Universidad de Murcia

Servicios Telemáticos

PCEO

Servidor Web: Entrega Voluntaria

Autor:
Jose Antonio Lorencio Abril
49196915F
joseantonio.lorencioa@um.es

Profesor:
Esteban Belmonte Rodríguez

Marzo 2020

Índice

Índice de cuadros

Índice de figuras

1. Descripción de la implementación del servicio web-SSTT HTTP

Para esta explicación, voy a seguir el flujo del programa. Es decir, voy a suponer que un cliente quiere contactar con mi servidor e intercambiar mensajes HTTP, explicando el funcionamiento del servidor en cada caso posible.

Primero, notar que para poder llamar www.sstt6915.org a mi servidor he modificado el fichero /etc/hosts del cliente. Añadiendo la siguiente línea:

192.168.56.104 www.sstt6915.org

El establecimiento de la conexión TCP se realizaba correctamente con el código original proporcionado por el equipo docente. Sabemos que se crea un hijo y este pasa a procesar la petición web. A este procedimiento, *process_web_request*, le he añadido un nuevo parámetro, *int port*, para poder conocer el puerto en el que estoy operando en todo momento.

En este momento nos encontramos dentro de este procedimiento, que procesará peticiones entrantes hasta que:

1. El cliente cierre la conexión TCP
2. Salte el timeout, es decir, el cliente pase más tiempo de lo que dura timeout sin enviar ninguna petición
3. El cliente envíe más cantidad de peticiones de las definidas como máximo (esta es una mejora que he realizado)

Para ello, el procesamiento de peticiones web se encuentra dentro de un bucle de la siguiente forma:

```
void process_web_request(int descriptorFichero, int port)
{
  fd_set client;
  FD_ZERO(&client);
  FD_SET(descriptorFichero, &client);

  struct timeval timeout;
  timeout.tv_sec = TIMEOUT;
  timeout.tv_usec = 0;

  int maxpeticiones=MAXPET;

  while (select(descriptorFichero + 1, &client, NULL, NULL, &timeout))
  {
        //LEER PETICIÓN
        if(LECTURA INCORRECTA){
            exit(-1)
        }
    else
    {
      if(maxpeticiones--<=0){
        //Tratar el caso en que se excede el número máximo de peticiones (MEJORA)
        //ENVIAR code 429 Too Many Requests
        break; //salir del bucle
      }

      //
      //  TRATAR LOS CASOS DE LOS DIFERENTES METODOS QUE SE USAN
      //

      //CHEQUEAR FORMATO DE CABECERA

      if(FORMATO CABECERA INCORRECTO){
          //ENVIAR CODE 400 Bad Request
          continue; //regresar al principio del bucle
      }
      else if (MÉTODO GET) == 0)
      {
        //PROCESAR GET Y RESPONDER
      }
      else if (MÉTODO POST)
      {
        //PROCESAR POST Y RESPONDER
      }
      else{ //Si no es GET ni POST
          //ENVIAR CODE 400 Bad Request
      }

            //Reiniciar timeout
      timeout.tv_sec = 15;
      timeout.tv_usec = 0;
    }
  }
  close(descriptorFichero); //Al acabar el bucle se cierra la conexión TCP
  exit(0);
}
```

Cuadro 1: Esquema general: Persistencia

Aquí muestro un boceto del procedimiento, pero se ve claramente la implementación de la persistencia y de la mejora. Como vemos, creo un conjunto de descriptores de fichero en el que introduzco el descriptor de fichero asociado al socket de conexión con el cliente. Establezco el timeout que considero oportuno (es un #define) y el número máximo de peticiones (otro #define) que puedo atender en una misma conexión TCP. Esta variable se decrementa con cada petición que llega y, al llegar a 0, responderé con un mensaje con código 429 Too Many Requests. Si ocurre esto o salta el timeout, se cerrará la conexión TCP. El cliente deberá establecer una nueva conexión con el servidor para poder continuar enviando peticiones

HTTP. Esto tiene repercusiones en la cabecera de los mensajes HTTP que envío, ahora iremos a ese punto.

