Universidad del Chubut

TP1 - Desarrollo de un Servidor Web

Informe Decisiones de Diseño

Alumnos:

Macarena Belén Garcia Arcija
Mauro Leiva
Lucas Isaac Soto

Profesor: Ing. Fabio Gabriel Salerno.
Cátedra: Redes y Seguridad Informática.
Carrera: Tecnicatura Universitaria en Desarrollo de Software.
Fecha de entrega: 21/04/2023.

Índice

Objetivo del trabajo práctico.

En este proyecto se nos pidió desarrollar un servidor web simplificado. Dicho servidor web debe respetar la versión **1.0** del protocolo **HTTP** y seguir la **RFC 1945**, la cual describe el funcionamiento de esta versión del protocolo.

El servidor tiene que aceptar peticiones de tipo **GET** o **HEAD**, en caso de una petición diferente a las anteriormente mencionadas el servidor envía un mensaje (**501 Not Implemented**). También implementa los siguientes códigos de estado:

- **200 "OK".**
- **400 "Bad Request".**
- **404 "Not Found".**
- **501 "Not Implemented".**

Todas las peticiones del cliente y respuestas del servidor se imprimen en la consola del servidor para llevar un registro visual de su funcionamiento.

A su vez, tiene que ser capaz de procesar los siguientes tipos **MIME**:

Extensión objeto	**Tipo MIME**
.htm .html	text/html
.txt	text/plain
.gif	image/gif
.jpg .jpeg	image/jpeg
.pdf	application/pdf
*	aplication/octet-stream

Nota: el tipo **MIME "aplication/octet-stream"** se utiliza para devolver cualquier otro tipo de extensión que no ha sido contemplada en el resto de tipos **MIME**.

Para el desarrollo del servidor se utilizó como referencia el proyecto "**passeidireto - httpServer - Redes de computadores**".

Modificaciones al proyecto referenciado.

Las diferencias con el proyecto que utilizamos como referencia fueron bastantes.
Una modificación importante a no dejar de lado, es que el código estaba en partes en portugues, nosotros modificamos este para que estuviera en español.

Otra de ellas es que el proyecto original la concurrencia se encontraba realizada con hilos, algo que tuvimos que modificar para realizar la concurrencia con **fork()** y el servicio con la funcion **excev()** como se especificaba en los requerimientos del proyecto.

Al utilizar **fork()** debimos manejar los procesos hijos cuando finalicen. De la mano de dicha modificación, tuvimos que agregar una función para estar atento a la finalización de un hijo y esta realice el **waitpid()**.

Para utilizar **excev()** para realizar el servicio tuvimos que transformar el número del **file descriptor** del cliente a una **string**, y pasarsela por **parámetros**. Posteriormente en el servicio esté **file descriptor** que está como cadena se transforma a entero para poder utilizarlo.

Del lado del servicio que se encarga de recibir la petición del cliente y devolver la respuesta con el recurso asociado también tuvimos que hacer varios cambios.

Uno de estos cambios fue modificar la función **get_mime_type()**, ya que esta, por defecto, en caso de que la extensión sea desconocida devolvia **NULL**. En nuestro caso, debimos adaptarlo para que devuelva **"application/octet-stream"**, como especificaba el trabajo práctico.

A la hora de enviar la cabecera de respuesta al cliente, también nos interesó imprimir por consola del servidor la cabecera enviada, así que este fue otro cambio que le hicimos al proyecto original.

Otro punto importante es que necesitábamos cerrar la conexión a penas se termine de enviar el archivo cosa que el proyecto original no hacía.

Un cambio grande fue el hecho de que el proyecto original armaba en código el html en caso de alguna respuesta de error del servidor, nosotros creamos archivos **html** que el servidor pueda ubicar y devolver al cliente sea el error correspondiente, de esta forma nos desligamos de tener que armar el **html** dentro del error.

