HTTP/2

Un nuevo protocolo para la web

Marcelo Fernández y Gabriel Tolosa
{fernandezm, tolosoft}@unlu.edu.ar

Laboratorio de Redes
http://www.labredes.unlu.edu.ar/

Universidad Nacional de Luján

Argentina, 2016

Versión

Este es un documento que se actualiza a medida que se va produciendo más contenido, tanto en especificaciones como ejemplos y código fuente (*living document*). Su objetivo principal es brindar una introducción didáctica al tema y surge inicialmente como un texto complementario para la asignatura "Teleinformática y Redes"[1] de la Licenciatura en Sistemas de Información de la Universidad Nacional de Luján[2], de la cual los autores son docentes.

La actual es la versión 1.0, de diciembre de 2016.

Contacto

Por cualquier comentario, error u omisión, se agradece contactar a los autores a las direcciones de correo indicadas en la tapa.

[1] http://www.labredes.unlu.edu.ar/
[2] http://www.unlu.edu.ar/

Índice

1. La Web actual y HTTP/1.x

El protocolo HTTP es una de las bases de la Web (junto con el lenguaje HTML) y desde su diseño original ha sufrido muy pocas modificaciones. Su primera versión, documentada en 1991 [3], es muy simple y fue pensada para recuperar un documento de hipertexto desde un servidor. Sin embargo, el crecimiento exponencial de Internet y el desarrollo de la web exigieron nuevas capacidades al protocolo por lo que la IETF[3] desarrolló y publicó una nueva versión en 1997, HTTP/1.1 [13], como nuevo estándard oficial. Esta versión incluye varias mejoras significativas como conexiones persistentes, *pipelining*(1.5), mejores mecanismos de caching, entre otras y se convirtió rápidamente en el protocolo utilizado por la mayoría de los servidores web desde entonces [19].

Básicamente, HTTP es un protocolo que funciona en texto plano, esto es, los mensajes se intercambian sin encriptar siguiendo el modelo cliente/servidor. En el modelo de referencia OSI [**?**] corresponde a un protocolo de capa de aplicación que opera tomando servicio del protocolo TCP en la capa de transporte.

En las primeras épocas, las páginas web eran simples y contenían pocos objetos, en general, pequeños (por ejemplo, imágenes de unos pocos KBytes). La actualidad es muy diferente ya que, por ejemplo, el tamaño promedio de una página web en 2016 es de 2.4 MB (de acuerdo al HTTP Achive[4]).

Con el tiempo, el cambio en la dinámica de uso de la web, el incremento de los tamaños de los objetos y la necesidad de conexiones seguras pusieron desafíos que se fueron mitigando con algunas técnicas específicas (por ejemplo, abrir múltiples conexiones TCP, utilizar sesiones TLS, entre otras). Algunas de estas técnicas son retomadas y explicadas en detalle en los capítulos siguientes.

Luego de más de 15 años, la web se ha transformado en una plataforma que soporta una gran variedad de servicios sobre la cual se han montado diferentes tecnologías (varias versiones de HTML, javascript, CSS, AJAX, lenguajes de programación, etc.) por lo que esta evolución ha generado algunas problemáticas que influyen de manera negativa, principalmente en el rendimiento de HTTP, de cara al usuario del servicio. De allí, la necesidad de revisar este protocolo y ofrecer mejoras acordes al presente de la web.

1.1. Introducción al funcionamiento de los clientes HTTP

Como se mencionó anteriormente, HTTP es un protocolo diseñado para operar bajo el modelo cliente/servidor. Los clientes, es decir, los navegadores web[5], requieren de los siguiente pasos para obtener una página web[6]:

1. Obtención del recurso (por ejemplo, un archivo HTML) utilizando un método provisto por el protocolo (en general, GET).

2. Parsing (interpretación) del recurso[7].

3. Identificación de nuevos recursos de los que éste depende (Extracción de otras URLs).

[3] Internet Engineering Task Force - http://www.ieft.org/

[4] HTTP Archive. HTTP Trends. http://httparchive.org/trends.php

[5] En general, cualquier aplicación que implemente HTTP como los clientes de consola *wget* y *curl*

[6] Para el modelo completo de procesamiento, ver: http://fetch.spec.whatwg.org/#fetching

[7] Parsing HTML standard: http://www.whatwg.org/specs/web-apps/current-work/multipage/parsing.html

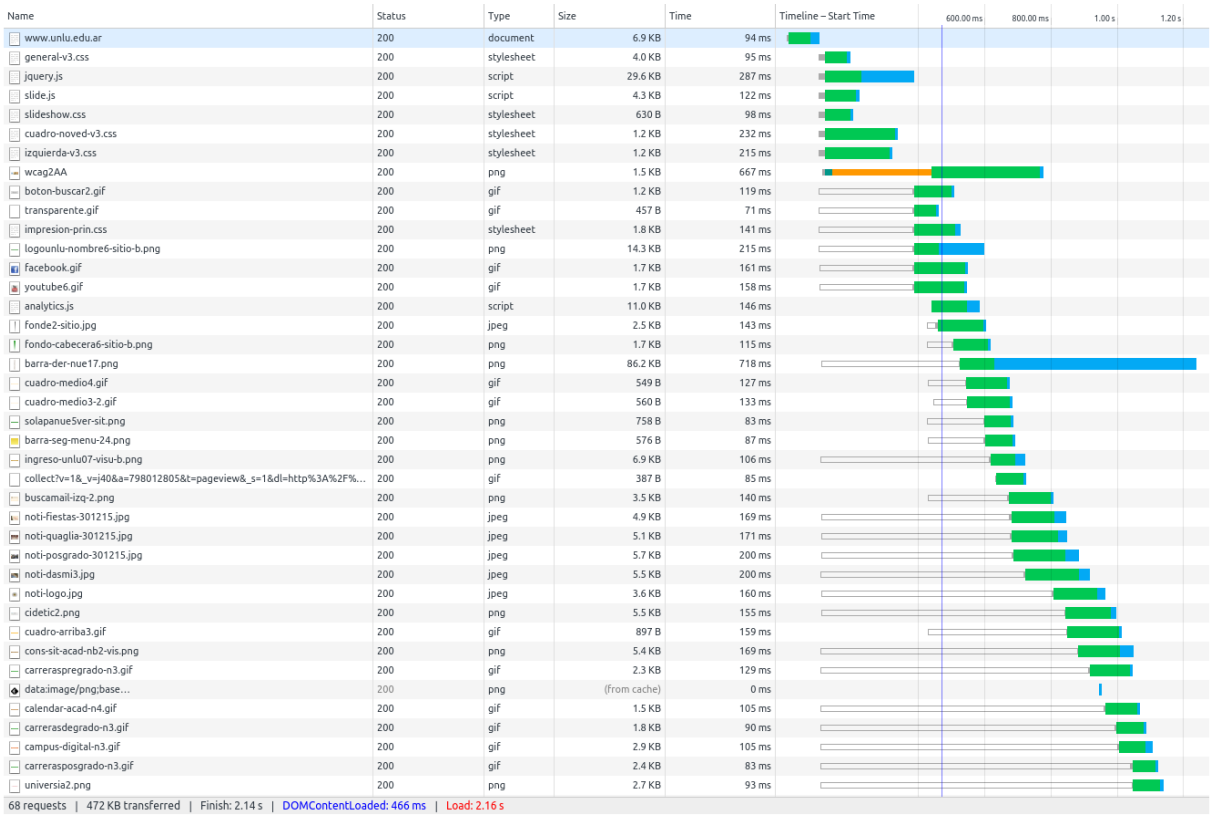


Figura 1: Obtención de recursos de una página web, gráfico en cascada.

4. Si hay nuevos recursos a obtener, éstos se clasifican según alguna prioridad y se descargan.

5. A medida que se van obteniendo los nuevos recursos, se vuelve a 2 teniendo ahora a éstos como referencia.

Los objetos que referencian a otros subrecursos suelen ser:

- Los documentos HTML, vía tags o atributos de éstos tales como `<a href="link"`, `<img src="link"`, `<link href="link"`, etc.
- Los CSS vía el valor `url("link")` aplicado a un estilo.
- Los scripts en lenguaje Javascript, por ejemplo, modificando el documento de la página web en forma dinámica.

El comportamiento resultante de las descargas realizadas para mostrar finalmente una página web se representa tradicionalmente en forma de "Gráfico de Cascada". En la Figura 1 se puede ver un listado de recursos de una página web, y su ubicación en una línea de tiempo de descarga (a la derecha), indicando el momento en que fue puesto en la cola de descargas, el orden en que fue efectivamente descargado el recurso y el tiempo empleado en obtenerlo del servidor.

1.2. Velocidad de Navegación y Experiencia del Usuario

Según estudios realizados por Akamai[8] las velocidades promedio en la conectividad de los usuarios en 2015 del Cono Sur de Sudamérica ronda los entre los 1,5 Mbps (Paraguay) y 5,9 Mbps (Uruguay), mientras que Argentina se encuentra un poco por encima del promedio, con 4,17 Mbps.

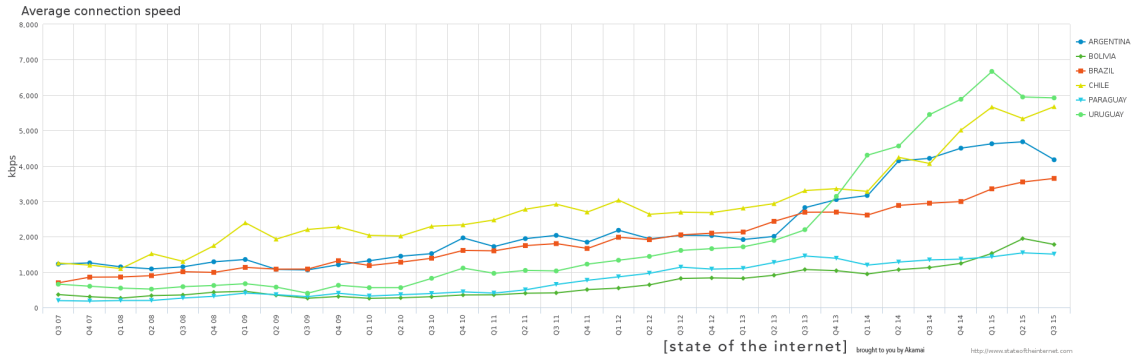


Figura 2: Conectividad promedio a Internet en algunos paises de sudamérica entre 2007 y 2015. Fuente: Akamai[9]

Sin embargo, a los efectos de cargar una página web completa, las tasas máximas de conexión a Internet expuestas anteriormente son morigeradas, porque amén de que las condiciones que afectan al rendimiento en Internet son variadas e impredecibles, un efecto negativo tanto o más importante que toma preponderancia es el de obtener todos los recursos de una página en forma progresiva y relativamente ordenada mediante HTTP.

Es decir, si una página web con todos sus recursos, pesa 1 MB y se dispone de una conexión de 10 Mbps, la página no tardará en descargar 0,8 segundos como indica el cálculo directo de 1 MB / 1,25 MBps = 0,8s, sino que tardará más (obviando el efecto de cualquier caché intermedia y/o local), y no sólo por efecto de las condiciones de la red, sino también por diferentes factores que se verán a continuación, todos relacionados con el protocolo HTTP y su uso por parte de los navegadores de internet.

1.3. Ancho de banda y latencia

Respecto a los parámetros de Ancho de Banda[10] y Latencia[11], los problemas se pueden encontrar en estudios realizados por Mike Belshe [2], cuyos resultados principales se exponen a continuación.

Por el lado del Ancho de Banda, Belshe realizó un test del tiempo de carga de una página completa, de las 25 páginas web más populares, con un porcentaje de pérdida de paquetes del 0 % y con un RTT (round trip time, la latencia del enlace al servidor medida de ida y vuelta)

[8] http://www.akamai.com/

[9] https://www.stateoftheinternet.com/trends-visualizations-connectivity-global-heat-map-internet-speeds-broadband-adoption.html

[10] El concepto de Ancho de Banda se utiliza como equivalente de Tasa de Transferencia solamente debido a que es un término ampliamente difundido como tal, aunque estrictamente incorrecto en cuanto a lo que representa.

[11] Se define aquí como latencia al tiempo que tarda un bit en salir de un extremo y llegar al otro. En la capa de enlace corresponde al tiempo en propagarse por el medio, de acuerdo a la distancia y la velocidad de la señal. En capas superiores, incluye atravezar todas las redes y dispositivos que unen los extremos que se están considerando.

de 60 ms, estableciendo el ancho de banda de 1 Mbps a 10 Mbps.

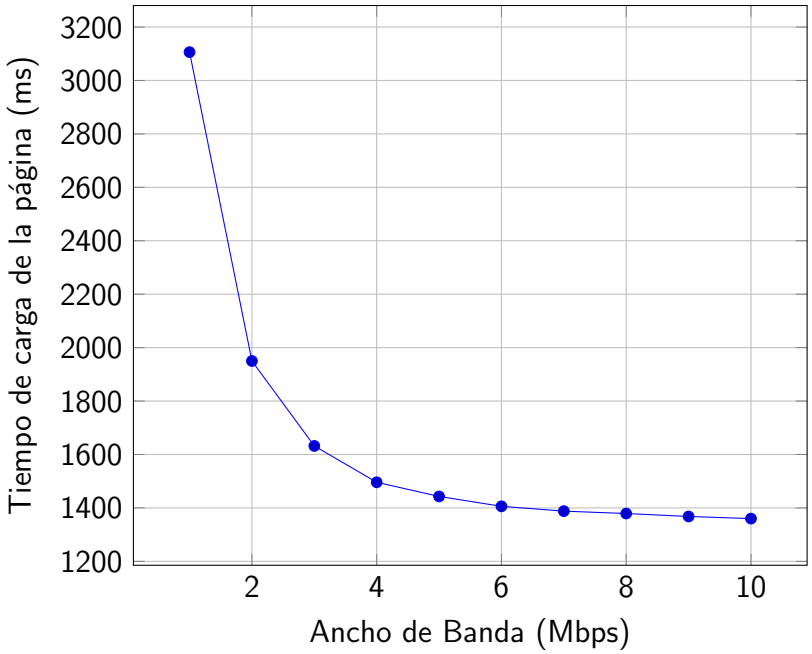


Figura 3: Relación entre el tiempo de carga de página y el Ancho de Banda [2]

En la Figura 3 se puede ver que el tiempo de carga de la página no varia significativamente a partir de una conexión de 4 Mbps; con lo cual se puede inferir que a partir de una conexión de este tipo, el ancho de banda que se posea no tiene influencia alguna sobre el tiempo de carga de una página web (téngase en cuenta que se está midiendo la descarga de solo una página).

