1. DESARROLLO.

El proceso de desarrollo de este proyecto se divide en dos secciones principales: la aplicación cliente y la aplicación servidor.

Aplicación de servidor

La aplicación servidor sirve como piedra angular de este proyecto, encapsulando la funcionalidad de un Proxy HTTP Invertido y un Balanceador de Carga.

Proxy:

- Intercepta las peticiones HTTP y las dirige al servidor web apropiado.
- Modifica la petición entrante para que se ajuste a la URL relativa y a las especificaciones de la cabecera Host.
- Añade una cabecera Via para indicar la presencia de un intermediario en la cadena de peticiones.
- **Equilibrador de carga:** Implementa un algoritmo Round Robin para distribuir las peticiones entrantes entre varios servidores.
- Mantiene la persistencia de la sesión para garantizar una experiencia de usuario fluida.
- **Almacenamiento en caché:** Almacena las respuestas de los servidores web en disco para una rápida recuperación en posteriores peticiones.
- Implementa TTL (Time-To-Live) para mantener la frescura de las respuestas almacenadas en caché.
- **Concurrencia:** Utiliza subprocesos para gestionar múltiples peticiones simultáneas de clientes, lo que permite al servidor funcionar con eficacia bajo carga.

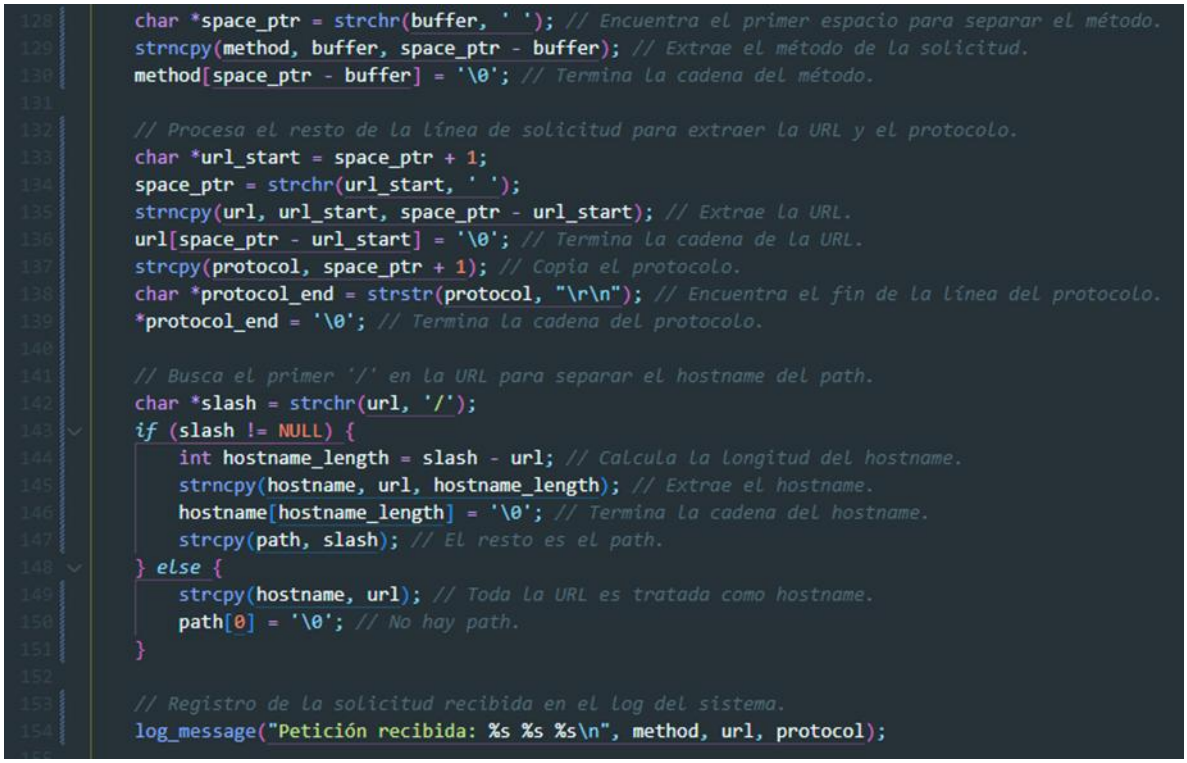
Implementación

- **Lógica del proxy y el equilibrador de carga**

El servidor proxy HTTP y el equilibrador de carga están escritos en C, aprovechando la potencia de los sockets Unix para gestionar las peticiones HTTP. Nuestro enfoque consiste en analizar directamente el protocolo HTTP, manipular las cabeceras e implementar un sistema de caché basado en MD5 para reducir la latencia y la carga del servidor.

