

Trabajo Sockets

Raúl Melgosa Salvador - Juan Carlos Velasco Sánchez

Índice

servidor.c	Página 2
servertcp	Página 5
serverudp	Página 10
cliente.c	Página 13
Resultados de la ejecución	Página 20

Servidor.c

El servidor, en cuanto al funcionamiento inicial, es muy similar al del programa de ejemplo, salvo que en este caso, se crean ambos sockets, el de UDP Y TCP, ya que el archivo cliente.c funciona para TCP y UDP. Se hace bind en ambos, y en TCP hacemos listen.

```
/* Create the listen socket. */
ls_TCP = socket (AF_INET, SOCK_STREAM, 0);
if (ls_TCP == -1)
{
    perror(argv[0]);
    fprintf(stderr, "%s: unable to create socket TCP\n", argv[0]);
    exit(1);
}

/* clear out address structures */
memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
memset ((char *)&clientaddr_in, 0, sizeof(struct sockaddr_in));

addrlen = sizeof(struct sockaddr_in);

/* Set up address structure for the listen socket. */
myaddr_in.sin_family = AF_INET;
/* The server should listen on the wildcard address,
 * rather than its own internet address. This is
 * generally good practice for servers, because on
 * systems which are connected to more than one
 * network at once will be able to have one server
 * listening on all networks at once. Even when the
 * host is connected to only one network, this is good
 * practice, because it makes the server program more
```

```
/* Bind the listen address to the socket. */
if (bind(ls_TCP, (const struct sockaddr *) &myaddr_in, sizeof(struct sockaddr_in)) == -1)
{
    perror(argv[0]);
    fprintf(stderr, "%s: unable to bind address TCP\n", argv[0]);
    exit(1);
}

/* Initiate the listen on the socket so remote users
 * can connect. The listen backlog is set to 5, which
 * is the largest currently supported.
 */
if (listen(ls_TCP, 5) == -1)
{
    perror(argv[0]);
    fprintf(stderr, "%s: unable to listen on socket\n", argv[0]);
    exit(1);
}

/* Create the socket UDP. */
s_UDP = socket (AF_INET, SOCK_DGRAM, 0);
if (s_UDP == -1)
{
    perror(argv[0]);
    printf("%s: unable to create socket UDP\n", argv[0]);
    exit(1);
}
```

Posteriormente, se hace al servidor un proceso daemon, se registra SIGTERM para una finalización ordenada y hacemos un bucle infinito.

```
setpggrp();
switch (fork())
{
    case -1: /* Unable to fork, for some reason. */
        perror(argv[0]);
        fprintf(stderr, "%s: unable to fork daemon\n", argv[0]);
        exit(1);

    case 0: /* The child process (daemon) comes here. */
        /* Close stdin and stderr so that they will not
         * be kept open. Stdout is assumed to have been
         * redirected to some logging file, or /dev/null.
         * From now on, the daemon will not report any
         * error messages. This daemon will loop forever,
         * waiting for connections and forking a child
         * server to handle each one.
         */
        fclose(stdin);
        fclose(stderr);

        /* Set SIGCLD to SIG_IGN, in order to prevent
         * the accumulation of zombies as each child
         * terminates. This means the daemon does not
         * have to make wait calls to clean them up.
         */
        if ( sigaction(SIGCHLD, &sa, NULL) == -1)
        {
            perror(" sigaction(SIGCHLD)");
            fprintf(stderr, "%s: unable to register the SIGCHLD signal\n", argv[0]);
            exit(1);
        }

        /* Registrar SIGTERM para la finalizacion ordenada del programa servidor */
        vec.sa_handler = (void *) finalizar;
        vec.sa_flags = 0;
```

```
/* Registrar SIGTERM para la finalizacion ordenada del programa servidor */
vec.sa_handler = (void *) finalizar;
vec.sa_flags = 0;
if ( sigaction(SIGTERM, &vec, (struct sigaction *) 0) == -1)
{
    perror(" sigaction(SIGTERM)");
    fprintf(stderr, "%s: unable to register the SIGTERM signal\n", argv[0]);
    exit(1);
}
```

Dependiendo si el socket es de TCP o UDP (comprobándolo con FD_ISSET), se hace accept, que devolverá un nuevo socket o recvfrom, respectivamente. En cada uno se ejecutará la función serverTCP o serverUDP, dependiendo del protocolo.

```

/* Comprobamos si el socket seleccionado es el socket TCP */
if (FD_ISSET(ls_TCP, &readmask))
{
    /* Note that addrln is passed as a pointer
     * so that the accept call can return the
     * size of the returned address.
     */
    /* This call will block until a new
     * connection arrives. Then, it will
     * return the address of the connecting
     * peer, and a new socket descriptor, s,
     * for that connection.
     */
    s_TCP = accept(ls_TCP, (struct sockaddr *) &clientaddr_in, &addrln);
    if (s_TCP == -1)
    {
        exit(1);
    }

    switch (fork())
    {
        case -1: /* Can't fork, just exit. */
            exit(1);

        case 0: /* Child process comes here. */
            close(ls_TCP); /* Close the listen socket inherited from the daemon. */
            serverTCP(s_TCP, clientaddr_in, argv[0]);
            exit(0);

        default: /* Daemon process comes here. */
            /* The daemon needs to remember
             * to close the new accept socket
             * after forking the child. This
             * prevents the daemon from running
             * out of file descriptor space. It
             * also means that when the server
             * closes the socket, that it will
             * allow the socket to be destroyed
             * since it will be the last close

```

```

if (FD_ISSET(s_UDP, &readmask))
{
    /* This call will block until a new
     * request arrives. Then, it will
     * return the address of the client,
     * and a buffer containing its request.
     * BUFFERSIZE - 1 bytes are read so that
     * room is left at the end of the buffer
     * for a null character.
     */
    cc = recvfrom(s_UDP, buffer, BUFFERSIZE - 1, 0, (struct sockaddr *)&clientaddr_in, &addrln);
    if (cc == -1)
    {
        perror(argv[0]);
        printf("%s: recvfrom error\n", argv[0]);
        exit(1);
    }

    /* Make sure the message received is
     * null terminated.
     */
    buffer[cc]='\0';
    serverUDP(s_UDP, buffer, clientaddr_in, argv[0]);
}
}

```

Una vez el bucle infinito termine, se cierran ambos sockets.

