Memoria Práctica 3

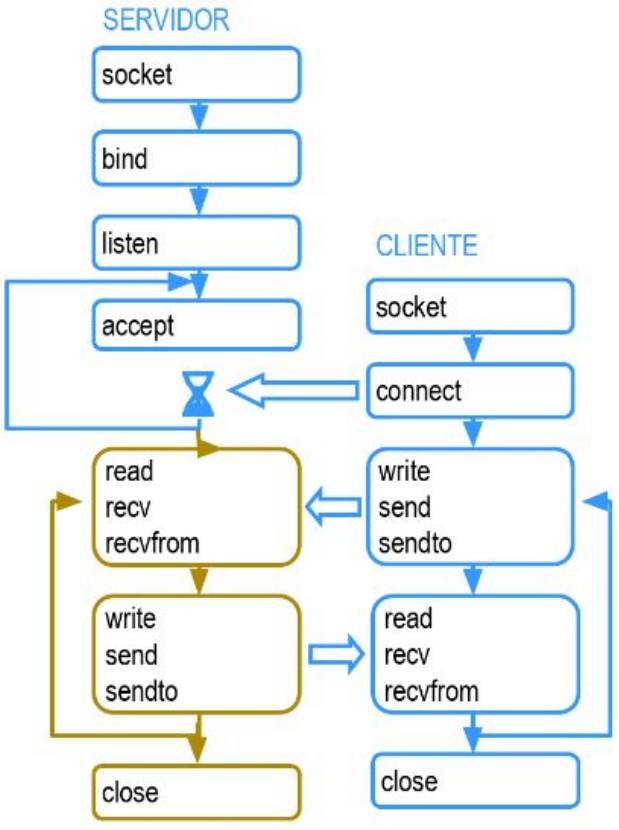
SERVICIOS DE RED

ESPECIFICACIONES

- Componentes del grupo: Rubén Rubio Martínez, Alejandro Castro Valero y Jesús Cuadra Tellez.
- Asignatura: Sistemas Distribuidos.
- Material: PC, máquina virtual VirtualBox, distribución Linux/Ubuntu, Compilador C, editor de texto Sublime Text, herramienta software Postman.
- Objetivos: Implementar un cliente (mi_wget) que sea capaz de solicitar recursos mediante peticiones HTTP a cualquier servidor, y un servidor (mi_httpd) que trate dicho tipo de peticiones de cualquier cliente y devuelva respuestas HTTP en función de sus recursos disponibles, usando el lenguaje de programación C.

DESARROLLO

CLIENTE



Para crear un cliente como podemos ver en el dibujo necesitamos un **socket** , un **connect**, un **write** y un **read**.

En primer lugar para realizar esta práctica realizamos un cliente web nosotros llamada main.c que únicamente podía realizar llamadas **GET**,
Este servidor crea un **socket**: **sockfd = socket(AF_INET, SOCK_STREAM, 0);** pasando por parámetro **AF_INET**, que significa que las peticiones que recibirá podrán ser de clientes que corren en el mismo ordenador o en otro distinto, y el parámetro **SOCK_STREAM**, que indica que el socket es orientado a conexión.

Hacemos un connect solicitando la IP, conexión en un puerto y lo comprobamos de esta forma **if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <0).**

```
100     if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <0){
101         pexit("coonnect() fallido");
102     }
```

A continuación realizamos el **write**. Al principio como el servidor solo podía hacer GET no comprobamos la llamada y directamente hacemos write de esta forma **write(sockfd, HTTP_request, (size_t)strlen(HTTP_request))**;

El http_request es un define en donde nosotros hemos introducido la llamada que queremos hacer. En este caso la llamada es la siguiente :
"GET %s HTTP/1.1\r\nHost: %s\r\nUser-Agent: %s\r\nConnection: Keep-alive\r\n\r\n"
Como podemos observar, el cliente pide un GET al servidor con el protocolo HTTP/1.1 y el host y user están sin definir para que se queden los valores por defecto.

Una vez hemos realizado la llamada al servidor solo nos queda realizar un **read** para obtener la respuesta:
while ((bLeidos = (int)read(sockfd, HTTP_response, sizeof(HTTP_response))) >0) {
write(STDOUT_FILENO, HTTP_response, bLeidos);}
Para ello realizamos un bucle para introducir toda la página web que recibimos en la variable HTTP_response.