Algo que hacía el proyecto original era verificar si lo que se le envió era un archivo o directorio. Si se trataba de un directorio, verificaba si se encontraba bien escrito, para luego agregar el **index.html** por defecto al final. Nuestro equipo le quito esto por cuestiones de practicidad del trabajo, por lo cual en caso de que el cliente pida cualquier otra cosa que no sea un archivo este devolverá el código de error **404 "Not Found",** indicando que dicho recurso no pudo ser localizado. También agregamos la verificación de la correcta escritura de la petición del cliente, para que en caso caso contrario el servidor devuelva el código de error **400 "Bad Request".**

Otro punto a tener en cuenta es que nuestro trabajo atiende los requerimientos cuyos métodos sean **HEAD** y **GET**, a diferencia del original que solo funciona con **GET**, así que agregamos esta funcionalidad también.

También modificamos la forma de leer y almacenar la petición de un cliente, el original utilizaba **strtok()**, lo cambiamos por **sscanf()**. Eliminamos la estructura **HOSTS** ya que nosotros no le damos uso a esta. En el caso de la estructura **CR_returns** agregamos que almacene el mensaje correspondiente al código que devolverá.

Una modificación bastante grande del archivo original fue modularizarlo en lugar de mantenerlo monolítico, ya que este estaba escrito todo en un solo código fuente. Esto permite la escalabilidad del proyecto en un futuro, además de una lectura más limpia y amigable del código.

El proyecto original mostraba por la consola del servidor cadenas que no nos interesaban, además de un formato que no nos gustaba a estas las cambiamos.

Problemas encontrados.

- Al tomar como esqueleto el proyecto de otro tuvimos varios problemas, uno de ellos fue que al usar **excev()** debíamos pasar por parámetros el file descriptor del cliente, sino no podríamos enviarle mensajes.
- Al separar el código por partes nos encontramos con un problema de que estamos incluyendo más de 1 vez algunos archivos **header**.
- Varias veces al utilizar variables de tipo **char ***, no realizamos un **malloc** para almacenar cadenas, también no habíamos utilizado free para liberar el espacio pedido.
- Algunas veces cuando queriamos dividir las peticiones del cliente no las dividiamos correctamente, por lo que el servidor siempre respondía con error **400**.
- Tuvimos problemas con la función **stat()** para recuperar información del archivo, entonces no pudimos mostrar información del archivo como **lenght** entre otras.
- Hubo veces donde al aceptar un cliente, este se cerraba enseguida sin realizar el servicio al mismo. Fue por un problema del file descriptor pasado por parámetros a **excev().**
- Otro error reciente fue que armamos el path al archivo incorrectamente por eliminar líneas de código sin pensar en las consecuencias de dicha acción. Nos dimos cuenta enseguida y lo revertimos, pero dejando la línea que nos interesaba.

El código a grandes rasgos.

Nuestro código básicamente se encarga de la creación y configuración de un socket **TCP** en el puerto **8000** con dirección **localhost**.
Este atiende los clientes del puerto, a continuación crea un proceso hijo que lo atienda y vuelve a esperar nuevos clientes y así sucesivamente hasta finalizar el proceso servidor.
El proceso hijo se encarga de la tarea de leer la petición del cliente y darle una respuesta apropiada con el recurso que deba entregar.

Decisiones de diseño.

Tuvimos que tomar varias decisiones a la hora de hacer nuestro Trabajo práctico. Dividimos nuestro código fuente en 2, por un lado el **servidor.c** y por el otro el **servicio.c**. Esta decisión fue tomada en base a la responsabilidad que tendría cada archivo fuente, por una parte de la aplicación se concentra la atención de clientes(el **servidor.c**) y por otra se concentra el servicio que se le da al cliente **(servicio.c)**. Ya que estos dos no interactúan más que para pasar parámetros podemos dividirla sin problemas, al hacer esto logramos una mejor modularización y separación por responsabilidad.