- **Análisis de peticiones**

El mecanismo de gestión de peticiones está diseñado para analizar las peticiones HTTP entrantes. Aísla el método, el URL y la versión del protocolo, lo que permite al proxy procesar y redirigir la petición de acuerdo con el algoritmo de equilibrio de carga.



```
char *space_ptr = strchr(buffer, ' '); // Encuentra el primer espacio para separar el método.
strncpy(method, buffer, space_ptr - buffer); // Extrae el método de la solicitud.
method[space_ptr - buffer] = '\0'; // Termina la cadena del método.

// Procesa el resto de la línea de solicitud para extraer la URL y el protocolo.
char *url_start = space_ptr + 1;
space_ptr = strchr(url_start, ' ');
strncpy(url, url_start, space_ptr - url_start); // Extrae la URL.
url[space_ptr - url_start] = '\0'; // Termina la cadena de la URL.
strcpy(protocol, space_ptr + 1); // Copia el protocolo.
char *protocol_end = strstr(protocol, "\r\n"); // Encuentra el fin de la línea del protocolo.
*protocol_end = '\0'; // Termina la cadena del protocolo.

// Busca el primer '/' en la URL para separar el hostname del path.
char *slash = strchr(url, '/');
if (slash != NULL) {
    int hostname_length = slash - url; // Calcula la longitud del hostname.
    strncpy(hostname, url, hostname_length); // Extrae el hostname.
    hostname[hostname_length] = '\0'; // Termina la cadena del hostname.
    strcpy(path, slash); // El resto es el path.
} else {
    strcpy(hostname, url); // Toda la URL es tratada como hostname.
    path[0] = '\0'; // No hay path.
}

// Registro de la solicitud recibida en el log del sistema.
log_message("Petición recibida: %s %s %s\n", method, url, protocol);
```

- **Equilibrio de carga**

Empleamos un método de programación Round Robin para distribuir la carga uniformemente entre los servidores disponibles. Esto se consigue manteniendo un índice actual y rotando a través de una matriz de estructuras de servidores en cada solicitud entrante.

```
// Si no está en caché, determina si debe ser manejada por un servidor Apache configurado utilizando Round Robin.
int is_for_apache = 0;
Server selected_server;
for (int i = 0; i < SERVER_COUNT; i++) {
    if (strcmp(hostname, servers[i].hostname) == 0) {
        is_for_apache = 1;
        selected_server = servers[current_server];
        current_server = (current_server + 1) % SERVER_COUNT;
        break;
    }
}
```

- **Mecanismo de caché**

La caché utiliza hashing MD5 para crear claves únicas para cada solicitud, que luego se utilizan para almacenar y recuperar respuestas del servidor. Cada archivo de caché comienza con una marca de tiempo, seguida del contenido de la respuesta HTTP.

```
156     // Manejo de la caché: genera un nombre de archivo basado en la URL para la caché.
157     char cache_file_name[2 * MD5_DIGEST_LENGTH + 1];
158     generar_nombre_archivo_cache(url, cache_file_name);
```

```
161     if (obtener_respuesta_cache(cache_file_name, respuesta_cache, ttl_global)) {
162         // Si la respuesta está en caché, la envía directamente al cliente.
163         log_message("Respuesta para %s servida desde caché.\n\n", url);
```

Cuando es una petición nueva que no se encuentra en el caché:

```
222         almacenar_respuesta_cache(cache_file_name, buffer);
```

Así mismo cada ciertos minutos se hace un barrido en el directorio donde se encuentra todo el caché para eliminar los archivos de caché que ya tengan vencido el TTL y así no queden gastando espacio en memoria, esto hecho con hilos para no afectar el funcionamiento actual del servidor.

```
262     // Inicializa un hilo de limpieza del caché.
263     pthread_t hilo_de_limpieza;
264     // Inicializa los argumentos para la función de limpieza del caché
265     struct LimpiarArgs limpiarArgs;
266     limpiarArgs.ttl = ttl_global; // Establece el valor del TTL
267     if (pthread_create(&hilo_de_limpieza, NULL, funcion_limpiar, (void *)&limpiarArgs) != 0) {
268         perror("No se pudo crear el hilo de limpieza del caché");
269         exit(EXIT_FAILURE);
270     }
271     pthread_detach(hilo_de_limpieza); // Asegura que los recursos del hilo se liberen automáticamente al terminar.
```

