Cátedra de Redes y Seguridad Informática.
TP1 Servidor Web.

INFORME TRABAJO PRACTICO N°1

REDES Y SEGURIDAD INFORMATICA

PROGAMADO EN C

ACTIVIDADES

DESARROLLAR UN SERVIDOR WEB PARA QUE SE AJUSTE A LA VERSION 1.1 DEL PROTOCOLO HTTP

EN CASO DE RECIBIR UNA ORDEN DIFERENTE A GET O HEAD EL SERVIDOR DEBE CONTESTAR CON ESTADO “HTTP/1.1 501 NOT IMPLEMENTED”

AL SERVIDOR SE LE HARA UNA PETICION HTTP QUE SOLICITARA UN ARCHIVO HTML

ALUMNOS

NICOLAS MAXIMILIANO BARRIENTOS

FAVA MAXIMILIANO ARIEL

TEC. UNIV. EN DESARROLLO DE SOFTWARE

Cátedra de Redes y Seguridad Informática.
TP1 Servidor Web.

Introducción:

El presente informe detalla el proceso de desarrollo de un servidor web simplificado en el marco del trabajo práctico asignado en la materia de Redes y Seguridad Informática de la Tecnicatura en Desarrollo de Software de la Universidad del Chubut, curso 2024. El objetivo de este trabajo práctico es desarrollar un servidor web que cumpla con los estándares del protocolo HTTP/1.1 RFC 2616.

Objetivos del Trabajo Práctico:

- Desarrollar un servidor web que sea capaz de manejar solicitudes HTTP GET y HEAD.
- Implementar una respuesta adecuada en caso de recibir una orden diferente a GET o HEAD.
- Mostrar por pantalla todo lo recibido del cliente para fines de depuración y seguimiento del funcionamiento del servidor.

Decisiones de Diseño:

Entorno de trabajo:

Para realizar este primer trabajo práctico, la primera decisión fue usar la IDE ***Visual Studio Code*** como entorno de desarrollo, apoyándonos en una de las extensiones que nos brinda esta IDE llamada ***GitHub Copilot*** la cual está basada en la tecnología de **OpenAI**, y utilizando control de versiones ***GIT*** para subir el proyecto al sitio web www.Github.com que nos facilitó el trabajo remoto en conjunto.

- https://github.com/Mazo667/servidorTCP

El software que utilizamos para realizar las pruebas correspondientes durante el desarrollo del servidor web fue "***Postman***"

Primeras decisiones de programación:

Empezamos eligiendo el protocolo TCP de la familia TCP/IP para definir la comunicación entre el Servidor web y los clientes. Elegimos TCP porque es un protocolo orientado a la conexión para garantizar la entrega de los archivos solicitados y uno de los protocolos más usados mundialmente.

En el archivo **`server.c`** se define la estructura básica del servidor, su objetivo principal es manejar las solicitudes de los clientes de manera concurrente y proporcionar respuestas adecuadas según lo solicitado.

El servidor está configurado para escuchar en el puerto 8000 y puede manejar múltiples conexiones de clientes simultáneamente. Utiliza un enfoque basado en procesos para manejar la concurrencia: el proceso principal (Proceso Padre) está a la escucha de nuevas conexiones. Una vez un cliente se conecte al servidor; el servidor aceptará la conexión y delegará cada conexión a un proceso hijo para atender al cliente utilizando **`fork()`** y el hijo usara **`execv()`** para realizar la repuesta al cliente dependiendo de su solicitud.

En las llamadas a **`execv()`** que utilizara el proceso hijo se encuentran disponibles dos tipos de archivos .c:

- **`requestget.c`**: Este archivo llamado con **`execv()`** contiene lo necesario para responder a una solicitud de tipo GET, devuelve el archivo solicitado al cliente ya sea una página HTML o una imagen JPG.
  Está diseñado para diferenciar el tipo de extensión que el cliente requiera en la solicitud, ya sea HTML o JPG. Si alguna extensión es diferente a las ya mencionadas serán rechazadas.
- **`requesthead.c`**: Este archivo llamado con **`execv()`** contiene lo necesario para responder a una solicitud de tipo HEAD, devuelve la cabecera del archivo solicitado al cliente ya sea una página HTML o una imagen JPG.
  Está diseñado para diferenciar el tipo de extensión que el cliente requiera en la solicitud, ya sea HTML o JPG. Si alguna extensión es diferente a las ya mencionadas serán rechazadas.