Como vemos en el código anterior, el siguiente paso es leer la petición del cliente.

```
    // Definir buffer y variables necesarias para leer las peticiones
    char buffer[BUFSIZE] = {0}; //aquí guardaré el mensaje
    char header[BUFSIZE], body[BUFSIZE]; //aquí guardaré la cabecera y el cuerpo, resp.
    char *reqline[3]; //aquí guardaré los tres primeros campos de la cabecera
    int bytes_leidos; //total bytes leidos
    char *pwd = getenv("PWD"); //Directorio del servidor
    char demanded_url[PATH_MAX], rpath[PATH_MAX]; //url pedida y su resolución en el ordenador

    char host[FILENAME_MAX]; //nombre de host
    sprintf(host, "www.sstt6915.org:%d", port); //Así envío mi nombre de host

    bytes_leidos = leer(buffer, header, body, descriptorFichero); //Leo la petición

    if (bytes_leidos < 0)
    {
      //ERROR DE LECTURA
      exit(-1);
    }
    else if (bytes_leidos == 0)
    {
      //CLIENTE DESCONECTADO
      descriptorFichero);
      exit(-1);
    }
    else
    {
      //LA LECTURA HA IDO BIEN

      if(maxpeticiones--<=0){
        //Tratar el caso en que se excede el número máximo de peticiones (MEJORA)
        strcpy(demanded_url,"/home/alumno/Practicas/errors/429_TooManyRequests.html");
        send_file_from_url(429, "Too Many Requests", demanded_url, host, descriptorFichero);
        //Cierro la conexión por considerarlo un ataque a mi servidor
        break;
      }

      char *start_line = strtok(buffer, "\r\n");
      reqline[0] = strtok(start_line, " \t");
      reqline[1] = strtok(NULL, " \t");
      reqline[2] = strtok(NULL, " \t");

            //COMPROBAMOS EL FORMATO DE LA CABECERA
      if(CABECERA INCORRECTA){
        //ENVIAR ERROR 400 Bad Request
      }
      else if (strncmp(reqline[0], "GET\0", 4) == 0) //Compruebo si es un GET
      {
        //PROCESAR GET
      }
      else if (strncmp(reqline[0], "POST\0", 5) == 0) //Compruebo si es un POST
      {
        //PROCESAR POST
      }
      else{ //Si no es GET ni POST
        //ENVIAR ERROR 400 Bad Request
      }
      //  Como se trata el caso excepcional de la URL que no apunta a ningún fichero
      //  html
      timeout.tv_sec = 15;
      timeout.tv_usec = 0;
    }
  }
  close(descriptorFichero);
  exit(0);
```

Cuadro 2: Esquema general: lectura de la petición

Fijémonos primero en el array *reqline[3]*, en el que guardo los 3 campos de la primera línea del mensaje HTTP. Pongamos ahora nuestro ojo en la función *leer*. Que es así:

```
int leer(char buffer[BUFSIZE], char header[BUFSIZE], char body[BUFSIZE], int descriptorFichero){
  int bytes_leidos, bytes_acum=0; //tamaño leido en cada trama TCP, tamaño total
  int continue_reading=0; //si hay cuerpo, se activa esta variable
  int body_len=0; //si hay cuerpo, aquí guardo su tamaño
  while(((bytes_leidos = read(descriptorFichero, buffer+bytes_acum, BUFSIZE)) > 0) && !strstr(buffer+
    max(bytes_acum-3,0), "\r\n\r\n") && !continue_reading){
    bytes_acum+=bytes_leidos;
    char *con_len=NULL;
    //Si hay cuerpo, no paro en \r\n\r\n, si no cuando he leido todo
    if(!continue_reading && (con_len=strstr(buffer, "Content-Length: "))!=NULL){
      con_len += 16;
      char *endline=strstr(con_len, "\r\n");
      char cl[4];
      strncpy(cl, con_len, endline-con_len);
      continue_reading = body_len = atoi(cl);
    }
    if(continue_reading){
      continue_reading-=bytes_leidos;
    }
  }
  bytes_acum+=bytes_leidos; //Los últimos bytes leidos
  if (bytes_leidos < 0)
  {
    return(-1);
  }
  else if (bytes_leidos == 0)
  {
    return(0);
  }
  else
  {
    //
    // Si la lectura tiene datos válidos terminar el buffer con un \0
    //
    buffer[bytes_acum] = '\0';


    //Si el content_length entró en la última iteración, no lo compruebo dentro del while,
    //tengo que hacerlo fuera
    char *con_len=NULL;
    if(body_len == 0 && (con_len=strstr(buffer, "Content-Length: "))!=NULL){
      con_len += 16;
      char *endline=strstr(con_len, "\r\n");
      char cl[4];
      strncpy(cl, con_len, endline-con_len);
      body_len = atoi(cl);
    }

    char *header_ends = strstr(buffer, "\r\n\r\n");
    int head_len = header_ends-buffer;
    strncpy(header, buffer, head_len);
    if(body_len>0){
      strncpy(body, buffer+head_len+4, body_len);
    }
  }
}
```

Cuadro 3: Procedimiento *Leer* para leer la petición HTTP

Lo que hago es ir leyendo hasta que encuentro "\r\n\r\n", lo que significa que la cabecera la he recibido completamente. Si encuentro el campo *ContentLength* en la cabecera, significa que también hay cuerpo, por lo que debo leer una cantidad adicional de bytes, indicada en ese mismo campo. Para eso sirve la variable *continue_reading*.

Por otro lado, es importante destacar que la última comprobación del while no ejecuta el código interior, por lo que fuera debo sumar los últimos bytes leidos, así como comprobar si hay cuerpo.

Además, hay que notar que he supuesto que el mensaje que me van a enviar ocupa menos de 8K. Esta suposición es razonable en un servidor en pequeña escala como el nuestro, quizá para trabajos mayores habría que deshacerse de esta suposición.

