Capa de Aplicación, HTTP

Andres Barbieri [*]

11 de marzo de 2019

1. Aclaración para el Lector

Este texto fue generado en el año 2008 para el postgrado de Redes II y la materia Redes y Comunicaciones de la Fac. de Informática - UNLP. Con el tiempo fue sufriendo algunas modificaciones, pero parte de la información contenida se mantuvo. Por ejemplo los tests se realizaron con software que ya a quedado ampliamente superado. Si bien contiene referencias o información que con el tiempo ha ido cambiando, el autor lo cree útil para el entendimiento del protocolo HTTP del cual se adjuntan varias capturas. Las mismas fueron generadas con el propósito de acompañar el texto en el entendimiento del mismo. Es importante que la información aquí contenida sea complementada con otros textos, herramientas y ejemplos, como puede ser las tutorías del programa `cURL(1)` que pone disponible la cátedra.

2. Introducción

La World Wide Web, WWW (red de sistemas de hypertexto inter-linkeados accesibles vía Internet), fue desarrollada en 1990 por Tim Berners-Lee, que trabajaba en el Conseil Europeen pour la Recherche Nucleaire (CERN). Tim Berners-Lee inventó el HTML -HyperText Markup Language- (HTML Tags) y un sistema para poder accederlo vía TCP utilizando la infraestructura de Internet, el protocolo HTTP -HyperText Transfer Protocol-. HTML estuvo basado en SGML y HTTP se basó, en sus principios, en un protocolo conocido como Gopher, que se utilizaba para compartir archivos de una forma más amigable que FTP. La primera descripción de HTML y las implementaciones de lo que fue después HTTP son de 1991. La primera versión de HTTP, desarrollada por Tim Berners-Lee, es conocida como HTTP 0.9. Además, desarrolló un servidor llamado `httpd` y un cliente/editor, de tipo WYSIWG, llamado `WorldWideWeb`, el cual corría sobre una estación de trabajo NeXTSTEP. La segunda versión de HTML, que sería la 1.0, nunca fue establecida como un estándar formal. Tampoco, HTTP 0.9 fue establecido como un estándar.

El primer estándar de HTML es el 2.0 [RFC-1866] establecido mediante una RFC a través de la IETF en 1995. En 1993, el primer cliente/browser con interfaz gráfica para la Web, llamado Mosaic, fue desarrollado. Éste nació en la NCSA (National Center for Supercomputing Applications), que es una unidad de la University of Illinois, y, además, fue uno de los centro originales de la NSF (National Science Foundation). Inicialmente, sólo ejecutaba sobre UNIX ya que estaba montado sobre X Window System. Fue desarrollado por Marc Andreessen y Eric Bina.

[*]barbieri@cespi.unlp.edu.ar, con colaboración de Matías Robles: mrobles@info.unlp.edu.ar.

Tiempo después, Marc Andreessen abandonó la NCSA. Junto con Jim Clark (uno de los fundadores de SGI) y otros estudiantes de la Universidad de Illinois comenzaron Mosaic Communications Corporation, más tarde llamada Netscape Communications Corporation. En 1994, surge una nueva interfaz para la Web, el Netscape Navigator 1.0, el cual rápidamente llegó a toda la red. El Netscape Navigator era gratis para uso no comercial.

Microsoft intenta entrar en el mercado usando el código de Mosaic provisto por Spyglass, Inc., una empresa con base en Champaign, Illinois, y creada para comercializar las tecnologías de la NCSA. Hasta 1996, con el IE (Internet Explorer) 3.0 los productos de Microsoft en el tema eran altamente inferiores a los de Netscape. Pero, con el tiempo, Microsoft se fue imponiendo sobre Netscape. Muchos acreditan esto a las prácticas monopólicas de Microsoft, otros a la muy nueva tecnología de Java incluida en el desarrollo del Netscape Communicator, la cual no tenía la madurez necesaria en ese tiempo produciendo un procesamiento más lento. Otros navegadores conocidos son el Opera, desarrollado por una compañía Noruega, que tenía la fama de ser el más rápido, Safari, desarrollado para Mac OS por Apple Inc., y el Mozilla, de 1998, que está basado en el código abierto de Netscape Communicator y del cual deriva el Mozilla-Firefox. En Septiembre de 2008, el gigante Google lanzó su propio navegador, el Chrome. Para inicios de 2013 se encontraba en el top del ranking. En la figura 1 se muestra el share de los diferentes navegadores [BR1]. Esto da una idea de cuán dinámico son las herramientas y "mercado" de este protocolo.

En el ámbito de los servidores, Netscape también comercializó servidores Web como el Netscape FastTrack Server, Microsoft posee el Internet Information Server, pero el más usado es el Apache HTTP Server. El proyecto Apache comenzó en 1995 a partir del código del `httpd` v1.3 de Rob McCool, desarrollado en la NSCA. En la figura 2 se muestra el share de los diferentes servidores Web (fuente netcraft:[SR1]) hasta 2013.

En la actualidad, tanto HTML como HTTP están regidos por el World Wide Web Consortium, abreviado W3C [W3C]. Éste es un consorcio internacional que produce estándares para la World Wide Web creado, en 1994, por Tim Berners-Lee en el MIT. Los estándares del W3C son abiertos. A partir de HTML 3.0, las especificaciones no se manejan como RFC sino como estándares del W3C, por ejemplo: [HTML30], [HTML401], [XHTML1] y [HTML5].

Otro invento interesante de Tim Berners-Lee es la URL (Uniform Resource Locator). A través de una URL se referencia a un recurso u objeto HTTP. Existe también el término URI (Uniform Resource Identifier), el cual es más general. Una URI puede usar una URL o un nombre, URN (Uniform Resource Name) como identificador único de un objeto. formato de URL: `protocol://[user:pass@]host:[port]/[path]`, por ejemplo: `http://www.NN.unlp.edu.ar:8080/dir/index.html`.
formato de URN: `"urn:" <NID> ":" <NSS>`, por ejemplo:
`<xsd: ... targetNamespace="urn:mynamespace ... >"`.

3. Versiones del Protocolo HTTP

El funcionamiento del protocolo HTTP está basado en el modelo cliente/servidor, Request/Response, donde el cliente envía un requerimiento solicitando un objeto, en general, una página HTML y la recibe como respuesta desde un servidor WEB. Ver fig. 3. El protocolo corre sobre TCP porque requiere un protocolo de transporte confiable. Según [RFC-1945], podría basarse en cualquier protocolo que brinde

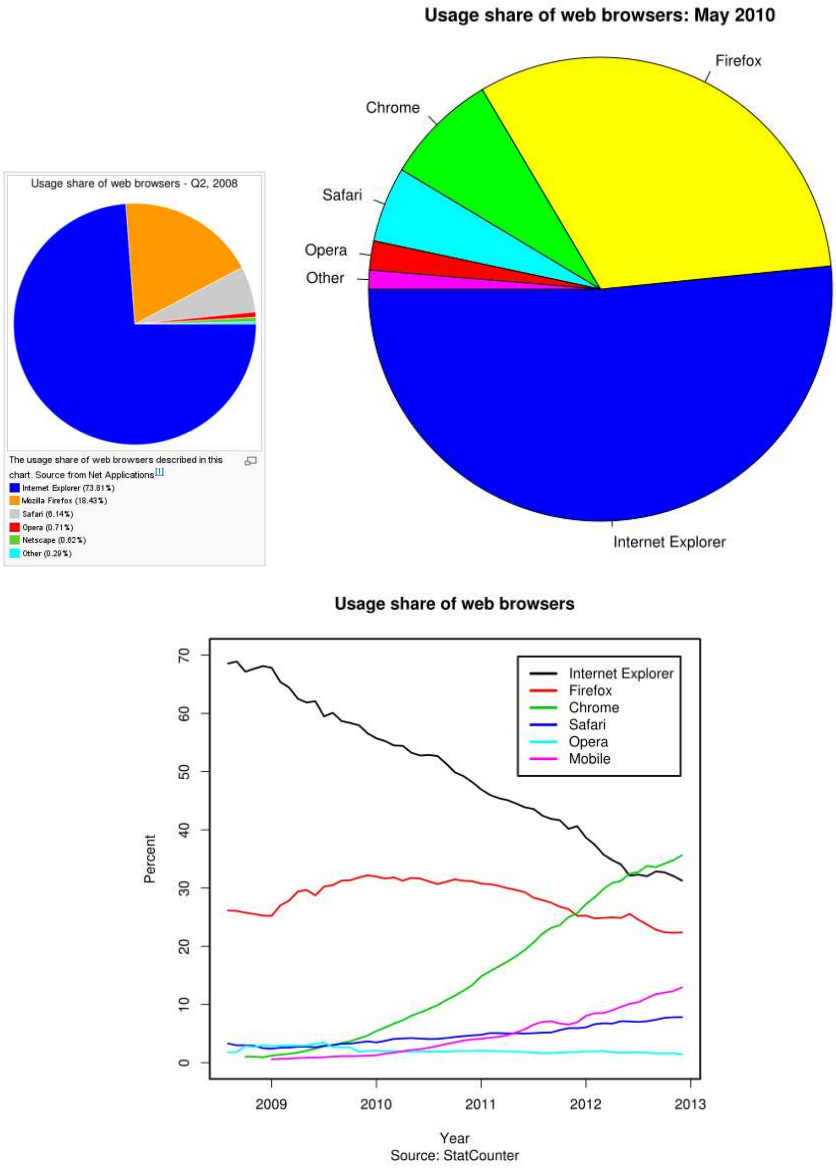

Figura 1: Share de Browsers para Septiembre de 2008, Mayo de 2010 y evolución hasta 2013.

ese servicio. Obviamente, UDP queda descartado.

3.1. HTTP 0.9

La primera versión de HTTP fue la 0.9 [HTTP0.9], nunca se estandarizó. Define un protocolo del tipo Request/Response muy sencillo. Sólo contiene un método o comando, `GET`. Los pasos para obtener un documento son:

1. Establecer la conexión TCP

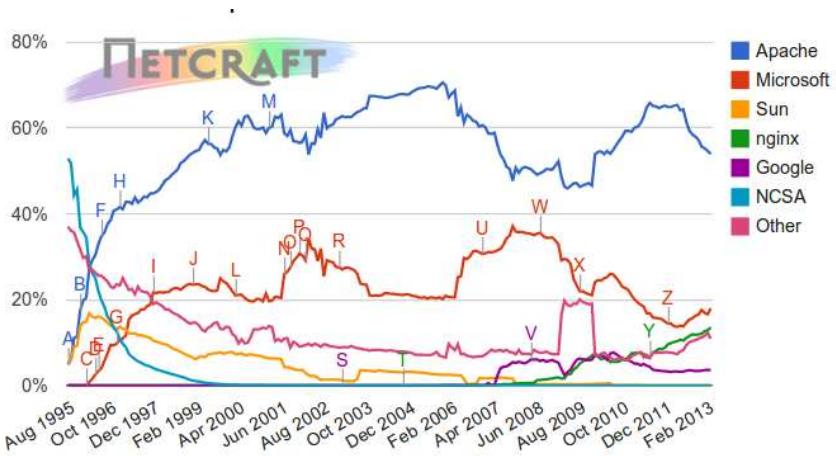

Figura 2: Evolución de share de servidores web hasta 2013.

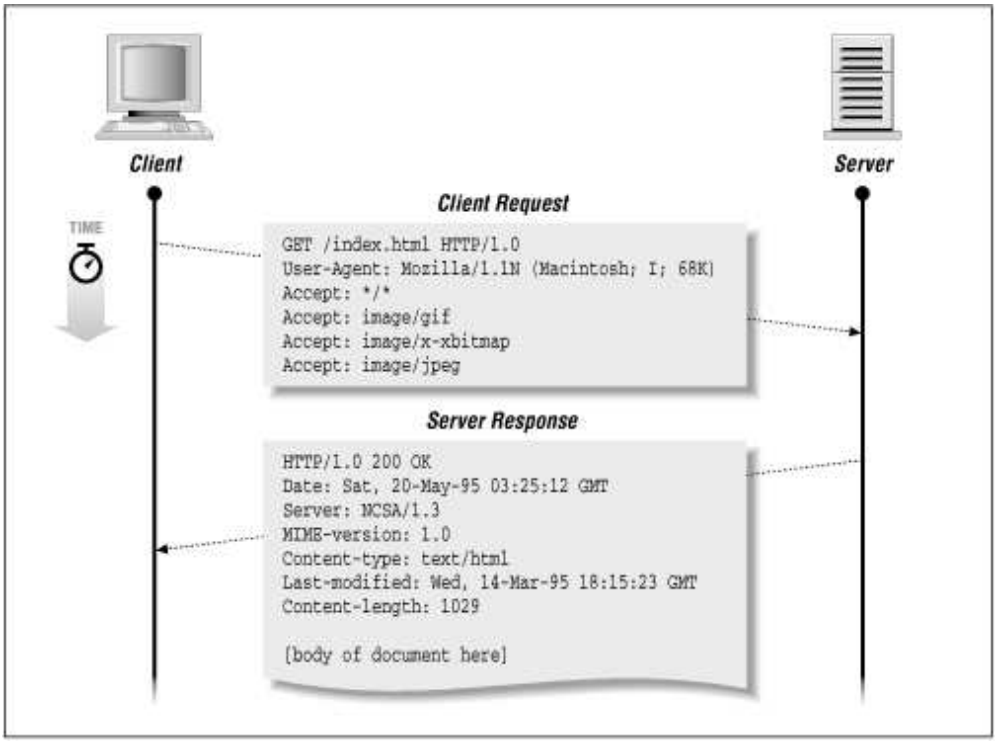

Figura 3: Ejemplo Request/Response.

2. HTTP Request vía comando `GET`.
3. HTTP Response enviando la página requerida.
4. Cerrar la conexión TCP por parte del servidor.

```
Request ::== GET <document-path> <CR><LF>

Response ::== ASCII chars HTML Document.
```

Solo existía una forma de hacer el requerimiento y, también, una única forma de responder. El cliente se conecta al puerto 80, puerto default de HTTP, aunque podría utilizar otro.

```
# grep "www" /etc/services
www             80/tcp          http            # WorldWideWeb HTTP
...
```

Se realiza un requerimiento de la siguiente manera:

```
GET /hello.html <CR><LF>
```

El servidor retorna el contenido de la página solicitada y cierra la conexión TCP. Si no existe el documento requerido, el servidor no envía nada y solo cierra la conexión. El cliente no mostará nada. No existe la posibilidad de enviar mucha información del cliente al servidor, solo vía el comando `GET`. Cada línea de la respuesta debe llevar un Carrier Return (CR) opcional y un Line Feed (LF) obligatorio. No existen códigos que distinguen errores de respuestas correctas. Ejemplo:

```
# tcpdump -n -i lo port 80 -s 1500 -w 01-http0.9.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET /

<HTML>
<H1> HOLA </H1>
</HTML>

# tail -f /var/log/apache2/access.log
127.0.0.1 - - [20/Apr/2008:21:24:26 -0300] "GET /" 200 31
```

Para este ejemplo se usó un servidor Apache local. Más adelante, en la pág. 7, se muestra la configuración. Los clientes/browsers actuales sólo se pueden configurar para que trabajen con HTTP 1.0 ó 1.1. Es por esto, que se utilizó el comando `telnet(1)` para generar las conexiones TCP.

