Práctica 1: Servidor Web

Introducción

En la actualidad el tráfico Web ha cobrado gran importancia debido a que la mayor parte de los servicios usados hoy en día pueden ser accedidos a través de una página Web (Gmail, Youtube, Facebook, Google, etc..).

Para acceder a estas páginas Web se usan navegadores que se conectan a servidores mediante el protocolo HTTP (Hyper Text Transfer Protocol) . En esta práctica se desarrollará un servidor Web sencillo que sea capaz de comunicarse con navegadores estándar (Mozilla, Internet Explorer, Google Chrome, wget etc) y servir tanto contenidos sencillos (archivos de texto o páginas web sencillas) como contenidos más complejos (imágenes o programas).

Tabla de Contenidos de la práctica

FUNDAMENTOS DEL PROTOCOLO HTTP

El protocolo HTTP [RFC 2616] funciona sobre TCP en el puerto 80 haciendo uso de texto plano para la comunicación. Este protocolo está basado en el modelo petición-respuesta. Cuando un navegador quiere acceder a un **recurso** localizado en un servidor, construye y envía una petición al mismo. El servidor comprueba el recurso solicitado y si le pertenece construye una respuesta al cliente que contiene tanto información referente al recurso como el recurso en sí.

Por ejemplo cuando realizamos una petición a www.google.esnuestro navegador genera una petición parecida a la siguiente:

```
GET / HTTP/1.1

Host: www.google.es

Connection: keep-alive

Accept:application/xml,application/xhtml+xml,text/html;q=0.
9,text/plain;q=0.8,image/png,*/*;q=0.5
```

```
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US)
AppleWebKit/534.7 (KHTML, like Gecko) Chrome/7.0.517.44
Safari/534.7

Accept-Encoding: gzip,deflate,sdch

Accept-Language: es-ES,es;q=0.8

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3

Cookie: NID=41=uBDgNAUYK-
J0rl6WIVYr9q4urtJ6U7BEEvH8EMIZ4mGA7BYZfC1PFaspJbYwHoN_6U71a
AGRdcdfhixCBjPuDUa1-
4ST9oQkP0Qjfz_Kl1GeiD8wkGFf20PeTvjcxeW3;
```

Esta petición es procesada por el servidor que nos envía una respuesta parecida a la siguiente:

```
HTTP/1.1 200 OK

Date: Tue, 14 Dec 2010 15:46:06 GMT

Expires: -1

Cache-Control: private, max-age=0

Content-Type: text/html; charset=UTF-8

Set-Cookie:
PREF=ID=024ab96d378d5a4b:U=71882ebd72f1d8f8:FF=0:LD=es:NR=1
0:TM=1277722809:LM=1292341566:SG=1:S=wLPiV-268UMeOOun;
expires=Thu, 13-Dec-2012 15:46:06 GMT; path=/;
domain=.google.es

Content-Encoding: gzip

Server: gws

Content-Length: 8360

X-XSS-Protection: 1; mode=block

<html>…….</html>
```

A continuación se detallarán algunos campos básicos en el protocolo HTTP.

Toda petición HTTP sigue la siguiente estructura:

```
MÉTODO RECURSO VERSIÓN\r\n
```

Opcion1\r\n

Opcion2\r\n

.

.

.

OpcionN\r\n

\r\n

Data

En HTTP existen dos métodos principales para realizar peticiones: GET y POST.

MÉTODO GET

Este método es el usado comúnmente cuando se quiere obtener toda la información referente a un recurso que tiene un servidor.

MÉTODO POST

Este método es usado para indicar al servidor que debe realizar acciones asociadas en ocasiones a datos contenidos en la petición. Este método es usado comúnmente para :

-Subir mensajes a listas de correo, foros, boletines, etc

-Proporcionar bloques de datos que son el resultado del envío formularios a un proceso que los trata.

-Extender una base de datos añadiendo datos

Adicionalmente existen otro métodos como por ejemplo: HEAD, PUT, DELETE, TRACE o CONNECT que no entran en el ámbito de esta práctica.

