through a web browser, but this feature will not be part of the criteria for grading

The program will essentially consist of a loop that goes on forever (until the program is killed), waiting for a client to connect to it. When a client does connect, the server accepts that connection, then calls `fork` to make a child process, where it handles communication with the newly-connected client. The parent process should continue to wait for more connections, accepting them and forking as necessary (avoiding fork-bombs).

The program will accept two mandatory command line parameters:

1. The port number for the server to listen on.
2. The path to a directory that will serve as the root directory of the web server (the *web root*)

For example, if your zid were `z123456`, you wanted to run your server on port `9001`, and the files to be served were in the `~/www` directory, you'd run the program as seen below:

```
./z123456 9001 ~/www
```

For the purposes of this assignment, the requests sent by the client to your program will be all be of the form:

```
GET /path
```

Where `/path` is the path, relative to the directory specified as the second command line parameter (the web root), of the file that the client is requesting. There are several rules on what can form a valid path:

1. It must begin with a "`/`"
2. It may contain additional "`/`" separators to access subdirectories
3. A single "`/`" character refers to the directory specified as argument two on the command line
4. A trailing "`/`" in the pathname can be ignored if the path refers to a directory that otherwise exists. If it refers to a normal file, or to a directory that doesn't exist, the *response* should include an error message.
5. There may be no spaces in the filename. In a real web server there is a way to encode them, but you should just avoid spaces for now. Once a space is found in the *request*, consider the string specifying the requested path to have been completed just before the space.
6. Any data in the request past the path should be ignored. (It is useful in *real* HTTP but you won't be worrying about it for this assignment.)
7. It may not contain the substring "`..`", in order to prevent files above the web root path from being accessed.

what to do with the request?

Once your program receives a request from the client (over the TCP/IP connection), it needs to generate and send a *response*. The response will depend on what the path requested is referencing.

If the path requested by the client refers to a directory, then:

- If there is a file named "`index.html`" in the directory, send the contents of that file to the client, byte for byte.
- If not, generate a list of the files in the requested directory and send it to the client. (Do not include any files that start with a "`.`". You can use `opendir` and `readdir` to accomplish this. If you have knowledge of HTML, feel free to format the output nicely and make the filenames work as links, but this is not required.

If the path refers to a file, then the contents of that file should be sent to the client, byte for byte

After finishing the response, your server should disconnect from the client immediately.

Error Checking

- Both of the command line arguments are mandatory. If they are not supplied, your program should instruct the user on how to run it properly.

- If the path given in the second argument doesn't exist or isn't a directory, print an appropriate error message to the standard error stream and quit with an appropriate return code.
- If any system call fails, the program should use `perror` to report what happened and then exit with an error code.
- If the path in the `GET` request is invalid, or if a file or directory cannot be accessed, then an appropriate error message should be sent to the client to notify them, and then the connection should be terminated.

Other Notes

- This is a simple TCP server. If you need a tool to test it, you can use the `telnet` command to connect to your server at the specified port. Type the request and then hit enter to send it. The client will show the response when it arrives. Here are some examples of what that could look like. These examples are running with the assumption that the server was run with the command line above. This may have a different port or root path in practice.

```
# Start the server first (in another terminal, or with & to launch in bg)
% ./z123456 9001 ~/www

# Client asked for a dir that has no index.html; server lists files in ~/www/
% telnet localhost 9001
Connected to localhost.
Escape character is '^]'.
GET /
fileOne.html fileTwo.html
Connection closed by foreign host.

# Client asked for a specific file, but it doesn't exist; server sends error message
% telnet localhost 9001
Connected to localhost.
Escape character is '^]'.
GET /fileOne
Error: fileOne not found
Connection closed by foreign host.

# Client asked for a specific file that does exist; server sends its contents
% telnet localhost 9001
Connected to localhost.
Escape character is '^]'.
GET /fileOne.html
[... contents of file ~/www/fileOne.html ...]
Connection closed by foreign host.
```

- Remember that testing will be done via `turing` and `hopper`, so make sure your program compiles, links and runs properly there before submitting it. This will necessitate testing it with `telnet` or `nc` like the example above, because of some restrictions on the ports that can be accessed from outside.
- Normal web servers usually run on the standard HTTP port, `80`, but users on `turing` and `hopper` without elevated privileges will not be able to bind to any ports under `9000`, so you will need choose a port number higher than that. The machine can only have one socket listening on a given port, so you may have to choose a different port in that range as you test, when a given port is already in use.
- As you are debugging, remember that some ASCII characters aren't printable, so a simple `cout` of a string you got over the network might not show you what is going on. Having extra, non-printable characters in, say, a filename, would likely cause your program to fail to open files, etc. It is fairly common for characters such as newlines and carriage returns to be present in the requests made over the network (particularly if testing with `telnet`), and you should make sure that your program cleans them off the ends of your strings before trying to use them.
- If you'd like to continue working on this after the requirements listed in this assignment, you can learn more about the rules for how a fully-featured HTTP (web) server should behave at the HTTP RFC:

  `https://tools.ietf.org/html/rfc2616`

  Specifically, you will need to send the appropriate HTTP header when the connection is established if you want a web browser to work with your server. (This is not required, but it can be cool to see a program that you wrote serving files to a real web browser.)

What to turn in?

Submit, via Blackboard, the following:

- The C++ source code file for your program, named as `tcp-z123456.cc`, but with your zid instead of `z123456`.