3.2. HTTP 1.0

Fue estandarizado vía [RFC-1945]. Define un formato de mensaje para el Request y otro para el Response.

- Se debe especificar la versión en el requerimiento del cliente.
- Para los Request, define diferentes métodos HTTP.

- Define códigos de respuesta.
- Admite repertorio de caracteres, además del ASCII, como: ISO-8859-1, UTF-8, etc.
- Admite MIME(Multipurpose Internet Mail Extensions), no solo sirve para descargar HTML e imágenes.

3.2.1. Formato del Request

```
<Method> <URI> <Version>
[<Headers Opcionales>]
<Blank>
[<Entity Body Opcional>]
<Blank>

<Method HTTP 1.0> ::== GET, POST, HEAD, PUT, DELETE, LINK, UNLINK.
```

3.2.2. Formato del response

```
<HTTP Version> <Status Code> <Reason Phrase>
[<Headers Opcionales>]
<Blank>
[<Entity Body Opcional>]
```

3.2.3. Ejemplos de respuestas HTTP/1.0

```
HTTP/<version> 200 OK
HTTP/<version> 301 Moved Permanently
HTTP/<version> 400 Bad Request
HTTP/<version> 403 Access Forbidden
HTTP/<version> 404 Not Found
HTTP/<version> 405 Method Not Allowed
HTTP/<version> 500 Internal Server Error (CGI Error)
HTTP/<version> 501 Method Not Implemented
```

3.2.4. Métodos HTTP1.0

GET: obtener el documento requerido. Puede enviar información, pero no demasiada. Es enviada en la URL del requerimiento. Con el formato `?var1=val1&var2=val2...`. La cantidad e info está restringida al tamaño de la URL. En general, 256 bytes.

HEAD: idéntico a GET, pero sólo requiere la meta información del documento, por ejemplo, su tamaño. No obtiene el documento en sí.

POST: hace un requerimiento de un documento, pero también envía información en el Body. Generalmente, usado en el fill-in de un formulario HTML(FORM). Puede enviar mucha más información que un GET.

PUT: usado para reemplazar un documento en el servidor. En general, deshabilitada. Utilizado, por ejemplo, por protocolos para compartir archivos y carpetas montados sobre HTTP, como WebDAV [WDV].

DELETE: usado para borrar un documento en el servidor. En general, deshabilitada. También, puede ser utilizada por WebDAV.

LINK, UNLINK: establecen/des-establecen relaciones entre documentos.

Los mensajes (operaciones) `PUT`, `DELETE`, `LINK`, `UNLINK` son pasados directamente a la aplicación si el servidor web los soporta.

3.2.5. Ejemplo, Configuración Apache

Se muestra la configuración inicial del servidor Web Apache. Se utilizó para los ejemplos la versión `apache2(8)` sobre GNU/Linux [LX].

```
# uname
Linux

# apache2ctl -v
Server version: Apache/2.2.4 (Ubuntu)
Server built:   Feb  4 2008 20:30:42


# cat /etc/apache2/apache2.conf
### Created By Andres 2005-10-18 (C) ###

ServerName www
UseCanonicalName On

Listen 0.0.0.0:80

ServerRoot "/etc/apache2"

DocumentRoot "/var/www/html"
LoadModule dir_module /usr/lib/apache2/modules/mod_dir.so
DirectoryIndex index.html

LockFile /var/lock/apache2/accept.lock
PidFile /var/run/apache2.pid

User www-data
Group www-data

TransferLog /var/log/apache2/access.log
ErrorLog /var/log/apache2/error.log
LogLevel Info
```

```
ServerTokens Full

#### EOF ####

# /etc/init.d/apache2 start
```

3.2.6. Ejemplo GET-200 HTTP/1.0 con telnet(1)

A continuación se muestra un ejemplo de un Request y un Response correcto.

```
# tcpdump -n -i lo port 80 -s 1500 -w 02-http1.0.pcap
```

Request:

```
? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET /index2.html HTTP/1.0
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
Accept: */*
```

Response:

```
HTTP/1.1 200 OK
Date: Mon, 21 Apr 2008 00:28:51 GMT
Server: Apache/2.2.4 (Ubuntu)
Last-Modified: Mon, 21 Apr 2008 00:18:14 GMT
ETag: "a3b36-1f-91d5d80"
Accept-Ranges: bytes
Content-Length: 31
Connection: close
Content-Type: text/plain

<HTML>
<H1> HOLA </H1>
</HTML>
Connection closed by foreign host.
```

En los logs correspondientes se puede observar el resultado de la operación. Como se detalló más arriba, el código 200 indica que la respuesta es OK.

```
# tail -f /var/log/apache2/access.log
127.0.0.1 - - [20/Apr/2008:21:28:51 -0300] "GET /index.html HTTP/1.0" 200 31
```

3.2.7. Ejemplo GET-404 HTTP/1.0 con telnet(1)

Pero, como se puede observar en el siguiente ejemplo, al solicitar un documento inexistente se obtiene el código de error 404.

```
# tcpdump -n -i lo port 80 -s 1500 -w 03-http1.0-notfound.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET /index2.html HTTP/1.0
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
Accept: */*

HTTP/1.1 404 Not Found
Date: Mon, 21 Apr 2008 00:56:31 GMT
Server: Apache/2.2.4 (Ubuntu)
Content-Length: 209
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /index2.html was not found on this server.</p>
</body></html>

Connection closed by foreign host.

# tail -f /var/log/apache2/access.log
127.0.0.1 - - [20/Apr/2008:21:56:31 -0300] "GET /index2.html HTTP/1.0" 404 209
```

3.2.8. Ejemplo HEAD HTTP/1.0 con telnet(1)

Usando el comando HEAD se pueden requerir los meta-datos del documento.

```
# tcpdump -n -i lo port 80 -s 1500 -w 04-http1.0-head.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
HEAD /index.html HTTP/1.0
```

```
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
Accept: */*

HTTP/1.1 200 OK
Date: Mon, 21 Apr 2008 00:40:25 GMT
Server: Apache/2.2.4 (Ubuntu)
Last-Modified: Mon, 21 Apr 2008 00:18:14 GMT
ETag: "a3b36-1f-91d5d80"
Accept-Ranges: bytes
Content-Length: 31
Connection: close
Content-Type: text/plain

Connection closed by foreign host.
# tail -f /var/log/apache2/access.log
127.0.0.1 - - [20/Apr/2008:21:40:25 -0300] "HEAD /index.html HTTP/1.0" 200 -
```

3.2.9. Ejemplo POST HTTP/1.0 con telnet(1)

Otro método es el POST. Su funcionamiento se muestra en el siguiente ejemplo. Se envían datos en el cuerpo del mensaje HTTP, un string de 10 caracteres ASCII: "1234567890". El servidor ignora la información enviada porque la página requerida es un HTML estático.

```
# tcpdump -n -i lo port 80 -s 1500 -w 05-http1.0-post.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
POST /index.html HTTP/1.0
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
Accept: */*
Content-Type: text/plain
Content-Length: 10

1234567890

HTTP/1.1 200 OK
Date: Mon, 21 Apr 2008 00:37:22 GMT
Server: Apache/2.2.4 (Ubuntu)
Last-Modified: Mon, 21 Apr 2008 00:18:14 GMT
ETag: "a3b36-1f-91d5d80"
Accept-Ranges: bytes
Content-Length: 31
Connection: close
```

```
Content-Type: text/plain

<HTML>
<H1> HOLA </H1>
</HTML>

Connection closed by foreign host.

# tail -f /var/log/apache2/access.log
127.0.0.1 - - [20/Apr/2008:21:37:22 -0300] "POST /index.html HTTP/1.0" 200 31
```

3.2.10. Otros ejemplos HTTP/1.0 telnet(1)

A continuación se muestran diferentes valores de retorno. Primero, un método deshabilitado, el DELETE.

```
# tcpdump -n -i lo port 80 -s 1500 -w 06-http1.0-otros.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
DELETE /index.html HTTP/1.0
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
Accept: */*

HTTP/1.1 405 Method Not Allowed
Date: Mon, 21 Apr 2008 00:39:00 GMT
Server: Apache/2.2.4 (Ubuntu)
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Length: 234
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>405 Method Not Allowed</title>
</head><body>
<h1>Method Not Allowed</h1>
<p>The requested method DELETE is not allowed for the URL /index.html.</p>
</body></html>

Connection closed by foreign host.
```

Ahora, se envía un método que no está definido en el protocolo.

```
? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
NOEXISTE /index.html HTTP/1.0
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
Accept: */*

HTTP/1.1 501 Method Not Implemented
Date: Mon, 21 Apr 2008 00:39:42 GMT
Server: Apache/2.2.4 (Ubuntu)
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Length: 220
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>NOEXISTE to /index.html not supported.<br />
</p>
</body></html>

Connection closed by foreign host.

# tail -f /var/log/apache2/access.log
127.0.0.1 - - [20/Apr/2008:21:39:00 -0300] "DELETE /index.html HTTP/1.0" 405 234
127.0.0.1 - - [20/Apr/2008:21:39:42 -0300] "NOEXISTE /index.html HTTP/1.0" 501 220
```

3.2.11. Ejemplo GET HTTP/1.0 con un browser

Para hacer requerimientos HTTP 1.0 con un browser se debe utilizar una versión que permita configurar HTTP 1.0 en lugar de HTTP 1.1. En este caso se utiliza Mozilla 1.7.13 Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.13). Para modificar esto se debe ir a "Edit", "Preferences", "Advanced", "HTTP Networking", como se puede ver en la figura 4. Recordar de hacer "Clear" de la cache antes de realizar el pedido.

```
# tcpdump -n -i lo port 80 -s 1500 -w 07-http1.0-mozilla-OLD.pcap

? tar -xvzf mozilla-i686-pc-linux-gnu-1.7.13.tar.gz
? cd mozilla
? ./mozilla http://127.0.0.1

<HTML>
```

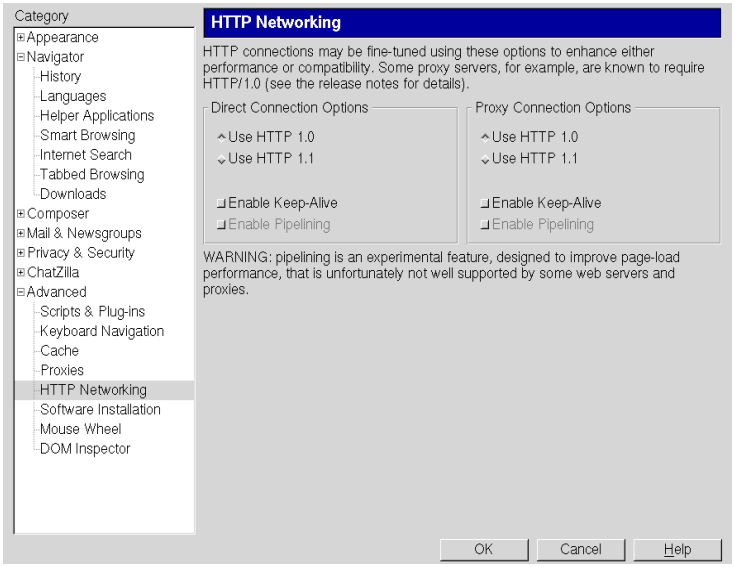

Figura 4: Ejemplo configuración Mozilla.

```
<H1> HOLA </H1>
</HTML>
```

Ver que el requerimiento se ve en texto plano. El HTML no se interpreta como tal y esto permite que se vea el código fuente del HTML. Por default, el contenido MIME se establece en texto plano o claro.

```
# tail -f /var/log/apache2/access.log
127.0.0.1 - - [21/Apr/2008:11:05:32 -0300] "GET / HTTP/1.0" 200 31
```

Tanto Mozilla como Apache2, por default, trabajan en HTTP 1.1.

Configurar el Apache para que envíe el contenido correcto (utilizar el MIME) y manejar los requerimientos como HTTP 1.0.

```
# vi /etc/apache2/apache2.conf
...
LoadModule mime_module /usr/lib/apache2/modules/mod_mime.so
TypesConfig /etc/mime.types
AddType text/html .html
...
LoadModule setenvif_module /usr/lib/apache2/modules/mod_setenvif.so
BrowserMatch "Mozilla/5" nokeepalive downgrade-1.0 force-response-1.0
...
```

Además, configurar para acceder por nombre en lugar de dirección IP. Esto requiere una entrada en el servidor de DNS o, de forma más sencilla, agregarlo en el archivo de búsqueda/resolver local, `/etc/hosts`.

```
# cat /etc/hosts
127.0.0.1       localhost www

# /etc/init.d/apache2 restart

# tcpdump -n -i lo port 80 -s 1500 -w 08-http1.0-mozilla-OLD-mime.pcap

? ./mozilla http://www
HOLA

# tail -f /var/log/apache2/access.log
127.0.0.1 - - [21/Apr/2008:11:22:02 -0300] "GET / HTTP/1.0" 200 31
```

3.2.12. Ejemplo GET HTTP/1.0 con Host Virtuales

Mediante el parámetro `Host` se pueden multiplexar varios servidores virtuales en el mismo servidor, en lugar de tener un servidor por cada puerto. Esto es posible a partir de HTTP/1.0. Primero, se deben crear los contenidos para cada uno de los hosts virtuales.

```
# mkdir /var/www/html2

# vi /var/www/html2/index.html

# cat /var/www/html2/index.html
<HTML>
<H1> HOLA 2</H1>
</HTML>

# chown -R www-data:www-data /var/www/html2

# cat /etc/hosts
127.0.0.1       localhost www www2
```

Una vez creado el "nuevo servidor", se debe modificar Apache2 para que sea capaz de trabajar con hosts virtuales.

```
# cat /etc/apache2/apache2.conf
### Created By Andres 2005-10-18 (C) ###

ServerName www
UseCanonicalName On

Listen 0.0.0.0:80

ServerRoot "/etc/apache2"

DocumentRoot "/var/www/html"
```

```
LoadModule dir_module /usr/lib/apache2/modules/mod_dir.so
DirectoryIndex index.html

LockFile /var/lock/apache2/accept.lock
PidFile /var/run/apache2.pid

LoadModule mime_module /usr/lib/apache2/modules/mod_mime.so
TypesConfig /etc/mime.types
AddType text/html .html
##DefaultType text/html

User www-data
Group www-data

TransferLog /var/log/apache2/access.log
ErrorLog /var/log/apache2/error.log
LogLevel Info
ServerTokens Full

LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined

LoadModule setenvif_module /usr/lib/apache2/modules/mod_setenvif.so
BrowserMatch "Mozilla/5" nokeepalive downgrade-1.0 force-response-1.0

NameVirtualHost *

<VirtualHost *>
        ServerName  www
        DocumentRoot /var/www/html
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
        ErrorLog /var/log/apache2/www.error.log
        LogLevel warn
        CustomLog /var/log/apache2/www.access.log combined
        ServerSignature On
</VirtualHost>

<VirtualHost *>
        ServerName  www2
        DocumentRoot /var/www/html2
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
```

```
        ErrorLog /var/log/apache2/www2.error.log
        LogLevel warn
        CustomLog /var/log/apache2/www2.access.log combined
        ServerSignature On
</VirtualHost>

#### EOF ####

# /etc/init.d/apache2 restart

# tcpdump -n -i lo port 80 -s 1500 -w 09-http1.0-mozilla-OLD-vhosts.pcap

? ./mozilla http://www
? ./mozilla http://www2
? ./mozilla http://127.0.0.1
```