Se han creado dos estructuras, estructura_mensajes_log, estructura_mensajes_servidorTCP y estructura_mensajes_servidorUDP que se usarán posteriormente. Como punto más destacable, el campo mensaje_respuesta_servidor, que contiene el mensaje de vuelta al cliente, y que es común tanto en estructura_mensajes_servidorTCP y estructura_mensajes_servidorUDP es muy distinto en ambos. En TCP, este campo es de 32500 bytes, mientras que en UDP es de tan sólo 512.

```
struct estructura_mensajes_log
{
    char comunicacion_realizada[500];
    char comunicacion_finalizada[500];
    char nombre_ejecutable[50];
    char hostname[100];
    char hostname_IP[100];
    char puerto_efimero_cliente[50];
    char mensaje[500];
};

struct estructura_mensaje_servidorTCP
{
    char mensaje_respuesta_servidor[32500]; //este buffer va a contener el mensaje de vuelta al cliente. Siempre empieza por HTTP/1.1
    char orden_cliente[100]; //este buffer va a contener la orden que ha realizado el cliente
    char orden_implementada[10]; //este buffer contiene la orden que implementa el servidor. En nuestro caso solo implementamos la orden GET
    char paginaHTML[100]; //este buffer va a contener el nombre de la pagina html a obtener
    char directoriopaginaHTML[100]; //este buffer va a contener la ruta completa de la pagina html para poder abrirla con fopen
    char cabecera[3]; //este buffer contiene la cabecera \r\n que llevan todas las lineas //puede que sea eliminado
    char servidor[100]; //este buffer seguramente sea eliminado
    char connection[100]; //este buffer contiene el tipo de conexion (close o keep-alive)
    char contenido[100]; //este buffer seguramente sea eliminado
    char longitud_contenido[100]; //este buffer contiene la longitud de la pagina html enviada
    char contenido_paginaHTML[32000]; //este buffer va a contener el contenido de la pagina HTML
};

struct estructura_mensaje_servidorUDP {
    char mensaje_respuesta_servidor[512]; //este buffer va a contener el mensaje de vuelta al cliente. Siempre empieza por HTTP/1.1
    char orden_cliente[100]; //este buffer va a contener la orden que ha realizado el cliente
    char orden_implementada[10]; //este buffer contiene la orden que implementa el servidor. En nuestro caso solo implementamos la orden GET
    char paginaHTML[100]; //este buffer va a contener el nombre de la pagina html a obtener
    char directoriopaginaHTML[100]; //este buffer va a contener la ruta completa de la pagina html para poder abrirla con fopen
    char cabecera[3]; //este buffer contiene la cabecera \r\n que llevan todas las lineas //puede que sea eliminado
    char servidor[100]; //este buffer seguramente sea eliminado
    char connection[100]; //este buffer contiene el tipo de conexion (close o keep-alive)
    char contenido[100]; //este buffer seguramente sea eliminado
    char longitud_contenido[100]; //este buffer contiene la longitud de la pagina html enviada
    char contenido_paginaHTML[400]; //este buffer va a contener el contenido de la pagina HTML
};
```

serverTCP

En primer lugar, se crean dos variables de tipo estructura_mensajes_servidorTCP y estructura_mensajes_log.

```

void serverTCP(int s, struct sockaddr_in clientaddr_in, char *argv)
{
    int reqcnt = 0;    /* keeps count of number of requests */
    char buf[TAM_BUFFER]; /* This example uses TAM_BUFFER byte messages. */
    char hostname_decimal_punto[MAXHOST];
    char hostname[MAXHOST]; /* remote host's name string */

    int len, len1, status;
    struct hostent *hp; /* pointer to host info for remote host */
    long timevar; /* contains time returned by time() */

    struct linger linger; /* allow a lingering, graceful close; */
                          /* used when setting SO_LINGER */

    struct estructura_mensajes_log mensaje_log;
    struct estructura_mensaje_servidorTCP mensaje_servidor;

```

Se intenta obtener el nombre del cliente dada su IP mediante la función `getnameinfo`. Después, se abre el fichero `peticiones.log` con `fopen` y se escribe la hora y el nombre del ejecutable.

```

status = getnameinfo((struct sockaddr *)&clientaddr_in, sizeof(clientaddr_in), hostname, MAXHOST, NULL, 0, 0);
if (status)
{
    /* The information is unavailable for the remote
    * host. Just format its internet address to be
    * printed out in the logging information. The
    * address will be shown in "internet dot format".
    */
    /* inet_ntop para interoperatividad con IPv6 */
    if (inet_ntop(AF_INET, &(clientaddr_in.sin_addr), hostname, MAXHOST) == NULL)
    {
        perror(" inet_ntop \n");
    }
}

/* Log a startup message. */
time (&timevar);
/* The port number must be converted first to host byte
* order before printing. On most hosts, this is not
* necessary, but the ntohs() call is included here so
* that this program could easily be ported to a host
* that does require it.
*/

printf("Startup from %s port %u at %s", hostname, ntohs(clientaddr_in.sin_port), (char *) ctime(&timevar));

/* Set the socket for a lingering, graceful close.
* This will cause a final close of this socket to wait until all of the
* data sent on it has been received by the remote host.
*/
linger.l_onoff = 1;
linger.l_linger = 1;
if (setsockopt(s, SOL_SOCKET, SO_LINGER, &linger, sizeof(linger)) == -1)
{
    errout(hostname);
}

```

Se rellenan los campos de la estructura del mensaje del log con el tipo de protocolo, puerto efímero y el nombre del host en formato decimal punto, obtenido con `inet_ntop`. Una vez está relleno lo escribimos en el log. Hay un bucle `while` que va a hacer `recv`.

```

while (len = recv(s, buf, TAM_BUFFER, 0))
{
    if (len == -1)
    {
        errout(hostname);
    }
    /* error from recv */
    /* The reason this while loop exists is that there
    * is a remote possibility of the above recv returning
    * less than TAM_BUFFER bytes. This is because a recv returns
    * as soon as there is some data, and will not wait for
    * all of the requested data to arrive. Since TAM_BUFFER bytes
    * is relatively small compared to the allowed TCP
    * packet sizes, a partial receive is unlikely. If
    * this example had used 2048 bytes requests instead,
    * a partial receive would be far more likely.
    * This loop will keep receiving until all TAM_BUFFER bytes
    * have been received, thus guaranteeing that the
    * next recv at the top of the loop will start at
    * the beginning of the next request.
    */

    bzero((char *)&mensaje_log, sizeof(mensaje_log));
    bzero((char *)&mensaje_servidor, sizeof(mensaje_servidor));
}

```

Por cada iteración de este bucle, se rellenan los campos de la estructura estructura_mensajes_servidorTCP, recogiendo la página html. Una vez recogida, se comprueba que ésta se ubica en el directorio donde se encuentran los archivos.html.

Con la función stat, se obtiene el tamaño de la página, que la dividiremos en trozos si supera los 32000 bytes. Este número de trozos se guarda en la variable num_trozos.