En un segundo test se invirtieron los roles, fijándose el ancho de banda en 5 Mbps y pero variando el RTT del enlace entre 0 y 240 ms. Los resultados fueron:

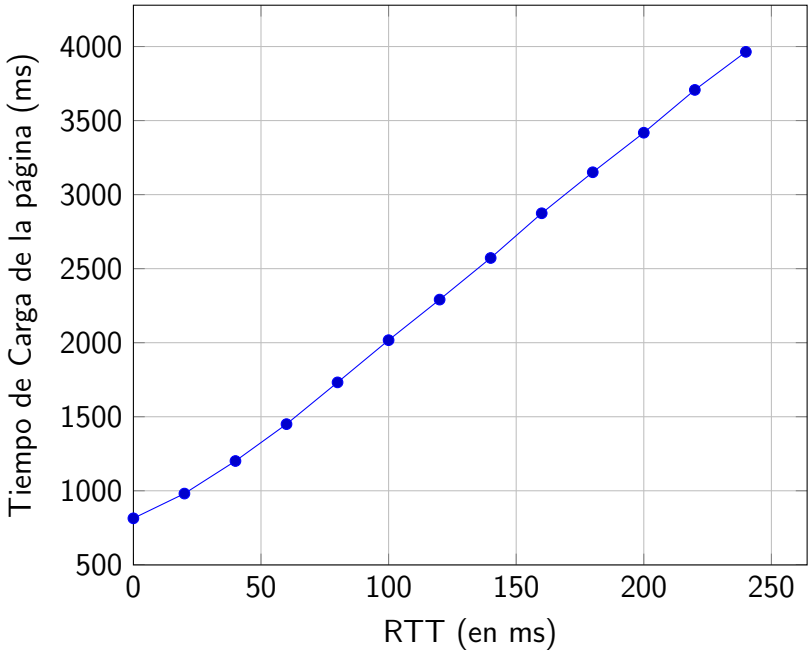


Figura 4: Relación entre el Tiempo de Carga de Página y la Latencia [2]

En la Figura 4 se observa que la relación entre el tiempo de carga de página y el RTT posee un comportamiento lineal y directamente proporcional: a mayor RTT, mayor es el tiempo de carga de la página[12]. Lamentablemente y como es de suponer, mejorar de forma significativa el RTT del camino entre cliente y servidor es virtualmente imposible por diversas razones: distancia, límites fisicos de los medios empleados, nivel de congestión de dispositivos intermedios, principalmente. Incluso, desde el lado del usuario, esto en inmanejable ya que depende de su proveedor de acceso a la red (quien a su vez es cliente de un transportista de datos mayor y así.). Entonces, tener múltiples conexiones a causa de la implementación del protocolo HTTP, que responden a los límites impuestos por el RTT, se vuelve un problema.

Un navegador, para obtener una página (desde cero, por ejemplo una "*home page*") debe hacer una o varias consultas mediante el protocolo DNS y establecer varias conexiones HTTP (sobre TCP) para obtener los recursos correspondientes. Cada una de estas conexiones es administrada en forma coordinada por el agente de usuario pero son independientes, con lo cual se incurre en las demoras del *Three-Way Handshake* (Saludo de Tres Vías) y el esquema de *Slow Start* de TCP **por cada conexión**, lo que aumenta considerablemente la latencia real por sobre la del enlace, a cambio de poder obtener varios recursos simultáneamente luego de que las conexiones TCP fueron establecidas.

Como se mencíonó, en lo comienzos de la web las páginas eran mucho más simples, referenciaban muchos menos recursos que ocupaban mucho menos espacio, y los enlaces eran mucho más lentos, no sólo en ancho de banda sino principalmente en latencia, de lo que hoy se dispone. En aquel momento, el mecanismo de varias conexiones TCP simultáneas tenía sentido.

La idea general en el análisis previo que dio motivos para pensar en una nueva versión del protocolo HTTP es que hoy en día usar HTTP sobre TCP (más DNS) involucra más tiempos de retardo de los necesarios para obtener una página web, muy por encima del mínimo físico que impone el enlace.

[12] En este caso, la serie ajusta a $f(x) = 13{,}55x + 694{,}32$ con $R^2 = 0{,}99$

1.4. Conexiones Persistentes

Básicamente, de trata de una conexión TCP por la que se puede realizar más de una petición HTTP. En este caso, el servidor web por defecto no cierra la conexión luego de responder la petición inicial y, en cambio, espera un tiempo (configurable desde el servidor) antes de hacerlo, posibilitando que el cliente realice una nueva petición.

Tanto la utilización de conexiones persistentes (*Keep Alive*) por defecto como el *Pipelining*, fueron definidos en HTTP/1.1 para incrementar la utilización de los sockets TCP (*caros* de establecer), ya que HTTP/1.0 permite solo una única petición por conexión[13].

Por ejemplo:

```
marcelo@neptuno:~$ nc -v www.google.com 80
Connection to www.google.com 80 port [tcp/http] succeeded!
GET / HTTP/1.1


HTTP/1.1 302 Found
[... headers ...]

<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/?gws_rd=cr">here</A>.
</BODY></HTML>

GET /favicon.ico HTTP/1.1

HTTP/1.1 200 OK
[... headers ...]
[... contenido ...]

Connection closed by foreign host.
```

Esto sucede siempre en HTTP/1.1 a menos que en los headers el cliente se incluya `Connection: close`, con lo cual luego de terminar la transferencia de la petición, la conexión TCP se cierra. De esta manera, luego de recibir un recurso, el cliente puede volver a pedir otros más por la misma conexión, disminuyendo el tiempo de espera (medido en RTTs) de la apertura de nuevas conexiones TCP.

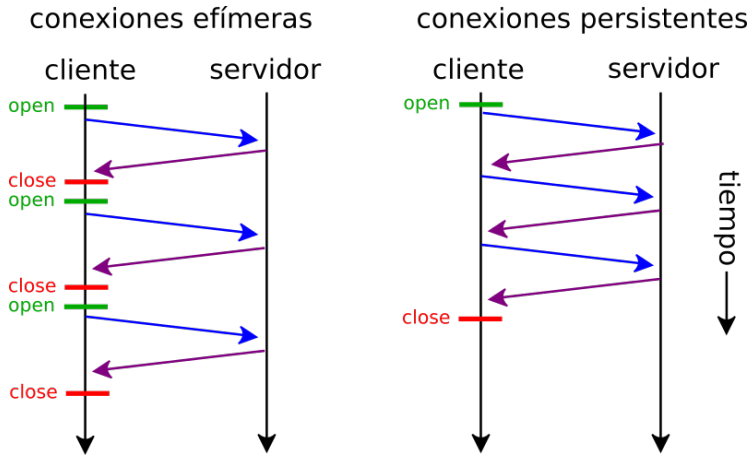


Figura 5: Esquema de peticiones sobre conexiones efímeras vs. conexiones persistentes

Sin embargo, y aunque claramente es una mejora respecto al modelo HTTP/1.0 de "una petición, una conexión" ya que se eliminan los RTTs del establecimiento de los sockets TCP,

[13] http://www8.org/w8-papers/5c-protocols/key/key.html

se sigue incurriendo en un RTT por cada petición que haga, ya que hasta no obtener un recurso completamente, no se puede solicitar otro.

En la Figura 6 se muestra un ejemplo del impacto de la apertura a nivel TCP sobre una configuración con *Keep-Alive* considerando un enlace con 28ms de latencia. La apertura de la conexión requiere 56ms (un RTT, ya que en la segunda confirmación envía el pedido de capa de aplicación), mientras que el intercambio HTTP de dos objetos, 172ms. Es decir, un 24 % del tiempo fue requerido por la conexión en capa de transporte. Esto se hubiese agravado (37 %) de no tratarse de HTTP persistente.

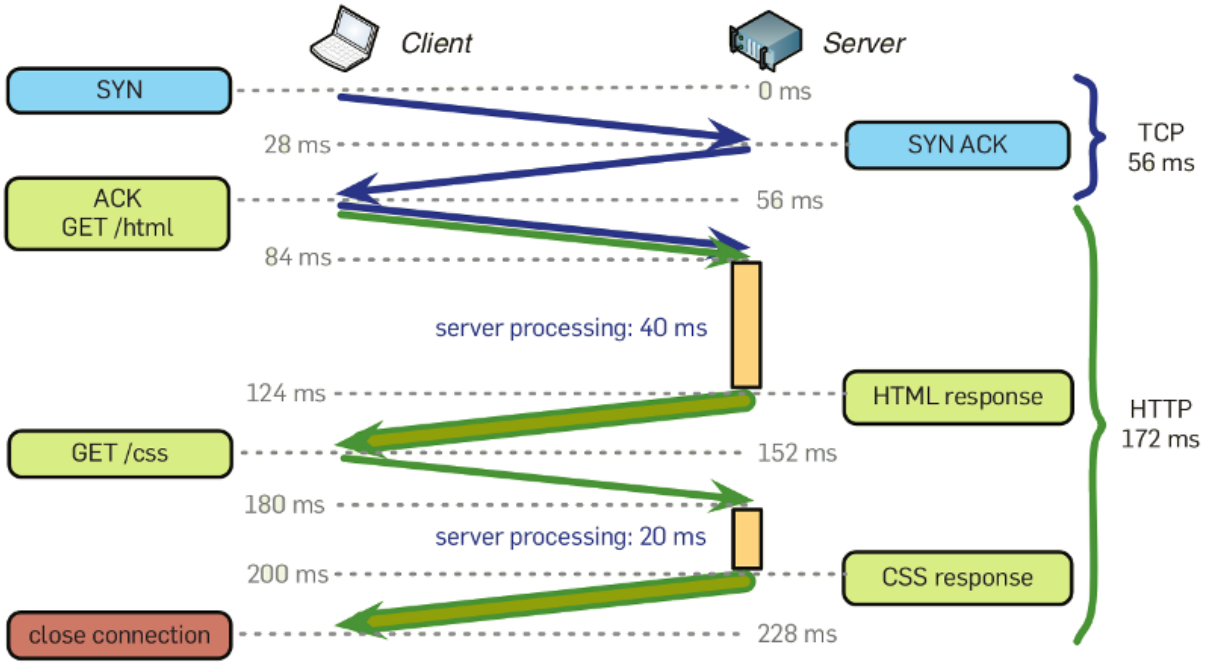


Figura 6: Rol de la latencia para obtener dos recursos HTTP usando una conexión TCP y HTTP persistente [12].

El impacto de esta desventaja, dadas las características de la web de 1999, fue minimizado. Pero actualmente hay estadísticas realizadas sobre un número representativo de páginas web[14] que indican que se necesitan alrededor de 50 peticiones al mismo dominio para cargar sólo una página; con lo cual serían 50 RTTs en promedio a emplearse en la obtención de una página web si se utilizara una única conexión TCP al servidor. Estos RTT oscilan en la Argentina entre 180 y 200ms a un servidor ubicado en Estados Unidos, con lo que siguiendo el caso hipotético se desperdiciarían en total 10 segundos (200ms x 50 peticiones) sólo en esperar que nuestra petición llegue al servidor y comenzar a obtener la respuesta, de todos los recursos de una página web.

Es por esto que si bien en HTTP/1.1 se permite un máximo de 2 conexiones simultáneas contra el servidor[15], en la práctica los navegadores utilizan 6 conexiones[16] [17] para mejorar la paralelización de la obtención de recursos. Utilizando 6 conexiones simultáneas, el tiempo mínimo de las 50 peticiones bajaría a 1,6 segundos; aunque claro, sólo se están contemplando

[14] http://httparchive.org/trends.php#numDomains&maxDomainReqs

[15] http://tools.ietf.org/html/rfc2616#section-8.1.4

[16] La restricción es de N conexiones máximas por un mismo origen ("*same-origin policy*"). Un origen para el navegador está definido por: protocolo, FQDN y puerto. Por ejemplo, en una referencia a http://img.unlu.edu.ar:8080, el protocolo es http, el FQDN es img.unlu.edu.ar y el puerto 80.

[17] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

los tiempos de transferencia teóricos de la red y no los tiempos de *Slow Start* de TCP, consumo de CPU, variaciones en las condiciones de la red y otros agentes, procesamiento y descubrimiento de los recursos a obtener por parte del navegador, el uso de caché en distintos niveles, etc.

Entonces, si sabiendo que 6 peticiones concurrentes alivian el problema ¿por qué no usar 8, 10 o más conexiones simultáneas? Por varios motivos:

- Para evitar congestión en la red (el mismo protocolo lo aconseja al definir el límite de 2).
- Porque comienza a incurrirse en la saturación de otros recursos, como el ancho de banda del cliente para una serie de peticiones a descargar del servidor sin prioridad entre sí a nivel de red. Es decir, que si se utilizan 10 conexiones simultáneas para descargar imágenes, el uso parece acertado, pero no lo es. Por ejemplo, si 8 son imágenes y 2 de ellas son para hojas de estilos (CSS), fundamentales para que los navegadores le puedan comenzar a mostrar algo al usuario en pantalla y no se llegue a un el FOUC (Flash of unstyled content – Flash de contenido sin estilo)[18]. Lo mismo sucede con los scripts en Javascript, que pueden tener dependencia entre sí y/o pueden obligar al navegador a obtener más recursos prioritarios. Es decir, son pocos los casos en donde se podría paralelizar abriendo muchas más conexiones y realmente sea beneficioso (por ejemplo, una galería de muchas imágenes).
- Si se piensa en aún más conexiones, se estaría en el caso de HTTP/1.0, "un recurso, una conexión", del cual ya se explicó los problemas que involucra.
- El problema es realmente complejo desde el punto de vista del navegador, porque además éstos lógicamente no saben de antemano cuántos recursos insumirá la obtención de una página, lo van calculando a medida que van interpretándola.