3.2.13. Ejemplo GET HTTP/1.0 Condicional (Cliente Cache)

Al solicitar una página, el funcionamiento del browser varía dependiendo de si la página está o no en su cache. Para comprobarlo, dar enter sobre la URL, sin borrar la cache del browser, y ver que el navegador muestra el contenido de su cache, pero no conecta contra el servidor. En cambio, si se hace "Reload" si se conecta al servidor, pero éste le dice que esta página no ha cambiado.

```
? ./mozilla http://www

# tcpdump -n -i lo port 80 -s 1500 -w 10-http1.0-mozilla-OLD-cached.pcap
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 1500 bytes
10 packets captured
20 packets received by filter
0 packets dropped by kernel

# tcpdump -n -i lo port 80 -s 1500 -w 10-http1.0-mozilla-OLD-cached.pcap
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 1500 bytes
0 packets captured
0 packets received by filter
0 packets dropped by kernel

GET / HTTP/1.0
Host: www
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.13) Gecko/20060417
Accept: text/xml,application/xml,application/xhtml+xml,text/html;
        q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: close
If-Modified-Since: Mon, 21 Apr 2008 00:18:14 G
```

```
If-None-Match: "a3b36-1f-91d5d80"
Pragma: no-cache

HTTP/1.0  304 Not Modified
Date: Mon, 21 Apr 2008 14:47:30 GMT
Server: Apache/2.2.4 (Ubuntu)
Connection: close
ETag: "a3b36-1f-91d5d80"
```

Si observamos en la captura anterior a este flujo HTTP, el servidor indicó al cliente mediante el Tag: `Last-Modified` cual fue la fecha de modificación del archivo entregado. De esta forma el cliente sabe a partir de cuando solicitar los cambios. Ver archivo: `09-http1.0-mozilla-OLD-vhosts.pcap`

```
HTTP/1.0 200 OK
Date: Mon, 21 Apr 2008 14:43:22 GMT
Server: Apache/2.2.4 (Ubuntu)
Last-Modified: Mon, 21 Apr 2008 00:18:14 GMT
ETag: "a3b36-1f-91d5d80"
...
```

De esta forma se permite ahorrar en ancho de banda y hacer más eficiente y ágil el flujo de la información. El servidor no necesita enviar el contenido si el objeto solicitado no ha cambiado. Este un principio básico de como un cliente puede realizar el "caching" coordinando con el servidor.

3.2.14. ETags (Cliente Cache)

Si se observan en los requerimientos y en las respuestas encontramos los tags o marcadores en los encabezados que indican una comparación mediante valores similares a "hashes": por ejemplo en el request: `If-None-Match: ‘‘a3b36-1f-91d5d80’’` o en las respuestas `ETag: ‘‘a3b36-1f-91d5d80’’`. Un ETag (entity tag) son marcadores en los encabezados definidos como parte del HTTP. Son parte de diferentes mecanismos que permiten realizar cache al cliente mediante GET condicionales. Funcionan como fingerprints que pueden compararse y detectar si el objeto requerido cambio o no. Al cambiar el valor asignado debe ser distinto al de la versión anterior. `If-None-Match` se puede usar de la misma forma que los timestamps con `If-Modified-Since`. Los primeros son más eficientes con contenido dinámico.
Los ETags podrían ser utilizados para realizar control de concurrencia, por ejemplo previniendo la actualización simultanea del mismo objeto y así evitar la sobre-escritura del mismo.

3.2.15. Ejemplo de permiso denegado

Reconfigurar Apache2 para obtener permisos denegados sobre algún recurso cuando se lo quiere acceder. Ver el mensaje de error que genera el servidor.

```
# mkdir /var/www/html/priv

# cat /etc/apache2/apache2.conf
...
#LoadModule access_module /usr/lib/apache2/modules/mod_access.so
```

```
LoadModule authz_host_module /usr/lib/apache2/modules/mod_authz_host.so
...
<VirtualHost *>
        ServerName  www
        ...
        <Directory /var/www/html/priv>
         Order Deny,Allow
         Deny from All
        </Directory>

# /etc/init.d/apache2 restart
```

Realizar el requerimiento.

```
# tcpdump -n -i lo port 80 -s 1500 -w 11-http0.9-deny.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET /priv/index.html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access /priv/index.html
on this server.</p>
<hr>
<address>Apache/2.2.4 (Ubuntu) Server at www Port 80</address>
</body></html>

Connection closed by foreign host.
```

3.2.16. Ejemplo de Autenticación

A partir de HTTP/1.0 se contempla la autenticación Web agregando WWW-Authenticate Headers. A partir de estos encabezados, el cliente y el servidor pueden intercambiar información referente al proceso de autenticación. El proceso más sencillo es "Basic Authentication". El servidor, ante un requerimiento de un documento que requiere autenticación, enviará un mensaje 401 indicando la necesidad de autenticación y un String representando un Dominio/Realm para el cual se requiere dicho proceso. El navegador solicitará al usuario los datos de user/password, si es que no los tiene cachedos para el Realm solicitado, y se los enviará en texto claro al servidor. El servidor dará o no acceso en base a esos valores. Para los siguientes requerimientos, el navegador usará los valores que tiene almacenados para el Realm solicitado.

Para probar esto se debe configurar Apache con autenticación, generando los archivos de usuarios. Las passwords se puede almacenar en DES o en MD5.

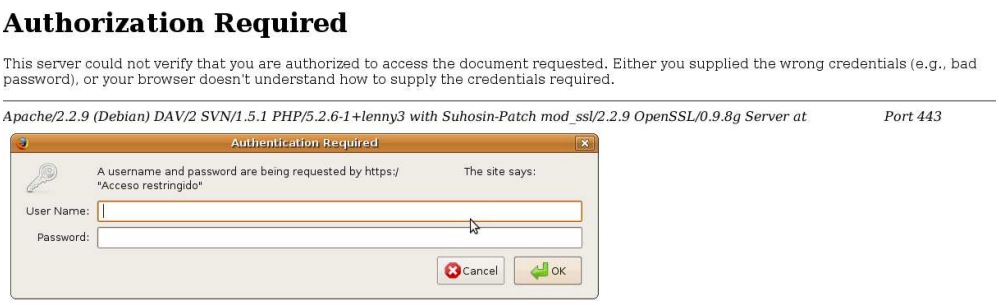

Figura 5: Ventana de autenticación HTTP.

```
# htpasswd -c /etc/apache2/password-www andres
New password:
Re-type new password:
Adding password for user andres
# cat /etc/apache2/password-www
andres:cI9hPLPWdeMRg

# htpasswd -m -c /etc/apache2/password-www andres
New password:
Re-type new password:
Adding password for user andres

# cat /etc/apache2/password-www
andres:$apr1$lpcsH/..$SmFevBck5JDdCB3O3fZ9E1
```

Configurar el Apache2 y probar la autenticación.

```
# cat /etc/apache2/apache2.conf
...
LoadModule auth_basic_module /usr/lib/apache2/modules/mod_auth_basic.so
LoadModule authn_file_module /usr/lib/apache2/modules/mod_authn_file.so
LoadModule authz_user_module /usr/lib/apache2/modules/mod_authz_user.so

...
  <Directory /var/www/html/restrict>
                AuthType Basic
                AuthName "Restrict"
                AuthUserFile  /etc/apache2/password-www
                #Require user  pepe
                #Require group uno
                Require valid-user
   </Directory>
```

```
...

# /etc/init.d/apache2 restart

# cp /var/www/html/index.html /var/www/html/restrict/

# tcpdump -n -i lo port 80 -s 1500 -w 12-http1.0-access.pcap

? ./mozilla http://www/restrict
? ./mozilla http://www/restrict/index.html
? ./mozilla http://www/restrict/otro.html
```

3.2.17. Ejemplo de HTTP/1.0 con conexiones persistentes

En HTTP/1.0 no se contemplaron las conexiones persistentes, es decir, no abrir y cerrar la conexión TCP por cada transacción. A partir de HTTP/1.1 [RFC-2068], se contemplan este tipo de conexiones y, en la actualidad, es la forma de trabajar por default. Para HTTP/1.0, el default es sin conexiones persistentes, pero éstas se pueden negociar explícitamente mediante los headers HTTP y pueden utilizarse.

A continuación se muestra como probar conexiones persistentes en HTTP 1.0. Si no se usan las conexiones persistentes, por cada reload del Mozilla se ve que se establece una nueva conexión.

```
? mozilla http://www

? netstat -atn | grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:59142         127.0.0.1:80            TIME_WAIT
tcp        0      0 127.0.0.1:80            127.0.0.1:59138         TIME_WAIT
tcp        0      0 127.0.0.1:80            127.0.0.1:59140         TIME_WAIT
tcp        0      0 127.0.0.1:80            127.0.0.1:59139         TIME_WAIT
tcp        0      0 127.0.0.1:80            127.0.0.1:59141         TIME_WAIT
tcp        0      0 127.0.0.1:80            127.0.0.1:59137         TIME_WAIT
```

Además, si se usa una página con referencias a otros objetos, cada referencia es una nueva conexión.

```
# tcpdump -n -i lo port 80 -s 1500 -w 13-http1.0-image-mozilla.pcap

# cat /var/www/html/page.html
<HTML>
<TITLE> TEST </TITLE>
<BODY>
  <IMG SRC="firefox.jpg" />
</BODY>
</HTML>

? mozilla http://www/page.html
```

Para trabajar con conexiones HTTP persistentes se las debe habilitar en el servidor.

```
# cat /etc/apache2/apache2.conf
...
KeepAlive On
### Se recomienda valor alto para mejor rendimiento, ilimitado = 0
MaxKeepAliveRequests 100
### Números de segundos para dar con tiempo fuera una conexión permanente
KeepAliveTimeout 15
...
#BrowserMatch "Mozilla/5" nokeepalive downgrade-1.0 force-response-1.0
BrowserMatch "Mozilla/5" downgrade-1.0
...

# /etc/init.d/apache2 restart
```

Ahora, al realizar la conexióm vía `telnet(1)` se puede observa que la conexión permanece abierta entre varios requerimientos consecutivos.

```
? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET /index.html HTTP/1.0
Connection: Keep-Alive
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
Accept: */*

HTTP/1.1 200 OK
Date: Tue, 22 Apr 2008 01:31:56 GMT
Server: Apache/2.2.4 (Ubuntu)
Last-Modified: Mon, 21 Apr 2008 00:18:14 GMT
ETag: "a3b36-1f-91d5d80"
Accept-Ranges: bytes
Content-Length: 31
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html

<HTML>
<H1> HOLA </H1>
</HTML>
GET /index.html HTTP/1.0
Connection: Keep-Alive
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
```

```
Accept: */*

HTTP/1.1 200 OK
Date: Tue, 22 Apr 2008 01:32:14 GMT
Server: Apache/2.2.4 (Ubuntu)
Last-Modified: Mon, 21 Apr 2008 00:18:14 GMT
ETag: "a3b36-1f-91d5d80"
Accept-Ranges: bytes
Content-Length: 31
Keep-Alive: timeout=15, max=99
Connection: Keep-Alive
Content-Type: text/html

<HTML>
<H1> HOLA </H1>
</HTML>
Connection closed by foreign host.

? netstat -atn | grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
```

Lo mismo sucede si se permiten conexiones persistentes desde el cliente Mozilla.

```
# tcpdump -n -i lo port 80 -s 1500 -w 14-http1.0-persist-mozilla.pcap

? ./mozilla http://www/

? netstat -atn | grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:60950         127.0.0.1:80            ESTABLISHED
tcp        0      0 127.0.0.1:80            127.0.0.1:60950         ESTABLISHED

Varios Reload

? netstat -atn | grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:60950         127.0.0.1:80            ESTABLISHED
tcp        0      0 127.0.0.1:80            127.0.0.1:60950         ESTABLISHED
```

Luego de un tiempo sin utilizar.

```
? netstat -atn | grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:80            127.0.0.1:60952         TIME_WAIT
```

Para cerrar, se puede ver como es el request de otro cliente HTTP. En este caso, un cliente no gráfico.

```
# tcpdump -n -i lo port 80 -s 1500 -w 15-http1.0-lynx.pcap

? lynx http://www/
```

3.3. HTTP 1.1

HTTP/1.1 se establece en 1997 con la [RFC-2068] y, más tarde, se actualiza con [RFC-2616]. De esta versión se puede hacer notar como importantes:

- Nuevos mensajes HTTP 1.1: `OPTIONS, TRACE, CONNECT`.
- Conexiones persistentes por omisión.
- Pipelining.