Posteriormente, se hace un bucle que itera tantas veces como el número de trozos que haya, donde comprobaremos si existe la orden “GET”.

```

int num_trozos = 1;
char * todo;
struct stat st;
stat(mensaje_servidor.directoriopaginaHTML, &st);
num_trozos = num_trozos + st.st_size / 32000;
for (int t = 0; t < num_trozos; t++)
{
    for (i = 0; i < sizeof(buf); i++)
    {
        if (buf[i] == '/')
        {
            indice = i;
            strncpy(mensaje_servidor.orden_cliente, buf, indice-1);
            break;
        }
    }

    if (strcmp(mensaje_servidor.orden_cliente, "GET") == 0)
    {
        if (fp == NULL)
        {
            sprintf(mensaje_log.mensaje, "\n404 Not Found. La pagina %s no se ha encontrado", mensaje_servidor.paginaHTML);
            fwrite(mensaje_log.mensaje, strlen(mensaje_log.mensaje), 1, flog);

            //char orden_no_encontrada[100] = "404 Not Found";
            //char pagina_no_encontrada[100] = "<html><body><h1>404 Not Found</h1></body></html>";
            strcat(mensaje_servidor.contenido_paginaHTML, "<html><body><h1>404 Not Found</h1></body></html>\n");
            sprintf(mensaje_servidor.longitud_contenido, "%ld", strlen(mensaje_servidor.contenido_paginaHTML));

            strcat(mensaje_servidor.mensaje_respuesta_servidor, "HTTP/1.1 ");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "404 Not Found");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "Server: ");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, hostname);
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
        }
    }
}

```


Después, se realiza una comprobación de errores, mediante la cual si el archivo es null, se escribe el mensaje de error 404 en el log y en el servidor escribiremos `<html><body><h1>404 Not Found</h1></body></html>` en el contenido de la página, mientras que si lo encuentra, devuelve 200. Si estamos en la primera iteración del bucle mencionado al principio del párrafo, se reserva con malloc una cantidad de memoria equivalente al tamaño de la página, obtenido anteriormente con la función stat, para así, poder ir recogiendo trozo a trozo con strncpy.

```
else
{
    if (t == 0)
    {
        sprintf(mensaje_log.mensaje, "\n200 OK. La pagina %s se ha encontrado correctamente", mensaje_servidor.paginaHTML);
        fwrite(mensaje_log.mensaje, strlen(mensaje_log.mensaje), 1, flog);
    }

    //char orden_encontrada[100] = "200 OK";
    //char pagina_encontrada[1000];
    //bzero(mensaje_servidor.contenido_paginaHTML, sizeof(mensaje_servidor.contenido_paginaHTML));

    if (t == 0)
    {
        todo = malloc(st.st_size);
        bzero(todo, sizeof(todo));
        fread(todo, st.st_size, 1, fp); //revisar si el 3º PARAMETRO NO DEBE SER UNA UNIDAD MENOR
        //printf("ahi vaaaaa %s", todo);
        //fflush(stdout);
    }

    if (num_trozos != t+1)
    {
        strncpy(mensaje_servidor.contenido_paginaHTML, &todo[32000 * t], 32000);
    }
    else
    {
        strncpy(mensaje_servidor.contenido_paginaHTML, &todo[32000 * t], st.st_size - (32000 * t));
    }

    fread(mensaje_servidor.contenido_paginaHTML, sizeof(mensaje_servidor.contenido_paginaHTML), 1, fp); //revisar si el 3º PARAMETRO NO DEBE SER UNA UNIDAD MENOR
    sprintf(mensaje_servidor.longitud_contenido, "%ld", strlen(mensaje_servidor.contenido_paginaHTML));
}
```

Por otro lado, si no existe la orden “GET”, se escribe el mensaje de error 501 en el log y `<html><body><h1>501 Not Implemented</h1></body></html>` en el campo página html de la estructura de TCP. Al final de todas las condiciones para ver si existe la orden “GET” o no, se hace un sleep de una centésima para simular el procesamiento real de un servidor, y posteriormente hacemos el send, pasándole la estructura que hemos rellenado.

```
else
{
    num_trozos = 1;

    sprintf(mensaje_log.mensaje, "\n501 Not Implemented. La orden %s no esta implementada", mensaje_servidor.orden_cliente);
    fwrite(mensaje_log.mensaje, strlen(mensaje_log.mensaje), 1, flog);

    //char orden_no_implementada[100] = "501 Not Implemented";
    //char pagina_no_implementada[100] = "<html><body><h1>501 Not Implemented</h1></body></html>";
    strcat(mensaje_servidor.contenido_paginaHTML, "<html><body><h1>501 Not Implemented</h1></body></html>\n");
    sprintf(mensaje_servidor.longitud_contenido, "%ld", strlen(mensaje_servidor.contenido_paginaHTML));

    strcat(mensaje_servidor.mensaje_respuesta_servidor, "HTTP/1.1 ");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "501 Not Implemented");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "Server: ");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, hostname);
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
}
```

Con bzero, vaciamos los campos de las estructuras rellenadas anteriormente para dejarlas listas para la siguiente iteración. Por último, comprobamos si en la orden hay una k o una c o no hay nada, para ver si la conexión es de tipo keep-alive, en caso de que haya una k o close, en caso de que haya una c o nada. Si no es de tipo keep-alive, hacemos un shut-down para indicar la finalización de la transmisión de mensajes y un break para salir del bucle.

```

sleep(1); //simulamos el procesamiento real de un servidor

if (send(s, mensaje_servidor.mensaje_respuesta_servidor, sizeof(mensaje_servidor.mensaje_respuesta_servidor), 0) != sizeof(mensaje_servidor.mensaje_respuesta_servidor))
{
    errout(hostname);
}

bzero(mensaje_servidor.mensaje_respuesta_servidor, sizeof(mensaje_servidor.mensaje_respuesta_servidor));
bzero(mensaje_servidor.contenido_paginaHTML, sizeof(mensaje_servidor.contenido_paginaHTML));
} //FIN for (int t=0; t < num_trozos; t++)

for (i = 0; i < strlen(mensaje_servidor.connection); i++)
{
    if (mensaje_servidor.connection[i] == ':')
    {
        i = i + 2;
        break;
    }
}

if (strcmp(mensaje_servidor.connection + i, "keep-alive") != 0)
{
    fprintf(fpe, "\nConexión es %s", mensaje_servidor.connection + i);
    if (shutdown(s, SHUT_RDWR) == -1)
    {
        //perror(argv[0]);
        //fprintf(stderr, "%s: unable to shutdown socket\n", argv[0]);
        exit(1);
    }
    break;
}
} //FIN while (len = recv(s, buf, TAM_BUFFER, 0))