Así mismo dividimos de cada archivo **servidor.c** y **servicio.c** el desarrollo de sus funciones en otros 2 archivos fuentes más el **server-function.c** y el **servicio-function.c**, los cuales tienen funciones correspondientes a lo que necesitaba el código del servidor y del servicio según correspondiera. Con esto logramos poder separar el programa servidor y cliente, de las funciones que estos necesitan, de esta manera pudimos hacer una abstracción extra, por lo cual los archivos servidor y servicios deben incluir **servidor-function.h** y **servicio-function.h** respectivamente, de esta forma el servidor sabe las funciones que puede utilizar pero no sabe cómo están hechas y de la misma forma el servicio. Por otro lado también hicimos los archivos **estructuras.h** y **includes-defines.h,** los cuales contienen las **estructuras** y las bibliotecas y constantes(**includes y define**) que utilizan cada uno, esto lo hicimos para no duplicar líneas de código en ambos archivos, tener separada la estructura del código y un código mucho más limpio en **servidor.c** y **servicio.c**

Finalmente creamos una carpeta dentro del proyecto llamada **recursos** donde tenemos los distintos archivos que el cliente nos puede solicitar. Algunos de ellos se encuentran en recursos/redes/index.html entre muchísimos más, esto para que se vea que el tipo **MIME** lo devuelve correctamente. También tenemos un archivo **Makefile** que ayuda a la compilación de la aplicación de una forma más transparente.

Funciones, Procedimientos y Estructuras de la aplicación

Funciones del servidor

int setServer(Host *servidor, int puerto)

Función que se encarga de crear y configurar un socket.

@Parametro *servidor: la estructura que guardará el servidor creado y configurado.
@Parametro puerto: el número de puerto donde el socket estará escuchando.

@Salida: devuelve 1 si hubo un error al setear el socket, retorna 3 si el error ocurrió al nombrarlo, un 4 si se trata de un error con el listen del socket, y finalmente un 0 si termina exitosamente.

int setSocket(int *sock)

Crea un socket de la familio **TCP** y lo guarda en lo apuntado por sock, controla los errores devolviendo 1 si fallo la creación.

@parametro *sock: hace referencia a donde se guardará el file descriptor de la función socket.

@Salida: devuelve 0 si se creó el socket exitosamente sino 1.

int setBind(Host *host)

Esta función se encarga de nombrar a un socket asignándole un puerto y dirección **IP.**

@Parametro *host: socket que se le asigna IP y PUERTO.

@Salida: si logra realizar el bind() retorna 0, si no retorna 1.

int setListen(Host host)

Asigna a un socket la cantidad de clientes pendientes que podrá encolar para atender.

@Parametro host: socket que se le asigna la cantidad de clientes pendientes.

@Salida: devuelve 0 si listen() se ejecutó exitosamente sino 1.

bool initAccept(Host *server, Host *client)

La función se encarga de esperar una petición de un cliente y establecer la conexión entre el cliente y el servidor.

@Parametro *servidor: estructura que guarda información del servidor, el cual escuchará peticiones.
@Parametro *cliente: estructura que guarda información del cliente, guardará el file descriptor del cliente.

@Salida: Si el cliente no puede conectarse al devuelve false, si no devuelve true.

void salidaError(char *msg)

Imprime el error indicando el nombre del servidor y el mensaje. Se indica que hubo un estado de salida fallido (la función terminó con alguna falla).

@Parametro *msj: Mensaje de error.

@Salida no tiene.

void sig_chld()

Atiende la señal **SIGCHLD** para completar la destrucción del proceso hijo y envía un mensaje por consola del servidor y se muestra su **PID**.

@Parametros no tiene

@Salida no tiene.

Funciones del servicio

Request readRequest(Host cliente, FILE *f)

Obtiene la línea de solicitud y la organiza en una estructura de tipo **Request** separando método, recurso y protocolo.

@Parametro cliente: contiene la estructura y el socket del cliente.
@Parametro *f: referencia a archivo donde obtendremos los datos para las variables método, recurso y protocolo.

@Salida Estructura Request con sus valores correspondientes cargados.

CR_returns checkRequest(Request peticion_cliente)

Analiza la petición del cliente y este arma la respuesta que se le debe devolver según las distintas condiciones que podemos tener con la petición del cliente.

@Paramtro peticion_cliente: utilizamos los datos guardados en ella para analizar si la petición está armada correctamente.