3.3.1. Ejemplo GET HTTP/1.1 con conexiones persistentes

Primero, reconfigurar Apache2 para trabajar con conexiones persistentes.

```
# cat /etc/apache2/apache2.conf
...
#LoadModule setenvif_module /usr/lib/apache2/modules/mod_setenvif.so
#BrowserMatch "Mozilla/5" nokeepalive downgrade-1.0 force-response-1.0
#BrowserMatch "Mozilla/5" downgrade-1.0
...
KeepAlive On
### Se recomienda valor alto para mejor rendimiento 0=ilimitado
###MaxKeepAliveRequests 100
### Números de segundos para dar con tiempo fuera una conexión permanente
###KeepAliveTimeout 15
...

# /etc/init.d/apache2 restart
```

Parece observarse que los servers con HTTP 1.1 son más exigente en el formato del Request. Si se realiza el primer requerimiento con HTTP/1.0 responde OK.

```
? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 22 Apr 2008 03:07:23 GMT
Server: Apache/2.2.4 (Ubuntu)
Last-Modified: Mon, 21 Apr 2008 00:18:14 GMT
ETag: "a3b36-1f-91d5d80"
```

```
Accept-Ranges: bytes
Content-Length: 31
Connection: close
Content-Type: text/html

<HTML>
<H1> HOLA </H1>
</HTML>
Connection closed by foreign host.
```

Pero, si el mismo se realiza en HTTP/1.1 no.

```
? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET / HTTP/1.1

HTTP/1.1 400 Bad Request
Date: Tue, 22 Apr 2008 03:05:33 GMT
Server: Apache/2.2.4 (Ubuntu)
Content-Length: 294
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
<hr>
<address>Apache/2.2.4 (Ubuntu) Server at www Port 80</address>
</body></html>
Connection closed by foreign host.
```

Si se acomoda el encabezado responde OK.

```
telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET / HTTP/1.1
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
```

```
HTTP/1.1 200 OK
Date: Sun, 27 Apr 2008 22:03:15 GMT
Server: Apache/2.2.4 (Ubuntu)
Last-Modified: Mon, 21 Apr 2008 00:18:14 GMT
ETag: "a3b36-1f-91d5d80"
Accept-Ranges: bytes
Content-Length: 31
Content-Type: text/html

<HTML>
<H1> HOLA </H1>
</HTML>
Connection closed by foreign host.
```

Primero, probar conexiones persistentes con `firefox(1)`. En este caso se usa Firefox 2.0.0.13.

```
# tcpdump -n -i lo port 80 -s 1500 -w 16-http1.1-firefox.pcap

? firefox http://www/page.html

    Si se hacen varios varios reload.

? netstat -atn | grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:80            127.0.0.1:59087         ESTABLISHED
tcp        0      0 127.0.0.1:59087         127.0.0.1:80            ESTABLISHED
```

Ahora, probar las conexiones persistentes con `telnet(1)`.

```
# tcpdump -n -i lo port 80 -s 1500 -w 17-http1.1-telnet.pcap

?  telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET /index.html HTTP/1.1
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
Accept: */*

HTTP/1.1 200 OK
Date: Tue, 22 Apr 2008 02:41:00 GMT
Server: Apache/2.2.4 (Ubuntu)
Last-Modified: Mon, 21 Apr 2008 00:18:14 GMT
ETag: "a3b36-1f-91d5d80"
Accept-Ranges: bytes
```

```
Content-Length: 31
Content-Type: text/html

<HTML>
<H1> HOLA </H1>
</HTML>
GET /index.html HTTP/1.1
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com
Accept: */*

HTTP/1.1 200 OK
Date: Tue, 22 Apr 2008 02:41:05 GMT
Server: Apache/2.2.4 (Ubuntu)
Last-Modified: Mon, 21 Apr 2008 00:18:14 GMT
ETag: "a3b36-1f-91d5d80"
Accept-Ranges: bytes
Content-Length: 31
Content-Type: text/html

<HTML>
<H1> HOLA </H1>
</HTML>
Connection closed by foreign host.
```

3.3.2. Ejemplo GET HTTP/1.1 con Pipelining

Otra facilidad que se agrega a partir de HTTP/1.1 es el *Pipelining* de conexiones. Esto se refiere a no esperar a que me llegue la respuesta a un requerimiento para realizar el próximo. Esta opción no parece estar disponible con HTTP/1.0. Esto, solamente, se puede usar con conexiones persistentes.

```
# tcpdump -n -i lo port 80 -s 1500 -w 17b-http1.1-pipe-mozilla.pcap

# cat /var/www/html/page2.html
<HTML>
<TITLE> TEST </TITLE>
<BODY>
  <IMG SRC="firefox.jpg" />
  <IMG SRC="linux.jpg" />
</BODY>
</HTML>

? ./mozilla http://www/page2.html
```

Y sin esta opción:

```
# tcpdump -n -i lo port 80 -s 1500 -w 17c-http1.1-pipe-mozilla.pcap
```

3.3.3. Ejemplo TRACE HTTP/1.1

Esta operación es usada para hacer un debugging. El servidor debe copiar el requerimiento en la respuesta al cliente. De esta forma, el cliente puede observar que es lo que está recibiendo el servidor. Es muy útil cuando se utiliza a través de proxies.

```
# tcpdump -n -i lo port 80 -s 1500 -w 23-http1.1-trace.pcap

?  telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
TRACE /page2.html HTTP/1.1
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com

HTTP/1.1 200 OK
Date: Sun, 27 Apr 2008 22:23:28 GMT
Server: Apache/2.2.4 (Ubuntu)
Transfer-Encoding: chunked
Content-Type: message/http

59
TRACE /page2.html HTTP/1.1
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com

0

Connection closed by foreign host.
```

3.3.4. Ejemplo OPTIONS HTTP/1.1

Este comando sirve para consultar las opciones de operaciones/métodos que ofrece un server sobre un documento.

```
# tcpdump -n -i lo port 80 -s 1500 -w 24-http1.1-options.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
OPTIONS / HTTP/1.1
User-Agent: telnet/andres (GNU/Linux)
Host: estehost.com

HTTP/1.1 200 OK
```

```
Date: Sun, 27 Apr 2008 22:51:07 GMT
Server: Apache/2.2.4 (Ubuntu)
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Length: 0
Content-Type: text/html
Connection closed by foreign host.
```

3.3.5. Otros ejemplos: Redirects/CGIs/Javascript

A continuación se muestran otras características que no son propias de HTTP 1.1, ya que también son soportadas por la versión anterior.

Redirects

Los redirect son mensajes que se utilizan para enviar al user-agent (browser) a otra ubicación debido a que el recurso ha sido movido.

```
# tcpdump -n -i lo port 80 -s 1500 -w 18-http1.1-redirect.pcap

# cat /etc/apache2/apache2.conf
...
LoadModule alias_module /usr/lib/apache2/modules/mod_alias.so

NameVirtualHost *
<VirtualHost *>
        ServerName  www
        DocumentRoot /var/www/html

        RedirectMatch ^/$ http://www2

...

# /etc/init.d/apache2 restart
```

En este caso se realiza un Redirect temporal 302, indicando la nueva URL/URI, donde se encuentra la respuesta. Se puede enviar un POST como requerimiento, pero la nueva URL debe ser alcanzada con un GET. El primer requerimiento podría activar el script o CGI del lado del servidor. La nueva URL no es un reemplazo de la consulta en el primer requerimiento. Este tipo de respuesta no debe ser cacheada, si la obtenida a partir del segundo GET (redirigido). La respuesta debe darse en el campo `Location` y/o en un HTML corto. El user-agent no debería re-direccionarlo salvo que el usuario confirme, aunque varios browser lo hacen. Otra respuesta de Redirect temporal, pero que permite que el user-agent re-direccione es la 303 [RFC-2616].

```
302 Found:
Note: RFC 1945 and RFC 2068 specify that the client is not allowed
 to change the method on the redirected request.  However, most
 existing user agent implementations treat 302 as if it were a 303
 response, performing a GET on the Location field-value regardless
 of the original request method. The status codes 303 and 307 have
```

```
      been added for servers that wish to make unambiguously clear which
      kind of reaction is expected of the client.

     303 See Other:
     Note: Many pre-HTTP/1.1 user agents do not understand the 303
      status. When interoperability with such clients is a concern, the
      302 status code may be used instead, since most user agents react
      to a 302 response as described here for 303.
```

Cuando una URL/URI es reemplazada permanentemente por otra URL/URI se debería utilizar un mensaje `301 Moved Permanently`. En este caso se indica que cualquier acceso futuro debe realizarse sobre la nueva ubicación. Para este caso el user-agent tampoco debería re-direccionar automáticamente aunque si se hace. Este mensaje es comúnmente usado cuando se genera un hostname, donde reside un servidor web, sin y con el prefijo +www+ en el nombre. Es importante marcar que ambos sean el mismo y se utilice una redirección permanente, si no lo fuesen podrían surgir problemas, por ejemplo con las Cookies. Este mensaje es útil también para los motores de búsqueda ya que pueden presentar la URL final.

```
...
        RedirectMatch permanent ^/$ http://www2
        Redirect 303 /otro http://www3
...

# /etc/init.d/apache2 restart

Canonical URIs are good. Multiple URIs for the same resource are bad. Period.
```

CGIs (Common Gateway Interface)

Una CGI es una aplicación que interactúa con un servidor Web para obtener información de los requerimiento generados por el cliente y así poder responder. Existe un estándar del W3C para las CGIs. De esta manera, se le agrega cierto dinamismo a los sitios que trabajan en la WWW. A continuación se muestran varios ejemplos. Para que Apache2 permita los CGIs, se lo debe reconfigurar.

```
# mkdir /var/www/cgi-bin/

# cat  /etc/apache2/apache2.conf
...
####################################
LoadModule cgi_module /usr/lib/apache2/modules/mod_cgi.so
ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"
ScriptLog /var/log/apache2/cgi-error.log

<Directory "/var/www/cgi-bin">
 AllowOverride None Options None
 Order Allow,Deny
 Allow from all
</Directory>
```

```
###################################
...
```

El cliente puede comparar lo que envía con lo que recibe.

También, se crean algunos programas CGI en Bash o Perl.

```
# cat /var/www/cgi-bin/sample.cgi
#!/bin/bash

echo "Content-type: text/html"
echo

echo "<HTML>"
DATE=‘date‘
echo "<H1> Hi: $REMOTE_HOST </H1>"
echo "<H1> CGI executed: on $DATE </H1>"
echo "</HTML>"
```

```
# cat /var/www/cgi-bin/vars.cgi
#!/bin/bash

echo "Content-type: text/html"
echo

echo "<HTML>"
echo "<UL>"
echo "<LI>$QUERY_STRING"
echo "<LI>$SERVER_PORT"
echo "<LI>$SERVER_PROTOCOL"
echo "<LI>$REQUEST_METHOD"
echo "<LI>$SERVER_SOFTWARE"
echo "<LI>$REMOTE_IDENT"
echo "<LI>$HTTP_USER_AGENT"
echo "</UL>"
echo "</HTML>"
```

```
# cat /var/www/cgi-bin/vars.pl
#!/usr/bin/perl

use CGI;

printf ("Content-type: text/html\n");
printf ("\n");

$query=new CGI;
$query->import_names('vars');
```

```
print "<HTML>";
print "<H1> $vars::var1 $vars::var2 </H1>";
print "</HTML>";
```

```
# cat /var/www/cgi-bin/vars2.cgi
#!/bin/bash

echo "Content-type: text/html"
echo

read INPUT
echo "<HTML>"
echo "<UL>"
echo "<LI>$QUERY_STRING"
echo "<LI>$SERVER_PORT"
echo "<LI>$SERVER_PROTOCOL"
echo "<LI>$REQUEST_METHOD"
echo "<LI>$SERVER_SOFTWARE"
echo "<LI>$REMOTE_IDENT"
echo "<LI>$HTTP_USER_AGENT"
echo "<LI>$INPUT"
echo "</UL>"
echo "</HTML>"
```

Por último, se le debe dar permiso de ejecución a estos programas.

```
# chmod +x /var/www/cgi-bin/*
```

Los programas se invocan desde un formulario HTML.

```
# cat /var/www/html/formget.html
<html>
<form  name="in" action="cgi-bin/vars2.cgi" method="GET">
        <input type="text" name="var1" value="uno">
        <input type="text" name="var2" value="dos">
        <input type="submit" value="Send Info">
</form>
</html>
```

```
# cat /var/www/html/formpost.html
<html>
<form  name="in" action="cgi-bin/vars2.cgi" method="POST">
        <input type="text" name="var1" value="uno">
        <input type="text" name="var2" value="dos">
        <input type="submit" value="Send Info">
```

```
</form>
</html>
```

Probar mandando información con `telnet(1)` usando un `GET`. Como se ve, la CGI obtiene la información de una variable de entorno llamada `QUERY_STRING`.

```
# tcpdump -n -i lo port 80 -s 1500 -w 19-http1.1-telnetget.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
GET /cgi-bin/vars2.cgi?var1=uno&var2=dos HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 22 Apr 2008 03:04:10 GMT
Server: Apache/2.2.4 (Ubuntu)
Connection: close
Content-Type: text/html

<HTML>
<UL>
<LI>var1=uno&var2=dos
<LI>80
<LI>HTTP/1.0
<LI>GET
<LI>Apache/2.2.4 (Ubuntu)
<LI>
<LI>
<LI>
</UL>
</HTML>
Connection closed by foreign host.
```

Lo mismo pero usando el navegador. Usar `formget.html`.

```
# tcpdump -n -i lo port 80 -s 1500 -w 20-http1.1-formget.pcap

? firefox http://www/formget.html

* var1=uno&var2=dos
* 80
* HTTP/1.1
* GET
* Apache/2.2.4 (Ubuntu)
* Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.13) Gecko/20080325 \
  Ubuntu/7.10 (gutsy) Firefox/2.0.0.13
*
```

Ahora, realizar el requerimiento usando un `POST`. Usar `formpost.html`.

```
? mozilla http://www/formput.html

*
* 80
* HTTP/1.1
* POST
* Apache/2.2.4 (Ubuntu)
* Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.13) Gecko/20080325 \
  Ubuntu/7.10 (gutsy) Firefox/2.0.0.13
* var1=uno&var2=dos
```

Ver que en este caso, los datos los obtiene leyendo la entrada estándar (STDIN). La salida, en ambos casos, la comunica mediante la salida estándar (STDOUT). Estos métodos están definidos en el estándar del W3C: CGI 1.1. Todo el procesamiento de CGI es realizado del lado del servidor: **server-side**, el cliente solo genera los datos y los envía para que luego sean procesados. Los programas CGI mayormente se los relaciona con código binario, compilado previamente, aunque los ejemplos mostrados son sobre lenguajes de script. La evolución de los CGI a dado nuevos lenguajes de scripting que ejecutan del lado del servidor interpretando y/o compilando al vuelo los programas a ejecutar, interactuando como CGI o módulo binario con el servidor web. Un caso es PHP (PHP Hypertext Pre-processor), diseñado originalmente a partir de Perl para generar contenido web de forma dinámica. Este lenguaje fue creado en 1994. Otros lenguajes o sistemas como ASP (Active Server Page), JSP (Java Server Page) existen con el mismo propósito.

Ejemplo con Server-Side Script

Para que Apache2 permita PHP, se lo debe reconfigurar.

```
# cat  /etc/apache2/apache2.conf
...
DirectoryIndex index.htm index.html index.cgi index.pl index.php index.xhtml index.htm
...
####################################
LoadModule php5_module /usr/lib/apache2/modules/libphp5.so

AddType application/x-httpd-php .php .phtml .php3
AddType application/x-httpd-php-source .phps
####################################
...
```

También, se crean algunos scripts PHP. Uno que trabaja con POST y otro con GET. Uno se escribe como un script completo en PHP y otro con sentencias embebidas en HTML. Estos scripts leen variables de un formulario y las muestran en la salida en HTML.

```
# cat /var/www/html/formpost-php
root@laplata:/var/www# cat html/formpost-php.html
<html>
<form  name="in" action="vars2.php" method="POST">
```

```
        <input type="text" name="var1" value="uno">
        <input type="text" name="var2" value="dos">
        <input type="submit" value="Send Info">
</form>
</html>
```

```
# cat /var/www/html/formpost-php-embeded.html
<html>
<form  name="in" action="vars2-embeded.php" method="GET">
        <input type="text" name="var1" value="uno">
        <input type="text" name="var2" value="dos">
        <input type="submit" value="Send Info">
</form>
</html>
```

```
# cat /var/www/html/vars2.php
<?php
echo "<H1> TEST PHP </H1>";
echo "<B>VAR1:</B>".$_REQUEST["var2"];
echo "</BR>";
echo "<B>VAR2:</B>".$_REQUEST["var1"];
?>
```

```
# cat /var/www/html/vars2-embeded.php
<HTML>
<H1> TEST PHP </H1>
<B>VAR1:</B> <?php echo $_GET["var1"]; ?>
</BR>
<B>VAR2:</B> <?php echo $_GET["var2"]; ?>

</HTML>
```

Para probarlos:

```
? mozilla http://www/formpost-php.html

? mozilla http://www/formpost-php-embeded.html
```

Ejemplo con Client-Side Script

De la misma forma que existen lenguajes de script que ejecutan del lado del servidor, los hay que lo hacen del lado del cliente, dentro del navegador/browser. El primer más popularizado fue JavaScript. Este fue creado por Netscape Communications, y bautizado en un principio como Mocha y LiveScript. Luego en un convenio entre Sun Microsystems y Netscape se lo rebautizó como JavaScript (pues tiene similitudes de sintaxis al lenguaje Java de Sun), hoy ya es un estándar. El W3C ha definido un modelo de objetos conocido como DOM (Document Object Model), sobre el cual trabaja el lenguaje Javascript y lo pueden hacer otros, como JScript o VBScript. A continuación se muestra un ejemplo ilustrativo.

```
# cat /var/www/html/formpost-java.html
<html>
<head>
<!-- script type="text/javascript" src="js.js" -->
<script language="JavaScript" type="text/javascript">
<!--
function printVars ( form )
{
  // Imprime las variables del formulario

   alert("VAR1; "+form.var1.value+" VAR2: "+form.var2.value);
   document.write("VARIABLES IMPRESAS");
   return false;
}
//-->
</script>
<noscript>
 <p>NO tiene soporte de JavaScript en su navegador</p>
</noscript>
</head>
<body>
<form  name="in" action="#" method="POST" onsubmit="return printVars(this);">
        <input type="text" name="var1" value="uno">
        <input type="text" name="var2" value="dos">
        <input type="submit" value="Send Info">
</form>
</body>
</html>
```

3.3.6. Cookies

Las Cookies [COOKIE] son un mecanismo que permite a las aplicaciones web desde el servidor (como CGIs o scripts) almacenar y leer información en el lado del cliente, de esta forma poder manejar de forma sencilla estados persistentes. Esta información también puede ser consultada o generada por aplicaciones del lado del cliente como por ejemplo desde JavaScript. Las Cookies fueron inventadas por Nestcape Communicator. Habitualmente son utilizadas para:

- Autenticación por aplicación.
- Carritos de compras (shopping carts).
- Preferencias, recomendaciones.
- Estado de sesión de usuario (user session state, e.g. web-email).