Finalmente lo único que nos queda es **cerrar** el canal **close(sockfd);**
En el momento que el servidor realizaba correctamente las peticiones get que mandábamos por el cliente decidimos mejorar el cliente para que pudiera realizar otro tipo de llamadas.

Para ello introducimos al cliente la opción de hacer peticiones HEAD, DELETE y PUT.

Para lograrlo lo que hicimos fue introducir un fichero más que se leería desde el terminal de esta manera: **/mi_wget HTTP localhost 8080 /index.html HEAD**
Al ejecutar el cliente de esta forma nos permite desde el cliente hacer la comprobación y dependiendo de la llamada, hacer al servidor una petición con una cabeza o otra.

```
104     // WRITE: Realizamos la solicitud (http)
105     if(strcmp(dirIP,"localhost")==0 || puerto==8080){
106         //TODO se compara con strcmp
107         if(strcmp(accion,"GET")==0){
108 
109             sprintf(HTTP_request, HTTP_REQUEST_PATTERN_GET, recurso, dirIP, "generic server");
110             log(" * Enviando %d bytes:\n\n%s\n", (int)strlen(HTTP_request), HTTP_request);
111             write(sockfd, HTTP_request, (size_t)strlen(HTTP_request));
```

Finalmente para llevar el servidor a un buen funcionamiento utilizamos postman ya que es un cliente mucho más avanzado y nos permite visualizar una gran cantidad de opciones que nuestro cliente no es capaz.

SERVIDOR

El servidor es el encargado de tratar las peticiones HTTP que le envíe el cliente devolviendo o rechazando el recurso solicitado en función de las especificaciones de dicha petición. Para ello, lo que debe hacer el servidor es escuchar y conectarse con algún cliente.

Primero, debemos obtener el recurso a ofrecer. Para ello, usamos el segundo argumento que nos pasan al programa que corresponde con el fichero de configuración **mi_httpd.conf**, lo abrimos con permisos de lectura mediante la función **fopen(argv[2],"r");** y lo almacenamos en la variable **FILE** *archivoconf*. Posteriormente, invocamos la función **rellenarCamposConf(archivoconf,document_root,topeclientes,uri,puerto);** para dar valor a esas variables, con los valores de las directivas que contiene dicho archivo de configuración.

```
71      //Leemos el archivo .conf
72      if(argc>1){
73 
74          if(strcmp(argv[1],"-c")==0){
75              archivoconf = fopen(argv[2],"r"); //Leemos el archivo
76              if(archivoconf==NULL){
77                  printf("No existe el archivo .conf");
78              }else{
79                  rellenarCamposConf(archivoconf,document_root,topeclientes,uri,puerto);
80              }
81          }
82      }
```

Usando la función **strtok(NULL,"\n");** repetidas veces obtenemos los valores buscados de las directivas. Éstas son: *DocumentRoot*, donde especifica el directorio del recurso; *DirectoryIndex*, correspondiente al nombre del recurso; *MaxClients*, número máximo de clientes que puede atender de forma concurrente; y *Listen*, que especifica el puerto por donde establecerá conexión.

```
219     //Bucle hasta final de archivo
220     while(read=getline(&linea,&length,archivoconf)!=-1){
221         char *aux;
222
223         aux = strtok(linea," ");
224         if(strcmp(aux,"DocumentRoot")==0){
225             aux = strtok(NULL,"\n");
226             strcpy(document_root,aux);
```

Por último, cerramos al archivo **mi_httpd.conf**, con la función **fclose(archivoconf)**.

```
239     fclose(archivoconf);
240     if(linea){
241         free(linea);
242     }
```

Una vez tenemos el recurso, ejecutamos el método **LanzarServicio();** para conectarnos con el cliente.