```

Una vez fuera del bucle del receive, volvemos a anotar el tiempo y escribimos en el log que la comunicación ha finalizado con el tiempo anotado. Por último, cerramos el socket y escribimos un mensaje por pantalla.

```

fwrite("\n", 1, 1, flog);

fwrite((char *) ctime(&timevar), strlen(ctime(&timevar)), 1, flog);
fwrite(argv, strlen(argv), 1, flog);
fwrite("\n", 1, 1, flog);

strcat(mensaje_log.comunicacion_realizada, "Comunicación finalizada: ");
strcat(mensaje_log.comunicacion_realizada, hostname);

if (inet_ntop(AF_INET, &(clientaddr_in.sin_addr), hostname, MAXHOST) == NULL)
{
    perror(" inet_ntop \n");
}

strcat(mensaje_log.comunicacion_realizada, " ");
strcat(mensaje_log.comunicacion_realizada, hostname);
strcat(mensaje_log.comunicacion_realizada, " ");
strcat(mensaje_log.comunicacion_realizada, "TCP"); //OJO ESTO CAMBIARLO CUANDO SE IMPLEMENTE BIEN
strcat(mensaje_log.comunicacion_realizada, " ");
sprintf(mensaje_log.puerto_efimero_cliente, "%u", ntohs(clientaddr_in.sin_port));
strcat(mensaje_log.comunicacion_realizada, mensaje_log.puerto_efimero_cliente);
strcat(mensaje_log.comunicacion_realizada, "\n");
fwrite(mensaje_log.comunicacion_realizada, strlen(mensaje_log.comunicacion_realizada), 1, flog);

/* The loop has terminated, because there are no
 * more requests to be serviced. As mentioned above,
 * this close will block until all of the sent replies
 * have been received by the remote host. The reason
 * for lingering on the close is so that the server will
 * have a better idea of when the remote has picked up
 * all of the data. This will allow the start and finish
 * times printed in the log file to reflect more accurately
 * the length of time this connection was used.
 */
close(s);

/* Log a finishing message. */
time(&timevar);
/* The port number must be converted first to host byte
 * order before printing. On most hosts, this is not
 * necessary, but the ntohs() call is included here so
 * that this program could easily be ported to a host
 * that does require it.
 */
printf("Completed %s port %u, %d requests, at %s\n", hostname, ntohs(clientaddr_in.sin_port), reqcnt, (char *) ctime(&timevar));

```

serverUCP

Al igual, que en serverTCP, creamos otra vez la estructura de mensajes del log y en este caso creamos la estructura del tipo estructura_mensajes_servidorUDP.

Nuevamente, se intenta obtener el nombre del cliente dada su IP mediante la función getnameinfo. Después, se abre el fichero peticiones.log con fopen y se escribe la hora y el nombre del ejecutable. Se rellenan los campos de la estructura del mensaje del log con el tipo de protocolo, puerto efímero y el nombre del host en formato decimal punto, obtenido con inet_ntop. Una vez está relleno lo escribimos en el log.

```
void serverUDP(int s, char * buf, struct sockaddr_in clientaddr_in, char * argv)
{
    struct in_addr reqaddr; /* for requested host's address */
    struct hostent *hp;      /* pointer to host info for requested host */
    int nc, errcode, status;

    struct addrinfo hints, *res;

    char hostname[MAXHOST];
    char hostname_decimal_punto[MAXHOST];

    struct estructura_mensajes_log mensaje_log;
    struct estructura_mensaje_servidorUDP mensaje_servidor;

    int addrlen;
    long timevar;

    addrlen = sizeof(struct sockaddr_in);

    memset (&hints, 0, sizeof (hints));

    status = getnameinfo((struct sockaddr *)&clientaddr_in, sizeof(clientaddr_in), hostname, MAXHOST, NULL, 0, 0);
    if (status)
    {
        /* The information is unavailable for the remote
        * host. Just format its internet address to be
        * printed out in the logging information. The
        * address will be shown in "internet dot format".
        */
        /* inet_ntop para interoperatividad con IPv6 */
        if (inet_ntop(AF_INET, &(clientaddr_in.sin_addr), hostname, MAXHOST) == NULL)
        {
            perror(" inet_ntop \n");
        }
    }
}
```

Se vuelve a extraer el directorio, y se abre el archivo con el nombre del archivo.html. Comprobamos el tamaño del archivo, y en base a ello se trocea en un número de trozos que se almacena en la variable num_trozos. En este caso, y a diferencia del protocolo TCP, se trocea en partes de 400 bytes.

```

int num_trozos = 1;
char * todo;
struct stat st;
stat(mensaje_servidor.directoriopaginaHTML, &st);
num_trozos = num_trozos + st.st_size / 400;
for (int t = 0; t < num_trozos; t++)
{
    if (strcmp(mensaje_servidor.orden_cliente, "GET") == 0)
    {
        if (fp == NULL)
        {
            sprintf(mensaje_log.mensaje, "\n404 Not Found. La pagina %s no se ha encontrado", mensaje_servidor.paginaHTML);
            fwrite(mensaje_log.mensaje, strlen(mensaje_log.mensaje), 1, flog);

            //char orden_no_encontrada[100] = "404 Not Found";
            //char pagina_no_encontrada[100] = "<html><body><h1>404 Not Found</h1></body></html>";
            strcpy(mensaje_servidor.contenido_paginaHTML, "<html><body><h1>404 Not Found</h1></body></html>\n");
            sprintf(mensaje_servidor.longitud_contenido, "%ld", strlen(mensaje_servidor.contenido_paginaHTML));

            strcat(mensaje_servidor.mensaje_respuesta_servidor, "HTTP/1.1 ");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "404 Not Found");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "Server: ");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, hostname);
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");

            strcat(mensaje_servidor.mensaje_respuesta_servidor, "Connexion: close");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");

            strcat(mensaje_servidor.mensaje_respuesta_servidor, "Content_Length: ");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, mensaje_servidor.longitud_contenido);
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
            strcat(mensaje_servidor.mensaje_respuesta_servidor, mensaje_servidor.contenido_paginaHTML);
        }
    }
}

```

Ahora se hace un bucle que itera tantas veces como el valor de la variable num_trozos. Se comprueba si existe la orden "GET". En el caso de que exista, si el archivo es null, se escribe el error 404 en el log y `<html><body><h1>404 Not Found</h1></body></html>` en el campo de contenido de la página html en la estructura de mensajes de UDP. Por el contrario, si el archivo no es null se le dará el valor 200 en el log, y si estamos en la primera iteración del bucle mencionado al principio del párrafo, se reserva con malloc una cantidad de memoria equivalente al tamaño de la página, obtenido anteriormente con la función stat, para así, poder ir recogiendo trozo a trozo con strncpy.

```

else
{
    if (t == 0)
    {
        sprintf(mensaje_log.mensaje, "\n200 OK. La pagina %s se ha encontrado correctamente", mensaje_servidor.paginaHTML);
        fwrite(mensaje_log.mensaje, strlen(mensaje_log.mensaje), 1, flog);
    }

    //char orden_encontrada[100] = "200 OK";
    //char pagina_encontrada[1000];
    //bzero(mensaje_servidor.contenido_paginaHTML, sizeof(mensaje_servidor.contenido_paginaHTML));

    if (t == 0)
    {
        todo = malloc(st.st_size);
        fread(todo, st.st_size, 1, fp); //revisar si el 30 PARAMETRO NO DEBE SER UNA UNIDAD MENOR
        //printf("ahi vaaaaa %s", todo);
        //fflush(stdout);
    }

    if (num_trozos != t+1)
    {
        strncpy(mensaje_servidor.contenido_paginaHTML, &todo[400 * t], 400);
    }
    else
    {
        strncpy(mensaje_servidor.contenido_paginaHTML, &todo[400 * t], st.st_size - (400 * t));
    }

    sprintf(mensaje_servidor.longitud_contenido, "%ld", strlen(mensaje_servidor.contenido_paginaHTML));
    fprintf(fpe, "\nLa pagina (tamano %ld) tiene el contenido %s", sizeof(mensaje_servidor.contenido_paginaHTML), mensaje_servidor.contenido_paginaHTML);

    strcat(mensaje_servidor.mensaje_respuesta_servidor, "HTTP/1.1 ");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "200 OK");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "Server: ");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, hostname);
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");

    strcat(mensaje_servidor.mensaje_respuesta_servidor, "Conexion: close");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");

    strcat(mensaje_servidor.mensaje_respuesta_servidor, "Content_Length: ");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, mensaje_servidor.longitud_contenido);
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
}