- **Modelo de concurrencia**

La aplicación utiliza hilos POSIX (pthreads) para manejar conexiones concurrentes. Cada solicitud del cliente genera un nuevo hilo, lo que garantiza que el proxy pueda gestionar varias conexiones simultáneas sin bloquearse.

```
        // Crea un nuevo hilo para manejar la solicitud del cliente.
        pthread_t tid;
        if (pthread_create(&tid, NULL, handle_request, (void*)client_socket_ptr) != 0) {
            perror("Failed to create thread for client request");
            close(*client_socket_ptr);
            free(client_socket_ptr);
        } else {
            pthread_detach(tid); // Desacopla el hilo para evitar la necesidad de unirse explícitamente más tarde.
        }
    }
```

Retos y soluciones

- **Gestión de la memoria**

Las versiones iniciales del proyecto se enfrentaban a fugas de memoria debido a la asignación dinámica de memoria sin la adecuada liberación. Utilizando herramientas como Valgrind, identificamos y solucionamos estas fugas, asegurándonos de que todas las llamadas a malloc tuvieran sus correspondientes llamadas a free.

- **Sincronización de hilos**

La gestión de varios subprocesos que accedían a recursos compartidos, como la caché, provocaba condiciones de carrera. Introdujimos bloqueos mutex para garantizar la seguridad de los subprocesos en las operaciones con datos compartidos, evitando la corrupción y asegurando la integridad de los datos.

- **Análisis de casos extremos**

El tratamiento de varios casos extremos en el análisis sintáctico de peticiones HTTP fue todo un reto. Mediante pruebas rigurosas con distintos formatos de petición y la aplicación de reglas de validación estrictas, mejoramos la solidez del analizador sintáctico de peticiones.

Uso

- **Compilación y/o ejecución Servidor HTTP Proxy + balanceador de carga**

Para compilar el Servidor Proxy Invertido HTTP ubíquese en la carpeta server y compile el código fuente utilizando GCC con el makefile proporcionado, ejecute los siguientes comandos:
gcc -c cache/manejoCache.c -o cache/manejoCache.o

gcc -c proxy_server.c -o proxy_server.o -Icache

gcc -o proxy_server proxy_server.o cache/manejoCache.o -lpthread -lcrypto

Para iniciar el Servidor Proxy HTTP, ubíquese en la carpeta server y ejecute el siguiente comando:

./proxy_server <ttl> <port> logProxyServer.log

./proxy_server: es el ejecutable de la aplicación.

Ttl: Tiempo TTL en segundos para los recursos de cache.

port: es el puerto en el cual se escucharán las peticiones por parte de los clientes. Para efectos de este proyecto debe ser el puerto 8080.

logProxyServer.log: representa la ruta y nombre del archivo que almacena el log.

Ejemplo:

./proxy_server

60 8080 logProxyServer.log

- **Compilación y/o ejecución Cliente:**

Para compilar el Cliente ubíquese en la carpeta CLIENT y compile el código fuente utilizando GCC con el makefile proporcionado, ejecute los siguientes comandos:

gcc -c cache/manejoCacheClient.c -o cache/manejoCacheClient.o

gcc -c final_client.c -o final_client.o -Icache

gcc -o final_client final_client.o cache/manejoCacheClient.o -lpthread -lcrypto

Para iniciar el cliente, ubíquese en la carpeta CLIENT y ejecute el siguiente comando:

./final_client logClient.log <url:port>

./final_client: es el nombre del archivo ejecutable.

logClient.log: representa la ruta y nombre del archivo que almacena el log.

<url:port>: URL y puerto donde se localiza el recurso a solicitar.

Ejemplo:

./final_client logClient.log example.com:80/

Nota importante: en la parte de url:port después del port (en el ejemplo :80) siempre se debe ingresar con un / al final si no vas a una página en específico como por ejemplo el home.

2. ASPECTOS LOGRADOS Y NO LOGRADOS.

Logros

El Servidor Proxy HTTP y el Equilibrador de Carga gestionan con éxito las peticiones básicas HTTP GET y HEAD, con logros significativos como:

- Respuestas de baja latencia gracias a un eficiente sistema de caché.
- Distribución uniforme de la carga del servidor gracias al algoritmo Round Robin.
- Manejo robusto de conexiones simultáneas de clientes.

Limitaciones

Aunque el servidor cumple los requisitos básicos, existen áreas de desarrollo futuro:

- Actualmente, el servidor sólo soporta HTTP/1.1. Futuras iteraciones podrían incluir soporte para HTTP/2.
- El sistema de caché no tiene en cuenta las directivas de caché HTTP de clientes o servidores.
- La estrategia actual de equilibrio de carga no tiene en cuenta la salud del servidor ni los tiempos de respuesta.