Dificultades y problemáticas:

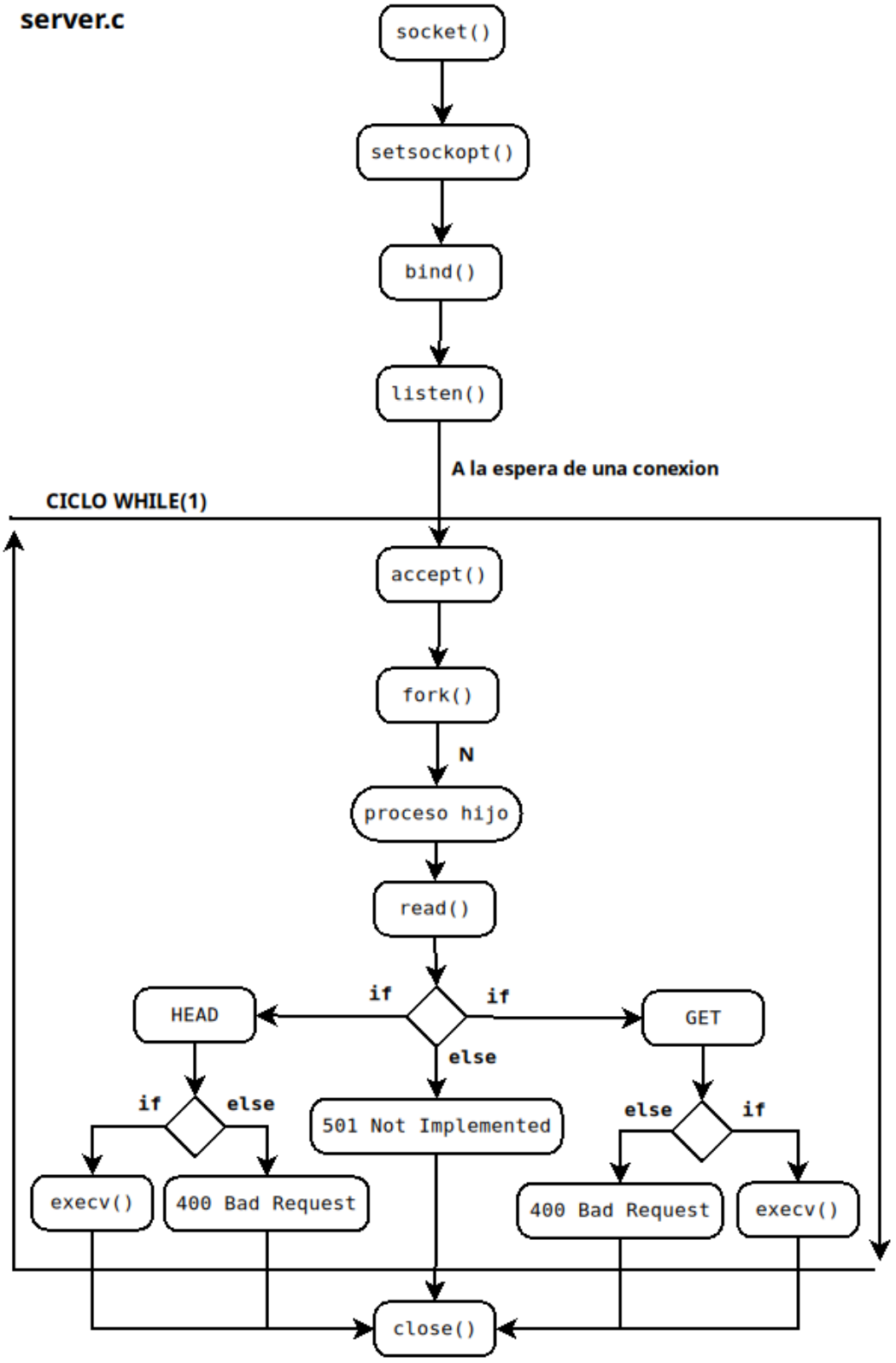
Una de las principales problemáticas que se nos presentó a la hora de realizar el trabajo práctico fue en el momento de implementar los llamados a la familia execv(), se nos dificulto hacer que el servidor devuelva la respuesta solicitada. Tuvimos que investigar la forma de pasar por parámetro más de una variable al **`execv()`** y poder implementar lo investigado.