Ahora pasamos a comprobar si la cabecera del mensaje es correcta. Para ello he creado la función *check_header(char header[BUFSIZE], int port)*, que es como sigue:

```
int check_header(char header[BUFSIZE], int port){

      regex_t regex_line;
      char *reqline[3];
      reqline[0]=strtok(header, " \t");
      reqline[1] = strtok(NULL, " \t");
      reqline[2] = strtok(NULL, " \t\n");

      int must=1;
      for(int i=0;i<3;i++){
        if(reqline[i]==NULL || !strcmp(reqline[i]," "))
          must=0;
      }
      if(!must) return 0;

      if(strncmp(reqline[2], "HTTP/1.0", 8) != 0 && strncmp(reqline[2], "HTTP/1.1", 8) != 0){
        return 0;
      }

      char *line = strtok(NULL, "\r\n");

      int match = regcomp(&regex_line, "([a-zA-Z][a-zA-Z0-9]*: ..*)|([a-zA-Z][a-zA-Z0-9]*=..*)",
    REG_EXTENDED);

      int hostexists=0;
      char *is_host=NULL;
      char host[30], ip[30];

      sprintf(host, "www.sstt6915.org:%d", port);
      sprintf(ip, "192.168.56.104:%d", port);

      while(line!=NULL){
        match = regexec(&regex_line, line, (size_t) 0, NULL, 0);

        if (match != 0) {
          regfree(&regex_line);
          return 0;
        }

        if(!hostexists && (((is_host=strstr(line, host))!=NULL) || ((is_host=strstr(line, ip))!=NULL)))
    {
          hostexists=1;
        }
        line = strtok(NULL, "\r\n");
      }
      regfree(&regex_line);
      return hostexists;
}
```

Cuadro 4: Parseo de la cabecera

Como se puede apreciar, he recurrido a expresiones regulares para realizar esta tarea. El funcionamiento es el siguiente:

1. Cojo la primera línea, para lo que utilizo la función strtok. Compruebo que esta línea tiene 3 campos.
2. Compruebo que el tercero de esos campos me indica alguna de las versiones soportadas de HTTP por mi servidor
3. Avanzo de linea
4. Si la linea no es vacía, le paso la expresión regular. Si es vacía, fin, retornar 1.
5. Si hay error, retornar 0. La cabecera es incorrecta
6. Si no hay error, puede ocurrir:
   a) Aun no he encontrado el host. En este caso compruebo si el host se encuentra en esta línea y si contiene un nombre de host correcto, y si es así pongo la variable que indica esto a 1 (true)
   b) Ya he encontrado el host, seguimos
7. Volver al paso 3

Tras comprobar la correción de la cabecera vemos de qué método HTTP se trata. Si es un GET, el código ejecutado será el siguiente:

```
else if (strncmp(reqline[0], "GET\0", 4) == 0) //Compruebo si es un GET
  {
    if (strncmp(reqline[1], "/\0", 2) == 0)
      reqline[1] = "/index.html"; //Si no se indica nada, devuelvo el index.html

    strcpy(demanded_url, pwd);

    strcat(demanded_url, reqline[1]);

    //
    //  Como se trata el caso de acceso ilegal a directorios superiores de la
    //  jerarquia de directorios
    //  del sistema
    //
    realpath(demanded_url, rpath);
    if ((strstr(rpath, pwd) != NULL))
    {
      send_file_from_url(200, "OK", rpath, host, descriptorFichero);
    }
    else
    {
      //Tratar el caso en que acceden a un directorio superior al mio
      strcpy(demanded_url,"/home/alumno/Practicas/errors/403_Forbidden.html");
      send_file_from_url(403, "Forbidden", demanded_url, host, descriptorFichero);
      //Cierro la conexión por considerarlo un ataque a mi servidor
      break;
    }

  }
```

Cuadro 5: Método GET

Recurrimos ahora a *reqline[0]*, si el método es GET la comprobación del código será correcta.

Si en *reqline[1]* hay /, entonces lo cambio por */index.html*. Tomo *demanded_url*, le pongo el pwd y le concateno *reqline[1]*, ya sabemos la ruta relativa del recurso pedido. Con *realpath* obtenemos la ruta absoluta, comprobamos si esta ruta contiene la ruta absoluta del directorio del servidor y, de no ser así, sabemos que están intentando acceder a directorios superiores o paralelos de la jerarquía de directorios, por lo que mandamos un error 403 Forbidden y cerramos la conexión por considerarlo un ataque al servidor.

En otro caso, el recurso pedido estará debajo de nuestro directorio, enviaríamos entonces, en principio (si la extensión es correcta y si existe, lo que se comprueba después), un code 200 OK y el recurso pedido.