@Salida: devuelve la respuesta armada que debe devolverse a la petición del cliente.

void sendRequest(Host cliente, CR_returns respuesta, char metodo[10])

El procedimiento se encarga de obtener el estado del archivo/objeto requerido en la petición, abrirlo solo para lectura, llamar a la función **sendFile** para que lo envíe al cliente, y por último cerrar dicho archivo.

@Paramtro cliente: estructura del cliente donde usaremos el file descriptor para enviarle la respuesta.
@Paramtro respuesta: es la respuesta armada que se enviará al cliente.
@Paramtro metodo: método pedido por el cliente, según este la respuesta tendrá un comportamiento diferente.

@Salida: al tratarse de un procedimiento, no genera salida.

void sendFile(int cliente_socket, CR_returns respuesta, char metodo[10])

Determina el tamaño del archivo, crea la cabecera del header de respuesta y envía el archivo al cliente.

@Parametro cliente_socket: Socket del cliente a enviar el archivo.
@Parametro respuesta: Estructura donde están guardados los datos del requisito validado anteriormente que se enviará.
@Parametro metodo[10]: Método de solicitud (POST, GET, HEAD, etc), se obtiene de la estructura Request.

@Salida: No tiene.

void httpHeader(int cliente_socket,char *tipo_mime, int tamanio, time_t ultima_actualizacion, int codigo_respuesta, char *msj_codigo)

A Través de la información por parámetros, se encarga de armar la cabecera correcta a devolver al file descriptor del cliente cada vez que arma una línea este se la envía al cliente.

@Parametro cliente_socket: el file descriptor del cliente al cual se enviaran la cabecera.
@Parametro tipo_mime: el tipo mime del recurso que solicitó el cliente.
@Parametro tamanio: el tamaño del recurso en bytes solicitado por el cliente.
@Parametro ultima_actualizacion: la fecha y hora de la última modificación al recurso solicitado.
@Parametro codigo_respuesta: el código de respuesta de la petición del cliente.
@Parametro msj_codigo: mensaje correspondiente al código de respuesta.

@Salida: No tiene.

char *get_mime_type(char *name)

Realiza un corte de la cadena a la ruta del archivo obteniendo el tipo de dato del archivo y se compara con los tipos de datos que acepta el servidor para obtener el tipo MIME.

@Parametro *name: recibe por parámetro la ruta a la dirección del archivo.

@Salida: Función que devuelve una cadena de caracteres con el tipo MIME (tipos de datos de archivos).

Estructuras

Host.

Estructura utilizada para almacenar un socket junto a su dirección destino

```
typedef struct{
   int socket;
   struct sockaddr_in destino;
}Host;
```

Request.
Estructura utilizada para organizar la petición de un cliente, haciendo la división del path "directorio" + protocolo + versión

```
typedef struct{
   char line[MAXBUF];        //Esto es el mensaje entero del cliente.
   char metodo[10];          //Método GET o POST.
   char recurso[1000];       //Recurso cliente . Ex: index.html ; random/image.gif...
   char protocolo[20];       //Versión del Protocolo http 1.0.
}Request;
```

CR_returns.
Estructura utilizada para guardar los returns de la función **checkRequest**.
Esta estructura tiene información sobre directorio del archivo local, código de respuesta del servidor, el código y mensaje motivo y una subestructura que comprueba el estado de los archivos.

```
typedef struct{
   char dir[MAXBUF];              // Directorio local del archivo.
   struct stat statBuffer;        // Estructura para verificar el archivo.
   int n;                         // Abrir el archivo local como interno.
   int codigo;                    // Respuesta del servidor HTTP. Ex: 200, 404 ...
   char msj_codigo[20];           // Mensaje respecto al código.
}CR_returns;
```

Bibliografía

Proyecto referenciado: passeidireto - httpServer - Redes de computadores
https://www.passeidireto.com/arquivo/53083495/http-server
HTTP 1.0 https://www.w3.org/Protocols/HTTP/1.0/spec
RFC 1945 https://www.rfc-editor.org/rfc/rfc1945
Requerimientos del proyecto: TP1 - Desarrollo de un servidor web.