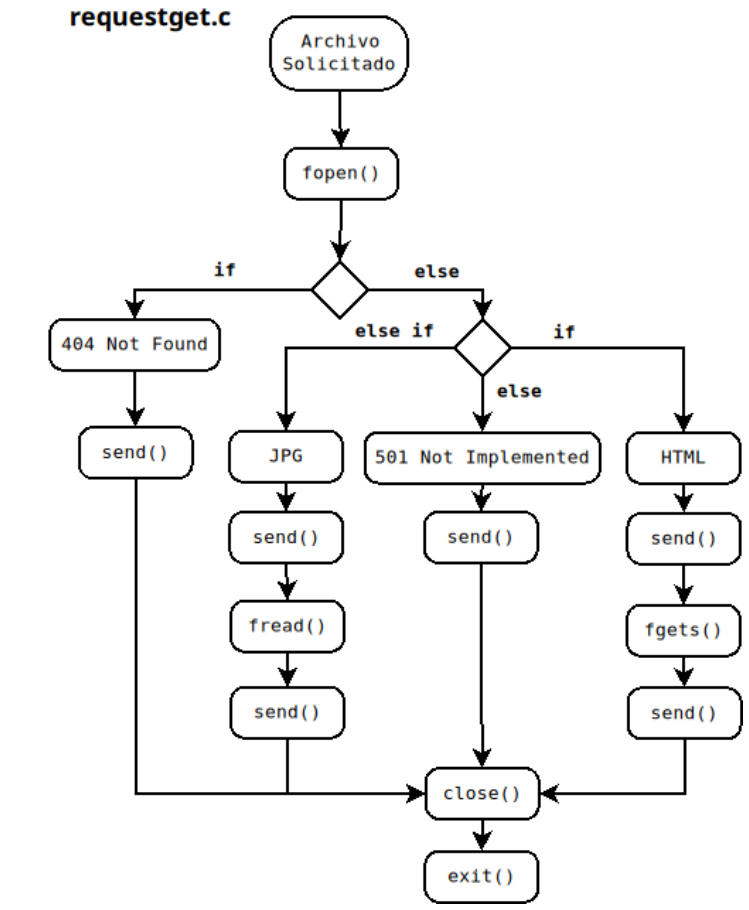
También se nos dificulto la implementación a la hora de que el **`requestget.c`** y **`requesthead.c`** diferenciara el tipo de extensión que llegaba en la solicitud.