RECURSOS

Los recursos de un servidor se especifican mediante un URI (Uniform Resource Identifier). Un URI es una cadena de texto que indica la ruta de un recurso dentro de Internet. Por ejemplo http://example.org/absolute/URI/with/absolute/path/to/resource.txt

Existen URI's absolutas y relativas. En el caso del protocolo HTTP se suelen usar rutas relativas al directorio de trabajo del servidor. Por ejemplo:

/recurso.txt

recurso.html

/

El URI / es un caso especial que en la mayoría de los servidores Web se suele interpretar como la página Web por defecto de ese servidor.

VERSIÓN

Existen diferentes versiones del protocolo HTTP que son producto de la evolución del mismo para adaptarse a las necesidades y redes que han ido surgiendo. Los valores permitidos son HTTP/1.0 y HTTP/1.1 siendo esta última la más usada actualmente.

RESPUESTAS HTTP

Toda respuesta HTTP sigue la siguiente estructura:

VERSIÓN CÓDIGO_ESTADO DESCRIPCIÓN_CÓDIGO\r\n

Opción1\r\n

Opción2\r\n

.

.

.

OpciónN\r\n

\r\n

Data

A continuación se muestra una tabla con los códigos más usados y sus descripciones:

Codigo de estado	Descripción
200	OK
304	Not Modified
404	Not Found
403	Forbidden
500	Internal Error

OPCIONES

Existen un gran número de opciones que pueden acompañar una petición o respuesta HTTP. A continuación se detallarán las más importantes para la realización de la

práctica. Para más información acerca las mismas se recomienda la lectura de la RFC 2626.

Content-Length:

Esta opción indica la longitud en bytes de los datos adjuntos tras la cabecera HTTP.

Content-Type:

Esta opción indica el tipo MIME de los datos adjuntos tras la cabecera HTTP. Algunos valores típicos text/html, image/gif, image/jpeg, audio/mpeg o application/octet-stream. El tipo de MIME de un archivo pueden determinarse a partir de su extensión consultando el fichero /etc/mime.types que se encuentra en la mayor parte de las distribuciones GNU/Linux.

Server:

Esta opción aporta información acerca del software que actúa como servidor y los subproductos del mismo.

Host:

Esta opción indica la dirección IP o nombre de host del servidor al que se realiza una petición.

APIs

SOCKETS STREAM (TCP)

Las primitivas básicas de la API para usarlas en el servidor son:

Crear socket:

int socket(int *socket_family*, **int** *socket_type*, **int** *protocol*);

Ej: sock = socket(AF_INET, SOCK_STREAM, 0);

Enlazar puerto y socket:

int bind(int *sockfd*, **const struct sockaddr** **my_addr*, **socklen_t** *addrlen*);

Ej: bind(sock,(struct sockaddr *)&sin,sizeof(sin));

Habilitar socket para recibir conexiones:

int listen(int *sockfd*, **int** *backlog*);

Ej: listen(socket, max_conection);

Aceptar conexiones:

int accept(int *sockfd*, **struct sockaddr** **addr*, **socklen_t** **addrlen*);

Ej: sock_aux=accept(sock,(struct sockaddr*)&pin,sizeof(struct sockaddr*));

Enviar:

ssize_t send(int *s*, **const void** **buf*, **size_t** *len*, **int** *flags*);

Ej: c = send(sock,data, strlen(data), 0);

Recibir:
ssize_t recv(int *s*, **void** **buf*, **size_t** *len*, **int** *flags*);

Ej: c = recv(sock, buffer, sizeof(buffer), 0);

Cerrar socket:
int close(int fd);

Ej: close(sock);

Para más información, consultad el manual de las funciones.

Se pueden encontrar multitud de manuales del uso de sockets. Por ejemplo, se puede consultar un ejemplo de uso de sockets TCP como servidor.