```

Por otro lado, si no existe la orden “GET” se escribe el error 501 en el log y `<html><body><h1>501 Not Implemented</h1></body></html>` en el campo de contenido de la página html en la estructura de mensajes de UDP. Una vez fuera de las condiciones para ver si está o no la orden “GET”, se hace un sleep de una centésima para simular el comportamiento de un servidor real y hacemos el sendto pasándole toda la estructura, y vaciamos los campos de las estructuras del log y la que usa el send para dejarlas listas para la siguiente iteración.

```

else
{
    num_trozos = 1;

    sprintf(mensaje_log.mensaje, "\n501 Not Implemented. La orden %s no esta implementada", mensaje_servidor.orden_cliente);
    fwrite(mensaje_log.mensaje, strlen(mensaje_log.mensaje), 1, flog);

    //char orden_no_implementada[100] = "501 Not Implemented";
    //char pagina_no_implementada[100] = "<html><body><h1>501 Not Implemented</h1></body></html>";
    strcat(mensaje_servidor.contenido_paginaHTML, "<html><body><h1>501 Not Implemented</h1></body></html>\n");
    sprintf(mensaje_servidor.longitud_contenido, "%ld", strlen(mensaje_servidor.contenido_paginaHTML));

    strcat(mensaje_servidor.mensaje_respuesta_servidor, "HTTP/1.1 ");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "501 Not Implemented");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "Server: ");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, hostname);
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");

    strcat(mensaje_servidor.mensaje_respuesta_servidor, "Conexion: close");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");

    strcat(mensaje_servidor.mensaje_respuesta_servidor, "Content_Length: ");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, mensaje_servidor.longitud_contenido);
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, "\r\n");
    strcat(mensaje_servidor.mensaje_respuesta_servidor, mensaje_servidor.contenido_paginaHTML);
}

```

Una vez fuera del bucle que hace los sendto correspondientes, se escribe en el log un mensaje de comunicación finalizada, acompañado de la hora actual obtenida con la función ctime y se cierra el fichero de flog.

```
time (&timevar);

char comunicacion_finalizada[500] = "Comunicacion finalizada: ";
//char puerto_efimero_cliente[50];

fwrite("\n", 1, 1, flog);
fwrite("\n", 1, 1, flog);

fwrite((char *) ctime(&timevar), strlen(ctime(&timevar)), 1, flog);
fwrite(argv, strlen(argv), 1, flog);
fwrite("\n", 1, 1, flog);

strcat(mensaje_log.comunicacion_realizada, "Comunicacion finalizada: ");
strcat(mensaje_log.comunicacion_finalizada, hostname);

if (inet_ntop(AF_INET, &(clientaddr_in.sin_addr), hostname, MAXHOST) == NULL)
{
    perror(" inet_ntop \n");
}

strcat(mensaje_log.comunicacion_finalizada, " ");
strcat(mensaje_log.comunicacion_finalizada, hostname);
strcat(mensaje_log.comunicacion_finalizada, " ");
strcat(mensaje_log.comunicacion_finalizada, "UDP");
strcat(mensaje_log.comunicacion_finalizada, " ");
sprintf(mensaje_log.puerto_efimero_cliente, "%u", ntohs(clientaddr_in.sin_port));
strcat(mensaje_log.comunicacion_finalizada, mensaje_log.puerto_efimero_cliente);
strcat(mensaje_log.comunicacion_finalizada, "\n");
fwrite(mensaje_log.comunicacion_finalizada, strlen(mensaje_log.comunicacion_finalizada), 1, flog);

fclose(flog);
```

Destacar que, a diferencia de serverTCP, aquí no existe un bucle while que vaya recibiendo mensajes, si no que solamente se hará una vez. Es por ello que aquí no hay conexiones de tipo keep-alive.

cliente.c

A diferencia de los programas de ejemplo, vamos a tener en el mismo archivo las funciones de los protocolos TCP y UDP. Se va a definir una estructura de tipo estructura_mensaje_cliente.

```
struct estructura_mensaje_cliente
{
    char mensaje[500];
    char orden_usuario[500];
    char hostname[100];
    char connection[100];
};
```

En primer lugar, vamos a coger el argumento del protocolo, que en este caso es argv[2]. Dependiendo de si es TCP o UDP haremos unas funciones u otras.

En el caso de que sea TCP, creamos el socket de TCP e inicializamos la estructura `sockaddr_in`. Posteriormente, se realiza la función `connect` para conectar con el servidor y con `getsockname` conseguimos la dirección IP y el puerto efímero del socket creado. Se escribe el mensaje con `connected` acompañado de la hora obtenida con la función `ctime`.

```
if (strcmp(argv[2], "TCP") == 0)
{
    /* Create the socket. */
    s = socket (AF_INET, SOCK_STREAM, 0);
    if (s == -1)
    {
        perror(argv[0]);
        fprintf(stderr, "%s: unable to create socket\n", argv[0]);
        exit(1);
    }

    /* clear out address structures */
    memset ((char *) &myaddr_in, 0, sizeof(struct sockaddr_in));
    memset ((char *) &servaddr_in, 0, sizeof(struct sockaddr_in));

    /* Set up the peer address to which we will connect. */
    servaddr_in.sin_family = AF_INET;

    /* Get the host information for the hostname that the
     * user passed in. */
    memset (&hints, 0, sizeof (hints));
    hints.ai_family = AF_INET;

    /* esta función es la recomendada para la compatibilidad con IPv6 gethostbyname queda obsoleta */
    errcode = getaddrinfo (argv[1], NULL, &hints, &res);
    if (errcode != 0)
    {
        /* Name was not found. Return a
         * special value signifying the error. */
        fprintf(stderr, "%s: No es posible resolver la IP de %s\n", argv[0], argv[1]);
        exit(1);
    }
    else
    {
        /* Copy address of host */
        servaddr_in.sin_addr = ((struct sockaddr *) res->ai_addr)->sin_addr;
    }

    freeaddrinfo(res);

    /* puerto del servidor en orden de red */
    servaddr_in.sin_port = htons(PUERTO);
}
```