En la Sección 1.6 se introducen algunas optimizaciones forzadas que tiene disponible el desarrollador web para superar algunos de estos obstáculos y manejar algunas de estas variables, aún de manera relativamente precaria.

1.5. Pipelining

Con la misma idea de mejorar los RTT insumidos por una conexión persistente y funcionar un poco más en forma de "ráfagas" o de "proceso batch", en HTTP/1.1 también se adicionó la posibilidad del Pipelining, que no es más que permitir que el cliente haga todas las peticiones al inicio de la conexión, y luego el servidor devolverá los recursos en el mismo orden en que fueron solicitados.

[18] http://en.wikipedia.org/wiki/Flash_of_unstyled_content

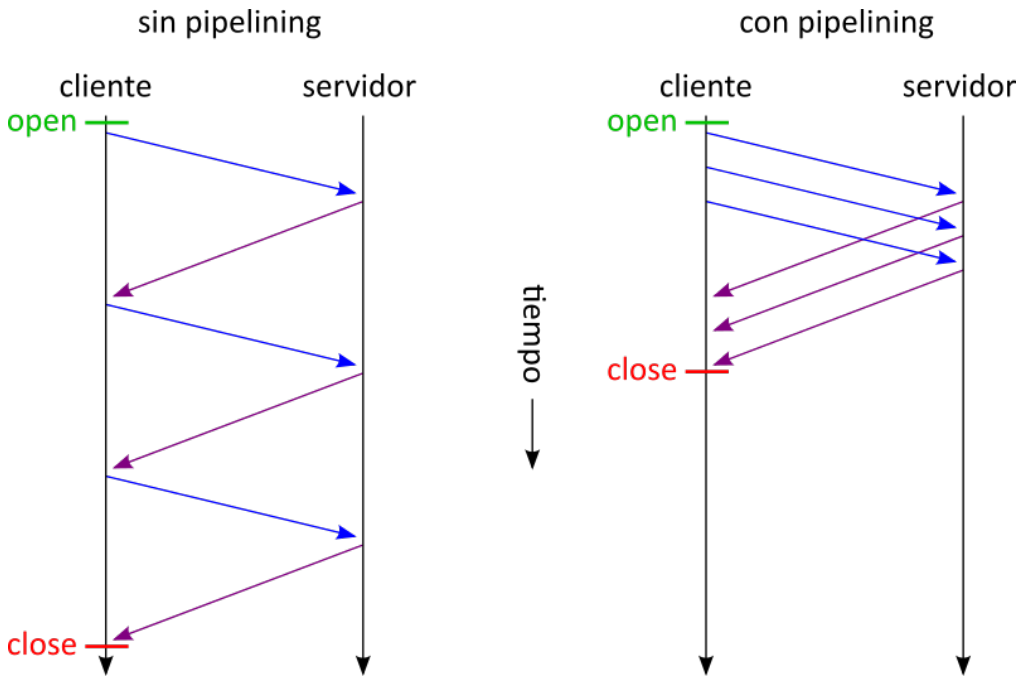


Figura 7: Esquema de peticiones sobre conexiones sin pipelining vs. con pipelining

Ejemplo:

```
marcelo@neptuno:~$ nc www.unlu.edu.ar 80
GET / HTTP/1.1
Host: www.unlu.edu.ar
Accept-Encoding: text/plain
Connection: keep-alive

GET /css/slideshow.css HTTP/1.1
Host: www.unlu.edu.ar
Accept-Encoding: text/plain
Connection: keep-alive

GET /js/jquery.js HTTP/1.1
Host: www.unlu.edu.ar
Accept-Encoding: text/plain
Connection: keep-alive

HTTP/1.1 200 OK
Date: Sat, 30 Jan 2016 13:44:29 GMT
Server: Apache
Cache-Control: max-age=3600
Expires: Sat, 30 Jan 2016 14:44:29 GMT
Vary: Accept-Encoding
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1

24e6
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="es">
<head>
<title>Universidad Nacional de Luján</title>
[... resto de la salida omitida ...]
2775
[... resto de la salida omitida ...]
149e
[... resto de la salida omitida ...]

0

HTTP/1.1 200 OK
```

```
Date: Sat, 30 Jan 2016 13:52:35 GMT
Server: Apache
Last-Modified: Thu, 21 Feb 2013 17:36:01 GMT
ETag: "a464-162-4d63f821c0240"
Accept-Ranges: bytes
Content-Length: 354
Cache-Control: max-age=604800
Expires: Sat, 06 Feb 2016 13:52:35 GMT
Vary: Accept-Encoding
Keep-Alive: timeout=15, max=99
Connection: Keep-Alive
Content-Type: text/css
[... resto de la salida omitida ...]
HTTP/1.1 200 OK
Date: Sat, 30 Jan 2016 13:52:35 GMT
Server: Apache
Last-Modified: Tue, 19 Feb 2013 13:25:53 GMT
ETag: "a514-14fa5-4d613c7e15a40"
Accept-Ranges: bytes
Content-Length: 85925
Cache-Control: max-age=604800
Expires: Sat, 06 Feb 2016 13:52:35 GMT
Vary: Accept-Encoding
Keep-Alive: timeout=15, max=98
Connection: Keep-Alive
Content-Type: application/javascript

/*!
 * jQuery JavaScript Library v1.5.2
 * http://jquery.com/
[... resto de la salida omitida ...]
```

Como puede verse, el navegador hace 3 peticiones iniciales e inmediatamente después el servidor fue devolviendo los recursos en forma secuencial y en el mismo orden que fueron solicitados. Un detalle del ejemplo es el uso de la opción `chunked` del header `Transfer-Encoding` en la primer respuesta, que indica que el tamaño de respuesta es desconocido por el servidor web en ese momento[19] y lo va a ir devolviendo por partes (o *chunks*). Es por eso que si la respuesta es *chunked* tendrá intercalados números hexadecimales con el tamaño en bytes de la parte que sigue a continuación, finalizando con una parte de 0 bytes. Los dos recursos restantes tienen un tamaño conocido y llevan el header `Content-Length` indicando su tamaño.

Sin embargo, esta característica es raramente utilizada y actualmente se encuentra implementada por los navegadores web pero deshabilitada por defecto[20] [21]. Algunos de los motivos son[22]:

- Problemas de Interoperabilidad: la falta de especificación de los diferentes casos de error y condiciones de falla en el protocolo original dio a lugar a implementaciones y comportamientos muy distintos entre diferentes componentes de software. Esto se suma a la falta de definición sobre el comportamiento de intermediarios (*proxies*) HTTP, como pueden ser productos corporativos, proxy-caché transparentes e incluso software antivirus domésticos, que deshabilitan o soportan pipelining de manera muy primitiva o incorrecta, incluso "rompiendo" las conexiones[23].
- *Head of Line blocking*[24]: Se trata de una problemática clásica: tiene que ver conque si el primer recurso –o cualquiera subsiguiente– que se encuentre en la cabeza de la

[19] Probablemente esto sea así por el uso de un script del lado del servidor que hará la respuesta dinámica y el tamaño del recurso en el momento de devolver los encabezados es incierto.
[20] http://kb.mozillazine.org/Network.http.pipelining
[21] https://code.google.com/p/chromium/issues/detail?id=8991
[22] https://insouciant.org/tech/status-of-http-pipelining-in-chromium/
[23] https://www.mnot.net/blog/2007/06/20/proxy_caching
[24] https://en.wikipedia.org/wiki/Head-of-line_blocking

lista de recursos a devolver por el servidor tarda sensiblemente más que el resto, toda la lista debe esperar (estará "bloqueada") a que el primero sea despachado. En el ejemplo anterior, si el script de servidor que genera la página inicial de la Universidad de Luján se demora en exceso, el resto de los recursos (`/css/slideshow.css` y `/js/jquery.js`) deben esperar a que éste termine aún cuando podrían ser enviados en simultáneo por una conexión paralela.

- Mejoras de velocidad no tan concluyentes: Aún cuando bajo ciertas condiciones es mucho más probable que acelere la navegación del usuario (como enlaces satelitales o de mucha latencia), se desprende del punto anterior que las mejoras de velocidad no son definitivas, más aún considerando el retardo típico de las conexiones actuales y la estructura de la web en la actualidad[25] [26].

Desde que se definió HTTP/1.1, hubo algunas propuestas para su revisión y/o modificación[27] [16], que no prosperaron demasiado.

En 2007, se conformó el Httpbis Working Group del IETF, que trabajó en varios RFC con el objetivo de actualizar la especificación HTTP/1.1 [9, 10, 8, 5, 6, 7]; y se han escrito recomendaciones con el fin de mejorar la interoperabilidad del pipelining[28], pero casi en paralelo, desde 2009, estos esfuerzos se dedicaron para pasar a una nueva versión de HTTP.

1.6. "Afinando" las páginas web para HTTP/1.1

Por último y por si no fuera poco, desde hace varios años han proliferado muchas técnicas para que el desarollador web fuerce y/o personalice la manera o "el algoritmo" en que el agente de usuario obtiene los recursos de una página web[29] [18], para mejorar la percepción del rendimiento que obtendría el usuario por defecto con HTTP/1.1. Algunos de estos son:

- *Domain Sharding*: Es una manera de moderar el límite de 6 conexiones que impone la política de mismo origen, haciendo referencia desde el código HTML a recursos en otros subdominios o dominios del mismo sitio o externos. Permite que se paralelicen las peticiones.

- Recursos *inline*[30]: Es una técnica para embeber en el mismo código HTML o CSS recursos que de otra manera generarían una petición HTTP aparte, como ser scripts Javascript, código CSS y hasta imágenes.

- Mapas de imágenes: Un mapa de imágenes permite insertar diferentes enlaces dentro de regiones de una gran imagen, evitando tener una petición por imagen, por ejemplo, para una barra de menú.

[25] HTTP Pipelining - Not So Fast...(Nor Slow!): http://www.guypo.com/http-pipelining-not-so-fast-nor-slow/

[26] HTTP Pipelining is a Hit and Miss: https://www.subbu.org/blog/2012/07/http-pipelining-hit-and-miss

[27] Waka (2002): https://www.ietf.org/proceedings/83/slides/slides-83-httpbis-5.pdf

[28] Making HTTP Pipelining Usable on the Open Web: http://tools.ietf.org/html/draft-nottingham-http-pipeline-01

[29] Yahoo! Best Practices for Speeding Up Your Site: https://developer.yahoo.com/performance/rules.html

[30] https://en.wikipedia.org/wiki/Data_URI_scheme

- CSS Image Sprites[31]: La idea es almacenar muchas imágenes a utilizar en una página web en un único archivo, en una gran imagen que contiene a todas; luego mediante CSS se selecciona una imagen a mostrar definiendo el offset dentro de la gran imagen. Esto también ahorra peticiones HTTP.
- Ordenamiento del código fuente y dependencias de código Javascript: Normalmente se indica que se ubiquen los estilos CSS en el comienzo del código HTML, y el Javascript al final, para que el navegador primero obtenga lo mínimo indispensable para mostrar la página y por último el código que aporta la funcionalidad/interactividad una vez ya se ve ésta en pantalla.

Como se puede ver, estas técnicas no tienen que ver con la programación web en sí, sino en optimizar la velocidad de carga de las páginas web según el contenido a mostrar; esto agrega trabajo al desarrollador, brindando soluciones que sólo mitigan de manera imperfecta los problemas de fondo de HTTP/1.1, como se ha visto anteriormente en este capítulo.

[31] https://css-tricks.com/css-sprites/

2. Características principales de HTTP/2

Como se vio anteriormente, las motivaciones principales que llevan a desarrollar mejoras a HTTP/1.1 son algunas de sus características, que en combinación con otros factores como el protocolo TCP subyacente, más el tamaño, tipo y diversidad de objetos que la Web provee hoy en día, hacen que se desaprovechen los medios y recursos que se disponen. Estos son, y por citar algunos ejemplos, baja latencia y alto ancho de banda de los enlaces, capacidad de procesamiento tanto en el cliente como en el servidor, la posibilidad de reproducir audio y video bajo demanda, etc.

Previo a HTTP/2, Google desarrolló un protocolo denominado SPDY[15] que inicialmente fue soportado por su navegador Chrome (y sus servidores web) con el objeto de abordar los problemas antes mencionados. SPDY es un protocolo completamente binario, que abre solo una conexión TCP con un servidor (opera sobre TLS/SSL) y multiplexa pedidos y respuestas (denominados *streams*). Adicionalmente, introduce el concepto de "prioridad" permitiendo que los clientes recuperen primero algunos objetos que se consideran importantes para el renderizado y comportamiento de la página (por ejemplo, plantillas de estilo y/o scripts). Otras dos características novedosas son el concepto de "server push", que habilita a que el servidor envíe determinado contenido sin requerir la petición explícita, y compresión de los encabezados (para reducir el volumen de información intercambiada).

2.1. Objetivos de Diseño

Los objetivos de diseño que en particular persigue el desarrollo de HTTP/2 son heredados mayoritariamente de SPDY, a saber:

- Evitar cambios en el protocolo de Transporte: modificar TCP e implementar un nuevo protocolo en Internet requeriría un esfuerzo tal que podría nunca terminarse.
- Evitar cambios en los contenidos: Implementar HTTP/2 no debería traer asociados cambios radicales en los servicios Web provistos.
- Modificar únicamente los componentes de software Cliente-Servidor relacionados: Tanto el navegador como el servidor web serían los únicos afectados en la migración.
- Rendimiento siempre superior, nunca inferior, a HTTP/1.1.
- Si bien el aprovechar al máximo el ancho de banda disponible es un objetivo siempre presente, es importante destacar la identificación de la latencia en la transferencia general y global de los contenidos de una página web como uno de los problemas principales a atacar.

2.2. Diseño Interno

En la especificación de HTTP/2 se identifican y se separan dos capas internas respecto de lo que es HTTP/1.1:

- La de bajo nivel (la **sintaxis**) que especifica cómo manejar las conexiones TCP para obtener los recursos, cómo es el intercambio de mensajes por sobre el socket TCP y la definición estricta de cómo estos deben escribirse en el socket. Ésta es la sección que HTTP/2 redefine por completo.