El servidor cuando retorna un recurso (un objeto HTTP, como una página HTML) puede solicitar al cliente que almacene de forma temporal información de estado. Este estado es almacenado en el cliente y asociado con una fecha de expiración. El cliente en los sucesivos requerimientos al mismo servidor enviará

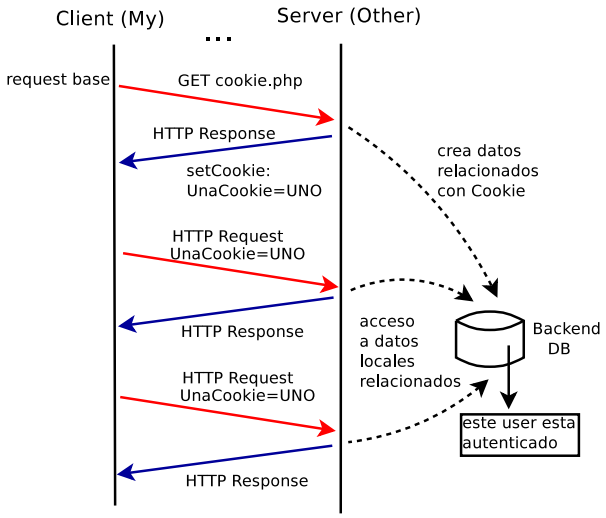

Figura 6: Flujo de una Cookie.

estas Cookies para que el otro extremo las pueda utilizar. Las cookies se almacenan por site y puede haber varias por c/u.
Mediante este agregado se provee a un protocolo sin estados la posibilidad de simularlos permitiendo generar aplicaciones web más complejas, por ejemplo aplicaciones de venta on-line, manejo de autenticación y preferencias de usuarios.
Una Cookie es introducida al cliente mediante el mensaje en el header `Set-Cookie:` de una respuesta HTTP. Este mensaje indica un par (nombre,valor) e incluye una fecha de expiración. El cliente en cada requerimiento luego de haber almacenado la Cookie se la enviará al servidor con el header `Cookie:`.
A continuación se muestra como crear una Cookie con PHP y poder consultarla. Luego crear con JavaScript. Por último un muy sencillo ejemplo de manejo de sesiones en PHP. Tener en cuenta que el código de creación o borrado de Cookies debe estar al principio, sino se generan problemas con el cambio de Header del documento.

```
# cat /var/www/html/cookie.php
<?php
// Poner al principio sino no funciona
$value = "UNO";
setcookie("UnaCookie", $value, time()+3600);  /* expira en 1 hora */
?>
<html>
<head>
<title>Cookie Test - Crear</title>
</head>
<body>
<a href='ver-cookie.php'>Ver Cookie</a>
<a href='del-cookie.php'>Borrar Cookie</a>
</body>
</html>
```

```
# cat /var/www/html/ver-cookie.php
<html>
<head>
<title>Cookie Test - Ver</title>
</head>
<body>
<B>Cookie:</B> <?php echo $_COOKIE["UnaCookie"]; ?>
</body>
</html>

# cat /var/www/html/del-cookie.php
<?php
// Poner al principio sino no funciona
setcookie("UnaCookie", "" , time()-3600);  /* Ya exipro */
?>
<html>
<head>
<title>Cookie Test - Borrar</title>
</head>
<body>
<B> Cookie Borrada</B>
</body>
</html>
```

Generando las conexiones y capturando el tráfico.

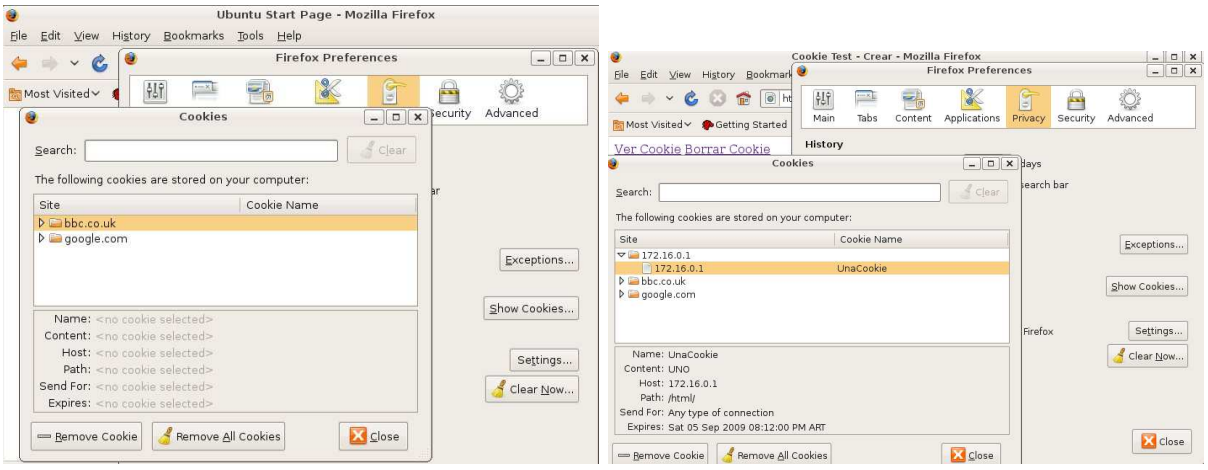

Figura 7: Cookies en el navegador.

```
? firefox http://172.16.0.1/html/cookie.php

# tcpdump -n -i br0 port 80 -s 1500 -w 29-php-cookie-OK.pcap
```

En la captura se ven los mensajes en el encabezado.

```
GET /html/cookie.php HTTP/1.1
Host: 172.16.0.1
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.13)
Gecko/2009080315 Ubuntu/8.10 (intrepid) Firefox/3.0.13
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive

HTTP/1.1 200 OK
Date: Sat, 05 Sep 2009 17:12:09 GMT
Server: Apache/2.2.8 (Ubuntu) PHP/5.2.4-2ubuntu5.7 with Suhosin-Patch
X-Powered-By: PHP/5.2.4-2ubuntu5.7
Set-Cookie: UnaCookie=UNO; expires=Sat, 05-Sep-2009 18:12:09
Content-Length: 164
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html

<html>
<head>
<title>Cookie Test - Crear</title>
</head>
<body>
<a href='ver-cookie.php'>Ver Cookie</a>
<a href='del-cookie.php'>Borrar Cookie</a>
</body>
</html>

GET /html/ver-cookie.php HTTP/1.1
Host: 172.16.0.1
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.13)
Gecko/2009080315 Ubuntu/8.10 (intrepid) Firefox/3.0.13
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://172.16.0.1/html/cookie.php
Cookie: UnaCookie=UNO
```

```
HTTP/1.1 200 OK
Date: Sat, 05 Sep 2009 17:12:16 GMT
Server: Apache/2.2.8 (Ubuntu) PHP/5.2.4-2ubuntu5.7 with Suhosin-Patch
X-Powered-By: PHP/5.2.4-2ubuntu5.7
Content-Length: 98
Keep-Alive: timeout=15, max=98
Connection: Keep-Alive
Content-Type: text/html

<html>
<head>
<title>Cookie Test - Ver</title>
</head>
<body>
<B>Cookie:</B> UNO</body>
</html>

GET /html/del-cookie.php HTTP/1.1
Host: 172.16.0.1
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.13)
Gecko/2009080315 Ubuntu/8.10 (intrepid) Firefox/3.0.13
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: UnaCookie=UNO

HTTP/1.1 200 OK
Date: Sat, 05 Sep 2009 17:12:24 GMT
Server: Apache/2.2.8 (Ubuntu) PHP/5.2.4-2ubuntu5.7 with Suhosin-Patch
X-Powered-By: PHP/5.2.4-2ubuntu5.7
Set-Cookie: UnaCookie=deleted; expires=Fri, 05-Sep-2008 17:12:23
Content-Length: 106
Keep-Alive: timeout=15, max=97
Connection: Keep-Alive
Content-Type: text/html

<html>
<head>
<title>Cookie Test - Borrar</title>
</head>
<body>
<B> Cookie Borrada</B>
</body>
```

```
</html>
```

El código para JavaScript sería el siguiente:

```
# cat /var/www/html/cookie-java.html
<html>
<head>
<!-- script type="text/javascript" src="js.js" -->
<script language="JavaScript" type="text/javascript">
<!--
function createCookie ()
{
  var date = new Date();
  date.setTime(date.getTime()+(3600000));
  var expires = "; expires="+date.toGMTString();
  var path = "; path=/";
  document.cookie = "OtraCookie=DOS"+expires+path;
  alert("Cookie Creada");
}
//-->
</script>
<noscript>
 <p>NO tiene soporte de JavaScript en su navegador</p>
</noscript>
</head>
<body onLoad="createCookie()" >
</body>
</html>
```

Ejemplo de sesión PHP manejada con Cookies. En el caso de las sesiones PHP el servidor almacena en un pool de sesiones la información y le da una Cookie al cliente para que mantenga una referencia o identificador dentro del pool de sesiones.

```
# cat /var/www/html/cookie-java.html
<html>
<head>
<!-- script type="text/javascript" src="js.js" -->
<script language="JavaScript" type="text/javascript">
root@laplata:/var/www# cat /var/www/html/start.php
<?php
    // Start Session using Cookies
    session_start();
    $_SESSION['user'] = 'andres';
    $_SESSION['id'] = '123';
?>
<html>
<head>
```

```
<title>Session Test</title>
</head>
<body>
<a href='next.php'>Ver variables de session</a>
</body>
</html>

# cat /var/www/html/next.php
<?php
    // Inspecciona Variables de session
    session_start();
    echo "<H1>".$_SESSION['user']."</H1>";
    echo "<H1>".$_SESSION['id']."</H1>";
    $_SESSION['id']= $_SESSION['id']+1;
?>
```

A continuación se realizan las pruebas.

```
? firefox http://172.16.0.1/html/start.php

# tcpdump -n -i br0 port 80 -s 1500 -w 30-php-session.pcap
```

Dentro de la captura se encuentra el seteo de la Cookie.

```
GET /html/start.php HTTP/1.1
Host: 172.16.0.1
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.13)
Gecko/2009080315 Ubuntu/8.10 (intrepid) Firefox/3.0.13
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: UnaCookie=UNO

HTTP/1.1 200 OK
Date: Sat, 05 Sep 2009 18:04:38 GMT
Server: Apache/2.2.8 (Ubuntu) PHP/5.2.4-2ubuntu5.7 with Suhosin-Patch
X-Powered-By: PHP/5.2.4-2ubuntu5.7
Set-Cookie: PHPSESSID=e46039e8d37bc404e5fb4685f33b28a7; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 122
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
```

```
Content-Type: text/html
...
```

3.3.7. Ejemplo PUT HTTP/1.0 o HTTP/1.1

El método `PUT` ya existía en HTTP/1.0. En este caso se muestra un ejemplo muy sencillo de como implementarlo en un servidor Web Apache2. Al servidor se le indica que script/CGI es el encargado de manejar los PUTs. En este ejemplo no se tiene en cuenta la seguridad ni los problemas que puede ocasionar el mal uso de este script, solo se muestra con un objetivo didáctico.
Otra forma de subir archivos es usando el método `POST` y programando una CGI adecuada, por ejemplo, incluyendo este código en el HTML.

```
...
<!-- The data encoding type, enctype, MUST be specified as below -->
<form enctype="multipart/form-data" action="__URL__" method="POST">
    <!-- MAX_FILE_SIZE must precede the file input field -->
    <input type="hidden" name="MAX_FILE_SIZE" value="30000" />
    <!-- Name of input element determines name in $_FILES array -->
    Send this file: <input name="userfile" type="file" />
    <input type="submit" value="Send File" />
</form>
...
```

Subir archivos usando `PUT`. El ejemplo aquí mostrado se obtuvo en base a [PUTs] y está programado en Perl.

```
# cat /var/www/cgi-bin/put.pl
#!/usr/bin/perl

## Check we are using PUT method
if ($ENV{'REQUEST_METHOD'} ne "PUT")
{
   &reply(500, "Request method is not PUT");
}

# Check we got a destination filename
$filename = $ENV{'PATH_TRANSLATED'};
if (!$filename)
{
  &reply(500, "No PATH_TRANSLATED");
}

# Check Length
$clength = $ENV{'CONTENT_LENGTH'};
if (!$clength)
 {
  &reply(500, "Content-Length missing or zero ($clength)");
```

```
}

## Read the content itself
## $toread = $clength;

$content = "";
$toread = $clength;
$content = "";
while ($toread > 0)
{
    $nread = read(STDIN, $data, $clength);
    &reply(500, "Error reading content") if !defined($nread);
    $toread -= $nread;
    $content = $data;
}

## Write it out
open(OUT, "> $filename") || &reply(500, "Cannot write to $filename");
print OUT $content;
close(OUT);

# Everything seemed to work, reply with 204 (or 200). Should reply with 201
# if content was created, not updated.
&reply(204);

exit(0);

#########################################################################
#
# Send back reply to client for a given status.
#

sub reply
{
    local($status, $message) = @_;

    print "Status: $status\n";
    print "Content-Type: text/html\n\n";

    if ($status == 200) {
        print "<HEAD><TITLE>OK</TITLE></HEAD><H1>Content Accepted</H1>\n";
    } elsif ($status == 500) {
        print "<HEAD><TITLE>Error</TITLE></HEAD><H1>Error Publishing File</H1>\n";
        print "An error occurred publishing this file ($message).\n";
    }
    exit(0);
```

```
}

### EOF ###
```

Se configuran los permisos y se configura Apache2 para que responda a los PUTs invocando este CGI.

```
# chmod +x /var/www/cgi-bin/put.pl

# cat /etc/apache2/apache2.conf
...
LoadModule actions_module /usr/lib/apache2/modules/mod_actions.so
...
<VirtualHost *>
        ServerName  www
        DocumentRoot /var/www/html

        <Directory />
                Options FollowSymLinks
                AllowOverride None
                Script PUT /cgi-bin/put.pl
        </Directory>
...

# /etc/init.d/apache2 restart
```

Finalmente, se lo prueba.

```
# tcpdump -n -i lo port 80 -s 1500 -w 22-http1.1-put.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
PUT /otro.txt HTTP/1.0
Host: otrohost.com
Accept-Encoding: gzip, deflate
Content-Type: text/xml
User-Agent: telnet/andres (GNU/Linux)
Content-Length: 10

0123456789
HTTP/1.1 204 No Content
Date: Tue, 22 Apr 2008 04:13:57 GMT
Server: Apache/2.2.4 (Ubuntu)
Content-Length: 0
Connection: close
Content-Type: text/html
```

```
Connection closed by foreign host.

? ls -l /var/www/html/otro.txt
-rw-r--r-- 1 www-data www-data 10 2008-04-22 01:13 /var/www/html/otro.txt
? cat /var/www/html/otro.txt
0123456789
```

La diferencia con el POST es que el archivo puede no existir y el script o CGI que maneja el request es siempre el mismo de forma implícita.