Se crea un bucle infinito, y por cada iteración se comprobará si la cadena contiene la palabra `ADIOS`, en cuyo caso hará `shutdown` y saldrá del bucle. Por otro lado, por defecto el tipo de conexión será `close`. Se hace una comprobación para ver si hay una `k`, en cuyo caso el tipo de conexión será `keep-alive`.

```

while (a == 0)
{
    bzero( (char *) &mensaje_cliente, sizeof(mensaje_cliente));

    if (fgets(mensaje_cliente.orden_usuario, sizeof(mensaje_cliente.orden_usuario),
    {
        exit(1);
    }

    for (int k = 0; k < sizeof(mensaje_cliente.orden_usuario); k++)
    {
        if (mensaje_cliente.orden_usuario[k] == '\n')
        {
            mensaje_cliente.orden_usuario[k] = '\0';
        }
    }

    if (strcmp(mensaje_cliente.orden_usuario, "ADIOS") == 0)
    {
        if (shutdown(s, 1) == -1)
        {
            perror(argv[0]);
            fprintf(stderr, "%s: unable to shutdown socket\n", argv[0]);
            exit(1);
        }
        break;
    }
}

```

```

int i;
strcpy(mensaje_cliente.connection, "close");
for (i = 0; i < sizeof(mensaje_cliente.orden_usuario); i++)
{
    if (mensaje_cliente.orden_usuario[i] == 'k' && mensaje_cliente.orden_usuario[i-1] == ' ')
    {
        strcpy(mensaje_cliente.connection, "keep-alive");
        break;
    }
}

```

Posteriormente, se rellenan los campos de la estructura estructura_mensaje_cliente y se hace el send.

```

strcat(mensaje_cliente.mensaje, mensaje_cliente.orden_usuario);
strcat(mensaje_cliente.mensaje, " HTTP/1.1");
strcat(mensaje_cliente.mensaje, "\r\n");
strcat(mensaje_cliente.mensaje, "Host: ");
strcat(mensaje_cliente.mensaje, mensaje_cliente.hostname);
strcat(mensaje_cliente.mensaje, "\r\n");
strcat(mensaje_cliente.mensaje, "Connection: ");
strcat(mensaje_cliente.mensaje, mensaje_cliente.connection);
strcat(mensaje_cliente.mensaje, "\r\n");
strcat(mensaje_cliente.mensaje, "\r\n");

if (send(s, mensaje_cliente.mensaje, sizeof(mensaje_cliente.mensaje), 0) != sizeof(mensaje_cliente.mensaje))
{
    fprintf(stderr, "%s: Connection aborted on error ", argv[0]);
    fprintf(stderr, "on send number %d\n", i);
    exit(1);
}
//FIN else if (strcmp(mensaje_cliente.orden_usuario, "ADIOS") == 0)

```


Ahora, para la respuesta, tendremos que hacer tantos recv como trozos haya. Para ello, dividimos el tamaño de la página entre 32000 y se hará un bucle for que itera tantas veces como trozos haya, y en cada iteración del bucle se hará un recv. En caso de que en la respuesta haya un error 501, se saldrá de este bucle. Al final de cada iteración se hace un bzero a la cadena que almacena la respuesta de manera que se pueda almacenar el siguiente trozo en la siguiente iteración.

```
int num_trozos = 1;

struct stat st;
stat(pagina, &st);
num_trozos = num_trozos + st.st_size / 32000;

char respuesta[32500];
printf("\n");
for (int t = 0; t < num_trozos; t++)
{
    i = recv(s, respuesta, sizeof(respuesta), 0);
    if (i == -1)
    {
        perror(argv[0]);
        fprintf(stderr, "%s: error reading result\n", argv[0]);
        exit(1);
    }
}
```

Una vez hayamos salido del for, en caso de que el tipo de conexión sea close se hará un shutdown y se saldrá del bucle infinito. Además, se hace un fclose y se cierra el socket y se imprime por pantalla un mensaje que informa de que se ha completado acompañado de el tiempo.

```
if (strcmp(mensaje_cliente.connection, "close") == 0)
{
    if (shutdown(s, 1) == -1) {
        perror(argv[0]);
        fprintf(stderr, "%s: unable to shutdown socket\n", argv[0]);
        exit(1);
    }
    break;
}
} // FIN while (a == 0)
```

```

        if (strcmp(&respuesta[i], "<html><body><h1>501 Not Implemented</h1></body></html>\n") == 0)
        {
            break;
        }

        bzero(respuesta, sizeof(respuesta));
    } // FIN for (int t = 0; t < num_trozos; t++)

    if (strcmp(mensaje_cliente.connection, "close") == 0)
    {
        if (shutdown(s, 1) == -1) {
            perror(argv[0]);
            fprintf(stderr, "%s: unable to shutdown socket\n", argv[0]);
            exit(1);
        }
        break;
    }
} // FIN while (a == 0)

fclose(fp);
fclose(fpe);
close(s);

/* Print message indicating completion of task. */
time(&timevar);
printf("All done at %s", (char *)ctime(&timevar));

```

Ahora vamos con UDP. En primer lugar se crea el socket de UDP, se hace bind y getsockname igual que con TCP. Una vez conectado se imprime por pantalla el mensaje que indica que está conectado acompañado de la hora.

```

else if (strcmp(argv[2], "UDP") == 0)
{
    s = socket (AF_INET, SOCK_DGRAM, 0);
    if (s == -1)
    {
        perror(argv[0]);
        fprintf(stderr, "%s: unable to create socket\n", argv[0]);
        exit(1);
    }

    /* clear out address structures */
    memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
    memset ((char *)&servaddr_in, 0, sizeof(struct sockaddr_in));

    /* Bind socket to some local address so that the
     * server can send the reply back. A port number
     * of zero will be used so that the system will
     * assign any available port number. An address
     * of INADDR_ANY will be used so we do not have to
     * look up the internet address of the local host.
     */

    myaddr_in.sin_family = AF_INET;
    myaddr_in.sin_port = 0;
    myaddr_in.sin_addr.s_addr = INADDR_ANY;
    if (bind(s, (const struct sockaddr *) &myaddr_in, sizeof(struct sockaddr_in)) == -1)
    {
        perror(argv[0]);
        fprintf(stderr, "%s: unable to bind socket\n", argv[0]);
        exit(1);
    }

    addrlen = sizeof(struct sockaddr_in);
    if (getsockname(s, (struct sockaddr *) &myaddr_in, &addrlen) == -1)
    {
        perror(argv[0]);
        fprintf(stderr, "%s: unable to read socket address\n", argv[0]);
        exit(1);
    }

    /* Print out a startup message for the user. */
    time(&timevar);
}

```

Debido a que es UDP, vamos a registrar SIGALRM para no quedar bloqueados en los `recvfrom` y vamos a definir un número de intentos con la variable `n_retry` que obtendrá el valor de la macro `RETRIES`.

```
vec.sa_handler = (void *) handler;
vec.sa_flags = 0;
if ( sigaction(SIGALRM, &vec, (struct sigaction *) 0) == -1)
{
    perror(" sigaction(SIGALRM)");
    fprintf(stderr, "%s: unable to register the SIGALRM signal\n", argv[0]);
    exit(1);
}
```