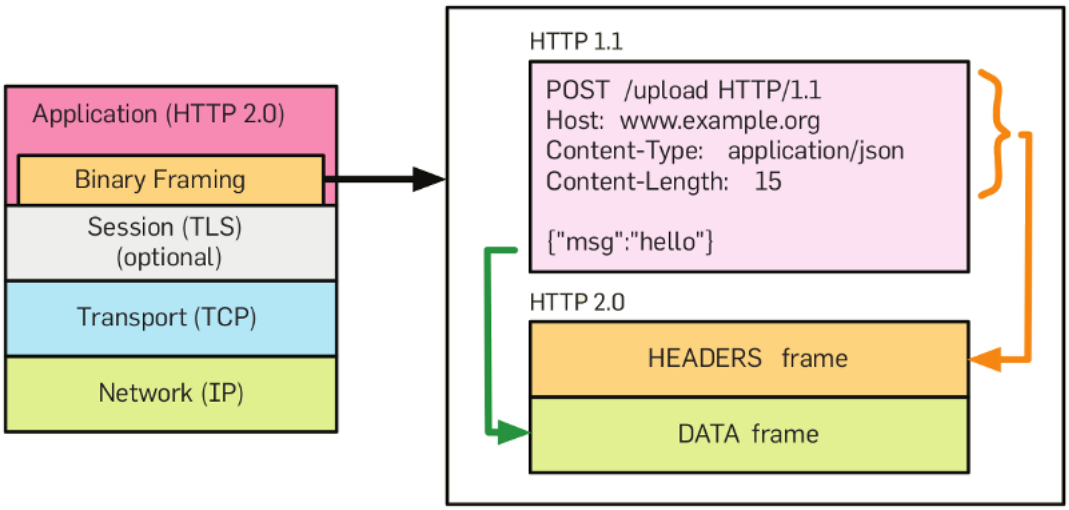


Figura 8: HTTP/2 - Capa de Framing binario y peticiones HTTP[12].

- La de alto nivel (la **semántica**), que contiene el significado de los mensajes: métodos, encabezados (Headers, Cookies, Encoding., etc.), que en HTTP/2 no sufre modificación alguna, de manera tal que la web actual no sufra modificaciones al pasar al nuevo protocolo.

En síntesis, la nueva sintaxis HTTP/2 define una capa de entramado (framing[32]) de datos en formato **binario** (no más de texto plano) situada sobre **una única sesión TCP** que provee:

- Segmentación en **Frames** de todo el tráfico de una sesión. Un frame es la unidad básica de información del protocolo.Hay frames de diferentes tipos, por ejemplo HEADERS y DATA (Figura 8), que se verán en profundidad en el capítulo siguiente.
- Multiplexación de **Streams**. Se introduce el concepto de *stream* como una conexión independiente dentro de la sesión principal de la capa de transporte. Dichos streams fluyen en paralelo, asincrónicamente pero dentro de la misma sesión TCP, sin estar relacionados unos con otros. En la práctica, cada Frame tiene un campo para identificar el Stream ID al cual pertenece ese Frame; los Frames viajan libremente por la conexión TCP sin bloquearse unos con otros (Figura 9).
- Priorización de streams. Los streams pueden tener prioridades.
- Los frames poseen un header (encabezado) pequeño (de sólo 8 Bytes).

Y también se define una capa semántica superior, que interactúa con la capa de Aplicación, que se la podría denominar "Capa HTTP", que encapsula y traduce peticiones HTTP. Sus características son:

- Multiplexación de peticiones HTTP tradicionales y concurrentes en una única sesión HTTP/2. Para esto utiliza los streams provistos por la capa inferior descripta anteriormente.

[32]No confundir esta idea con la capa de enlace, donde la estructura de datos de protocolo (PDU) se denomina *frame*. Aquí se hace uso del mismo concepto, nombrando a la PDU también como *frame* probablemente por tratarse de datos en formato binario.

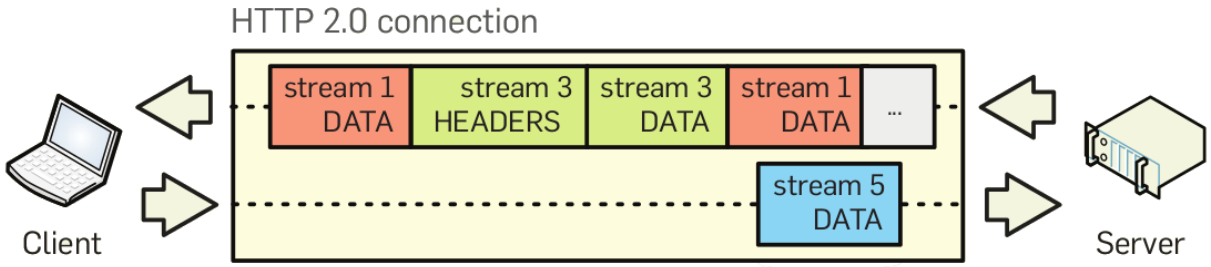


Figura 9: Multiplexación de peticiones HTTP en streams por medio de una sesión HTTP/2.

- Encabezados HTTP comprimidos mediante un nuevo algoritmo llamado HPACK definido en un RFC aparte [17], lo cual hace que éstos se conviertan a un formato binario y compacto, ya que trabaja con *deltas* o diferencias en el intercambio. De esta manera se reduce drásticamente el overhead que ocupan los headers.
- Niveles de prioridades en las peticiones de recursos HTTP.
- *Server-Pushed Streams*: Recursos enviados desde el servidor al cliente sin existir una petición previa de éste.
- Como se dijo anteriormente, se preserva por completo la semántica existente de HTTP: Cookies, headers, encoding, etc., funcionan de la misma manera que con HTTP/1.1.

En teoría, y si bien HTTP/2 fue definido para ser utilizado "en claro", es decir, sin capa TLS de cifrado por debajo, en la práctica casi todos los navegadores lo utilizan **únicamente sobre conexiones cifradas**[33], con lo cual hoy en día únicamente se utiliza con direcciones `https://`. Hay algunas propuestas intermedias para usar HTTP/2 con "Cifrado Oportunístico" (*Opportunistic Security*[34] [4]), que sería una especie de comunicación cifrada pero sin autenticación "fuerte" para URIs 'http', sobre HTTP/2 en texto plano.

Otra cuestión importante que define y necesita el protocolo para su integración en la web actual, refiere al método de UPGRADE o inicialización del protocolo HTTP/2 sobre HTTP/1.1. Si bien HTTP/1.1 contempla un mecanismo de negociación de un protocolo futuro o diferente sobre la conexión actual [35], como se verá posteriormente, el uso de TLS "por debajo" facilita el descubrimiento por parte del navegador web de las posibilidad de usar HTTP/2 contra el servidor, descartando el uso del header de Upgrade de HTTP/1.1.

[33] http://caniuse.com/#feat=http2
[34] Opportunistic Security for HTTP: https://tools.ietf.org/html/draft-ietf-httpbis-http2-encryption
[35] https://tools.ietf.org/html/rfc7230#section-6.7

3. Breve descripción del protocolo

3.1. Inicio de sesión

Existen tres maneras de obtener recursos vía HTTP/2 desde un servidor:

1. Sabiendo que el servidor sí **soporta HTTP/2 de antemano**, se puede inicializar la conexión a nivel HTTP/2 directamente, ya sea mediante una lista preconfigurada en el cliente, o métodos de descubrimiento alternativos como el header HTTP Alt-Svc[36] o un registro de tipo SRV anunciado por DNS[37]. Hoy en día prácticamente no se utiliza, aunque podría ser útil si HTTP/2 ya es el único estándar en pie, o en algunos casos como entornos controlados, aplicaciones especiales, sistemas embebidos, etc.

2. Para URIs de tipo **'http' sobre texto plano**, utilizando el mecanismo de UPGRADE de HTTP/1.1[38]. A fin de citar un ejemplo, el siguiente es un caso exitoso de upgrade, utilizando una versión de curl que soporta HTTP/2:

```
marcelo@neptuno:~/devel$ curl --http2 -v http://nghttp2.org
* Rebuilt URL to: http://nghttp2.org/
*   Trying 106.186.112.116...
* Connected to nghttp2.org (106.186.112.116) port 80 (#0)
> GET / HTTP/1.1
> Host: nghttp2.org
> User-Agent: curl/7.47.1
> Accept: */*
> Connection: Upgrade, HTTP2-Settings
> Upgrade: h2c
> HTTP2-Settings: AAMAAABkAAQAAP__
>
< HTTP/1.1 101 Switching Protocols
< Connection: Upgrade
< Upgrade: h2c
* Received 101
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* TCP_NODELAY set
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=21
* Connection state changed (MAX_CONCURRENT_STREAMS updated)! (SETTINGS Frame received)

* Sending HTTP/2 Magic
> PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n

< HTTP/2.0 200
< date:Tue, 09 Feb 2016 15:49:24 GMT
< content-type:text/html
< content-length:6680
< last-modified:Mon, 25 Jan 2016 11:01:34 GMT
< etag:"56a6008e-1a18"
< link:</stylesheets/screen.css>; rel=preload; as=stylesheet
< accept-ranges:bytes
< x-backend-header-rtt:0.000835
< server:nghttpx nghttp2/1.8.0-DEV
< via:1.1 nghttpx
<

<!DOCTYPE html>
[... resto de la salida omitida ...]
```

3. Para **URIs de tipo 'https', utilizando una sesión TLS** tradicional pero con la extensión NPN (*Next-Protocol Negotiation*[39]) o ALPN (*Application Layer Protocol Negotia-*

[36] HTTP Alternative Services: https://tools.ietf.org/html/draft-ietf-httpbis-alt-svc-12

[37] HTTP/2 DNS-based Upgrade: https://github.com/http2/http2-spec/issues/381

[38] HTTP/1.1 Upgrade header: https://en.wikipedia.org/wiki/HTTP/1.1_Upgrade_header

[39] TLS NPN extension: https://tools.ietf.org/html/draft-agl-tls-nextprotoneg-04

tion [11]). NPN se desarrolló para dar soporte a SPDY, y como éste, está siendo dejado de lado por ALPN que es el estándar formal aprobado por el IETF.

Casi todas las bibliotecas más populares para implementar TLS ya soportan ALPN, como ser OpenSSL[40], GnuTLS[41] y NSS[42].

Ejemplo de sesión tradicional:

```
marcelo@marcelo-notebook:~/devel$ curl --http2 -v https://www.google.com
* Rebuilt URL to: https://www.google.com/
*   Trying 173.194.42.115...
* Connected to www.google.com (173.194.42.115) port 443 (#0)
* Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL:!LOW:!RC4:@STRENGTH
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/certs/ca-certificates.crt
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* NPN, negotiated HTTP2 (h2)
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Client hello (1):
* TLSv1.2 (OUT), TLS handshake, Unknown (67):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES128-GCM-SHA256
* Server certificate:
*   subject: C=US; ST=California; L=Mountain View; O=Google Inc; CN=www.google.com
*   start date: Jan 27 13:58:11 2016 GMT
*   expire date: Apr 26 00:00:00 2016 GMT
*   subjectAltName: www.google.com matched
*   issuer: C=US; O=Google Inc; CN=Google Internet Authority G2
*   SSL certificate verify ok.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* TCP_NODELAY set
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0

* Sending HTTP/2 Magic
> PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n

* Using Stream ID: 1 (easy handle 0x1e30b90)
> GET / HTTP/1.1
> Host: www.google.com
> User-Agent: curl/7.47.1
> Accept: */*
>
* Connection state changed (MAX_CONCURRENT_STREAMS updated)!
< HTTP/2.0 302
< cache-control:private
< content-type:text/html; charset=UTF-8
< location:https://www.google.com.ar/?gfe_rd=cr&ei=9hC6Vs-LDMOB8QeJsbuQCA
< content-length:263
< date:Tue, 09 Feb 2016 16:16:54 GMT
< server:GFE/2.0
< alternate-protocol:443:quic,p=1
< alt-svc:quic="www.google.com:443"
<
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
[... resto de la salida omitida ...]
```

Si bien hay mucha más información para cada situación, lo más importante a destacar en este momento es que todo lo que figura previo a las líneas "*Using HTTP2*" de cada log

[40] https://www.openssl.org/news/openssl-1.0.2-notes.html

[41] http://www.gnutls.org/manual/html_node/Application-Layer-Protocol-Negotiation-_0028ALPN_0029.html

[42] https://bugzilla.mozilla.org/show_bug.cgi?id=959664

trata de cada mecanismo de upgrade o inicio de sesión de HTTP/2 sobre una conexión TCP existente. De ahí en más, ambos pares deben intercambiar el **prefijo de inicio de conexión** ("*Connection Preface*": [1], Sección 3.5) HTTP/2 para luego comenzar el libre intercambio de Frames con peticiones, respuestas y todo lo que éste especifica, de manera binaria.

Desde el cliente hacia el servidor, éste prefijo es un string fijo llamado "Magic" en la jerga[43]:

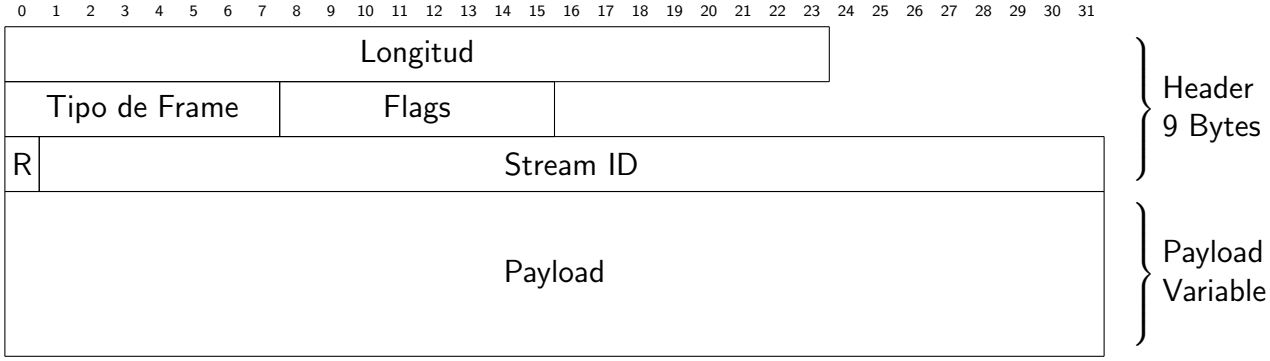
```
PRI * HTTP/2.0\r\n\r\n SM\r\n\r\n
```

más un Frame de tipo SETTINGS. Por su parte, el prefijo de inicio de conexión desde el servidor al cliente es sólo un Frame de SETTINGS que define las reglas básicas de la sesión (más sobre los tipos de Frames en la próxima sección).

