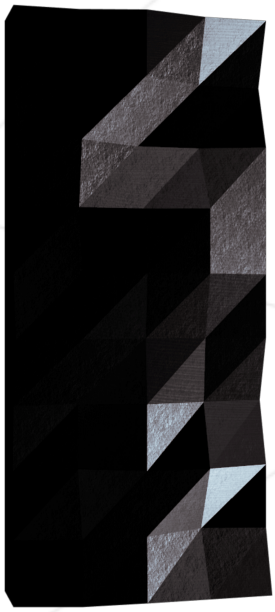

Webserv

Por fin, finalmente entenderás por qué una URL empieza con HTTP

Resumen: Este proyecto existe para hacerte escribir tu propio servidor HTTP. Serás capaz de probarlo con un navegador real. HTTP es uno de los protocolos más utilizados en internet. Saber que es algo arcaico es útil, aunque no trabajarás en una web.

Índice general

Capítulo I

Introducción

El Hypertext Transfer Protocol (HTTP) es un protocolo a nivel de aplicación para sistemas distribuidos, colaborativos, y de información hipermedia.
HTTP es la base de la comunicación de datos para la World Wide Web, donde los documentos hipertexto incluyen hiperenlaces a otros recursos a los que el usuario puede acceder fácilmente, por ejemplo con un clic del ratón o pulsando la pantalla en un navegador web.
HTTP se desarrolló para facilitar el hipertexto y la World Wide Web.
La función principal de un servidor web es alojar, procesar y entregar páginas web a clientes.
La comunicación entre cliente y servidor ocurre utilizando el Hypertext Transfer Protocol (HTTP).

Las páginas entregadas más frecuentemente son documentos HTML, que pueden incluir imágenes, hojas de estilo, y scripts aparte del texto de la página.

Múltiples servidores web se pueden utilizar para sitios web con mucho tráfico.

Un user agent, comunmente un navegador web o web crawler, inicia la comunicación solicitando un recurso específico utilizando HTTP, el servidor entonces responde con el contenido de ese recurso o un mensaje de error en caso de que no se pueda entregar. El recurso es frecuentemente un archivo real en el almacén secundario del servidor, pero esto no es obligatorio y depende de cómo se implemente el servidor web.

Mientras que la función principal es servir contenido, una implementación completa de HTTP también permite recibir contenido de los clientes. Esta característica es utilizada para enviar formularios web, incluyendo la subida de archivos.

Capítulo II

Parte obligatoria

Nombre de programa	`webserv`
Archivos a entregar	
Makefile	`Sí`
Argumentos	
Funciones autorizadas	`Todo en C++ 98. htons, htonl, ntohs, ntohl, select, poll, epoll, kqueue, socket, accept, listen, send, recv, bind, connect, inet_addr, setsockopt, getsockname, fcntl`
Se permite usar libft	`No`
Descripción	`Escribe un servidor HTTP en C++ 98`

- Debes escribir un servidor HTTP en C++ 98.
- Si necesitas funciones de C, puedes utilizarlas pero con preferencia C++.
- El estándar C++ debe ser C++ 98. Tu proyecto debe cumplir con esto.
- No se permiten librerías externas, ni Boost, ni otras.
- Intenta siempre programar en la forma más "C++" posible (por ejemplo, utilizando <cstring>sobre <string.h>).
- Tu servidor debe ser compatible con el servidor web de tu elección.
- Consideraremos que Nginx cumple el estándar HTTP 1.1 y puede utilizarse para comparar headers y otros comportamientos.
- En el subject y en la evalaución mencionamos select pero puedes utilizar equivalentes como epoll, poll, kqueue.
- No deberá bloquearse y utilizar solo 1 select (o equivalente) para todos los IO entre el cliente y el servidor (listens incluidos).

- select (o equivalente) deberá comprobar lectura y escritura a la vez.
- Tu servidor nunca deberá bloquearse y el cliente deberá hacer bounce correctamente en caso de ser necesario.
- Nunca deberás realizar una operación de lectura o de escritura sin pasar por select (o equivalente).
- Comprobar el valor de errno está estrictamente prohibido después de una operación de lectura o escritura.
- Una petición a tu servidor nunca debe quedarse esperando para siempre.
- Tu servidor deberá tener páginas de error por defecto en caso de que no se de ninguna.
- Tu programa no deberá tener leaks de memoria y nunca deberá fallar, (incluso si no queda memoria en caso de que haya iniciado correctamente).
- Evidentemente, no puedes hacer execve a otro servidor.
- Tu programa deberá tener un archivo de configuración como argumento o utilizar la ruta por defecto.
- No necesitas utilizar select (o equivalente) antes de leer tu archivo de configuración.
- Deberás poder servir un sitio web completamente estático.
- El cliente deberá ser capaz de subir archivos.
- Tus códigos de respuesta HTTP deben ser correctos.
- Necesitas implementar como mínimo GET, POST y DELETE.
- Estresa a tu servidor, deberá estar disponible a toda costa.
- Tu servidor debe poder escuchar múltiples puertos (mira el archivo de configuración).