Se hará un bucle `while` que iterará mientras el número de intentos sea mayor que cero. Dentro de este bucle se hará el `sendto`.

```
while (n_retry > 0)
{
    int status;
    status = getnameinfo((struct sockaddr *)&myaddr_in, sizeof(myaddr_in), mensaje_cliente.hostname, sizeof(mensaje_cliente.hostname), NULL, 0, 0);
    if (status)
    {
        if (inet_ntop(AF_INET, &myaddr_in.sin_addr, mensaje_cliente.hostname, sizeof(mensaje_cliente.hostname)) == NULL)
        {
            perror(" inet_ntop \n");
        }
    }

    strcat(mensaje_cliente.mensaje, mensaje_cliente.orden_usuario);
    strcat(mensaje_cliente.mensaje, " HTTP/1.1");
    strcat(mensaje_cliente.mensaje, "\r\n");
    strcat(mensaje_cliente.mensaje, "Host: ");
    strcat(mensaje_cliente.mensaje, mensaje_cliente.hostname);
    strcat(mensaje_cliente.mensaje, "\r\n");
    strcat(mensaje_cliente.mensaje, "Connection: close");
    strcat(mensaje_cliente.mensaje, "\r\n");
    strcat(mensaje_cliente.mensaje, "\r\n");

    if (sendto(s, mensaje_cliente.mensaje, strlen(mensaje_cliente.mensaje), 0, (struct sockaddr *)&servaddr_in, sizeof(struct sockaddr_in)) == -1)
    {
        perror(argv[0]);
        fprintf(stderr, "%s: unable to send request\n", argv[0]);
        exit(1);
    }
}
```

Para la respuesta, tendremos que conseguir el número de trozos, y se hará un bucle que iterará tantas veces como número de trozos haya y en cada iteración. se pondrá un alarm con un `timeout` y se hace un `recvfrom`.

```

int num_trozos = 1;

struct stat st;
stat(pagina, &st);
//printf("la longitud del cliente es %ld y el nombre del fichero es %s", st.st_size, pagina);
//respuesta = (char *) malloc(TAMANO_MENSAJE_MAXIMO + 500);

num_trozos = num_trozos + st.st_size/ TAMANO_MENSAJE_MAXIMO;
/* Set up a timeout so I don't hang in case the packet
 * gets lost. After all, UDP does not guarantee
 * delivery.
 */
for (int t = 0; t < num_trozos; t++) {
    alarm(TIMEOUT);
    /* Wait for the reply to come in. */
    char respuesta[1000];
    if (recvfrom(s, respuesta, sizeof(respuesta), 0, (struct sockaddr *)&servaddr_in, &addrlen) == -1)
    {
        if (errno == EINTR)
        {
            /* Alarm went off and aborted the receive.
             * Need to retry the request if we have
             * not already exceeded the retry limit.
             */
            printf("attempt %d (retries %d).\n", n_retry, RETRIES);
            n_retry--;
        }
        else
        {
            printf("Unable to get response from");
            exit(1);
        }
    }
}

```

En caso de que `n_retry` sea cero se imprimirá un mensaje por pantalla. Se cerrará el socket y el archivo.

```

if (n_retry == 0)
{
    printf("Unable to get response from");
    printf(" %s after %d attempts.\n", argv[1], RETRIES);
}

close(s);
fclose(fp);

```

Si `argv[2]` no es ni TCP ni UDP hace lo siguiente:

```

else
{
    fprintf(stderr, "Usage: %s <nameserver> <TCP/UDP> <fichero-ordenes>\n", argv[0]);
    exit(1);
}

```

Resultados de ejecuciones:

Ejemplo de TCP:

```
melgo@melgo-VirtualBox:~/Escritorio/Practica Sockets HTTP-202112137202110Z-001/Practica Sockets HTTP con trozeado (ordenado)-202112147093907Z-001/Sockets_HTTP$ ./cliente localhost TCP ordenes3.txt
Connected to localhost on port 53976 at Tue Dec 14 15:25:10 2021
Startup from localhost port 53976 at Tue Dec 14 15:25:10 2021
<html lang="es">
<head>
<title>Departamento de Informática y Automática - Universidad de Salamanca</title>
</head>
<body>
<table>
<tr>
<td>Redes de Computadores I</td>
</tr>
</table>
</body>
</html>
<HTML lang="es">
<head>
<title>Departamento de Informática y Automática - Universidad de Salamanca</title>
</head>
<body>
<table>
<tr>
<td>Redes de Computadores I</td>
</tr>
</table>
</body>
</html>
<html><body><h1>501 Not Implemented</h1></body></html>

<html><body><h1>404 Not Found</h1></body></html>
All done at Tue Dec 14 15:25:10 2021
melgo@melgo-VirtualBox:~/Escritorio/Practica Sockets HTTP-202112137202110Z-001/Practica Sockets HTTP con trozeado (ordenado)-202112147093907Z-001/Sockets_HTTP$ Completed 127.0.0.1 port 53976, 0 request
s, at Tue Dec 14 15:25:11 2021
melgo@melgo-VirtualBox:~/Escritorio/Practica Sockets HTTP-202112137202110Z-001/Practica Sockets HTTP con trozeado (ordenado)-202112147093907Z-001/Sockets_HTTP$
```

Fichero del puerto efímero:



```
1 |
2 |
3 | El mensaje enviado por el cliente es: GET /docencia.html HTTP/1.1
4 | Host: localhost
5 | Connection: keep-alive
6 |
7 |
8 | El mensaje recibido del servidor es: HTTP/1.1 200 OK
9 | Server: localhost
10 | Connection: keep-alive
11 | Content-Length: 198
12 |
13 | <HTML lang="es">
14 | <head>
15 | <title>Departamento de Informática y Automática - Universidad de Salamanca</title>
16 | </head>
17 | <body>
18 | <table>
19 | <tr>
20 | <td>Redes de Computadores I</td>
21 | </tr>
22 | </table>
23 | </body>
24 | </html>
25 |
26 | El mensaje enviado por el cliente es: GET /docencia.html HTTP/1.1
27 | Host: localhost
28 | Connection: keep-alive
29 |
30 |
31 | El mensaje recibido del servidor es: HTTP/1.1 200 OK
32 | Server: localhost
33 | Connection: keep-alive
34 | Content-Length: 198
35 |
36 | <HTML lang="es">
37 | <head>
38 | <title>Departamento de Informática y Automática - Universidad de Salamanca</title>
39 | </head>
40 | <body>
41 | <table>
42 | <tr>
43 | <td>Redes de Computadores I</td>
44 | </tr>
45 | </table>
46 | </body>
47 | </html>
```