Como se dijo anteriormente, el caso que más común en Internet es este último ya que cuenta con todo el soporte de los navegadores, y es el que se trabajará en los ejemplos y las prácticas de este documento. Es por ello que para utilizar HTTP/2 se debe contar con algunos conocimientos básicos de infraestructura de clave pública en Internet[44]. Esto, que quizás sea una barrera de entrada para su adopción –junto con lo binario del protocolo– se sobrelleva con algunos conceptos breves más algunas herramientas de análisis y depuración que se verán más adelante.

3.2. Estructura y tipos de Frame

La estructura común de todos los tipos de Frames HTTP/2 es la siguiente:



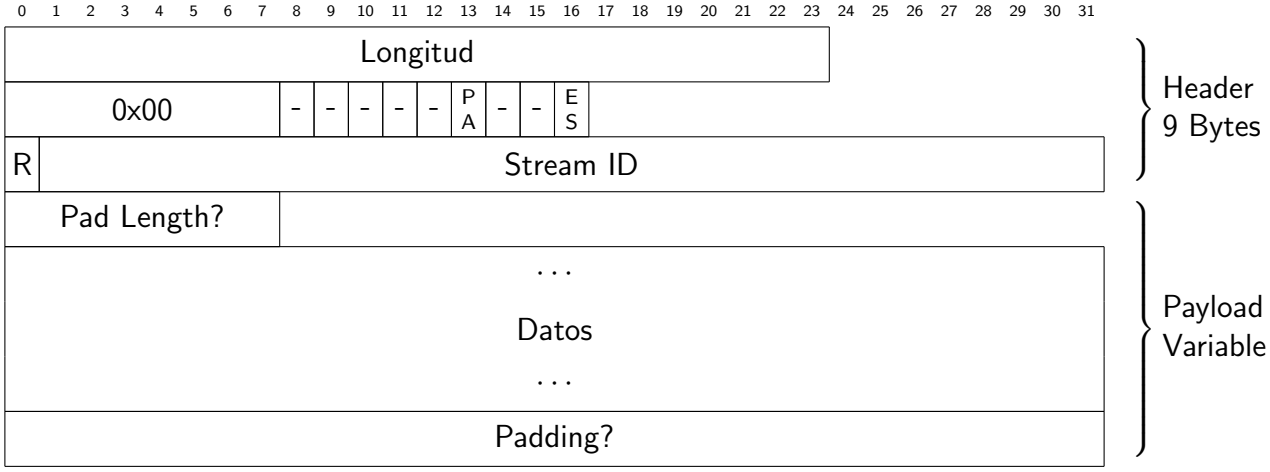
El campo Longitud es el tamaño en bytes del Payload, sin contar el propio encabezado de 9 octetos. Los distintos Tipos de Frame (1 Byte) se verán a continuación, y el campo de Flags (1 Byte) especifica opciones que tomarán significado según el Tipo de Frame en cuestión. Por último, hay un bit R (Reservado) antes del Stream ID (31 bits), que contiene el número de Stream al que hace referencia el Tipo de Frame de dicha trama.

Debe recordarse que en esta instancia los diferentes *Streams* son una conexión "virtual" dentro de una conexión TCP, por ende en una única sesión de transporte viajarán múltiples frames de distintas conexiones "virtuales", es decir, Stream IDs. Éstos son números secuenciales crecientes a medida que se van creando Streams, comenzando por el 1, donde el cliente utilizará los impares y el servidor (en el caso de aquellos inicializados por éste mediante SERVER_PUSH) pares. El Stream ID 0 se utiliza únicamente para mensajes de control de conexión.

[43] Se lo llama así porque es un string que en caso de ser interpretado por un agente HTTP/1.x, hará que éste aborte la conexión lo antes posible: http://www.chmod777self.com/2013/07/http2-wait-theres-more.html

[44] https://en.wikipedia.org/wiki/Public_key_infrastructure

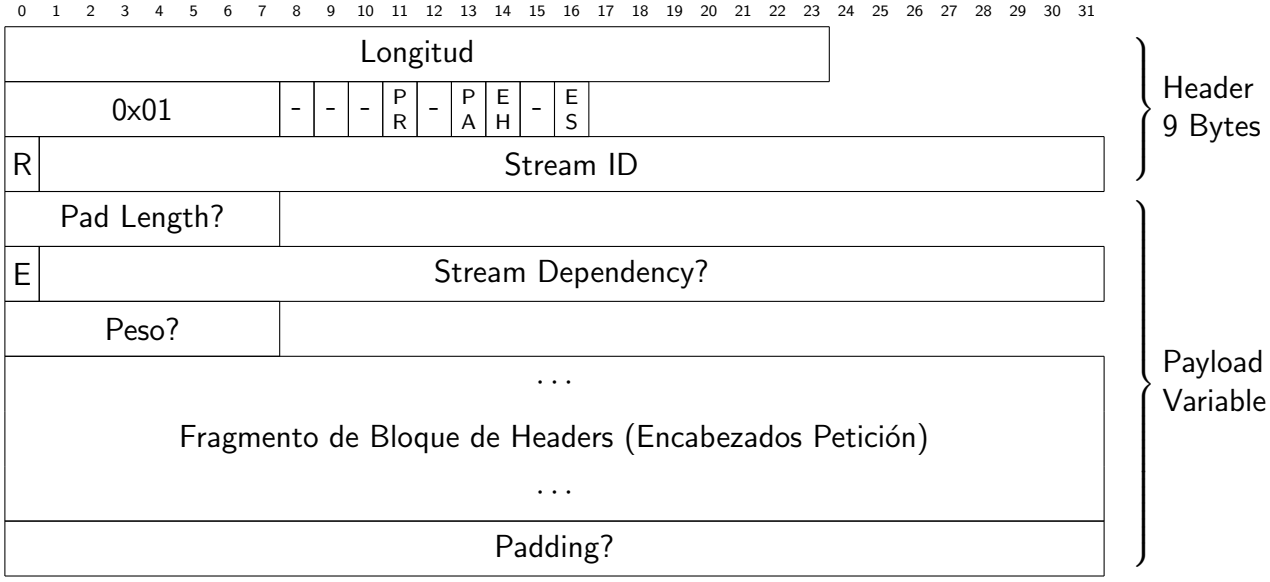
3.2.1. DATA Frame (0x00)



Los DATA Frames transportan los recursos de la capa superior (Aplicación) entre cliente y servidor, como por ejemplo contenido HTML, CSS, Javascript, Imágenes, etc, equivalente a lo que transporta el Message Body en HTTP/1.1[45] ([1], Sección 3). Existe la posibilidad de insertar un *padding* en el contenido como característica de seguridad, mediante el campo opcional Pad Length cuando el bit P (PADDED=0x08) del campo Flags esté en 1.

El bit ES (END_STREAM=0x01) indica que luego de procesarse este Data Frame en el receptor, el Stream ID en cuestión se cierra, por lo que queda en un estado Half-Close en el sentido origen-destino (Diagrama de estados de un Stream: [1], Sección 5.1). Este es un caso de aplicación de la técnica de *Piggy-Backing*, ya que también este bit aparece en el Frame HEADERS, alterando el estado del Stream de la misma manera.

3.2.2. HEADERS Frame (0x01)



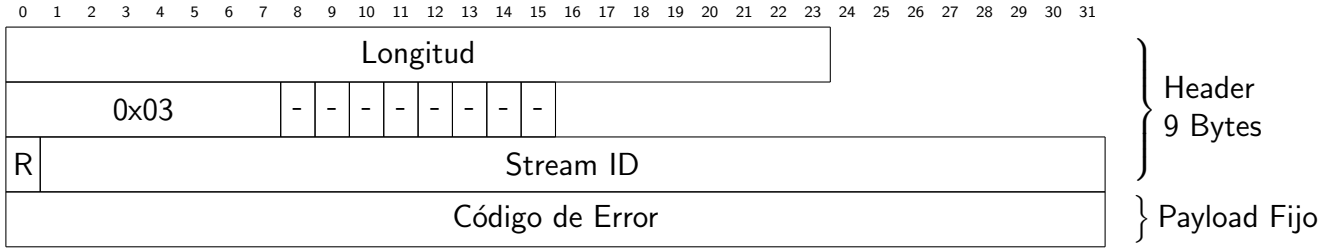
El HEADERS Frame se utiliza para hacer intercambiar peticiones y respuestas abriendo Streams, y tiene lugar en su payload para incluir los encabezados de las peticiones HTTP, codificados primero mediante el algoritmo de compresión HPACK [17], luego serializados a bytes, y por último fragmentados en 1 o más bloques que viajan en uno o más Frames de tipo HEADERS, PUSH_PROMISE o CONTINUATION. Tiene la misma característica respecto al padding opcional más el bit ES del Data Frame, utilizando los mismos campos.

[45] https://en.wikipedia.org/wiki/HTTP_message_body

En cambio, los campos Stream Dependency, bit E y Weight se utilizan cuando el bit de prioridad PR (PRIORITY=0x20) está en 1. Éstos campos permiten definir prioridades del Stream en cuestión (indicando su peso) y/o su dependencia diferentes Streams, con el fin de que el cliente realice peticiones apenas descubra los recursos que va a necesitar pero definiendo a la vez un orden de urgencia o prioridad. Esto permite que un navegador obtenga los recursos más importantes primero; por ejemplo, aquellos que le permiten dibujar la página, como CSS y código Javascript primero, imágenes después. Si el bit PR está en 0, el nuevo Stream depende del Stream 0x00, y se asigna implícitamente el peso por defecto que es 0x0F (16). Una posibilidad extra que permite HTTP/2 es repriorizar Streams existentes "al vuelo", mediante el PRIORITY Frame (0x02) ([1], Sección 6.3).

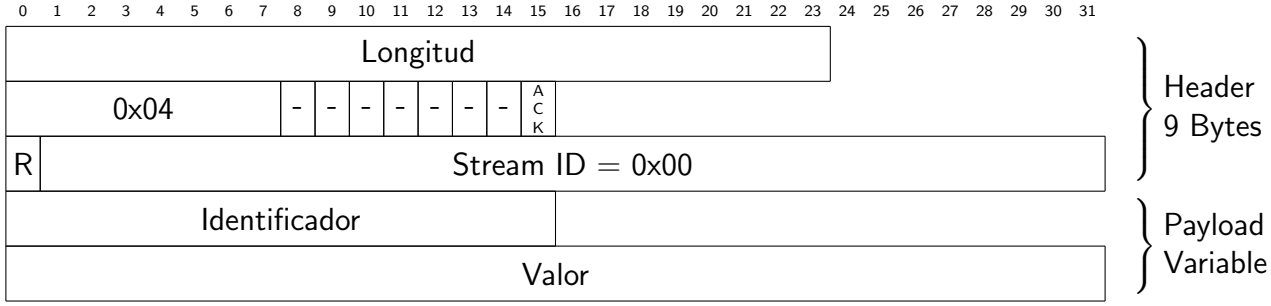
Dado que los headers – aún comprimidos – pueden requerir ser enviados en más de un HEADERS Frame, el bit EH especifica si todos los headers de la petición se encuentran este frame (EH=0x04) o no, con lo cual está en 0. En caso de no contener todos los headers, se utiliza un CONTINUATION Frame (0x09), un Frame específico para este caso ([1], Sección 6.10).

3.2.3. RST_STREAM Frame (0x03)



El RST_STREAM finaliza inmediatamente un Stream, cancelándolo y eventualmente enviando un código de error al destinatario. Los códigos de error definidos en el protocolo son varios ([1], Sección 7).

3.2.4. SETTINGS Frame (0x04)



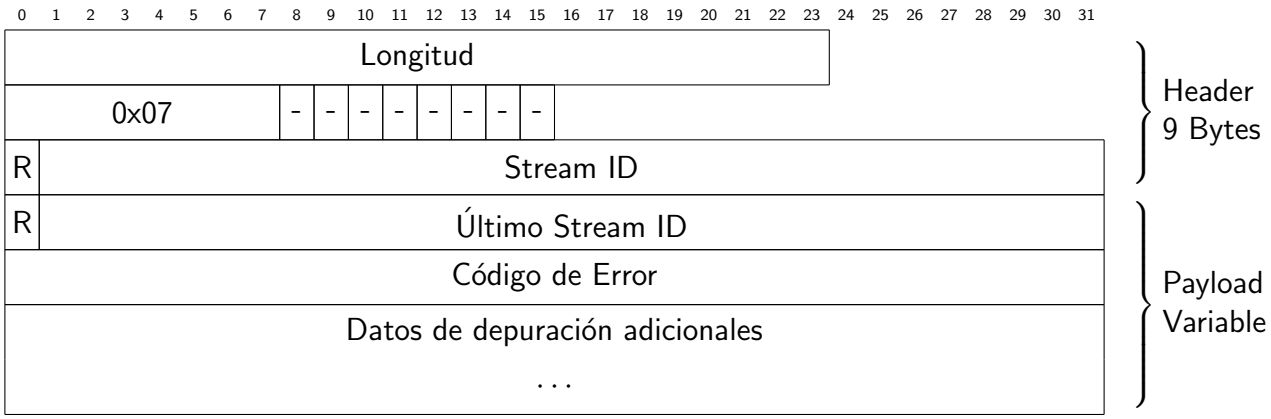
El SETTINGS Frame permite que un endpoint envíe diferentes parámetros globales relacionados con la sesión HTTP/2 al otro extremo, que son útiles para que el intercambio de tráfico sea fructífero; éstos deben confirmarse con otro SETTINGS Frame en el sentido inverso (Flag ACK=0x01), con su payload vacío (Longitud=0). Luego del inicio de la sesión es obligatorio que ambos pares intercambien un SETTINGS Frame, aunque también pueden aparecer en el medio de una sesión ya establecida; ante tal circunstancia, los nuevos valores reemplazan a los anteriores.