Antes de ver cómo funciona el procedimiento *send_file_from_url* voy a explicar el método POST, cuyo código queda así:

```
else if (strncmp(reqline[0], "POST\0", 5) == 0) //Compruebo si es un POST
  {
    char *email=NULL;
    if((email=strstr(body,"email=joseantonio.lorencioa\%40um.es"))!=NULL){
      strcpy(demanded_url, pwd);
      strcat(demanded_url, "/Privado/successful_login.html");
      send_file_from_url(200, "OK", demanded_url, host, descriptorFichero);
    }
    else
    {
      strcpy(demanded_url, pwd);
      strcat(demanded_url, "/login_gone_wrong.html");
      send_file_from_url(200, "OK", demanded_url, host, descriptorFichero);
    }
  }
  else{ //Si no es GET ni POST
    strcpy(demanded_url, "/home/alumno/Practicas/errors/400_BadRequest.html");
    send_file_from_url(400, NULL, demanded_url, host, descriptorFichero);
  }
```

Cuadro 6: POST y método incorrecto

Lo único que tengo que hacer es inspeccionar el cuerpo del mensaje y ver que contiene la línea

email=joseantonio.lorencioa@um.es

Caso de contenerla, envío code 200 OK y la página html diseñada específicamente para este caso, contenida en la carpeta Privado (que realmente no es privada, al menos por el momento).

Si no, envío otra página también diseñada para este caso, que indica que no se ha introducido un email correcto.

También se ve en este snippet de código, qué ocurre cuando la petición no es un GET ni un POST, que no es ni más ni menos que el envío de un error 400 Bad Request.

Vamos a ver ahora la función encargada de enviar las respuestas HTTP, *send_file_from_url*, que la he implementado como sigue:

```c
int send_file_from_url(int code, char *codemsg, char *url, char *host, int descriptorFichero)
{
  int fd, bytes_read, size, len;
  char send_buf[BUFSIZE];
  struct stat st;
  char *contype;
  char demanded_url[PATH_MAX];
  strcpy(demanded_url, url);
  strtok(url, ".");
  contype = strtok(NULL, ".");

  //
  //  Evaluar el tipo de fichero que se está solicitando, y actuar en
  //  consecuencia devolviendolo si se soporta u devolviendo el error correspondiente en otro caso
  //  Devuelvo un mensaje con código 400 Bad Request
  //  También puede ocurrir que se requiera un mensaje de este tipo desde otro lugar, por eso el 'or'
  //


  if (((contype = get_extension(contype))==NULL)){
    code = 415;
    codemsg = "Unsupported Media Type";
    contype = "text/html";
    strcpy(demanded_url, "/home/alumno/Practicas/errors/415_UnsMediaType.html");
  }
  else if(code == 400){
    code = 400;
    codemsg = "Bad Request";
    contype = "text/html";
    strcpy(demanded_url, "/home/alumno/Practicas/errors/400_BadRequest.html");
  }
  //
  //  En caso de que el fichero sea soportado, exista, etc. se envia el fichero con la cabecera
  //  correspondiente, y el envio del fichero se hace en blockes de un máximo de  8kB
  //
  if ((fd = open(demanded_url, O_RDONLY)) != -1) //El fichero existe a partir de nuestro directorio
  {
    fstat(fd,&st);
    size = st.st_size;
    // Aquí pedimos que se cree la cabecera correcta
    len = make_header(code, codemsg, url, host, size, contype, send_buf);

    //
    //Asumo que mi cabecera no ocupa 8kB, lo cual tiene sentido, pues mis cabeceras son cortas
    //y caben en send_buf[BUFSIZE]
    //

    write(descriptorFichero, send_buf, len);

    //Aquí vamos enviando el fichero como mucho en bloques de BUFSIZE=8kB
    while ((bytes_read = read(fd, send_buf, BUFSIZE)) > 0)
      write(descriptorFichero, send_buf, bytes_read);
    return 1;
  }
  else
  {
    //Si todo es correcto, pero el fichero no existe a partir del directorio del servidor,
    //enviamos error 404 Not Found
    strcpy(demanded_url,"/home/alumno/Practicas/errors/404_NotFound.html");
    send_file_from_url(404, "Not Found", demanded_url, host, descriptorFichero);
    return 0;
  }
}
```

Cuadro 7: Procedimiento *send_file_from_url*

Los parámetros son el código de envío (200, 400, 403,...), el mensaje asociado a ese código (OK, Bad Request,...), la ruta del fichero a enviar, el nombre del host y el descriptor de fichero asociado al socket en el que vamos a escribir.

Lo primero que hago es ver si el recurso pedido tiene una extensión soportada por nuestro servidor. Para eso creé la función *get_extension*, que explicaré ahora. Si la extensión es incorrecta, cambio el código, la extensión y la ruta del recurso por la de una página html que muestra un error 415. Después de esta comprobación, veo si el código es 400, en tal caso, cambio el mensaje del código, la extensión, y la ruta del recurso por la de una página html que muestra un error 400. Esto lo hago así porque hay varios lugares desde los que me puede llegar que envíe un Bad Request y consideré más sencillo desde el punto de vista lógico hacerlo así. Ahora ya sabemos que tenemos el recurso que queremos enviar a nuestra disposición, ya sea un error 400 o 415, o un archivo a priori correcto. Tratamos de abrirlo para ver si tenemos dicho recurso. Pueden ahora ocurrir dos cosas:

- El fichero se abre correctamente. Usamos la estructura *stat st* para ver el tamaño del recurso a enviar. Ahora creamos la cabecera con la función *make_header*, que también explicaré. Escribimos en el socket la cabecera, y, por último, escribimos en el socket el recurso, en un while porque puede ser mayor que BUFSIZE.
- El fichero no se abre correctamente. Estamos ante un error 404. Cambiamos la ruta del recurso demandado por la de un html que expresa este error y llamamos a esta misma función con los parámetros adecuados para enviar un error 404. Nótese que aquí podríamos entrar en un bucle infinito si este archivo no se encuentra donde indicamos. En mi servidor me aseguro, por supuesto, de que esto no ocurre.

