Práctica 1 - Servidor Web

Sitio: MOODLE DE GRADO
Curso: REDES DE COMUNICACIONES II
Libro: Práctica 1 - Servidor Web
Imprimido por: Javier Delgado del Cerro
Día: martes, 18 de junio de 2019, 19:50

Tabla de contenidos

1 Introducción

El objetivo de la primera práctica del curso es diseñar e implementar en lenguaje C un servidor Web, con algunas limitaciones, pero plenamente funcional. Para ello se recomienda seguir los siguientes pasos:

1. **Repasar** con cuidado el funcionamiento teórico del protocolo, hasta entender perfectamente sus componentes y cómo interaccionan entre sí. Además de las transparencias de teoría, la siguiente información también puede ser útil:
    1. Siempre es conveniente echar un ojo a los estándares, pues son la fuente original del resto de implementaciones de cualquier servidor Web completo, como Apache:
        1. HTTP/1.0
        2. HTTP/1.1
    2. HTTP basics
    3. HTTP Made Really Easy
2. "Jugar" con el protocolo, realizando diversas peticiones y observando las respuestas proporcionadas por el servidor Web. Una forma sencilla y rápida de chacerlo es utilizar una extensión para el navegador, como Restlest Cllient, o a través de la línea de comandos con cURL.
3. Tras el diseño, comenzar la **codificación** atacando el problema por partes:
    1. Escribir una librería que se encargue de la gestión de red a bajo nivel: apertura y cierre de sockets, conexión, etc. Probar de forma independiente, para descartar posibles errores futuros.
    2. Decidir el tipo e implementar el servidor. Inicialmente, no necesita "entender" ni responder con sintaxis HTTP, solo recibir conexiones y enviar alguna cadena. De esta manera se puede probar de forma exhaustiva, con muchas conexiones concurrentes.
    3. Añadir el resto de funcionalidad necesaria: archivo de configuración, ejecución de scripts, etc.
4. **Pruebas**: la práctica debe ser exhaustivamente probada antes de ser entregada. Además, se deben crear también una serie de scripts, que el servidor debe ejecutar correctamente.
5. **Documentación**: no se debe descuidar este aspecto, puesto que también tiene un peso importante en la nota final.

1.1 Tipo de servidor

Como hemos visto en teoría, existen diversos diseños de servidores, en función de su simplicidad, rendimiento y grado de concurrencia. En esta práctica, el estudiante deberá elegir el esquema que considere más oportuno, razonando siempre los motivos en la memoria de la misma.

Algunas referencias interesantes que pueden ayudar a tomar esta decisión son las siguientes:

- Stevens, W. Richard - Unix Network Programming, capítulo 30

1.2 Ejecución en modo demonio

El servidor a desarollar en esta práctica deberá ejecutar en modo daemon, es decir, como proceso no interactivo corriendo en segundo plano y desacoplado de cualquier terminal. En esta sección se enumeran las acciones que un programa debe llevar a cabo para iniciar como demonio (o daemon, en inglés), junto con una lista de funciones C a utilizar. Es tarea del alumno consultar la documentación de las mismas para comprender su funcionamiento.

En particular, se debe desarrollar una función `demonizar()` que reciba una cadena de caracteres que identifique al servicio y que devuelva cero en caso de ejecución correcta o un entero negativo en caso de error.

Dicha función deberá:

1. Crear un proceso hijo y terminar el proceso padre
2. Crear una nueva sesión de tal forma que el proceso pase a ser el líder de sesión
3. Cambiar la máscara de modo de ficheros para que sean accesibles a cualquiera
4. Establecer el directorio raíz / como directorio de trabajo
5. Cerrar todos los descriptores de fichero que pueda haber abiertos incluidos stdin, stdout, stderr
6. Abrir el log del sistema para su uso posterior

Es importante notar que un proceso daemon no está asociado a una terminal, y por tanto no debe utilizar funciones como `printf()` ni leer o escribir de entrada o salida estándar. Para notificar mensajes, este tipo de procesos utiliza el log del sistema, que típicamente se puede encontrar en /var/log/syslog .

Funciones a utilizar:

```
chdir, close, fork, getdtablesize, open, openlog, setsid, syslog, umask
```

Consultar en el manual de la función open el significado de los flags O_RDONLY y O_RDWR .

1.3 Desarrollo de la práctica

Depuración

Cada vez que un estudiante de la EPS decide depurar su programa utilizando printfs() y otras chapuzas similares, Bill Gates mata a un gatito. Tenlo en cuenta, por favor, por los gatitos del mundo. En serio, depurar un programa a base de este tipo de funciones es, a parte de incómodo, mucho más lento e incorrecto, impropio de un programador serio.