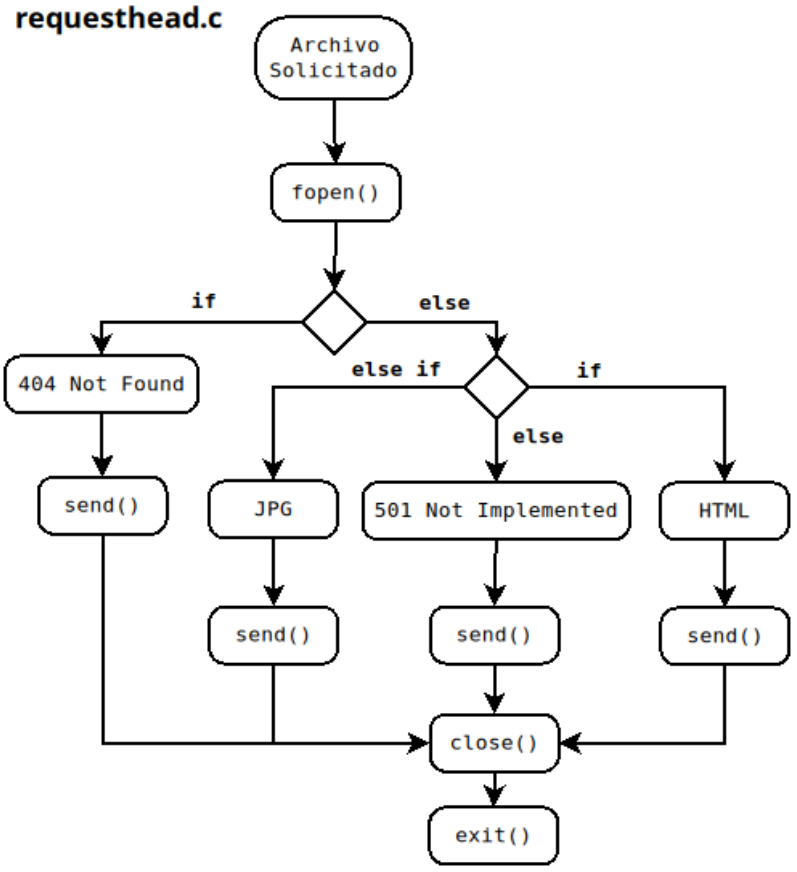
Al momento de que el cliente solicita un recurso web en nuestro servidor mantiene viva esa conexión, para que nuestro servidor responda. Por cuestiones de rendimiento y liberación de recursos, el servidor buscará el archivo solicitado y dependiendo si el archivo existe o no, el servidor enviará al cliente el recurso web solicitado junto al mensaje HTTP de respuesta correspondiente, finalmente cerrando la conexión.

Por último tuvimos dificultades a la hora de crear el archivo `Makefile`, utilizamos la extensión ***GitHub Copilot*** de **Visual Studio Code** para poder apoyarnos en la resolución del problema, una vez dimos con la respuesta a nuestra problemática pudimos realizar la creación de manera correcta y sin problemas de archivo `Makefile`.

Diseño del Proyecto (UML):




requestget.c
Archivo Solicitado
fopen()
if
else
404 Not Found
else if
if
else
send()
JPG
501 Not Implemented
HTML
send()
send()
send()
fread()
fgets()
send()
send()
close()
exit()


requesthead.c
Archivo Solicitado
fopen()
if
else
404 Not Found
else if
if
else
send()
JPG
501 Not Implemented
HTML
send()
send()
send()
close()
exit()

Configuración del Servidor:

```
int server_socket, client_socket;
struct sockaddr_in server_addr, client_addr;
socklen_t client_len;
char buffer[4096];

// Crear el socket del servidor
server_socket = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket < 0){
    perror("Error al crear el socket del servidor");
    exit(EXIT_FAILURE);
}
// Permitir la reutilizion del PORT de escucha
int option;
option = SO_REUSEADDR;
setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &option, sizeof(option));

// Configurar la estructura de dirección del servidor
memset((char *)&server_addr, 0, sizeof(server_addr)); //bzero despreciado

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT);
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0){
    perror("ERROR en bind");
    close(server_socket); // Cierro el servidor
    exit(EXIT_FAILURE);
}

listen(server_socket, 5);

printf("Servidor listo para recibir conexiones en el PORT %d\n", PORT);
```

Ciclo While() para atender a los Clientes

```
while (1){
    client_len = sizeof(client_addr);
    // Acepto la conexión del cliente
    client_socket = accept(server_socket, (struct sockaddr *)&client_addr, &client_len);
    if (client_socket < 0){
        perror("Error al aceptar la conexión");
        exit(EXIT_FAILURE);
    }
    // Eliminar basura del buffer
    memset(buffer, 0, sizeof(buffer));

    printf("Conexión aceptada\n");

    // Creó el proceso hijo para atender al cliente
    if (fork() == 0) {
        int pid = getpid();
        printf("Proceso hijo creado %d id\n", pid);
```

