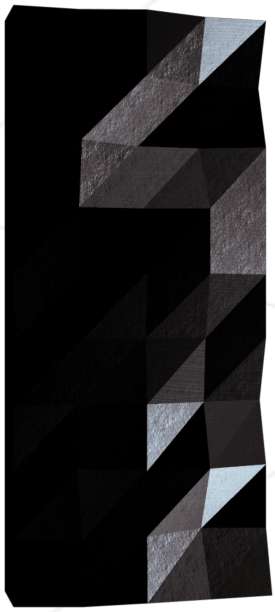

Webserv

This is when you finally understand why a url starts with HTTP

Summary: This project is here to make you write your own HTTP server. You will follow the real HTTP RFC and you will be able to test it with a real browser. HTTP is one of the most used protocol on internet. Knowing its arcane will be useful, even if you won't be working on a website.

Contents

Chapter I

Introduction

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems.
HTTP is the foundation of data communication for the World Wide Web, where hypertext documents include hyperlinks to other resources that the user can easily access, for example by a mouse click or by tapping the screen in a web browser.
HTTP was developed to facilitate hypertext and the World Wide Web.
The primary function of a web server is to store, process and deliver web pages to clients. The communication between client and server takes place using the Hypertext Transfer Protocol (HTTP).

Pages delivered are most frequently HTML documents, which may include images, style sheets and scripts in addition to the text content.

Multiple web servers may be used for a high traffic website.

A user agent, commonly a web browser or web crawler, initiates communication by making a request for a specific resource using HTTP and the server responds with the content of that resource or an error message if unable to do so. The resource is typically a real file on the server's secondary storage, but this is not necessarily the case and depends on how the web server is implemented.

While the primary function is to serve content, a full implementation of HTTP also includes ways of receiving content from clients. This feature is used for submitting web forms, including uploading of files.

Chapter II

Mandatory part

Program name	`webserv`
Turn in files	
Makefile	`Yes`
Arguments	
External functs.	`malloc, free, write, open, read, close, mkdir, rmdir, unlink, fork, wait, waitpid, wait3, wait4, signal, kill, exit, getcwd, chdir, stat, lstat, fstat, lseek, opendir, readdir, closedir, execve, dup, dup2, pipe, strerror, errno, gettimeofday, strptime, strftime, usleep, select, socket, accept, listen, send, recv, bind, connect, inet_addr, setsockopt, getsockname`
Libft authorized	`Yes`
Description	`Write a HTTP server in C++`

- You must write a HTTP server in C++
- It must be conditionnal compliant with the rfc 7230 to 7235 (http 1.1) but you need to implement only the following headers
    - Accept-Charsets
    - Accept-Language
    - Allow
    - Authorization
    - Content-Language
    - Content-Length
    - Content-Location
    - Content-Type
    - Date
    - Host

  - Last-Modified
  - Location
  - Referer
  - Retry-After
  - Server
  - Transfer-Encoding
  - User-Agent
  - WWW-Authenticate

- You can implement all the headers if you want to
- We will consider that nginx is HTTP 1.1 compliant and may be use to compare headers and answer behaviors
- It must be non blocking and use only 1 select for all the IO between the client and the server (listens includes).
- Select should check read and write at the same time.
- Your server should never block and client should be bounce properly if necessary
- You should never read or write without going through select
- A request to your server should never hang forever
- You server should have default error pages if none are provided
- Your program should not leak and should never crash, (even when out of memory if all the initialisation is done)
- You can't use fork for something else than CGI (like php or perl or ruby etc...)
- You can include and use everything in "iostream" "string" "vector" "list" "queue" "stack" "map" "algorithm"
- Your program should have a config file in argument or use a default path
- In this config file we should be able to:

`You should inspire yourself from the "server" part of nginx configuration file`

  - choose the port and host of each "server"
  - setup the server_names or not
  - The first server for a host:port will be the default for this host:port (meaning it will answer to all request that doesn't belong to an other server)

- setup default error pages
- limit client body size
- setup routes with one or multiple of the following rules/configuration (routes wont be using regexp):
  - define a list of accepted HTTP Methods for the route
  - define a directory or a file from where the file should be search (for example if url / is rooted to /tmp/www, url /pouic/toto/pouet is /tmp/www/pouic/toto/pouet)
  - turn on or off directory listing
  - default file to answer if the request is a directory
  - execute CGI based on certain file extension (for example .php)
    - You wonder what a CGI is go? https://en.wikipedia.org/wiki/Common_Gateway_I
    - Because you wont call the cgi directly use the full path as PATH_INFO
    - Just remember that for chunked request, your server need to unchunked it and the CGI will expect EOF as end of the body.
    - Same things for the output of the CGI. if no content_length is returned from the CGI, EOF will mean the end of the returned data.
    - Your program should set the following Meta-Variables
      AUTH_TYPE
      CONTENT_LENGTH
      CONTENT_TYPE
      GATEWAY_INTERFACE
      PATH_INFO
      PATH_TRANSLATED
      QUERY_STRING
      REMOTE_ADDR
      REMOTE_HOST
      REMOTE_IDENT
      REMOTE_USER
      REQUEST_METHOD
      REQUEST_URI
      SCRIPT_NAME
      SERVER_NAME
      SERVER_PORT
      SERVER_PROTOCOL
      SERVER_SOFTWARE
    - Your program should call the cgi with the file requested as first argument
    - the cgi should be run in the correct directory for relativ path file access
    - your server should work with php-cgi
  - make the route able to accept uploaded files and configure where it should be saved

You should provide some configuration files for evaluation

If you've got a question about one behavior, you should compare your program behavior with nginx. For example check how server_name works...

Please read the RFC and do some test with telnet and nginx before starting this project.

We've shared with you a small tester. Do not try evaluation if this tester fails.

The important things is resilience. Your server should never die. It will be stress tested.

Do not test with only one program, write your own test with a language quick to write/use, like python or golang or javascript or rails etc... you can even do your test in c or c++

Chapter III

Bonus part

- If the Mandatory part is not perfect don't even think about bonuses
- You don't need to do all the bonuses
- Make pluggins loadable/unloadable through terminal, like other compression system, charset convertor and so on... (repeatable bonus)
- Your program can have workers define as:
  - a worker can be either processes or threads (and use fork for them)
  - a worker should not be spawn for each client and must able to take care of an infinite number of requests
  - workers are not mandatory
  - you can use fork, wait, waitpid, wait3, wait4, dup, dup2, pipe or pthread_create, pthread_detach, pthread_join, pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_unlock
- Add any number of the following in the configuration file:
  - choose a number of worker (if your program implements workers)
  - Configure plugins (works with pluggins see above)
  - Make routes work with regexp
  - Configure a proxy to an other http/https server
  - Use an internal module for php or any other language. it means you aren't calling any external executable to generate a page with this language. (repeatable bonus)