Existen diversas alternativas:

- **Eclipse**, o cualquier otro IDE. Estos entornos disponen de un depurador integrado, fácil y sencillo de utilizar.
- **gdb**, un depurador que puede usarse desde la línea de comandos. Es ligero y viene instalado por defecto en casi cualquier distribución Linux. Más información sobre cómo utilizarlo aquí.

Otro aspecto esencial es que tu programa no tengas fugas de memoria, dado que va a ser programado en C. Para encontrarlas y solucionarlas, puedes (debes) utilizar Valgrind.

Orden de implementación

Es importante ordenar adecuadamente el desarrollo de las distintas características que debe cumplir la práctica para no retrasarte innecesariamente:

- Por ejemplo, como el servidor debe ejecutarse en forma de demonio, y ésto implica que la salida debe estar redirigida a un fichero de log y se trabaja con un proceso diferente al padre, depurar un programa en este modo es muy incómodo. Por eso, un consejo es desarrollar primera la función de demonización, comprobar que funciona y luego comentarla para desarrollar la práctica con un solo hilo de ejecución.
- El mismo comentario aplica a la concurrencia del servidor: puede resultar más efectivo actuar de la siguiente manera:
    - Elegir primero el esquema de servidor concurrente a desarrollar (con un proceso o thread para cade petición, con un pool de los mismos, etc.), implementarlo y comprobar que funciona correctamente sin ninguna funcionalidad HTTP, simplemente devolviendo una cadena predeterminada. Así se puede probar fácilmente el rendimiento del servidor y su estabilidad ante cargas muy altas.
    - A continuación, cuando se esté seguro de que esta parte está acabada, comentar las funciones necesarias para convertirlo de nuevo en un servidor iterativo (con un solo hilo de ejecución, sin threads ni procesos).
    - Así podrás centrarte ahora en desarrollar toda la funcionalidad de recepción, parseo y respuesta de peticiones HTTP, "aislando" esta parte del resto.

- Cuando esté acabada, volver a activar la concurrencia y demonización.

Como ves, se trata de atacar el desarrollo por "partes", aislando las distintas características, y facilitar así el desarrollo.

2 Funcionalidad

El objetivo de la práctica es diseñar y codificar un servidor Web que, aunque tenga una funcionlidad limitada, sea perfectamente usable y con un rendimiento aceptable. Para ello, deberá implementar como mínimo las siguientes características:

- Soporte para los verbos GET, POST y OPTIONS
- Configuración del servidor a través de un fichero
- Soporte para la ejecución de scripts

Aunque inicialmente puede parecer algo complicado, el funcionamiento de un servidor Web es, en realidad, muy sencillo, pues básicamente se dedica a leer ficheros de discos y transmitirlos al cliente. Tanto si estos ficheros son complicados ficheros HTML, PDFs o vídeos, para el servidor son simplemente recursos que transmite a la otra parte para que la procese.

Por tanto, a muy alto nivel, el funcionamiento general del servidor podría ser algo similar a lo siguiente:

1. Inicialización de entorno, procesos o threads, lectura del fichero de configuración, etc. Es decir, todas las tareas necesarias para poner el servidor en marcha.
2. Queda a la escucha de una conexión entrante. Cuando se produce:
    1. Realizar todas las comprobaciones necesarias: que el formato de la URI es correcto, que el verbo sea soportado, que el recurso solicitado existe, que se trata de un recurso soportado, etc. Si alguna comprobación falla, se devuelve el código de error adecuado.
    2. Si el recurso es un script (es decir, la URI acaba en .py o .php), se procesa éste: se ejecuta y se devuelve la salida.
    3. Se devuelve el recurso: se lee de disco y se envía al cliente.

Obviamente, este flujo de ejecución está MUY simplificado, y deberá ser adaptado y ampliado por el estudiante en función de las decisiones de diseño que vaya realizando.

Parseo de peticiones

Aunque el parseo de una petición HTTP es relativamente sencillo, y puede abordarse perfectamente "a mano", en la práctica está permitido utilizar también librerías que faciliten esta labor, como PicoHTTPParser.

2.1 Verbos soportados

Como se ha comentado, los verbos que el servidor debe soportar son GET, POST y OPTIONS. Antes de avanzar, asegúrate de comprender bien las diferencias entre unos y otros, especialmente entre los dos primeros.