El Payload de este Frame cuando ACK=0x00 es un conjunto variable de "clave=valor", que permite que el *peer* que envía el frame le señalice distintos parámetros al receptor del mismo:

1. SETTINGS_HEADER_TABLE_SIZE (0x1): Es un parámetro del algoritmo de compresión de encabezados (HPACK). 4096 es el valor por defecto.
2. SETTINGS_ENABLE_PUSH (0x2): Permite habilitar/deshabilitar la característica de "SERVER_PUSH" y el Tipo de Frame PUSH_PROMISE asociado a esta característica (0x05). Activa por defecto.
3. SETTINGS_MAX_CONCURRENT_STREAMS (0x3): Define la cantidad máxima de streams concurrentes en la sesión.
4. SETTINGS_INITIAL_WINDOW_SIZE (0x4): Indica el tamaño de ventana que el *peer* origen tiene disponible para recibir información, y por consiguiente inicializa el control de flujo. Por defecto es 2^{16}-1 (65.535 Bytes).
5. SETTINGS_MAX_FRAME_SIZE (0x5): Define el tamaño máximo del payload de un frame que el origen está dispuesto a recibir. Es de 2^{14} (16 KB) por defecto, y el máximo permitido por el protocolo para su valor es 2^{24} (16 MB).
6. SETTINGS_MAX_HEADER_LIST_SIZE (0x6): Informa al destinatario del tamaño máximo de la lista de headers una vez descomprimidos, en Bytes. Por defecto, ilimitado.

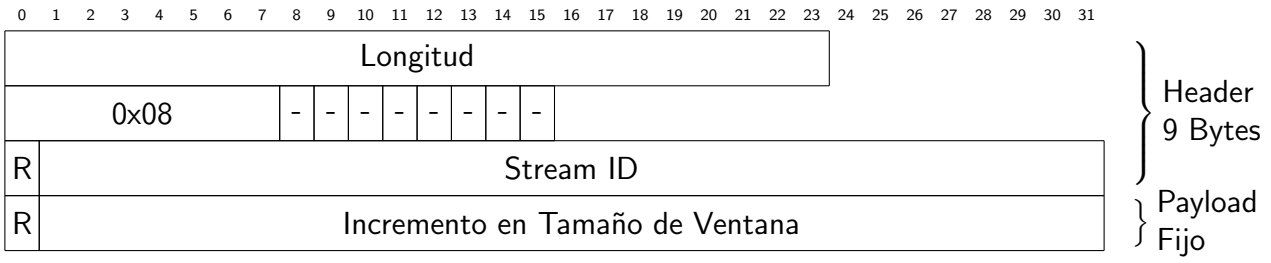
Dado que el payload de claves y valores es variable, pueden no estar todas las claves definidas, por lo que en caso de ausencia de clave/valor el destinatario asume los valores por defecto. Se trata de configuraciones en un sólo sentido, lógicamente permitiendo la asimetría de éstos hacia el lado contrario.

3.2.5. GOAWAY Frame (0x07)



GOAWAY es el frame que inicia el cierre de la conexión TCP, aunque también señaliza algunas condiciones condiciones de error. Una vez recibido este Frame, no se pueden iniciar más streams, quedando abiertos aquellos menores al definido en el campo Último Stream ID. Los códigos de error son los mismos del RST_STREAM Frame.

3.2.6. WINDOW_UPDATE Frame (0x08)



Este Frame permite implementar control de flujo, tanto a nivel global de la conexión (Stream ID=0x00), como a nivel de Stream individual (Stream ID>0x00), siempre aplicado para el Payload de los Frames de tipo DATA (0x00). El objetivo es que el par que envía datos lleve la cuenta de cuál es la capacidad de recepción del destinatario, por lo que tendrá prohibido enviar más información de lo que señala el valor de la ventana.

La ventana es por consiguiente un valor almacenado por cada uno de los pares que señala cuántos bytes es capaz de recibir el par del otro lado; existe una para cada Stream de forma individual, más una extra para toda la conexión en su totalidad. Este valor se actualiza en el origen al enviar un Frame de tipo DATA[46], y se incrementa cuando éste recibe un Frame de tipo WINDOW_UPDATE[47].

Ambos tipos de control de flujo –por conexión y por Streams– tienen lugar en conjunto, pero es importante que tener en consideración que el destinatario enviará dos tipos WINDOW_UPDATE por separado, uno por cada tipo de control de flujo, y probablemente en diferentes momentos, cuando lo considere necesario.

Si bien la ventana inicial luego de un inicio de sesión HTTP/2 es de 64KB (tanto para los Streams como para la conexión global), existe la particularidad que el primer SETTINGS Frame obligatorio posterior al inicio de sesión permite ajustar el valor de ventana para los Streams nuevos mediante SETTINGS_INITIAL_WINDOW_SIZE.

Por último, no es posible deshabilitar el control de flujo, aunque una manera *naive* de desactivarlo es definiendo el tamaño máximo de ventana igual a 2^{16}-1 (2MB) y enviando Frames de WINDOW_UPDATE luego de cada DATA que llegue.

3.2.7. Otros Frames

Hay otros tipos de Frame definidos por el protocolo, pero por su menor relevancia para el objetivo del presente documento se decidió dejar su definición de lado. Estos son:

- PRIORITY (0x02)
- PUSH_PROMISE (0x05)
- PING (0x06)
- CONTINUATION (0x09)

De todas maneras y como es lógico, se deja a criterio del lector consultar el RFC del protocolo y ahondar en la funcionalidad específica de cada uno.

[46] En esta situación se decrementa el valor de ambas ventanas al mismo tiempo: la de su Stream y la global, tantos bytes como el tamaño del Payload del Frame de tipo DATA.

[47] Se incrementarán tantos bytes adicionales como así lo señale el campo "Incremento en Tamaño de Ventana", para la ventana respectiva que indique ese Frame. Recordar que si el Stream ID=0x00 se trata de la ventana global.

4. Trabajando con HTTP/2

En este capítulo se describen algunas piezas de software como así también técnicas necesarias para estudiar el protocolo HTTP/2 y HPACK. Se recomienda contar con una instalación de una distribución GNU/Linux moderna[48]).

4.1. Herramientas de análisis y desarrollo

A continuación se introducirá brevemente en las herramientas y bibliotecas de desarrollo con las que se trabajará posteriormente.

Algunas referencias más se pueden obtener de esta página [49] aunque también el sitio donde tiene lugar el desarrollo del protocolo[50] mantiene una lista de Implementaciones de Software[51] y herramientas útiles afines a la especificación[52].

4.1.1. Nghttp2

Nghttp2[53] es una de las implementaciones de código abierto y multiplataforma del protocolo HTTP/2 y HPACK más utilizadas. Proporciona una biblioteca base en lenguaje C, cliente, servidor y proxy HTTP/2, una herramienta de benchmarking y testeo de carga, un servidor público de prueba en la web, APIs de más alto nivel en C++ y Python y más.

Fue adoptada por otros proyectos de software para incorporarle soporte de HTTP/2, como ser curl[54] y el servidor web Apache[55].

4.1.2. Curl

Curl[56] es una de las utilidades orientadas a la red que viene siendo desarrollada desde hace casi 20 años[57], incorporando soporte a infinidad de plataformas y protocolos a lo largo del tiempo. Es incluida en prácticamente todas las distribuciones de Linux, Mac OSX y mucho más, como ser dispositivos embebidos de cualquier tipo y proyectos de software de una multitud de fabricantes [58]. Como era de esperarse, acompañó el desarrollo de HTTP/2 y hoy lo soporta como un protocolo más.

4.1.3. Wireshark

Wireshark[59] es una de las herramientas de captura de tráfico de red más utilizdas. Soporta HTTP/2 nativamente desde la versión 1.12.

[48] Ubuntu 16.04, Debian Stretch, Fedora 24 o posteriores

[49] Tools for debugging, testing and using HTTP/2: https://blog.cloudflare.com/tools-for-debugging-testing-and-using-http-2/

[50] https://github.com/http2/

[51] https://github.com/http2/http2-spec/wiki/Implementations

[52] https://github.com/http2/http2-spec/wiki/Tools

[53] https://nghttp2.org/

[54] https://curl.haxx.se/docs/http2.html

[55] http://httpd.apache.org/docs/current/howto/http2.html#building

[56] https://curl.haxx.se/

[57] https://curl.haxx.se/docs/history.html

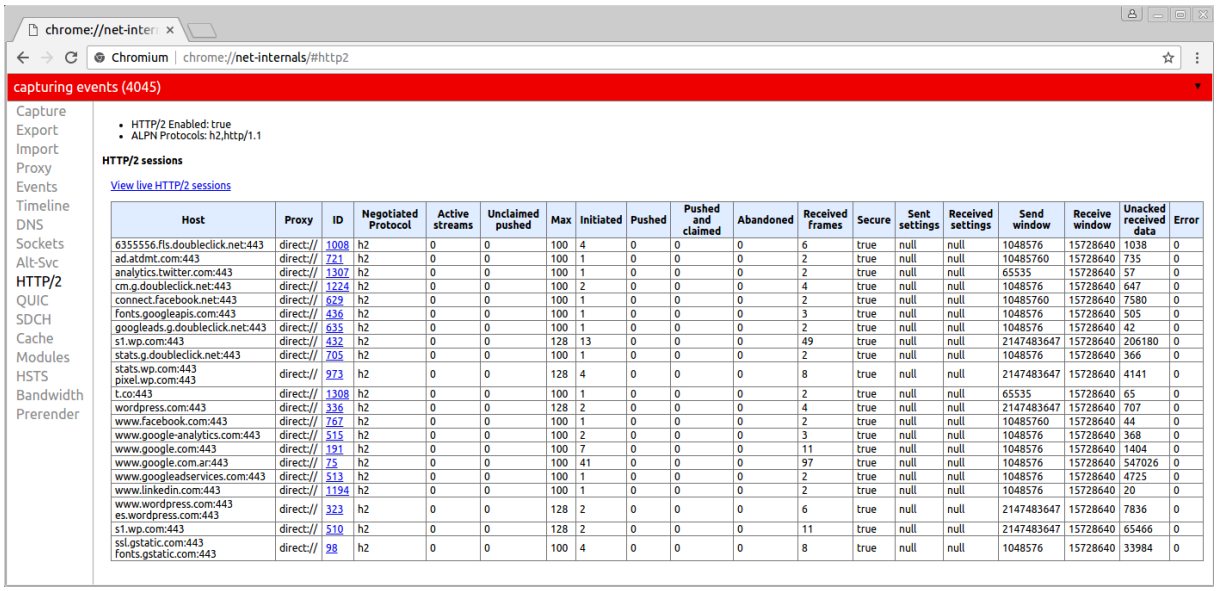
[58] https://curl.haxx.se/docs/companies.html

[59] https://www.wireshark.org/

4.1.4. Página net-internals de Chrome/Chromium

Si bien el objetivo principal del proyecto Chromium[60] es el de ser una aplicación para navegar la web, al ir incorporando nativamente herramientas para el desarrollo de páginas web, fue creciendo la necesidad de comprender cómo es el intercambio de tráfico en las capas inferiores, más aún cuando el uso de cifrado SSL/TLS dificulta la captura con utilidades tradicionales, donde las conexiones TCP contienen tráfico en protocolo HTTP sin cifrar.

En una de las páginas "especiales"[61] de Chrome/Chromium, además de poder acceder a diferentes cuestiones de manejo interno del módulo de red de la aplicación, se puede obtener y/o exportar una captura de tráfico, visualizar las sesiones HTTP/2 que mantiene concurrentemente el navegador, inspeccionar rápidamente algunas características de estas sesiones, etc.



Host	Proxy	ID	Negotiated Protocol	Active streams	Unclaimed pushed	Max	Initiated	Pushed	Pushed and claimed	Abandoned	Received frames	Secure	Sent settings	Received settings	Send window	Receive window	Unacked received data	Error
6355556.fls.doubleclick.net:443	direct://	1008	h2	0	0	100	4	0	0	0	6	true	null	null	1048576	15728640	1038	0
ad.atdmt.com:443	direct://	721	h2	0	0	100	1	0	0	0	2	true	null	null	10485760	15728640	735	0
analytics.twitter.com:443	direct://	1307	h2	0	0	100	1	0	0	0	2	true	null	null	65535	15728640	57	0
cm.g.doubleclick.net:443	direct://	1224	h2	0	0	100	2	0	0	0	4	true	null	null	1048576	15728640	647	0
connect.facebook.net:443	direct://	629	h2	0	0	100	1	0	0	0	2	true	null	null	10485760	15728640	7580	0
fonts.googleapis.com:443	direct://	436	h2	0	0	100	1	0	0	0	3	true	null	null	1048576	15728640	505	0
googleads.g.doubleclick.net:443	direct://	635	h2	0	0	100	1	0	0	0	2	true	null	null	1048576	15728640	42	0
s1.wp.com:443	direct://	432	h2	0	0	128	13	0	0	0	49	true	null	null	2147483647	15728640	206180	0
stats.g.doubleclick.net:443	direct://	705	h2	0	0	100	1	0	0	0	2	true	null	null	1048576	15728640	366	0
stats.wp.com:443 pixel.wp.com:443	direct://	973	h2	0	0	128	4	0	0	0	8	true	null	null	2147483647	15728640	4141	0
t.co:443	direct://	1308	h2	0	0	100	1	0	0	0	2	true	null	null	65535	15728640	65	0
wordpress.com:443	direct://	336	h2	0	0	128	2	0	0	0	4	true	null	null	2147483647	15728640	707	0
www.facebook.com:443	direct://	767	h2	0	0	100	1	0	0	0	2	true	null	null	10485760	15728640	44	0
www.google-analytics.com:443	direct://	515	h2	0	0	100	2	0	0	0	3	true	null	null	1048576	15728640	368	0
www.google.com:443	direct://	191	h2	0	0	100	7	0	0	0	11	true	null	null	1048576	15728640	1404	0
www.google.com.ar:443	direct://	75	h2	0	0	100	41	0	0	0	97	true	null	null	1048576	15728640	547026	0
www.googleadservices.com:443	direct://	513	h2	0	0	100	1	0	0	0	2	true	null	null	1048576	15728640	4725	0
www.linkedin.com:443	direct://	1194	h2	0	0	100	1	0	0	0	2	true	null	null	1048576	15728640	20	0
www.wordpress.com:443 es.wordpress.com:443	direct://	323	h2	0	0	128	2	0	0	0	6	true	null	null	2147483647	15728640	7836	0
s1.wp.com:443	direct://	510	h2	0	0	128	2	0	0	0	11	true	null	null	2147483647	15728640	65466	0
ssl.gstatic.com:443 fonts.gstatic.com:443	direct://	98	h2	0	0	100	4	0	0	0	8	true	null	null	1048576	15728640	33984	0

Figura 10: Sección HTTP/2 de página net-internals de Chromium v57

4.1.5. Nginx Web Server

El Servidor Web Nginx[62] es el segundo más utilizado entre los sitios activos de Internet[63], detrás de Apache. Dada la manera en que éste es distribuido y actualizado, resulta por lo general el software adecuado para proveer servicio web sobre HTTP/2.