IF() para determinar si es GET y procesar la solicitud

```
if (strcmp(method, "GET") == 0){
  // Pregunto si solicito un archivo
                  if (filename == NULL || (strstr(filename, ".") == NULL)){
                     printf("No se solicitó un archivo\n");
                     char response[4096];
                     sprintf(response,
                             "HTTP/1.1 400 Bad Request\r\n"
                             "Server: ServidorSejin/1.0\r\n"
                             "Content-Length: 0\r\n"
                             "Date: %s\r\n"
                             "Connection: close\r\n\r\n",
                             getActualTime());
                     send(client_socket, response, strlen(response), 0);
                     close(client_socket);
                     exit(0);
                  }else{
                     char client_socket_str[12];
                     // Convierto el descriptor de archivo a una cadena
                     sprintf(client_socket_str, "%d", client_socket);
                     char *args[] = {"./requestget", client_socket_str, filename, NULL};
                     execv(args[0], args);
                     exit(0);
                  }
```

Si se recibiera un método que no sea GET o HEAD

```
        }else {
              printf("Metodo no soportado\n");
                     char response[4096];
                     sprintf(response,
                             "HTTP/1.1 501 Not Implemented\r\n"
                             "Server: ServidorSejin/1.0\r\n"
                             "Date: %s\r\n"
                             "Content-Length: 0\r\n"
                             "Connection: close\r\n\r\n",
                             getActualTime());
                     // Envío la respuesta HTTP
                     send(client_socket, response, strlen(response), 0);
                     close(client_socket);
                     exit(0);
                }
```

Configuración para los execv() requestget.c y requesthead.c:

Pase por parametro de las variables a los execv():

```
int main(int argc, char *argv[]){
    // Verificar que se hayan pasado los argumentos correctos
    if (argc < 3){
        fprintf(stderr, "Uso: %s client_socket filename\n", argv[0]);
        exit(1);
    }

    // Convierte el argumento de client_socket de nuevo a un entero
    int client_socket = atoi(argv[1]);

    // El segundo argumento es el nombre del archivo
    char *filename = argv[2];

    printf("Archivo solicitado: %s\n", filename);
    char baseDir[] = "../";
    strcat(baseDir, filename);
    printf("Ruta del archivo: %s\n", baseDir);

    // abrir el archivo solicitado
    FILE *file = fopen(baseDir, "r");

    char *ext = strtok(filename,".");
    ext = strtok(NULL,".");
    printf("Extension: %s\n", ext);
```

Dentro de los execv() se valida si los archivos solicitados se pueden abrir:

```
/*PREGUNTO SI SE PUEDE ABRIR EL ARCHIVO*/
    if (file == NULL){
        printf("No se pudo abrir el archivo\n");
        char response[4096];
        sprintf(response,
                "HTTP/1.1 404 Not Found\r\n"
                "Server: ServidorSejin/1.0\r\n"
                "Content-Length: 0\r\n"
                "Date: %s\r\n"
                "Connection: close\r\n\r\n",
                getActualTime());
        // Envío la respuesta HTTP
        send(client_socket, response, strlen(response), 0);
        close(client_socket);
        exit(0);
    }
```

Valida el tipo de extensión que sea el archivo solicitado:

Si la extensión es de tipo HTML:

```
if(strcmp(ext,"html") == 0){
    // Preparar la respuesta HTTP
    char response[1024];
    sprintf(response,
            "HTTP/1.1 200 OK\r\n"
            "Server: ServidorSejin/1.0\r\n"
            "Content-Length: %ld\r\n"
            "Connection: close\r\n"
            "Date: %s\r\n"
            "Content-Type: text/html\r\n\r\n",
            getFileSize(baseDir), getActualTime());
    send(client_socket, response, strlen(response), 0);

    // Enviar el contenido del archivo al cliente
    char buffer[1024];
    while (fgets(buffer, sizeof(buffer), file) != NULL){
        send(client_socket, buffer, strlen(buffer), 0);
    }
    fclose(file);
    close(client_socket);
    return 0;
}
```