Solo quedan las dos últimas funciones que han quedado en el tintero. La primera de ellas es *get_extension*:

```
1  char* get_extension(char *contype){
2    char *aux="";
3    int i=0;
4    while(strcmp(aux, contype) && i<11){
5      aux = extensions[i].ext;
6      i++;
7    }
8    if(strcmp(aux, contype)){
9      return NULL;
10   }
11   return extensions[i-1].filetype;
12 }
```

Cuadro 8: Función *get_extension*

Es muy sencilla, lo único que hago es pasarle la extensión de un archivo, recorrer la estructura proporcionada en el código original del programa que contiene asociadas extensiones y lo que se envía en el mensaje HTTP para indicar esa extensión. Si encuentro mi extensión en la parte izquierda de la tabla, devuelvo la parte derecha. Si no la encuentro, devuelvo NULL.

Por último, vamos a ver la función *make_header*:

```
int make_header(int code, char *codemsg, char *url, char *host, int conlength, char *contype, char buf
    []){
  int n=sprintf(buf, "HTTP/1.1 %d %s\r\n", code, codemsg);
  n+=sprintf(buf+n, "Connection: Keep-Alive\r\n");

  if(host!=NULL){
    n+=sprintf(buf+n, "Host: %s\r\n", host);
  }

  n+=sprintf(buf+n, "Server: Ubuntu 16.04\r\n");

  if(url!=NULL){
  n+=sprintf(buf+n, "Content-Type: %s\r\n", contype);
  n+=sprintf(buf+n, "Content-Length: %d\r\n", conlength);
  }

  time_t date;
  time(&date);
  struct tm *local=localtime(&date);

  n+=sprintf(buf+n, "Date: %02d/%02d/%d %02d:%02d:%02d\r\n", local->tm_mday, local->tm_mon+1, local->
    tm_year+1900, local->tm_hour, local->tm_min, local->tm_sec);
  n+=sprintf(buf+n, "Keep-Alive: timeout=%d, max=%d\r\n\r\n", TIMEOUT, MAXPET);
  return n;
}
```

Cuadro 9: Función *make_header*

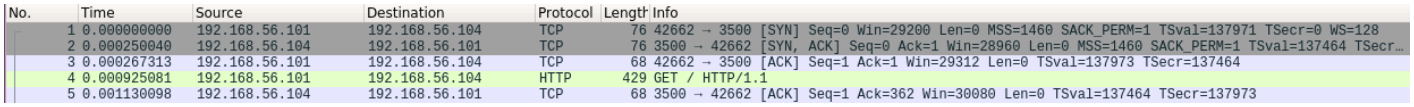
Que es también bastante simple. Le paso el código de envío, su mensaje asociado, la ruta del recurso, el host, el tamaño del cuerpo, la extensión del recurso y el buffer en el que quiero escribir la cabecera. Y simplemente voy escribiendo cada línea, acabándolas con un \r\n y no escribiendo las líneas que no debo. Para la fecha uso la estructura tm y el tipo de datos time_t, que sirven para esto. Escribo el timeout y el número máximo de peticiones que defino al comienzo, y termino con otro \r\n.

Creo que así queda bastante clara la implementación de mi servidor y todas sus funcionalidades.

2. Traza representativas del protocolo HTTP implementado

Vamos a ver unas cuantas trazas de Wireshark que considero representativas del funcionamiento de mi programa. En el archivo adjunto a la práctica *Captura_EntregaVoluntaria_49196915F.pcap* se puede observar también una traza representativa de lo que sucede. Si ponemos el filtro TCP podemos ver la conexión, así como el correcto funcionamiento de la persistencia. Si ponemos el filtro HTTP podremos observar el intercambio de mensajes HTTP de un par de sesiones TCP distintas.

Pasemos ahora a algunas tramas específicas, en la primera vamos a ver el establecimiento de la conexión TCP, una primera request HTTP GET y la respuesta y todo lo que se desencadena. Por último veremos una request HTTP POST y la respuesta.