El procesamiento correcto de estos verbos incluye, por supuesto, proporcionar las respuestas adecuadas. Obviamente, cuantas más posibles respuestas se implementen, mejor puntuación final tendrá la práctica, pero, en cualquier caso, es necesario implementar como mínimo las siguientes:

- **200 OK**: El procesamiento de la petición fue correcto.
- **400 Bad Request**: El servidor no pudo interpretar la petición correctamente, probablemente debido a un error de sintaxis.
- **404 Not Found**: El servidor no ha podido encontrar el recurso solicitado.

Existe también libertad para implementar las cabeceras de las respuestas del servidor, pero, de nuevo, deben implementarse como mínimo las siguientes:

- *Date*: con el formato adecuado, según el estándar.
- *Server*: nombre y versión de vuestro servidor.
- *Last-Modified*: fecha de última modificación del recurso solicitado.
- *Content-Length*: longitud del recurso solicitado, en bytes.
- *Content-Type*: tipo de contenido del recurso solicitado. El navegador utiliza esta cabecera para saber cómo "tratar" la salida proporcionada por el servidor. Los tipos que se deben soportar en la práctica son "`text/plain`", "`text/html`", "`image/gif`", "`image/jpeg`", "`video/mpeg`", "`application/msword`", y "`application/pdf`". Recordad que un servidor Web no procesa estos archivos de ninguna forma, sólo los sirve. Es decir, cuando un navegador solicita un archivo PDF, por ejemplo, el servidor se limita a buscarlo en disco, rellenar adecuadamente la cabecera `Content-Type` y enviar el fichero tal cual. Es el navegador quien lanzará un visor de PDFs o cualquier otra acción apropiada.

Las extensiones asociadas a cada etiqueta `Content-Type` son, en general, auto-explicativas, pero se recogen en la siguiente lista:

- **text/plain**: .txt
- **text/html**: .html, .htm

- **image/gif**: .gif
- **image/jpeg**: .jpeg, .jpg
- **video/mpeg**: .mpeg, .mpg
- **application/msword**: .doc, .docx
- **application/pdf**: .pdf

Esta lista significa que, por ejemplo, si una URL solicita un recurso cuya extensión es .jpeg o .jpg (se trata de una imagen), el servidor debe leer el fichero adecuado del disco, añadir la directiva `Content-Type` correcta (**image/jpeg**, en este caso) a sus cabeceras en la respuesta, y servir el fichero. Si se solicita un recurso de tipo no incluido en la lista, el servidor deberá responder con un código de error adecuado.

Ejemplos

Por ejemplo, ante una petición como la siguiente,

```
Date: Sun, 18 Oct 2009 10:32:05 GMT
Server: Apache/2.2.14 (Win32)
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Length: 215
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>501 Method Not Implemented</title>
</head><body>
<h1>Method Not Implemented</h1>
<p>get to /index.html not supported.<br />
</p>
</body></html>
```

2.2 Configuración del servidor

Como todo servidor Web, el de esta práctica también tendrá un fichero de configuración, que deberá llamarse 'server.conf' y situarse en el mismo directorio que el ejecutable del servidor. El fichero tendrá un formato muy sencillo de texto, basado en pares clave/valor, uno por línea, y considerando comentarios aquellas que comiencen por '#'.

Para facilitar el proceso, pueden utilizarse librerías externas que automaticen el parseo, como libconfuse. Aquí encontrarás ejemplos sencillos de cómo utilizarla.

En esta práctica, el servidor debe poder procesar correctamente las siguientes directivas:

- **server_root** : ruta al directorio que contiene los ficheros y recursos del servidor Web. Es un ruta relativa respecto al directorio donde se encuentra el fichero de configuración.
- **max_clients**: número máximo de clientes que el servidor podrá atender simultáneamente. Llegado este número, el servidor simplemente no aceptará más conexiones.
- **listen_port**: puerto en el que el servidor debe recibir las conexiones entrantes.
- **server_signature**: cadena que será devuelta en cada cabecera `ServerName` posterior.

Un fichero de configuración de ejemplo sería el siguiente:

```
# Fichero de configuración de la pareja 02 - Redes 2
server_root = htmlfiles/
max_clients = 10
listen_port = 8080
server_signature = MyCoolServer 1.1
```

El valor en **server_root** se concatena con la URL recibida para construir la ruta final donde acceder a un fichero. Por ejemplo, imaginemos la siguiente petición GET:

```
GET /docs/index.html HTTP/1.0
```

Con el fichero de configuración anterior, el servidor en este caso buscaría el fichero `index.html` en la ruta `htmlfiles/docs`.