Te hemos permitido utilizar fcntl porque MacOS X no implementa la escritura como otros sistemas Unix lo hacen.
Debes utilizar fd no bloqueantes para tener un resultado similar a otros sistemas operativos.

Dado que estás utilizando fd no bloqueantes, podrías utilizar read/recv o write/send sin select (o equivalentes) y tu servidor no estaría bloqueando. No queremos esto.
De nuevo, intentar usar read/recv o write/send en cualquier fd sin pasar por select (o equivalente) resultará en un 0 y el fin de tu evaluación.

Solo puedes utilizar fcntl de la siguiente manera: fcntl(fd, F_SETFL, O_NONBLOCK);
Cualquier otra flag está prohibida.

- En este archivo de configuración debemos ser capaces de:

`Te puedes inspirar en la parte de "server" del archivo de configuración de Nginx.`

  - Elige el puerto y el host de cada servidor.
  - Permite configurar los `server_names`.
  - El primer servidor para un host:puerto será el servidor por defecto para este host:puerto (lo que quiere decir que contestará todas las peticiones que no pertenezcan a otro servidor).
  - Permite configurar las páginas de error por defecto.
  - Limita el tamaño máximo del body.
  - Configura rutas con una o múltiples de las siguientes reglas/configuración (las rutas no utilizarán regexp):
    - Define una lista de los métodos HTTP aceptados para la ruta
    - Define una redirección HTTP.
    - Define un directorio o archivo desde el que se deberá buscar el archivo (por ejemplo una ruta desde /prueba hasta /tmp/www, de forma que la URL /prueba/esto/es/una/prueba sea /tmp/www/esto/es/una/prueba).
    - Activar o desactivar el listado de directorios
    - Un archivo por defecto para responder si la petición es un directorio.
    - Ejecutar CGI basado en las extensiones de algunos archivos (por ejemplo ṗhp).
      - ¿Te preguntas qué es un CGI?
      - Dado que no vas a llamar al CGI directamente utiliza la ruta absoluta como `PATH_INFO`.
      - Como recordatorio, para una petición de tipo `chunked`, tu servidor deberá juntar las piezas y el CGI esperará un EOF como fin del body.
      - Lo mismo para el resultado del CGIṠi no se devuelve un `content_length` del CGI, un EOF significará el fin de los datos devueltos.
      - Tu programa deberá llamar al CGI con el archivo solicitado como primer argumento.
      - El CGI deberá ejecutarse en el directorio correcto para acceder a archivos de rutas relativas.
      - Tu servidor deberá trabajar con un CGI (php-cgi, python, etc).
    - Permite a una ruta aceptar archivos subidos y configurar dónde deben quedar guardados.

Debes proporcionar algunos archivos de configuración y archivos básicos por defecto para probar y demostrar que cada característica funciona durante la evaluación.

Si tienes preguntas sobre algún comportamiento, deberás comparar el funcionamiento de tu programa con Nginx. Por ejemplo, verifica cómo funciona server_name...Hemos compartido contigo una pequeña prueba que no es obligatorio pasar si todo funciona correctamente con tu navegador y pruebas...Pero puede ayudar a cazar algunos bugs.

Por favor lee el RFC y realiza algunas pruebas con telnet y Nginx antes de empezar este proyecto.
Incluso si no tienes que implementar todo el RFC, leerlo te ayudará a poner en práctica los elementos solicitados.

Lo importante es la resiliencia. Tu servidor NUNCA debe morir.

No pruebes solo con un programa, escribe tus pruebas con lenguajes de rápida escritura/uso, como Python, Golang, JavaScript, etc. Puedes hasta escribir tus pruebas en C o C++.

Capítulo III

Parte extra

- Si la parte obligatoria no está perfecta, no pienses en bonus.
- Implementa cookies y gestión de sesiones (prepara ejemplos rápidos).
- Soporta múltiples CGI.