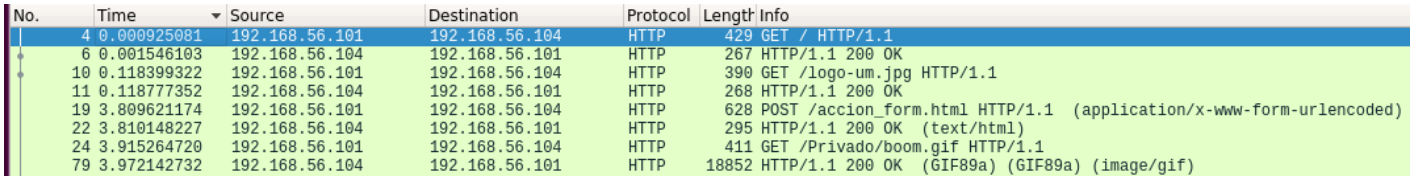


No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.56.101	192.168.56.104	TCP	76	42662 → 3500 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=137971 TSecr=0 WS=128
2	0.000250040	192.168.56.104	192.168.56.101	TCP	76	3500 → 42662 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=137464 TSecr…
3	0.000267313	192.168.56.101	192.168.56.104	TCP	68	42662 → 3500 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=137973 TSecr=137464
4	0.000925081	192.168.56.101	192.168.56.104	HTTP	429	GET / HTTP/1.1
5	0.001130098	192.168.56.104	192.168.56.101	TCP	68	3500 → 42662 [ACK] Seq=1 Ack=362 Win=30080 Len=0 TSval=137464 TSecr=137973

Figura 1: Establecimiento de la conexión y visualización general del intercambio de mensajes

Podemos observar como los dos primeros mensajes son los mensajes TCP SYN de establecimiento de la conexión. Ahora viene un ACK que ya contiene datos HTTP. En este caso una petición GET.

Vamos a pasar a ver los mensajes HTTP que se intercambian.



No.	Time	Source	Destination	Protocol	Length	Info
4	0.000925081	192.168.56.101	192.168.56.104	HTTP	429	GET / HTTP/1.1
6	0.001546103	192.168.56.104	192.168.56.101	HTTP	267	HTTP/1.1 200 OK
10	0.118399322	192.168.56.101	192.168.56.104	HTTP	390	GET /logo-um.jpg HTTP/1.1
11	0.118777352	192.168.56.104	192.168.56.101	HTTP	268	HTTP/1.1 200 OK
19	3.809621174	192.168.56.101	192.168.56.104	HTTP	628	POST /accion_form.html HTTP/1.1 (application/x-www-form-urlencoded)
22	3.810148227	192.168.56.104	192.168.56.101	HTTP	295	HTTP/1.1 200 OK (text/html)
24	3.915264720	192.168.56.101	192.168.56.104	HTTP	411	GET /Privado/boom.gif HTTP/1.1
79	3.972142732	192.168.56.104	192.168.56.101	HTTP	18852	HTTP/1.1 200 OK (GIF89a) (GIF89a) (image/gif)

Figura 2: Mensajes intercambiados para GET y POST

En esta figura observamos la secuencia de mensajes intercambiados. Que es, a alto nivel:

1. GET Index
2. 200 OK Index
3. GET Imagen_Dentro_de_Index
4. 200 OK Imagen_Dentro_de_Index
5. POST Formulario
6. 200 OK Formulario_Aceptado
7. GET Imagen_Dentro_de_Mensaje_Aceptación
8. 200 OK Imagen_Dentro_de_Mensaje_Aceptación

Y podemos ver ahora el interior de alguno de estos mensajes. Vamos a verlo del que nos devuelve un GIF.

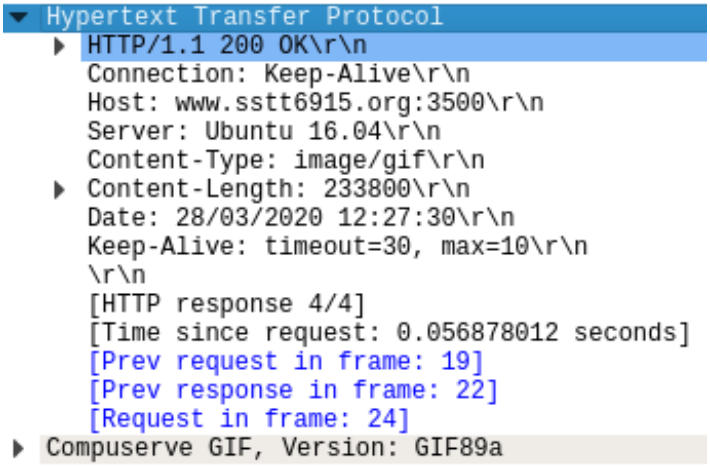


Figura 3: Mensaje HTTP que envía un GIF

Podemos observar los diferentes campos de la cabecera. Como el Connection, o el Keep-Alive, con el que indico el mecanismo de persistencia utilizado. Vemos que en content-type pone gif. Vemos también como en Content-Length se encuentra el tamaño del archivo gif. Y vemos como la cabecera cierra con una línea consistente en un único retorno de carro. Lo último es la forma en que se codifica un GIF, que no tiene interés para lo que estoy explicando.

Ahora voy a forzar un error 404 para poder verlo.