Si la extensión es de tipo JPG:

```
else if(strcmp(ext,"jpg") == 0){
    // Preparar la respuesta HTTP
    char response[1024];
    sprintf(response,
        "HTTP/1.1 200 OK\r\n"
        "Server: ServidorSejin/1.0\r\n"
        "Content-Length: %ld\r\n"
        "Connection: close\r\n"
        "Date: %s\r\n"
        "Content-Type: image/jpeg\r\n\r\n",
        getFileSize(baseDir), getActualTime());
    send(client_socket, response, strlen(response), 0);

    // Preparar el buffer para leer el archivo
    char buffer[1024];
    size_t count;

    // Leer el archivo y enviarlo al cliente
    while(feof(file) == 0){
        count = fread(buffer, sizeof(char), sizeof(buffer), file);
        send(client_socket, buffer, count, 0);
        memset(buffer, 0, sizeof(buffer));
    }

    // Comprobar si ha ocurrido un error al leer el archivo
    if (ferror(file)) {
        perror("Error al leer el archivo");
    }

    // Cerrar el archivo y el socket
    fclose(file);
    close(client_socket);
    return 0;
}
```

Si no es una extensión válida se rechaza:

```
else{
    char response[1024];
    sprintf(response,
            "HTTP/1.1 501 Not Implemented\r\n"
            "Server: ServidorSejin/1.0\r\n"
            "Date: %s\r\n"
            "Content-Length: 0\r\n"
            "Connection: close\r\n\r\n",
            getActualTime());
    // Envío la respuesta HTTP
    send(client_socket, response, strlen(response), 0);
    close(client_socket);
    exit(0);
}
```

Funciones Usadas:

Biblioteca string.h

- **`char *strtok(char *str, const char *delim)`**: Esta función rompe un string (str) en una serie de tokens usando el delimitador (delim), devuelve un puntero al primer token encontrado en la cadena. Se devuelve un puntero nulo si no quedan tokens para recuperar.
- **`int atoi(const char *str)`**: Esta función convierte un cadena (str) en un tipo entero (int).
- **`char *strcat(char *dest, const char *src)`**: Concatena la cadena (src) al final de la cadena (dest).
- **`int strcmp(const char *str1, const char *str2)`**: Devuelve 0 si las cadenas de texto son iguales (incluyendo mayúsculas y minúsculas). Si la primera cadena es mayor que la segunda, devuelve un número positivo; si es mayor la segunda, devuelve un valor negativo.
- **`void *memset(void *str, int c, size_t n)`**: Copia el carácter c al primero n caracteres a la cadena (str).

Biblioteca stdio.h

- **`int sprintf(char *str, const char *format, ...):`** Crea una cadena (str) según el formato que se especifique (format).
- **`char *fgets(char * str, int size, FILE * file)`**: Lee una cadena de texto (str) desde un archivo (file).
- **`size_t fread(void *data, size_t size, size_t n, FILE *file)`**: Lee datos (data) de un archivo (file).
- **`int feof (FILE* file)`**; Indica si ha llegado al final del archivo.

Biblioteca unistd.h

- **`fork()`:** crea un nuevo proceso hijo idéntico al proceso padre que lo llama.

Biblioteca socket.h

- **`int socket(int domain, int type, int protocol)`:** Crea un socket, en el cual podremos conectarnos a otro socket, devuelve un descriptor de archivo.
- **`int bind(int socket, const struct sockaddr *address, socklen_t address_len)`:** Une una dirección al socket con descriptor de archivo.
- **`int listen(int socket, int backlog)`**: Indica cuántas conexiones entrantes puede encolar.
- **`int accept(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len)`:** Extrae la primera conexión que está en la cola de conexiones pendientes.
- **`ssize_t send(int fdsock, const void *buf, size_t length, int flags)`:** Es usado para enviar un mensaje a otro socket (fdsock).

Bibliografía:

- TCP/IP Illustrated, Volume 1: The Protocols de W. Richard Stevens.
- https://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm
- https://www.aprendeaprogramar.com/
- https://github.com/sejin1031/TCPServer
- https://www.aprendeaprogramar.com/referencia/view.php?f=fread&leng=C
- https://www.ietf.org/rfc/rfc2616.txt
- OpenAI. Asistencia técnica proporcionada por Github Copilot/ChatGPT4 en el desarrollo del servidor web simplificado.