En primer lugar, abrimos un socket mediante la función **socket(AF_INET, SOCK_STREAM, 0);** pasando por parámetro **AF_INET**, para clientes independientemente del ordenador donde se sitúen, y el parámetro **SOCK_STREAM**, que indica que el socket es orientado a conexión, como se mencionó anteriormente. Devuelve un descriptor de archivo normal. Después, necesitamos alertar al sistema operativo para que asocie el socket que hemos abierto, a nuestro programa, y para ello usamos la función **bind(sfdServidor, (struct sockaddr *)&addr, sizeof(addr));**. Ahora, con la función **listen(sfdServidor, topeclientes);** el sistema empieza a atender la conexión de red especificada, para cualquier cliente, y estableciendo un sistema de colas en el caso de que el número clientes enviando peticiones sea superior al tamaño especificado en la directiva *MaxClients*.

```
448     // SOCKET: Abrimos un socket donde esperar peticiones de servicio
449     log("SOCKET: abrmos un socket...");
450     sfdServidor = socket(AF_INET, SOCK_STREAM, 0);
451     if(sfdServidor==-1){
452         fprintf(stderr, "ERROR [%s] abriendo socket\n",strerror(errno));
453         return -1;
454     }
```

```
461     error = bind(sfdServidor, (struct sockaddr *)&addr, sizeof(addr));
462     if(error < 0){
463         fprintf(stderr, "ERROR [%s] en bind\n",strerror(errno));
464         close(sfdServidor);
465         return -1;
466     }
```

```
467     // LISTEN: Nos ponemos a la escucha en el socket, indicando cuántas atendemos a la vez
468     log("LISTEN: indicamos que atienda hasta %d conexiones a la vez", topeclientes);
469     error = listen(sfdServidor, topeclientes);
470     if(error < 0){
471         fprintf(stderr, "ERROR [%s] en listen\n",strerror(errno));
472         close(sfdServidor);
473         return -1;
474     }
```

Justo después, entramos en un bucle infinito donde esperamos y tratamos peticiones.
Ejecutamos el método **ConectarConCliente(sfdServidor);** donde creamos un proceso hijo a partir del proceso padre con un **fork()**, pasando el socket o descriptor de archivo del proceso padre, y obtenemos el descriptor del cliente. Para ello se hace una llamada a la función **accept(sfdServidor, (struct sockaddr *)&cliente, &tamCliente);** que le indica al sistema operativo que nos pase al siguiente cliente de la cola. Si no hay clientes, el programa se bloqueará hasta que reciba uno.

```
485     // ACCEPT: Esperamos hasta que llegue una comunicación, para la que nos dan un nuevo socket
486     log("ACCEPT: esperando nueva conexion...");
487     sfdCliente = accept(sfdServidor, (struct sockaddr *)&cliente, &tamCliente);
```

Después hacemos una llamada a la función **gethostbyaddr((char *)&cliente.sin_addr, sizeof(cliente.sin_addr),cliente.sin_family);** que utiliza el protocolo DNS para traducir la dirección IP pasada por parámetro, al nombre del host.

```
489     log("Comprobando conexion entrante (resolucion inversa de la IP)");
490     host = gethostbyaddr((char *)&cliente.sin_addr, sizeof(cliente.sin_addr),cliente.sin_family);
491     log("Conexion aceptada a [%s] %s: %u",host->h_name, inet_ntoa(cliente.sin_addr),cliente.sin_port);
```

Pero únicamente la usamos para imprimirlo por pantalla, por cuestiones de comprensión, lo que realmente nos interesa es el descriptor devuelto por la función. Mientras que el proceso padre se queda esperando nuevas conexiones a otros clientes que solicitan sus recursos, el proceso hijo se encargará de atender aquel que se encuentren primero en la cola.

Una vez hemos completado todas las conexiones entre el cliente y el servidor procedemos a leer la petición que nos envía el cliente, esto lo conseguimos gracias a la función **read(sd, buffer, sizeof(buffer)-1);**

```
262     // HTTP REQUEST: leemos la cabecera (debe ser siempre menor que MAX_REQUEST_SIZE)
263     bLeidos = read(sd, buffer, sizeof(buffer)-1);
```

Esta función nos devuelve la cantidad de bytes leídos en la petición, si el número leído es menor a 0 entonces nos encontraremos con un error de tipo **"HTTP/1.1 500 Internal Server Error"**.

```
283     //No ha leido nada
284     if(bLeidos < 0){
285         error500(date,respuesta); // muestra el error
286         n = strlen(respuesta); //calcula el numero de caracteres
287         enviados = write(sd, respuesta, n); //escribimos la respuesta
288     }
```

Si por el contrario recibimos una cabecera válida nos dispondremos a leerla y tratarla de la siguiente manera:

1- La primera línea siempre estará compuesta por el método a realizar, seguida del recurso a buscar, y por último, el protocolo utilizado **"GET /index.html HTTP/1.1"**.

```
293     // Tratar la solicitud
294     for(i=0; i<bLeidos-3;i++){
295         if(buffer[i]=='\r' && buffer[i+1]=='\n' && buffer[i+2]=='\r' && buffer[i+3]=='\n'){
296             tambody = bLeidos-(i+4);
297             memcpy(body,&buffer[i+4],tambody);
298             //break;
299         }
300     }
```

2- Gracias a la función **strtok** dividiremos en 3 variables independientes dicha cabecera para más tarde poder analizarla.

```
304     metodo = strtok(buffer, " ");
305     uri = strtok(NULL," ");
306     protocolo = strtok(NULL, "\n\r");
```

Error 404 File Not Found

El **error 404** es uno de los más comunes a la hora de hacer las peticiones HTTP, para encontrar el archivo solicitado realizamos una concatenación entre el directorio raíz leído desde el archivo.conf y la uri introducida por el cliente, de la siguiente manera **strcat(document_root,uri);** esta será la ruta que le pasaremos a la función **fopen** para abrir el archivo **archivo = fopen(document_root,"r");** si la función nos

devuelve un NULL es que no se ha podido encontrar dicho archivo y devolveremos un error de tipo 404 File Not Found.

```
318         strcat(document_root,uri);
319         archivo = fopen(document_root,"r");
320         if(strcmp(protocolo,"HTTP/1.1")==0){
321
322
323             //No encontramos el archivo
324             if(archivo==NULL){
325                 error404(date,respuesta);
```

Error 505 HTTP version not supported

Este error lo detectamos mediante la comparación del protocolo que nos ha enviado el usuario, nuestro servidor solo soporta la versión HTTP/1.1 con lo cual todo protocolo distinto a ese devolverá el error **“505 HTTP version not supported”**.

```
320         if(strcmp(protocolo,"HTTP/1.1")==0){
```

```
361         }else{
362             //Version no soportada
363             error505(date,respuesta);
364         }
```

Método “GET”

El **método GET** consiste en devolver el body del archivo solicitado en la url acompañado de un HEAD con una información relevante al mismo. Como hemos mencionado anteriormente si no se consigue abrir el archivo se devolverá un error 404 File Not Found, de ser al contrario devolverá un 200 OK junto al HEAD y Body mencionados.

```
342         if(strcmp(metodo,"GET")==0){
343             //Es un GET
```

Método “HEAD”

El **método HEAD** es sin duda el más simple de todos ya que solo nos devuelve el HEAD referente al archivo solicitado por parámetro, sin el body asociado. Al igual que el resto en caso de no encontrarlo devolveremos un error 404 File Not Found.

```
362         }else if(strcmp(metodo,"HEAD")==0){
363             //Es un HEAD
```

Método “DELETE”

El **método DELETE** como el propio nombre dice se trata de eliminar el archivo que ha solicitado el cliente, para ello nos ayudaremos de la función **aux = remove(document_root);** que nos devolverá un -1 en el caso de que no se encuentre el archivo y devolviendo su error 404 File Not Found correspondiente, en caso contrario devolveremos un mensaje 200 OK acompañado de un HEAD con información y opcionalmente podremos adjuntar un body.

```
382         }else if(strcmp(metodo,"DELETE")==0){
383             //Es un DELETE
```

Método “PUT”

El **método PUT** nos basamos en el directorio solicitado por el cliente en modo escritura, si el archivo no tiene concedidos los permisos de escritura debemos devolver un error 403 Forbidden. Por otro, si no encontramos el archivo solicitado procederemos a crearlo y devolveremos la respuesta **“201 CREATED”** acompañada de un HEAD y de un body de manera opcional.

```
402         }else if(strcmp(metodo,"PUT")==0){
403             strcat(document_root,uri);
```

Error 405 Method Not Allowed

Como el propio nombre indica este error se trata de que el método solicitado no lo trata nuestro servidor, es decir, que el método es diferente a “GET”, “HEAD”, “DELETE” y “PUT”, si esto ocurre devolveremos el mensaje **“405 Method not allowed”** acompañado de un HEAD y un Body opcional.

```
428         }else{
429             error405(date,respuesta);
430         }
```