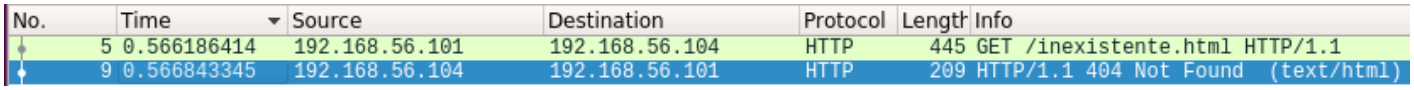


Figura 4: Intercambio de mensajes para forzar un error 404

Y ahora me centro en el propio mensaje de error:

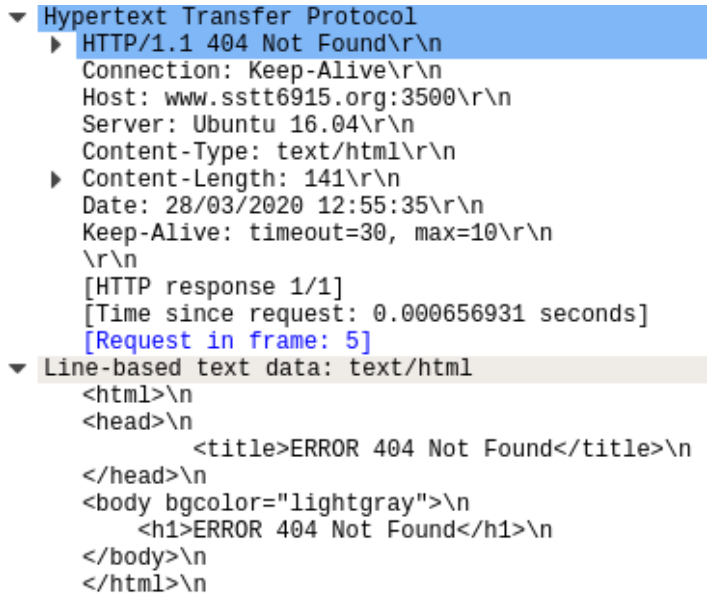


Figura 5: Mensaje de error 404

Seguimos observando la correción de la cabecera. El Content-Length indicando el tamaño del HTML que creé para

este caso, y que puede verse en el último campo del mensaje, el cuerpo.

Para acabar, voy a usar Telnet para pedir un recurso que esté fuera del directorio del servidor. Esto forzará un error 403 Forbidden, así como el cierre de la conexión TCP. Vamos a verlo:

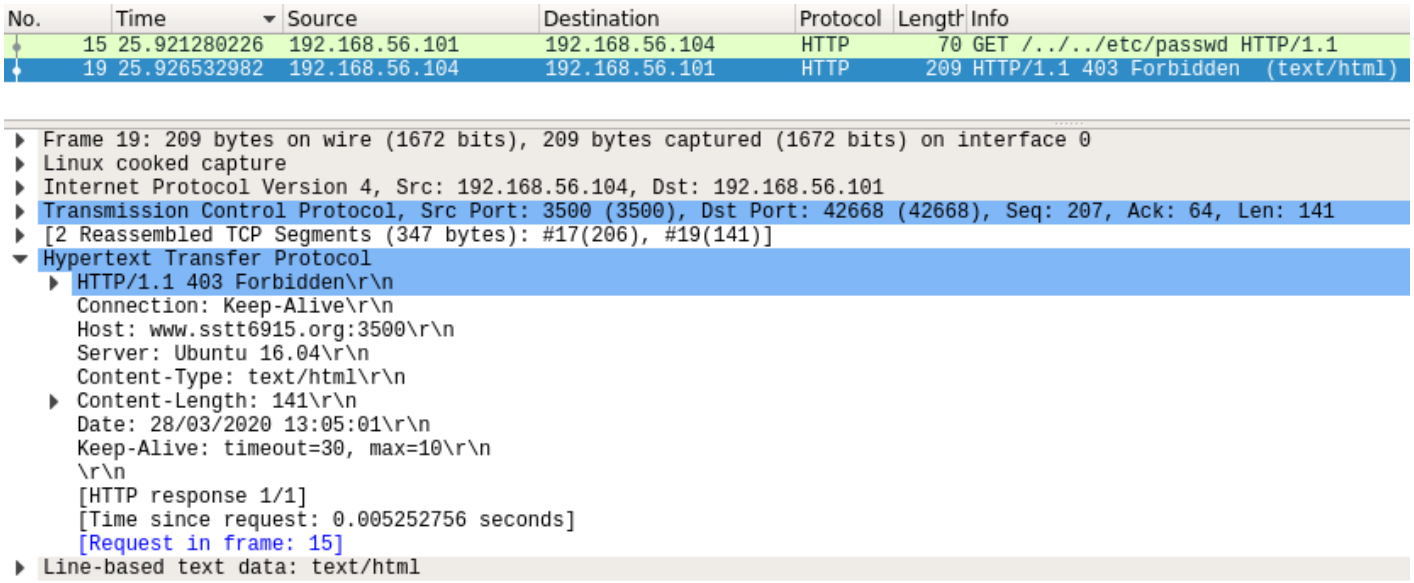


Figura 6: Intercambio HTTP ante una petición de un recurso fuera del servidor

Observamos como efectivamente nos llega el error 303 y podemos ver la cabecera en esta misma figura. Falta ver como se cierra la conexión:

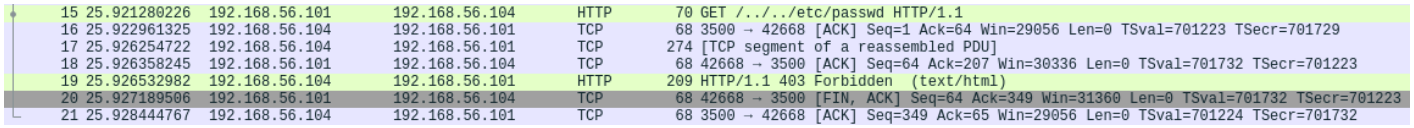


Figura 7: Fin de la conexión tras un error 403 Forbidden

Donde observamos como cuando nos llega el error 403, nos llega también el cierre de la conexión TCP.

Así, hemos visto trazas de momentos representativos por los que pasa nuestro servicio a lo largo de una conexión TCP e intercambio HTTP, desde el establecimiento hasta la finalización, comprobando la corección de las cabeceras y, en general, de los mensajes enviados.

3. Problemas encontrados en el proceso de desarrollo del escenario

Son varias las dificultades a las que me he enfrentado al abordar esta práctica, considero que la mayoría debidas a la novedad del trabajo planteado.

- **Comprender el trabajo del servidor:** al principio me encontraba un poco perdido, pues sentía que programaba sin un objetivo final claro. Hasta que no me esforcé en entender bien el protocolo HTTP me costó ver cómo debía proceder de forma global.
- **Manejo de buffers, lectura y escritura de mensajes:** esta ha sido la parte que más quebraderos de cabeza me ha producido. Me he apoyado en diversas librerias de C, como *string.h* o *regex.h*, pero aún así he tenido dificultades. Hay muchas comprobaciones a tener en cuenta a la hora de procesar una petición, y otras tantas en las que hay que tener cuidado al generar una respuesta. En ocasiones, sentía que solo estaba poniendo un parche encima de otro para ir arreglando problemas de funcionamiento, finalmente esto me obligó a redefinir desde cero algunas funciones y a crear algunas otras. Al menos me he familiarizado bastante bien con el manejo de strings y buffers en C en todo este proceso.
- **La persistencia:** no me ha producido excesivos problemas, aunque cuando la implementé, al principio, no funcionaba. El fallo no estaba en la implementación de esta, sino en que aún no había programado correctamente el envío de las cabeceras de respuesta. Una vez hice esto, funcionaba correctamente.
- **Uso de variables innecesarias:** a lo largo del desarrollo, he usado distintas variables que consideraba útiles en un principio, pero al avanzar en el proyecto dejaban de serlo y pasaban a ser ineficiencias del programa. Supongo que esto se debe a que debo planificar más a largo plazo mi programa en lugar de 'ir haciendo lo que toca'.

Estos son los principales problemas que he tenido con este proyecto. Considero que los he conseguido solventar todos satisfactoriamente (aunque es probable que aún haya ineficiencias en mi programa) y me han servido para mejorar como programador, y para entender mejor el protocolo HTTP.

4. Tiempo dedicado a la práctica

Tiempo dedicado a la Entrega Anticipada	
Programación y testeo	Memoria
14 horas de trabajo autónomo 6 horas de trabajo en clase	4 horas de trabajo autónomo

Cuadro 10: Desglose del tiempo de trabajo

5. Breve conclusión

Este trabajo está siendo bastante costoso, pero también resulta muy didáctico. He comprendido mucho mejor los conceptos dados en teoría, ya que he tenido que aplicarlos en un escenario, aunque pequeño, cercano a la realidad. En un principio era frustrante, como he indicado en los problemas que he tenido, no ver un objetivo general de lo que hacía. Pero una vez hecho el esfuerzo de arreglar esto, todo fue mucho mejor.

Lo principal que he aprendido es que antes de programar una funcionalidad, es conveniente pensar cómo esta se relacionará con las siguientes a implementar. De esta forma podemos evitar luego tener que andar rehaciendo código y parcheando ineficiencias. No siempre nos centramos en el diseño de lo que queremos hacer y pasamos muy rápido a hacerlo, cuando es más eficaz la otra forma de proceder. Aunque esto lo sabemos, la mayoría de proyectos de la carrera, al menos hasta el momento, no presentan una complejidad suficiente para requerir demasiada planificación. Considero que este sí.

Y ya termino, me he divertido haciendo esta práctica, si bien es cierto que en ocasiones he sentido un gran desasosiego. Cambiar una línea de código y que, de pronto, nada funcione es una sensación desagradable. Otra lección importante: hacer copias de seguridad del trabajo de una forma más o menos sistemática.

Última lección: cuando estoy mucho tiempo en algo que no consigo solucionar, me viene bien dejarlo y continuar en otro momento. A ser posible, al día siguiente: el problema sigue fresco en mi memoria, pero ya no estoy frustrado.