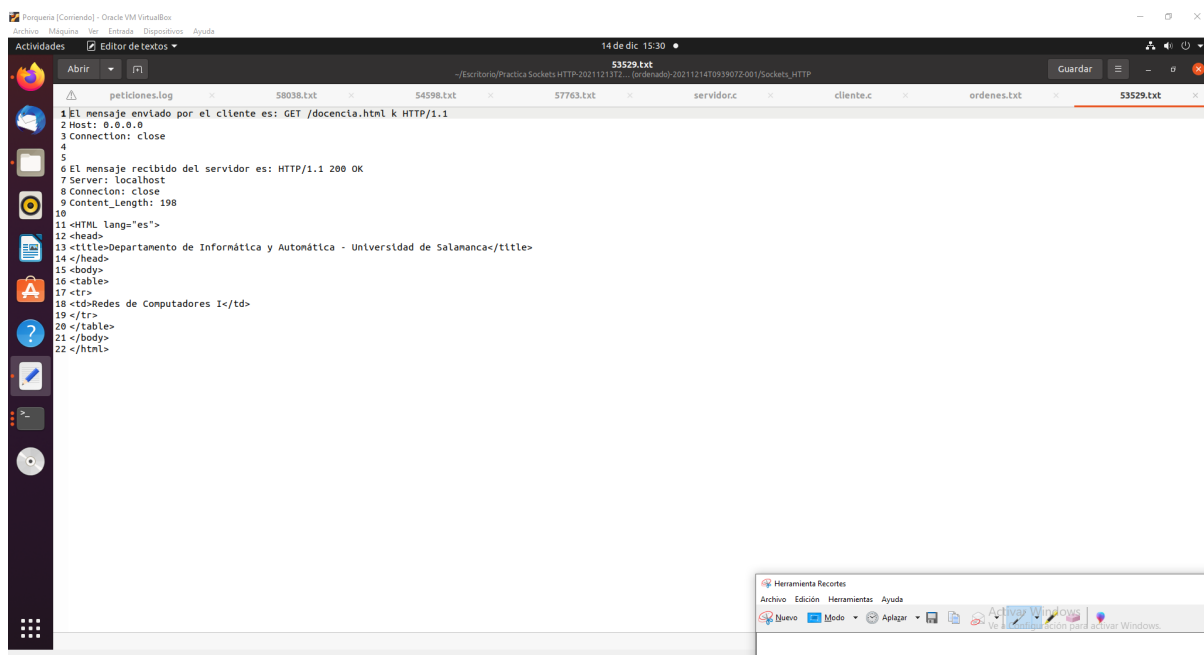


```
26 El mensaje enviado por el cliente es: GET /docencia.html HTTP/1.1
27 Host: localhost
28 Connection: keep-alive
29
30
31 El mensaje recibido del servidor es: HTTP/1.1 200 OK
32 Server: localhost
33 Connection: keep-alive
34 Content-Length: 198
35
36 <HTML lang="es">
37 <head>
38 <title>Departamento de Informática y Automática - Universidad de Salamanca</title>
39 </head>
40 <body>
41 <table>
42 <tr>
43 <td>Redes de Computadores I</td>
44 </tr>
45 </table>
46 </body>
47 </html>
48 =====
49 El mensaje enviado por el cliente es: GET /docencia2.html HTTP/1.1
50 Host: localhost
51 Connection: keep-alive
52
53
54 El mensaje recibido del servidor es: HTTP/1.1 501 Not Implemented
55 Server: localhost
56 Connection: keep-alive
57 Content-Length: 55
58
59 <html><body><h1>501 Not Implemented</h1></body></html>
60
61 =====
62 El mensaje enviado por el cliente es: GET /docencia2.html HTTP/1.1
63 Host: localhost
64 Connection: close
65
66
67 El mensaje recibido del servidor es: HTTP/1.1 404 Not Found
68 Server: localhost
69 Connection: close
70 Content-Length: 49
71
72 <html><body><h1>404 Not Found</h1></body></html>
```

Ejemplo de UDP:

```
rnelgo@rnelgo-VirtualBox:~/Escritorio/Practica Sockets HTTP-20211213T202110Z-001/Practica Sockets HTTP con trozado (ordenado)-20211214T093907Z-001/Sockets_HTTP$ ./cliente localhost UDP ordenes3.txt
connected to localhost on port 53529 at Tue Dec 14 15:28:50 2021
Startup from localhost port 53529 at Tue Dec 14 15:28:50 2021
<HTML lang="es">
<head>
<title>Departamento de Informática y Automática - Universidad de Salamanca</title>
</head>
<body>
<table>
<tr>
<td>Redes de Computadores I</td>
</tr>
</table>
</body>
</html>
```

Fichero del puerto efímero:



```
1 El mensaje enviado por el cliente es: GET /docencia.html k HTTP/1.1
2 Host: 0.0.0.0
3 Connection: close
4
5
6 El mensaje recibido del servidor es: HTTP/1.1 200 OK
7 Server: localhost
8 Connection: close
9 Content-Length: 198
10
11 <HTML lang="es">
12 <head>
13 <title>Departamento de Informática y Automática - Universidad de Salamanca</title>
14 </head>
15 <body>
16 <table>
17 <tr>
18 <td>Redes de Computadores I</td>
19 </tr>
20 </table>
21 </body>
22 </html>
```

Fichero peticiones.log:

```
146
147 Tue Dec 14 14:43:35 2021
148 ./servidor
149 Comunicacion finalizada: localhost 127.0.0.1 UDP 58418
150 Tue Dec 14 15:14:41 2021
151 ./servidor
152 Comunicacion realizada: localhost 127.0.0.1 UDP 58404
153
154 200 OK. La pagina index.html se ha encontrado correctamente
155
156 Tue Dec 14 15:14:43 2021
157 ./servidor
158 Comunicacion finalizada: localhost 127.0.0.1 UDP 58404
159 Tue Dec 14 15:25:10 2021
160 ./servidor
161
162 Comunicacion realizada: localhost 127.0.0.1 TCP 53976
163
164 200 OK. La pagina docencia.html se ha encontrado correctamente
165 200 OK. La pagina docencia.html se ha encontrado correctamente
166 505 Not Implemented. La orden DAME no esta implementada
167 404 Not Found. La pagina docencia2.html no se ha encontrado
168
169 Tue Dec 14 15:25:10 2021
170 ./servidor
171 Comunicacion finalizada: localhost 127.0.0.1 TCP 53976
172 Tue Dec 14 15:28:43 2021
173 ./servidor
174 Comunicacion realizada: localhost 127.0.0.1 UDP 39812
175
176 200 OK. La pagina index.html se ha encontrado correctamente
177
178 Tue Dec 14 15:28:46 2021
179 ./servidor
180 Comunicacion finalizada: localhost 127.0.0.1 UDP 39812
181 Tue Dec 14 15:28:50 2021
182 ./servidor
183 Comunicacion realizada: localhost 127.0.0.1 UDP 53529
184
185 200 OK. La pagina docencia.html se ha encontrado correctamente
186
187 Tue Dec 14 15:28:50 2021
188 ./servidor
189 Comunicacion finalizada: localhost 127.0.0.1 UDP 53529
```

Por último, decir que en nugal, por alguna razón en UDP no coge bien la orden GET, pero creemos que el fallo debe de ser menor, ya que TCP usa el mismo sistema y funciona bien. En Linux funciona todo correctamente.