EJERCICIOS

Ejercicio 1)

Implementar la funcionalidad básica de un servidor Web. El servidor debe ser capaz de servir archivos simples solicitados mediante una petición de tipo GET. El servidor debe leer la configuración inicial al arrancar de un fichero de configuración llamado **ARedes1.txt** que se localizará en el mismo directorio del ejecutable y que contendrá las siguientes variables separadas por un espacio en blanco en una única línea: **(es obligatorio usar este formato, NO SE ACEPTARÁN OTROS DE NINGUNA MANERA).**

- Puerto_de_escucha_del_servidor Directorio_donde_están_guardados_los_recursos Recurso_por_defecto

(esto es, en una sola línea, separados los campos por "blancos" y en estricto orden).

Un servidor web debería ser capaz de procesar peticiones de manera concurrente. Esto, cada vez que nos llegara una petición, debería crear un hilo para que la procesara, mientras el hilo principal se mantendría a la espera de nuevas peticiones. Sin embargo, por motivos de simplicidad, vamos a procesar las peticiones una a una, en serie.

Para empezar:

Descargarse el programa de ejemplo de servidor usando sockets TCP y entender cómo funciona. Para ello, compilarlo con el makefile que también se proporciona (o mediante el comando gcc -o ejecutable fichero.c) y ejecutarlo (desde la consola con el comando ./ejecutable) Con el servidor arrancado, acceder desde el navegador Firefox a http://localhost:2000/

Veréis que en el navegador sale un mensaje "XXX!" y en la consola donde está ejecutándose nuestro servidor saldrá impresa la petición HTTP que ha hecho el navegador.

¿Qué modificaciones tenemos que hacer al programa de ejemplo para implementar nuestro servidor web?

1. Antes de hacer la apertura del socket, debemos leer de un fichero de configuración los 3 parámetros que se piden.

2. Abrir el puerto que hemos leído del fichero de configuración, en vez del 2000 como está ahora.

3. Modificar la función process_request, para que procese la petición HTTP que nos va a llegar. Esto es, comprobar que es una petición GET, extraer el nombre del recurso y comprobar la versión. Si la petición es GET y la versión es la correcta, comprobamos la existencia del fichero que nos han solicitado.

4. En caso de que exista el recurso, construimos la respuesta "200 OK". Añadimos las opciones de la cabecera Content-type (por ahora valdrá text/html) y Content-Length (para saber cuánto ocupa un fichero podemos usar la función stat http://linux.die.net/man/2/stat, para cualquier duda sobre el uso de esta función o de otra, consultar la página del manual y preguntar al profesor). Después, enviamos el recurso (se lee del fichero local y se escribe en el socket).

5. En caso de que no exista el recurso, construimos la respuesta "404 Not Found". En este caso, no hacen falta más opciones ni datos.

Ejercicio 2)

Incrementar la funcionalidad del servidor haciendo que sirva todo tipo de contenidos. Para ello, se debe hacer uso del campo Content-Type. Para obtener el tipo MIME adecuado para cada petición, se mirará el fichero mime.types. En cado de que la extensión del recurso solicitado mediante el método GET no se encuentre en el fichero mime.types se debe contestar con un tipo MIME application/octet-stream.

El fichero de mime.types que vamos a usar como referencia es el de Apache.

CRITERIOS DE EVALUACIÓN

1. Fichero de configuración sigue las pautas solicitadas (nombre y formato). 0.5 puntos.
2. Las funciones usadas (especialmente, aquellas facilitadas) llevan cabecera. 0.5 puntos.
3. Sirve contenidos web sencillos (html sin imágenes) formando correctamente las cabeceras. 1 punto.
4. Sirve el contenido relativo al recurso por defecto (y cabecera correcta). 1 punto.
5. Devuelve cabecera de tipo 404 oportunamente y correctamente formada. 1 punto.
6. Sirve contenidos web avanzados (ejemplo, GIF). 2 puntos.
7. Control de errores ante fallos en las extensiones. 2 puntos.
8. Sirve contenidos de tipo PDF de tamaño superior a 1 GB. 2 puntos.

Si el punto 1 no es correcto no se evaluará el resto.
Si el punto 2 no es correcto no se evaluará el resto.
Todas las pruebas y la validación ser harán en Chrome y Firefox, es imprescindible que funcione correctamente en ambos navegadores.