3.3.8. Ejemplo CONNECT HTTP/1.1

El comando `CONNECT` es útil cuando se utiliza HTTP como medio de transporte de una conxión TCP (no necesariamente HTTP) a través de un proxy. Por ejemplo podría utilizarse HTTP para conectarse a un servidor de mail SMTP. A continuación se muetra el ejemplo de configuración de un servidor apache como proxy.

```
...
#####################################
LoadModule proxy_module /usr/lib/apache2/modules/mod_proxy.so
LoadModule proxy_connect_module /usr/lib/apache2/modules/mod_proxy_connect.so
AllowCONNECT 80 25 443
<IfModule mod_proxy.c>
ProxyRequests On
ProxyVia On
<Proxy *>
    Order deny,allow
    #Deny from all
    Allow from 10.168.1.0/24 127.0.0.1
</Proxy>
#####################################
...

# /etc/init.d/apache2 restart
```

Generar una conexión contra un servidor SMTP utilizando un servidor web como proxy.

```
# tcpdump -n -i lo port 80 or port 25 -s 1500 -w 25-http1.1-connect-25.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
CONNECT 127.0.0.1:25
HTTP/1.0 200 Connection Established
Proxy-agent: Apache/2.2.4 (Ubuntu)
```

```
220 khartum ESMTP Postfix (Ubuntu)
HELO otrohost.com
250 khartum
QUIT
221 2.0.0 Bye
Connection closed by foreign host.

? netstat -atn
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
...
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
...
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:54560         127.0.0.1:25            ESTABLISHED
tcp        0      0 127.0.0.1:51597         127.0.0.1:80            ESTABLISHED
tcp        0      0 127.0.0.1:80            127.0.0.1:51597         ESTABLISHED
tcp        0      0 127.0.0.1:25            127.0.0.1:54560         ESTABLISHED
...
```

De la misma forma que se generó una conexión TCP a otro servicio se puede hacer con un servicio web. Por ejemplo generar una conexión web externa utilizando un servidor web como proxy HTTP.

```
# tcpdump -n -i any port 80 -s 1500 -w 26-http1.1-connect-google.pcap

? telnet 127.0.0.1 80
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
CONNECT www.google.com:80
HTTP/1.0 200 Connection Established
Proxy-agent: Apache/2.2.4 (Ubuntu)

GET / HTTP/1.0

HTTP/1.0 302 Found
Location: http://www.google.com.ar/
Cache-Control: private
Set-Cookie: PREF=ID=39204e04fb2ab152:TM=1209339999:LM=1209339999:S=wO6gZq-KCH4pS2v9;
Date: Sun, 27 Apr 2008 23:46:39 GMT
Content-Type: text/html
Server: gws
Content-Length: 222
Connection: Close
```

```
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.com.ar/">here</A>.
</BODY></HTML>
Connection closed by foreign host.
```

3.3.9. Ejemplo de Proxy

Los ejemplos anteriores de Proxy Connect en la realidad son poco comunes. Pueden utilizarse en situaciones como gateway de aplicaciones a modo de firewall, de manera de solo permitir la salida de la red a un equipo en modo bastión. Es posible utilizar un servidor web como proxy HTTP sin necesidad de usar el método `CONNECT`. En este caso el equipo recibirá los requerimientos HTTP desde el navegador, pero incluyendo la URL completa en el `GET`, ya que el proxy debe salir a buscar el recurso solicitado a Internet. A continuación se muestra un ejemplo. Primero se configura el servidor para que actue como proxy.

```
...
####################################
LoadModule proxy_module /usr/lib/apache2/modules/mod_proxy.so
LoadModule proxy_http_module /usr/lib/apache2/modules/mod_proxy_http.so
<IfModule mod_proxy.c>
ProxyRequests On
ProxyVia On
<Proxy *>
    Order deny,allow
    Deny from all
    Allow from 10.168.1.0/24 127.0.0.1
</Proxy>
####################################
...

# /etc/init.d/apache2 restart
```

Luego configurar Mozilla para que utilice un proxy (en este caso local) y hacer el requerimiento HTTP.

```
? ./mozilla http://www.google.com
```

3.4. HTTP y sus agregados, una Continua Evolución

HTTP es el protocolo más difundido de la Internet y por lo tanto es el que más agregados y modificaciones sufre. A menudo no el protocolo mismo, sino los protcolos y sistemas que se basan en este. Algunos ejemplos son y fueron: AJAX, SOAP, web-Services y su derivados.

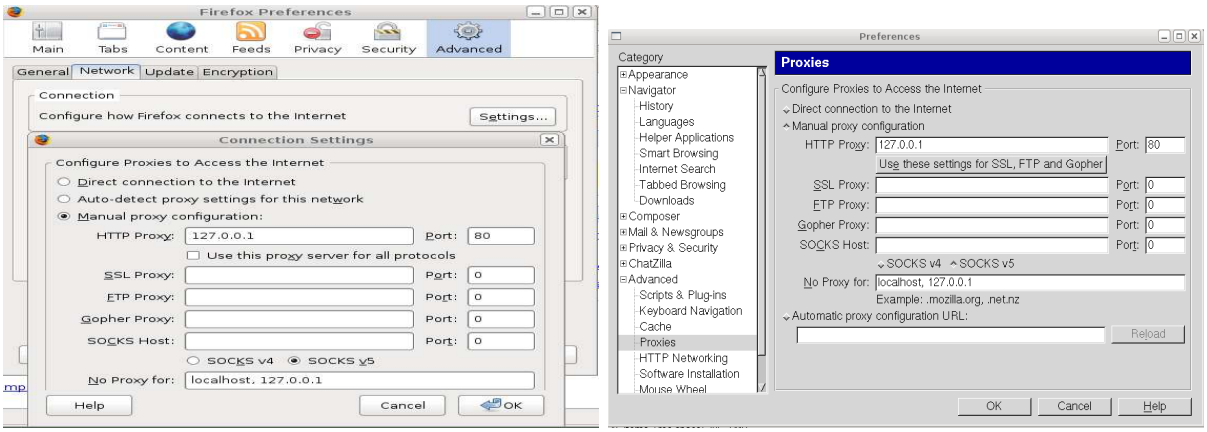

Figura 8: Ejemplos de config. de clientes de Proxy.

3.4.1. AJAX: El Cliente Cobra Vida

AJAX (Asynchronous JavaScript And XML) es una metodología de desarrollo de aplicaciones WEB a partir de la cual no se requiere recargar todo una página para generar un requerimiento y traer o enviar información hacia el servidor web. El cliente mediante scripting, en particular JavaScript, mantiene comunicación asincrónica con el servidor y puede comunicarse enviando y trayendo pequeños grupos de información para luego mostrar en el browser. Hoy existen numerosos frameworks que encapsulan esta funcionalidad brindando una interfaz de programación, API fácil de utilizar. La información que se solicita o envía en general va a estar codificada en XML, y a través de estos mensajes se puede tener un mecanismo de pasaje de mensajes. Ajax es una tecnología asíncrona, en el sentido de que los datos adicionales se requieren al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página. En este texto solo se mostrará un ejemplo sencillo a modo ilustrativo de la tecnología que esta basada sobre HTTP. El objetivo del ejemplo es mostrar como el procesamiento entre servidor y cliente se puede balancear con el objetivo de mejorar la interfaz de usuario. En el ejemplo se hace un requerimiento desde JavaScript del contenido de un archivo `data.txt`, para luego ser mostrado como el contenido de un campo de texto de un formulario. Los requerimientos desde el JavaScript comúnmente obtienen un XML que luego "parsean" en el cliente.

```
# cat /var/www/html/ajax-get.html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" dir="ltr" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

<script language="JavaScript">
<!--
function getDataAJAX()
{
var req = null;
```

```
document.ajaxform.dyntext.value="Started...";
if(window.XMLHttpRequest)
{
        // code for IE7+, Firefox, Chrome, Opera, Safari
req = new XMLHttpRequest();
}
else if (window.ActiveXObject)
{
        // code for IE6, IE5
req  = new ActiveXObject(Microsoft.XMLHTTP);
}
else
{
         //alert("Your browser does not support XMLHTTP!");
         alert("NO AJAX Support");
 return false;
}
// La propiedad "onreadystatechange" almacena la función que procesara el response
// desde el servidor. La función es almacenada en la propiedad y será invocada de forma
// automática
req.onreadystatechange = function()
{
document.ajaxform.dyntext.value="Wait server...";
if(req.readyState == 4)
{
          // Get teh Data from server
  if(req.status == 200)
  {
document.ajaxform.dyntext.value="Received:" + req.responseText;
  }
  else
  {
  // Error code Returned
  document.ajaxform.dyntext.value="Error: returned status code "
                                    + req.status + " " + req.statusText;
  }
        }
}; // end onreadystatechange

// Invoca la función
req.open("GET", "data.txt", true);
req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
req.send(null);
}
//-->
</script>
```

```
</head>

<body>
 <FORM name="ajaxform" method="POST" action="">
   <p><INPUT type="button" value="Submit (Get Data)"  onclick="getDataAJAX()"></p>
   <p><INPUT type="text" name="dyntext" size="32" value=""></p>
 </FORM>
</body>
</html>

# cat /var/www/html/data.txt
Prueba de Hola con AJAX
```

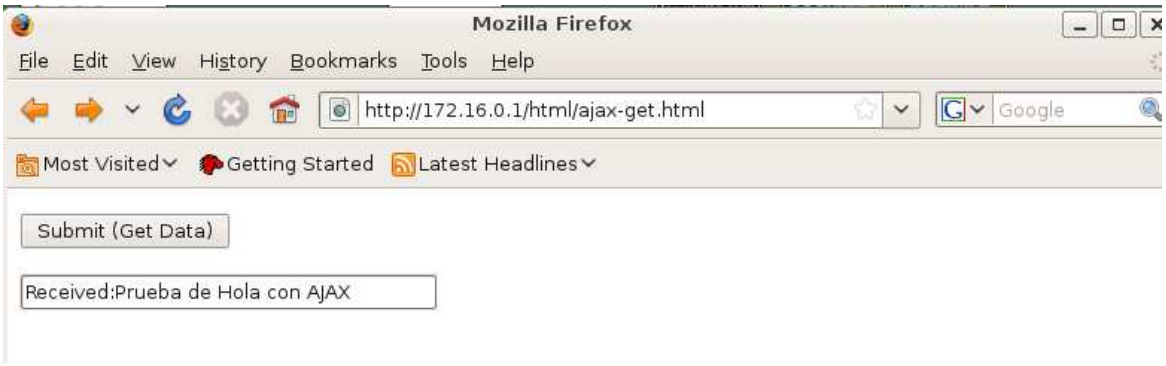

Figura 9: Ejemplo de AJAX.

3.4.2. Servicios sobre HTTP

El protocolo HTTP es utilizado actualmente para implementar servicios y objetos distribuidos en la Web. En un principio surgió la idea de implementar protocolos como RPC sobre la plataforma ofrecida por Internet y HTTP, lo que dio lugar a XML-RPC. A partir de esto derivo en diferentes arquitecturas y protocolos. Uno fue el protocolo de comunicación de objetos llamado SOAP (Simple Object Access Protocol). SOAP es un conjunto de convenciones para implementar RMI (Remote Method Invocation) sobre cualquier capa de transporte que permita texto (en particular HTTP) codificando los requerimientos y las respuestas mediante texto XML. Este fue desarrollado por: DevelopMentor, Microsoft, y UserLand Software. Hoy es un estándar del W3C [SOAP]. Previo a SOAP, como se indico se utilizaba XML-RPC, luego SOAP evoluciono dando lugar a WEB-services definidos por estándares como WSDL (Web Services Description Language) [WSDL] y UDDI (Universal Description Discovery and Integration). Los WEB-services agregaron nuevas definiciones y funcionalidad para completar la idea de SOAP. Como todo diseño e implementaciones en uso, tienen una continua evolución. De esta forma surge REST (Representational state transfer) [REST], que define una arquitectura con ideas de las anteriores pero más escalable, también desarrollada por el W3C. Los web-services y "derivados" como REST se puede implementar con CGIs, o con scripting del lado del servidor como PHP, JSP, ASP, extensiones de Python, Ruby o una infinidad de lenguajes. Siempre respaldados por desarrollos que corren del lado del cliente como AJAXs.

3.4.3. Conclusiones de Servicios sobre la Web

La ventaja de estos diseños e implementaciones es que permiten comunicación mediante puertos y protocolos estándares (habitualmente no filtrados) como http, https, web-cache e incluso SMTP brindando abstracción de como se implementa cada objeto o servicio. Se podría enmarcar todas estas ideas dentro de un diseño de arquitectura SOA(Service-oriented Architecture) [SOA]. Todos estos son temas que trascienden el objetivo de este texto, y, si el lector esta interesado se dejan varias referencias al final del mismo.

3.5. HTTP sobre SSL, HTTPS

Por último, y como cierre de HTTP, se verán ejemplos de la utilización de HTTP sobre una capa de "transporte" que ofrece seguridad, como cifrado y chequeo de identidad. Esta capa en el modelo OSI estaría a nivel presentación y de aplicación: SSL (Secure Sockets Layer). Éste es un protocolo de comunicación seguro que fue el predecesor del estándar TLS (Transport Layer Security). La combinación de HTTP + SSL se lo conoce como HTTPS o HTTP Seguro y corre, habitualmente, sobre el port TCP 443. La capa SSL/TLS no sólo sirve para HTTP, también se puede combinar con otros protocolos que no tienen seguridad integrada como SMTP, FTP, etc.

```
# grep https /etc/services
https           443/tcp                         # http protocol over TLS/SSL
...
```

SSL fue un protocolo inventado por Netscape y hoy conforma uno de los pilares de la seguridad en Internet. Se basa en el principio de claves asimétricas, aunque la mayor parte del cifrado se realiza con criptografía simétrica que es menos costosa. Requiere como parte importante de la infraestructura una jerarquía de CA (Autoridades Certificantes).