2.3 Ejecución de scripts

El servidor Web soportará, también, la ejecución de scripts en Python y PHP, con el fin de ofrecer la posibilidad de utilizar contenido dinámico. Para ello, el recurso solicitado debe acabar necesariamente en las extensiones .py o .php, respectivamente.

Contenido dinámico

Tradicionalmente, los servidores Web han generado contenido dinámico, en el que la respuesta HTML se genera en tiempo real a través de un script o programa que ejecuta el servidor al recibir la petición, a través de dos métodos básicos, CGI e intérpretes embebidos.

El primero de ellos es un estándar de finales de los 90, ya obsoleto y que no debería utilizarse en nuevos desarrollos, pero que aún sigue en uso en algunas aplicaciones antiguas. Es relativamente sencillo, y lanza un proceso para cada petición, que ejecuta el script, recoge su salida y la devuelve en forma de respuesta al cliente.

Por otro lado, hoy en día cualquier servidor Web serio, soporta la ejecución de lenguajes interpretados, como Python, PHP o Perl, de forma nativa, al tener embebido un intérprete en el propio código del servidor. De esta forma, la ejecución es mucho más rápida y segura que con el estándar CGI.

CGI

De estos dos enfoques, vamos a utilizar el de CGI en la práctica, para simplificarla y hacerla abordable, con alguna modificación relativa a cómo se pasan los parámetros al script. Por ejemplo, el estándar CGI los pasa en una variable de entorno en caso de una petición GET y por la entrada estándar en caso de una petición POST.

Para unificar, en nuestro servidor, tanto en peticiones GET como POST el procedimiento será el siguiente:

1. El servidor leerá los argumentos de la URL o del cuerpo de la petición, en función de si es una petición GET o POST, respectivamente.
2. Ejecutar localmente el script con el intérprete adecuado (python o php), pasándole los argumentos recibidos en su entrada estándar.
3. Leer la salida estándar del script y devolver estos datos en forma de respuesta HTTP. A priori, el servidor no sabe el tamaño de la respuesta del script, que puede ser de cualquier longitud. El script señalizará el final de sus datos con una línea que solo contenga los caracteres '\r\n' (igual que se indica la separación entre cabeceras y cuerpo en una petición HTTP).

Véamos un ejemplo. Imaginemos que el servidor recibe la siguiente petición:

```
GET /scripts/backend.py?var1=abcd&var2=efgh
```

Una vez hechas todas las comprobaciones necesarias, el servidor sabe que debe ejecutar el script 'backend.py' que se encuentra en la ruta relativa 'scripts/', para lo que deberá lanzar un nuevo proceso que se encargue de la ejecución del mismo.

Scripts de prueba

Una manera útil de probar la práctica y entender mejor su funcionalidad será desarrollar, además del servidor, una serie de programas de prueba (en Python o C) que:

- Reciba un petición GET con un argumento que guarde el nombre de un usuario, y responda con la cadena "Hola <<nombre>>!"
- Reciba un petición POST con un argumento que guarde una temperatura en grados Celsius, y devuelva la temperatura correspondiente en Farenheit.

Si se escriben, se debe entregar el código de estos scrips junto con el resto de la práctica, e información sobre cómo ejecutarlos.

2.4 Resumen de recursos

A lo largo de la descripción de la práctica se han ido enumerando una serie de recursos y librerías que pueden ser útiles para su realización. Se resumen a continuación:

- Descripción del protocolo HTTP y ejemplos de peticiones y respuestas típicas:
  - HTTP basics
  - HTTP Made Really Easy
- Ejemplos de código de distintos tipos de servidores:
  - Stevens, W. Richard - Unix Network Programming, capítulo 30

- Librería de parseo de peticiones y respuestas HTTP:
  - PicoHTTPParser
- Librería de parseo de ficheros de configuración:
  - libconfuse. Ejemplos de uso: aquí

3 Evaluación

La nota final de la práctica dependerá de varios factores, como la funcionalidad implementada, el funcionamiento de ésta, la calidad del código (comentarios, modularidad, creación de librerías) o de la documentación.

Sin embargo, en líneas generales, la funcionalidad que se considera mínima para que la práctica pueda ser aprobada (considerando que el resto de aspectos son igualmente correctos), sería la siguiente:

- El servidor procesa adecuadamente peticiones sencillas.
- El servidor acepta opciones básicas de configuración.
- La práctica dispone de scripts sencillos de prueba.