3.5.1. Como funciona SSL

En SSL se definen dos roles diferentes para las partes que conforman la conexión. Una será el servidor y otra el cliente. El cliente es quien siempre inicia la comunicación segura. En el caso más común, el web browser es el cliente y el sitio Web es el servidor.

1. El cliente se comunica contra el servidor y requiere una conexión segura presentando la versión de SSL más alta que soporta, una lista de Cipher Suites y algoritmos de compresión que desea usar. Un Cipher Suite está compuesto por tres elementos: un algoritmo de intercambio de claves, un cifrador y una función HMAC (Hash Message Authentication Code). Además, envía un valor aleatorio. Este mensaje se conoce como `ClientHello`.

2. El servidor contesta con tres mensajes, que podrían ir en el mismo segmento TCP. Un `ServerHello` donde indica el Cipher Suite y el algoritmo de compresión que seleccionó el servidor. En otro mensaje envía su certificado y, por último, envía un `ServerHelloDone` para indicar que el handshake inicial terminó. También, envía un número aleatorio, independiente del que recibió del cliente. El formato del certificado es X.509, pero existe la posibilidad que en algún momento se incluya el formato OpenPGP. El servidor podría solicitarle un certificado al cliente, quien se lo enviará después de recibir el `ServerHelloDone`.

3. El cliente chequeará el certificado del servidor de acuerdo a la CA (Certification Authority) que lo firma, al nombre del sitio para el cual se generó (que sea el mismo que la URL al cual se está conectando) y la validez en el tiempo. Luego, extraerá la clave pública del servidor desde el certificado.

4. El cliente genera una clave aleatoria, la cifra con la clave pública del servidor y se la envía. Esta clave se conoce como `PreMasterSecret` y viaja en un mensaje `ClientKeyExchange`. En este caso, se utiliza cifrado asimétrico.

5. El cliente y el servidor utilizan toda la información intercambiada previamente, entre ellos los número aleatorios y la clave `PreMasterSecret`, para computar una clave común secreta llamada `MasterSecret`. A partir de esta clave, tanto el cliente como el servidor, generan las claves de sesión, que son claves simétricas usadas para cifrar y descifrar la información intercambiada durante la sesión SSL y para verificar su integridad.

6. Desde el cliente se envía un mensaje `ChangeCipherSpec` para indicarle al servidor que de ahora en adelante se va a empezar a enviar los datos cifrados. El servidor, a su vez, enviará un mensaje similar.

7. Por último, cliente y servidor intercambian mensajes `Finished` cifrados y con MACs para comprobar mutuamente que el intercambio de claves y la autenticación ha sido exitosa. Estos son los primeros dos mensajes que van cifrados y autenticados con las claves recién negociadas. Estos mensajes se ven como `Encripted Handshake`.

Los mensajes `ClientKeyExchange`, `ChangeCipherSpec` y `Encripted Handshake` se pueden enviar todos juntos.

3.5.2. Características de implementación de SSL/TLS

Los certificados utilizados están en formato X.509. Estos certificados están en formato binario. Si se desea intercambiarlos en ASCII normal, se los pasa a formato denominado PEM (Privacy-enhanced Electronic Mail), formato que permitía cifrar documentos usando DES para intercambiarlos vía e-mail, y luego los trasladaba a ASCII de 7 bits agregando encabezado y pie. Hoy se usa para pasar certificados a formato ASCII estándar.

Los algoritmos de cifrado simétrico utilizados pueden ser RC2, RC4 de 40 bits (para exportación fuera de USA en versiones viejas) o 128 bits, DES de 56 o 3DES de 112 bits, CAST de 64, 80 o 128 bits, IDEA o AES.

Para intercambio de clave se usa RSA de 512, 1024 o 2048 bits. También se puede utilizar DH (Diffie-Hellman) o Digital Signature Algorithm (DSA/DSS). Para 2013 no se recomendaban más las claves de 1024 o menores.

Para MACs se usa MD5, SHA-1 o también podría incluirse RIPE-160. En versiones viejas se usaban algoritmos de hash MD2 y MD4.

A continuación se hace una captura accediendo al site de HomeBanking del Banco de la Nación Argentina.

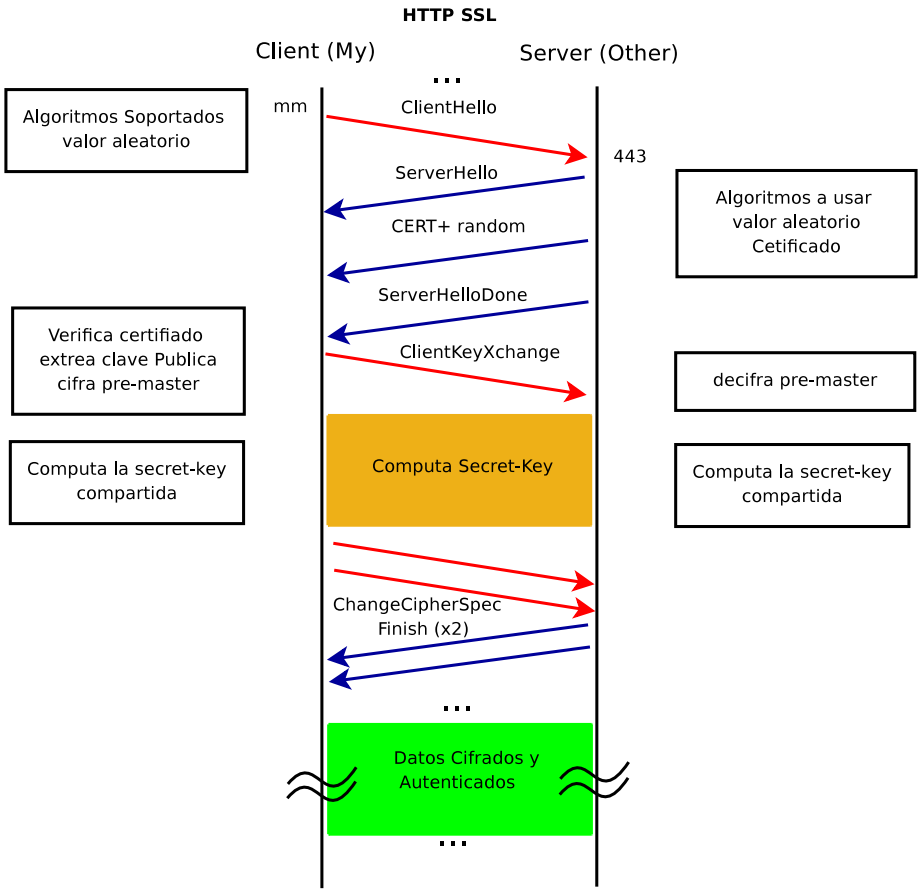

Figura 10: Intercambio de mensajes TLS/SSL.

```
# tcpdump -n -w 28-http-ssl.pcap -s 1500 port 80 or port 443 -i eth0

? firefox https://www.bna.com.ar
```

Se puede agregar este nivel de seguridad a un sitio particular. Para esto es necesario contar con una herramienta para generar el par de claves para el cifrado asimétrico. La clave pública y la privada deberán separarse y la clave pública deberá colocarse en un certificado. Luego, este certificado debe ser firmado por una CA (Certification Authority). Este trámite se debe realizar ante el front-end de la CA, la RA (Registration Authority), la cual corroborá la validez y autenticidad del solicitante y de su certificado. Una vez aprobado por la RA, se firmará con la clave privada de la CA. Los navegadores contienen pre-instalados certificados de las CA más utilizadas o conocidas, por lo que, cuando se navega a un site con un certificado generado por una de esas CAs, el chequeo es automático. Si el navegador no encuentra un certificado de una CA válida, generará un mensaje de advertencia. Habitualmente, la CA cuando otorga el certificado, también genera las claves en un proceso que no comprometa la clave privada generada para la entidad solicitante.

En los ejemplos mostrados se utiliza la herramienta [OpenSSL] sobre GNU/Linux y no se utiliza una

CA externa, sino que se firma con otro certificado propio simulando la CA. La otra opción sería firmar el certificado con la misma clave privada, auto-firma. Primero, se generan las claves privadas RSA de la CA y del Server. En este caso, la del server no se protege con clave con el algoritmo 3DES, si la de la CA. Se crean claves de 1024 bits.

```
# cd /etc/ssl/private/

# mkdir CA
# cd CA/

# openssl genrsa -des3 -out paraguil-ca.key 1024 ## Con password/passphrase
Generating RSA private key, 1024 bit long modulus
........++++++
.................................................++++++
e is 65537 (0x10001)
Enter pass phrase for paraguil-ca.key: *****
Verifying - Enter pass phrase for paraguil-ca.key: *****

# openssl genrsa      -out www-https-srv.key 1024 ## Sin passwd
Generating RSA private key, 1024 bit long modulus
.........++++++
...++++++
e is 65537 (0x10001)

# chmod 400 paraguil-ca.key
# chmod 400 www-https-srv.key
# mv www-https-srv.key ..
```

Luego, se crear un certificado para la CA a partir de la clave privada. El certificado incluye la clave pública. El último comando muestra su contenido.

```
# cd ../../certs/

# openssl req -new -x509 -days 3650 -key ../private/CA/paraguil-ca.key \
  -out paraguil-ca-x509.crt

# openssl x509 -in paraguil-ca-x509.crt -noout -text
```

Luego se crear un Requerimiento de Certificado para ser enviado a la CA Es necesario usar en el CN (Common Name) el nombre del servidor, en este caso “www”.

```
# openssl req -new -key ../private/www-https-srv.key -out www-https-srv.csr
...
Country Name (2 letter code) [AU]:AR
State or Province Name (full name) [Some-State]:Buenos-Aires
Locality Name (eg, city) []:La-Plata
Organization Name (eg, company) [Internet Widgits Pty Ltd]:khartum Test
```

```
Organizational Unit Name (eg, section) []:IT Testing WEB
Common Name (eg, YOUR name) []:www
Email Address []:tester@it.test
...
```

Paso siguiente, se firma el certificado del servidor con el de la CA.

```
# openssl x509 -req -in  www-https-srv.csr -out www-https-srv-x509.crt \
  -sha1 -CA paraguil-ca-x509.crt -CAkey ../private/CA/paraguil-ca.key \
  -CAcreateserial -days 3650
```

Este es el contenido de un certificado. Entre otras cosas se pueden ver quien firmó el certificado, la entidad dueña del certificado, su tiempo de validez, la clave pública de la entidad, etc.

```
# openssl x509 -in  www-https-srv-x509.crt -noout -text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            dc:31:78:3d:95:58:fb:14
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=AR, ST=Buenos-Aires, L=La-Plata, O=CA-Test-paraguil, OU=CA-Testing,
                CN=CA Operator/emailAddress=caoperator@ca-testing.test
        Validity
            Not Before: May 11 23:28:41 2008 GMT
            Not After : May  9 23:28:41 2018 GMT
        Subject: C=AR, ST=Buenos-Aires, L=La-Plata, O=khartum Test, OU=IT Testing WEB,
                 CN=www/emailAddress=tester@it.test
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:b0:e6:96:5c:67:c2:3c:50:bb:62:9e:d9:ee:1c:
                    4c:d8:a1:f4:c6:21:b6:93:83:bc:1c:f1:34:36:41:
                    da:e1:7e:25:87:54:de:fb:f6:56:87:dd:29:9c:5f:
                    7d:05:33:ae:e3:73:7c:e8:24:12:5a:bc:18:cb:0e:
                    19:83:35:c5:5f:49:7e:d1:16:38:5f:65:3b:39:26:
                    ed:55:f1:d3:f1:4b:08:84:ad:f1:47:87:42:ce:6a:
                    3a:ff:ac:9d:1a:1f:71:40:7e:a6:8b:80:65:73:bb:
                    14:eb:6c:b6:4d:27:8c:35:34:18:f3:38:4f:99:18:
                    3f:00:f7:0e:a9:c7:03:92:f1
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha1WithRSAEncryption
        4c:7c:b3:e6:7c:b9:9a:39:33:8a:52:56:56:ab:31:d3:82:a3:
        51:6e:a7:79:d1:7c:6d:ba:a0:42:50:b3:68:2c:c0:c2:10:f8:
        05:fa:2d:6e:b7:bb:5a:18:8c:6f:a7:aa:a0:90:c9:2e:b8:00:
        ab:da:94:ea:d8:e1:08:a4:d4:4c:c4:60:a9:ba:1e:44:4a:cc:
```

```
        b4:6e:13:7c:7b:e9:ff:d4:55:a6:c7:4e:cf:d3:96:f4:1b:09:
        fc:87:29:4a:f3:9e:5a:d7:25:68:6b:c5:f2:ee:0c:70:94:c6:
        57:08:3a:7a:86:0c:bf:ec:5c:b4:22:86:33:f8:5c:f9:df:de:
        80:54

# cat www-https-srv-x509.crt
-----BEGIN CERTIFICATE-----
MIICtTCCAh4CCQDcMXg9lVj7FDANBgkqhkiG9w0BAQUFADCBqDELMAkGA1UEBhMC
QVIxFTATBgNVBAgTDEJ1ZW5vcy1BaXJlczERMA8GA1UEBxMITGEtUGxhdGExGTAX
BgNVBAoTEENBLVRlc3QtcGFyYWd1aWwxEzARBgNVBAsTCkNBLVRlc3RpbmcxFDAS
BgNVBAMTC0NBIE9wZXJhdG9yMSkwJwYJKoZIhvcNAQkBFhpjYW9wZXJhdG9yQGNh
LXRlc3RpbmcudGVzdDAeFw0wODA1MTEyMzI4NDFaFw0xODA1MDkyMzI4NDFaMIGU
MQswCQYDVQQGEwJBUjEVMBMGA1UECBMMQnVlbm9zLUFpcmVzMREwDwYDVQQHEwhM
YS1QbGF0YTEVMBMGA1UEChMMa2hhcnR1bSBUZXN0MRcwFQYDVQQLEw5JVCBUZXN0
aW5nIFdFQjEMMAoGA1UEAxMDd3d3MR0wGwYJKoZIhvcNAQkBFg50ZXN0ZXJAaXQu
dGVzdDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAsOaWXGfCPFC7Yp7Z7hxM
2KH0xiG2k4O8HPE0NkHa4X4lh1Te+/ZWh90pnF99BTOu43N86CQSWrwYyw4ZgzXF
X0l+0RY4X2U7OSbtVfHT8UsIhK3xR4dCzmo6/6ydGh9xQH6mi4Blc7sU62y2TSeM
NTQY8zhPmRg/APcOqccDkvECAwEAATANBgkqhkiG9w0BAQUFAAOBgQBMfLPmfLma
OTOKUlZWqzHTgqNRbqd50XxtuqBCULNoLMDCEPgF+i1ut7taGIxvp6qgkMkuuACr
2pTq2OEIpNRMxGCpuh5ESsy0bhN8e+n/1FWmx07P05b0Gwn8hylK855a1yVoa8Xy
7gxwlMZXCDp6hgy/7Fy0IoYz+Fz5396AVA==
-----END CERTIFICATE-----
```

La alternativa más corta es crear un "Self-Signed Certificate" de la siguiente forma. No es necesario utilizar un CA.

```
# openssl x509 -req -in server.csr -out www-https-srv-self.crt \
          -signkey www-https-srv.key
```

Por último, se configura el Apache y se realizan las pruebas. En el servidor, se debe indicar donde se encuentran los certificados recién emitidos.

```
# less /etc/apache2/apache2.conf

...
NameVirtualHost *:443

Listen 443

######################################
##### SSL

# LoadModule ssl_module modules/mod_ssl.so
LoadModule ssl_module /usr/lib/apache2/modules/mod_ssl.so
# Cipher
SSLCipherSuite HIGH:MEDIUM
```

```
# Here I am allowing SSLv3 and TLSv1,
# I am NOT allowing the old SSLv2.
SSLProtocol all -SSLv2

# Server Certificate
SSLCertificateFile /etc/ssl/certs/www-https-srv-x509.crt

# Server Private Key
SSLCertificateKeyFile /etc/ssl/private/www-https-srv.key

# Server Certificate Chain
SSLCertificateChainFile /etc/ssl/certs/www-https-srv-x509.crt

#   Certificate Authority (CA):
SSLCACertificateFile /etc/ssl/certs/paraguil-ca-x509.crt

# Here, it throws:
# Illegal attempt to re-initialise SSL
# for server (theoretically shouldn't happen!)
#SSLEngine On
#####################################
...

<VirtualHost *:443>
   ServerName www
   DocumentRoot /var/www/html
   SSLEngine On
</VirtualHost>
...

# /etc/init.d/apache2 restart

? firefox https://www
```

3.5.3. Chequeos SSL/TLS desde el cliente

Cuando el programa cliente se conecta hace chequeo de los certificados ofrecidos por el servidor. El programa cliente de acuerdo a los certificados de las autoridades certificantes que tiene instalados validara las credenciales del servidor. También realizará otros chequeos como que el nombre del servidor concuerde con el ofrecido. En las figuras 11 se observan diferentes instancias de los chequeos realizados con el browser Google-Chrome.

3.6. HTTP: el pasado y hoy

Según [StevI], haciendo este referencia a otras fuentes, en 1994 el servicio que tenía más flujos de información era el SMTP, y FTP era el que acarreaba más datos. Para 1998, el protocolo HTTP había

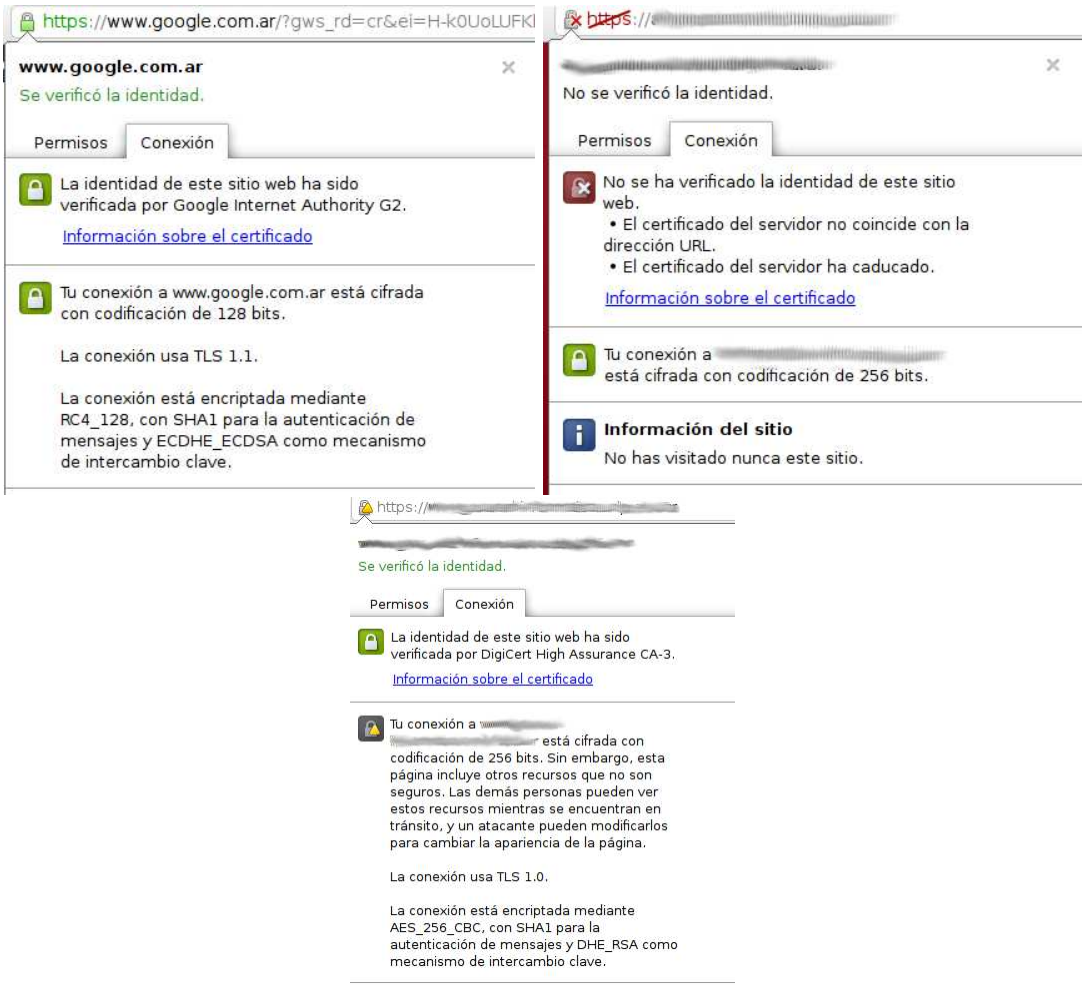

Figura 11: Chequeos de certificados por Google-Chrome

desplazado a ambos, siendo (dentro de los protocolos estándares) hasta hoy en día el dominante sobre la Internet. Para julio de 1998, según [CAIDAtnotb98], el uso de los protocolos estándares es dominado por HTTP por más del 50 %. Hoy, HTTP es utilizado para transferencia de archivos, navegación interactiva, hacer streaming, ver videos [YT], e incluso, para implementar protocolos tipo RPC (Remote Procedure Call) como WEB-Services o REST. La tendencia actual es montar "todo sobre HTTP" (o HTTPS) . HTTP genera mucho overhead comparado con otro protocolos, pero al ser un protocolo ASCII, admite compresión, lo que le permite ser optimizado, con lo que surgen los conceptos de Web-Accelerator. Estos combinan compresión, filtros, caching y otras optimizaciones, como es el enfoque de HTTP/2 [RFC-7540].

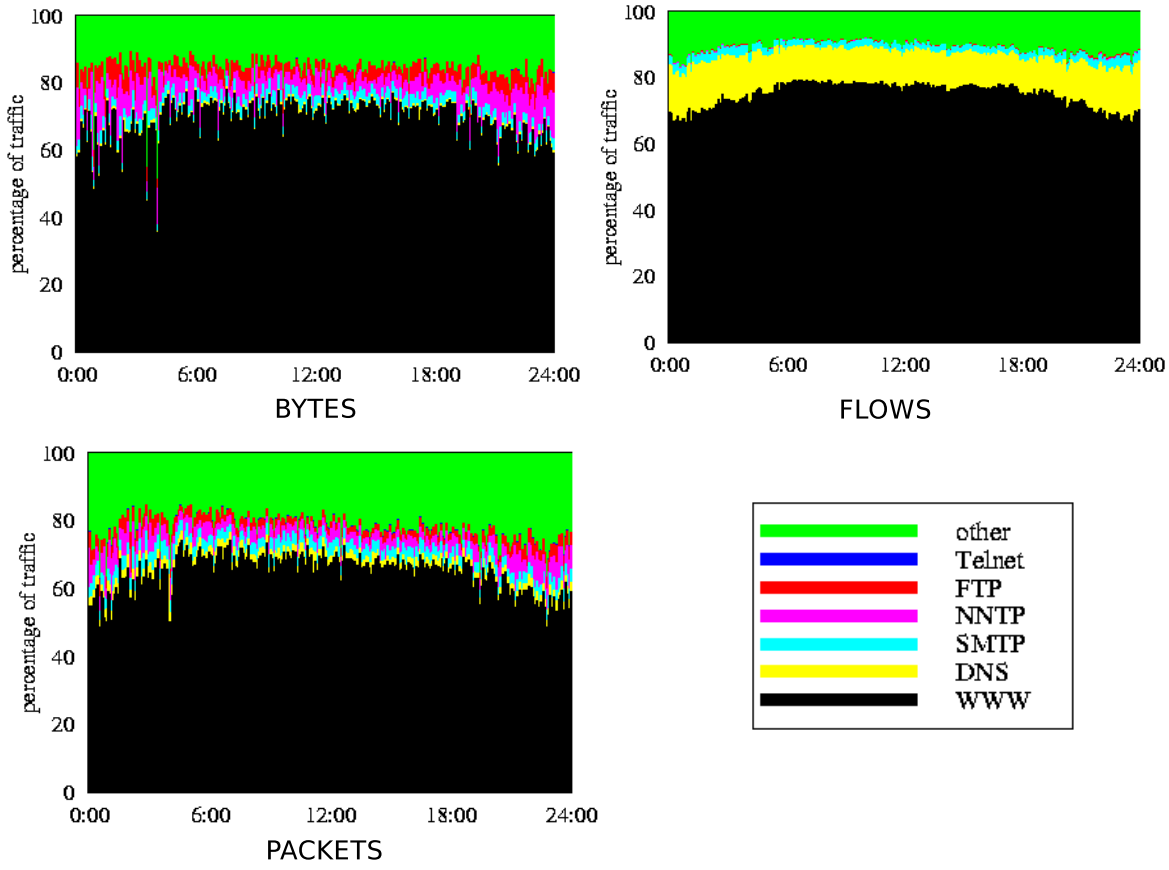

Figura 12: Uso de protocolos de aplicación en Internet en 1998.

Referencias

[StevI] TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley, (2nd. Ed). 2011. Kevin R. Fall, W. Richard Stevens.

[KR] Computer Networking: A Top-Down Approach, Addison-Wesley, (6th Edition). 2012. Kurose/Ross.

[LX] The Linux Home Page: http://www.linux.org/.

[Siever] Linux in a Nutshell, Fourth Edition June, 2003. O'Reilly. Ellen Siever, Stephen Figgins, Aaron Weber.

[RFC-793] http://www.rfc-editor.org/rfc/rfc793.txt. TCP Transmission Control Protocol (Jon Postel 1981 USC-ISI IANA).

[HTTP0.9] The Original HTTP as defined in 1991. http://www.w3.org/Protocols/HTTP/AsImplemented.html.

[RFC-1738] http://www.faqs.org/rfcs/rfc1738.html. Uniform Resource Locators (URL). (Berners-Lee, T., Masinter, L., and M. McCahill, CERN, Xerox PARC, University of Minnesota, 1994).

[RFC-1866] http://www.faqs.org/rfcs/rfc1866.html. Hypertext Markup Language - 2.0. (Berners-Lee (MIT/W3C) , D. Connolly, 1995).

[RFC-7540] https://tools.ietf.org/html/rfc7540. Hypertext Transfer Protocol Version 2 (HTTP/2). (M. Belshe (BitGo), R. Peon (Google), Ed. M. Thomson (Mozilla), 2015).

[HTML30] http://www.w3.org/MarkUp/html3/CoverPage. Raggett, D., "HyperText Markup Language Specification Version 3.0", September 1995. (Available at

[HTML401] http://www.w3.org/TR/html401 Raggett, D., et al., "HTML 4.01 Specification", W3C Recommendation, December 1999.

[XHTML1] http://www.w3.org/TR/xhtml1. "XHTML 1.0: The Extensible HyperText Markup Language: A Reformulation of HTML 4 in XML 1.0", W3C Recommendation, January 2000.

[HTML5] http://www.w3.org/TR/html5/

[RFC-1945] http://www.faqs.org/rfcs/rfc1945.html. Hypertext Transfer Protocol – HTTP/1.0. (T. Berners-Lee (MIT/LCS) , R. Fielding (UC Irvine) , H. Frystyk (MIT/LCS) , 1996).

[RFC-2068] HTTP/1.1. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and T. Berners-Lee, "Hypertext Transfer Protocol HTTP/1.1", RFC 2068, 1997.

[RFC-2616] http://www.faqs.org/rfcs/rfc2616.html. HTTP/1.1. (R. Fielding (UC Irvine) , J. Gettys (Compaq/W3C) , J. Mogul (Compaq) , H. Frystyk (W3C/MIT) , L. Masinter (Xerox) , P. Leach (Microsoft) , T. Berners-Lee (W3C/MIT), 1999)

[RFC-2246] http://www.ietf.org/rfc/rfc2246.txt. The TLS Protocol Version 1.0. (Dierks, C. Allen (Certicom), 1999).

[CGI1.1] http://www.w3.org/CGI/. The WWW Common Gateway Interface Version 1.1.

[PUTs] Referencias al método HTTP/1.0 PUT. http://www.apacheweek.com/features/put. http://www.w3.org/Amaya/User/Put.html. http://www.w3.org/Library/Examples/.

[APACHE] http://httpd.apache.org/docs/2.0/server-wide.html

[MOZ] Mozilla Firefox. http://www.mozilla.com.

[WCPP] http://www.oreilly.com/openbook/webclient/ Web Client Programming with Perl. Clinton Wong, 1997.

[OpenSSL] OpenSSL project: http://www.openssl.org/.

[WDV] WebDAV: http://www.webdav.org/.

[BR1] http://marketshare.hitslink.com.
http://www.nationmaster.com.
http://www.w3counter.com/globalstats.php.
http://en.wikipedia.org/wiki/Usage_share_of_web_browsers.
http://gs.statcounter.com/

[SR1] http://news.netcraft.com.

[CAIDAtnotb98] The nature of the beast: recent traffic measurements from an Internet backbone. K Claffy, caida, kc@caida.org. Greg Miller and Kevin Thompson, MCI/vBNS, gmiller,kthomp@mci.net. 1998.

[YT] http://www.youtube.com/.

[COM05] Ethereal, Wireshark. Autor original Gerald Combs, 2005.
http://www.ethereal.com/.
http://www.wireshark.org/.

[W3C] W3C - World Wide Web Consortium. http://www.w3.org/.

[SOAP] W3C Recommendation (27 April 2007). SOAP Version 1.2 Part 0: Primer (Second Edition). W3C. http://www.w3.org/TR/soap/.

[WSDL] Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl.

[REST] Learn REST: A Tutorial. http://rest.elkstein.org/.

[SOA] Service-Oriented Architecture. http://www.opengroup.org/soa/source-book/soa/soa.htm.

[COOKIE] Cookies Spec. http://curl.haxx.se/rfc/cookie_spec.html.

Índice

Índice de figuras