Actualmente no soporta Server Push, pero su implementación es estable y va madurando cada vez más a medida que pasa el tiempo.

4.2. Recuperación de una página web mediante HTTP/2

Como se dijo anteriormente, para capturar una sesión HTTP/2 sobre TLS no sólo se requiere contar con la captura, sino también se necesita descrifrar el contenido del payload de los segmentos TCP.

[60] http://www.chromium.org/Home
[61] chrome://net-internals
[62] http://nginx.org/
[63] https://news.netcraft.com/archives/2017/03/24/march-2017-web-server-survey.html

La manera más sencilla de hacer esto es solicitando a la aplicación que utilice su biblioteca criptográfica almacenando la clave maestra simétrica (*"master key"*) de la sesión TLS en un archivo en texto claro, con un formato estándar[64]. Existen varias bibliotecas criptográficas de código abierto muy utilizadas por navegadores, servidores web, etc., pero las más importantes son: OpenSSL, NSS, GnuTLS más algunas derivadas de ellas. Todas soportan el mismo método, que consiste en:

- Definir la variable de entorno `SSLKEYLOGFILE` y referenciarla a un archivo en el disco.
- Ejecutar la aplicación dentro del mismo entorno, de manera tal que ésta pueda leer la variable definida y configurar su biblioteca de criptografía en "modo depuración", guardando las claves de las sesiones TLS en este archivo.
- Realizar la captura normalmente, en otra ventana.

Por ejemplo, de esta manera se guardarán las claves de las sesiones TLS de Firefox en el archivo /tmp/sslkeylog.log, y se capturará todo el tráfico desde/hacia el puerto 443 contra https://www.google.com.ar:

```
alumno@unlu-http2:~$ export SSLKEYLOGFILE=/tmp/sslkeylog.log
alumno@unlu-http2:~$ firefox --private-window https://www.google.com.ar
```

Y en otra terminal, iniciamos la captura

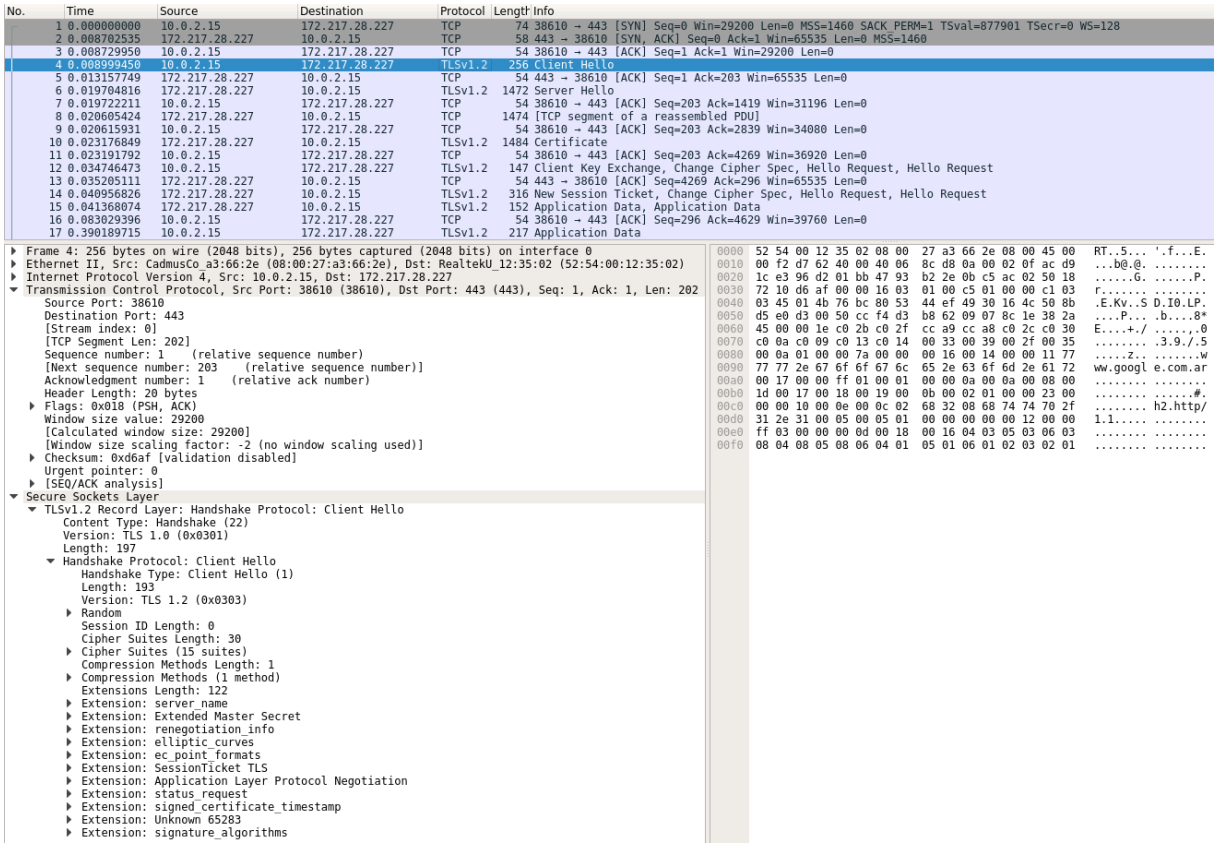
```
alumno@unlu-http2:~$ tshark -w /tmp/firefox-tls.cap tcp port 443 and host www.google.com.ar
Capturing on 'enp0s3'
[...] ^C
www.google.com.ar has address 172.217.28.227
www.google.com.ar has IPv6 address 2800:3f0:4002:800::2003
alumno@unlu-http2:~$
```

Advertencia: Esto hará que **todas** las comunicaciones mediante TLS realizadas por esa instancia de Firefox podrán ser descrifradas teniendo una captura de su tráfico y el correspondiente archivo *sslkeylogfile*, incluyendo cookies de sesión, información de formularios y demás. Es por esto que se sugiere establecer la variable de ambiente `SSLKEYLOGFILE` en una ventana de la terminal, ejecutar el navegador allí mismo, hacer la captura, y cerrarlo. De ninguna manera es aconsejable establecer esta variable de ambiente de manera permanente para el uso diario.

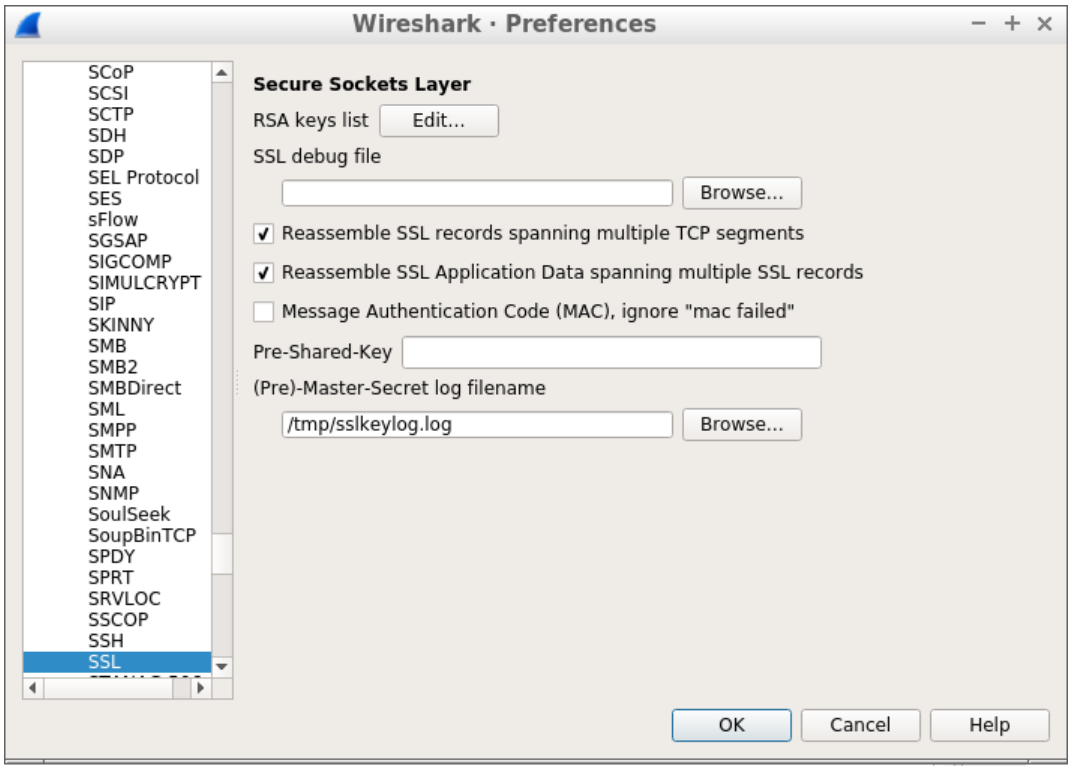
4.3. Análisis de una captura con Wireshark 2

Una vez obtenida la captura y el archivo *sslkeylogfile* de una sesión HTTP/2, debe abrirse la captura en Wireshark. Al hacer esto, se podrá ver que éste no puede descrifrar más allá de cierto tráfico TLS:

[64] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Key_Log_Format

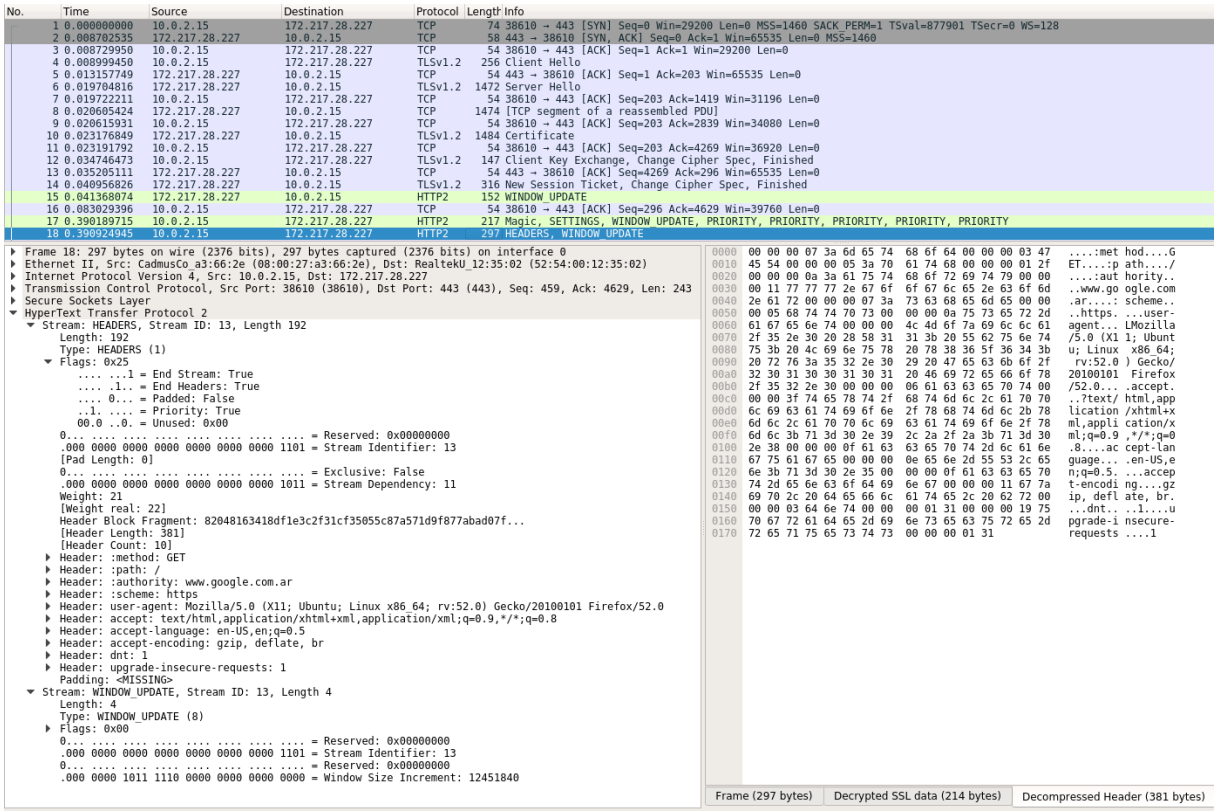


Para cargar el *sslkeylog*, hay que ir al menú Edit→Preferences→Protocols→SSL, como en la siguiente figura, seleccionar el archivo /tmp/sslkeylog.log en el campo "(Pre)-Master-Secret log filename", y clickear OK:

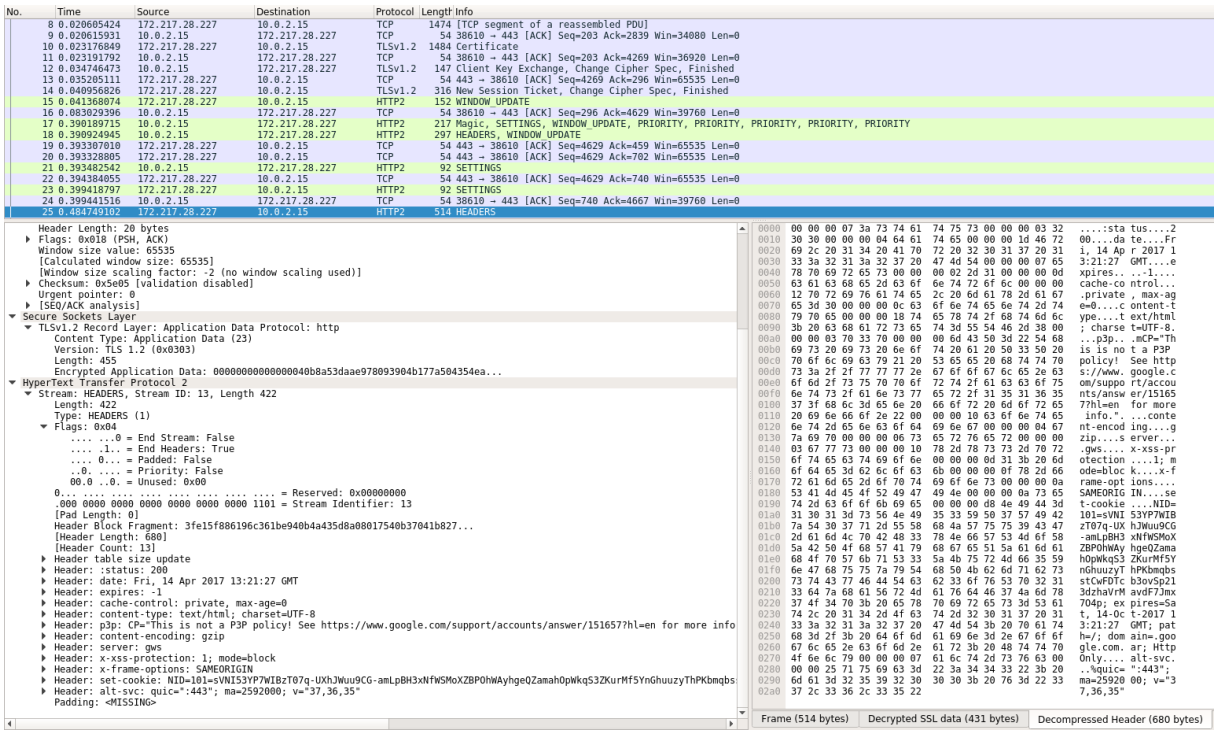


Luego de esto, Wireshark ya puede decodificar el contenido cifrado de la captura, recono-

ciendo y diseccionando el protocolo HTTP/2. Aquí se puede ver el GET / a www.google.com.ar, con los encabezados tradicionales HTTP de Firefox:



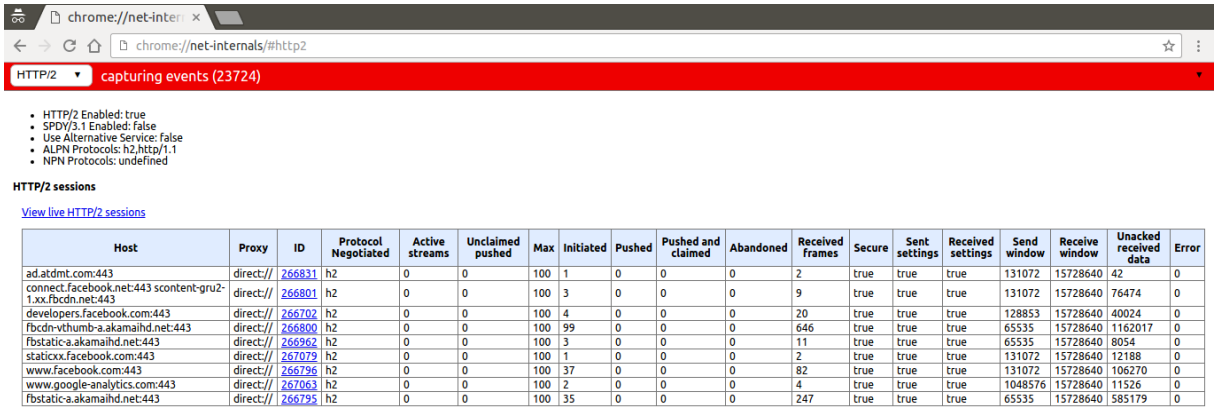
Se puede identificar que dentro del mismo Segmento TCP se incluyeron 2 Frames HTTP/2, uno de HEADERS y otro de WINDOW UPDATE, ambos referenciando al Stream ID 13. Y aquí se ve la respuesta del servidor, en un Frame de tipo HEADERS, también sobre el mismo Stream ID:



4.4. Frames y Streams de una sesión HTTP/2 con Chrome/Chromium

Si en Chrome/Chromium se abre la página interna de estadísticas HTTP2 de la capa de red[65], es posible visualizar un cuadro resumiendo la cantidad de:

- Streams iniciados por el browser (*Initiated*),
- Streams iniciados por el servidor mediante push (*Pushed*),
- Frames recibidos (*Received Frames*).



Host	Proxy	ID	Protocol Negotiated	Active streams	Unclaimed pushed	Max	Initiated	Pushed	Pushed and claimed	Abandoned	Received frames	Secure	Sent settings	Received settings	Send window	Receive window	Unacked received data	Error
ad.atdmt.com:443	direct://	266831	h2	0	0	100	1	0	0	0	2	true	true	true	131072	15728640	42	0
connect.facebook.net:443 scontent-gru2-1.xx.fbcdn.net:443	direct://	266801	h2	0	0	100	3	0	0	0	9	true	true	true	131072	15728640	76474	0
developers.facebook.com:443	direct://	266702	h2	0	0	100	4	0	0	0	20	true	true	true	128853	15728640	40024	0
fbcdn-vthumb-a.akamaihd.net:443	direct://	266800	h2	0	0	100	99	0	0	0	646	true	true	true	65535	15728640	1162017	0
fbstatic-a.akamaihd.net:443	direct://	266962	h2	0	0	100	3	0	0	0	11	true	true	true	65535	15728640	8054	0
staticxx.facebook.com:443	direct://	267079	h2	0	0	100	1	0	0	0	2	true	true	true	131072	15728640	12188	0
www.facebook.com:443	direct://	266796	h2	0	0	100	37	0	0	0	82	true	true	true	131072	15728640	106270	0
www.google-analytics.com:443	direct://	267063	h2	0	0	100	2	0	0	0	4	true	true	true	1048576	15728640	11526	0
fbstatic-a.akamaihd.net:443	direct://	266795	h2	0	0	100	35	0	0	0	247	true	true	true	65535	15728640	585179	0

Figura 11: Sesiones HTTP2 en una sesión de Chrome/Chromium

Además, haciendo click en un ID de sesión/conexión, se puede ver el detalle de frames enviados/recibidos con los headers correspondientes en aquellos en los que aplica. Tomar nota del stream_id que identifica la "conexión virtual" en varios frames distintos.

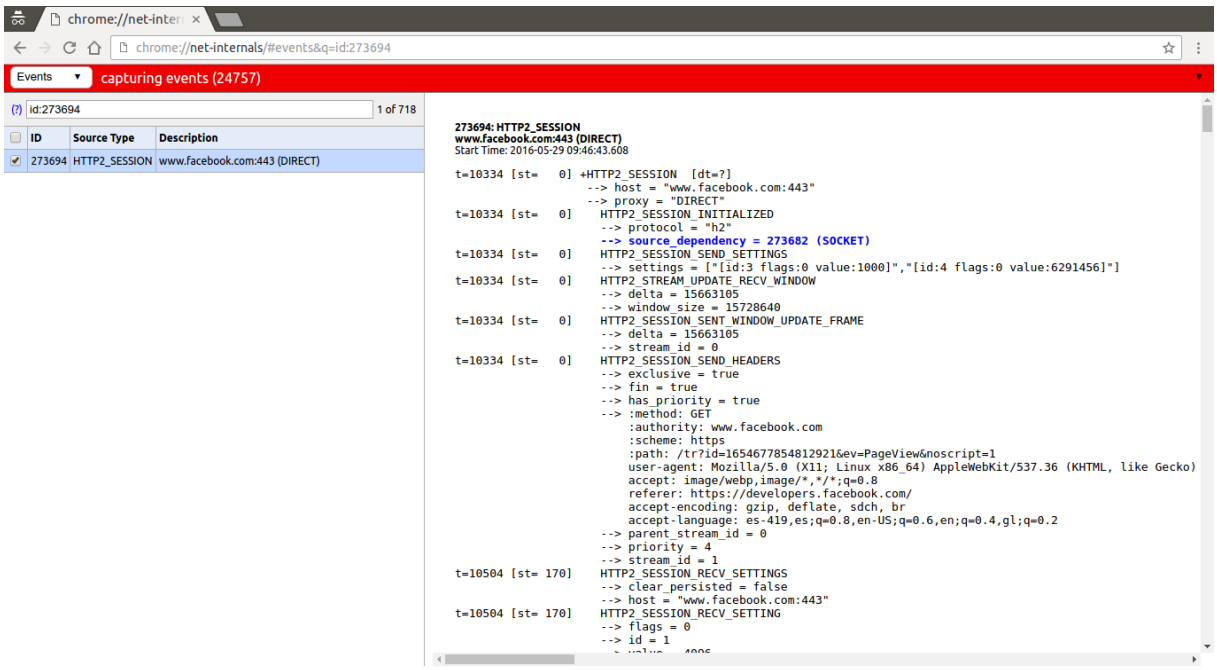


Figura 12: Detalle de una sesión HTTP2 en Chrome/Chromium

[65] chrome://net-internals/#http2

Se sugiere realizar esta navegación en una ventana privada para no mezclar las sesiones de navegación con las del usuario.

4.5. Identificando los bloqueos Head-of-Line de HTTP/1.x

Los bloqueos HoL tienen que ver con la cantidad de veces que el browser tuvo que esperar por tener un *pipe* TCP disponible para hacer efectiva la petición de un recurso. Sabiendo esto, en el gráfico de cascada de la carga de una página[66], el browser da el tiempo de "Queueing" que tuvo que esperar por la red. Es posible ver rápidamente que en la carga de una página mediante HTTP/1.1, los tiempos de encolamiento existen y son relevantes, mientras que en HTTP/2 son prácticamente nulos[67].

Los dos gráficos de cascada siguientes muestran la diferencia en los tiempos de encolamiento de los recursos según la carga de la página mediante HTTP/1.1 y HTTP/2. Nótese que en el gráfico de HTTP/1.1, las barras "blancas" o "transparentes" que refieren este encolamiento o espera en la descarga por parte del navegador aparecen luego de las primeras 6 conexiones TCP que se abren.

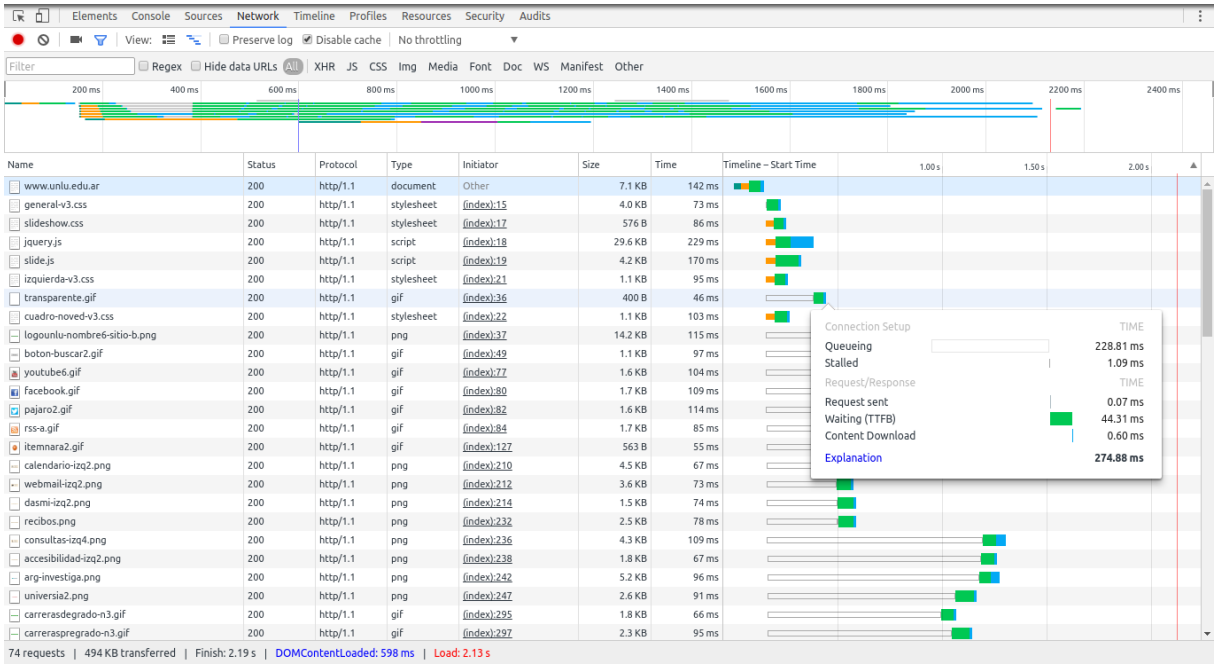


Figura 13: Queueing de un recurso HTTP/1.1 obtenido de `http://www.unlu.edu.ar`

En cambio, en HTTP/2. prácticamente no hay tiempo de espera por la disponibilidad de la conexión.

[66] Se accede al gráfico en cascada de una página dentro del apartado "Network", mediante las "Herramientas del Desarrollador", tanto en Chrome/Chromium como en Firefox, con la combinación de teclas Ctrl+Shift+I

[67] https://developers.google.com/web/tools/chrome-devtools/profile/network-performance/resource-loading#view-details-for-a-single-resource

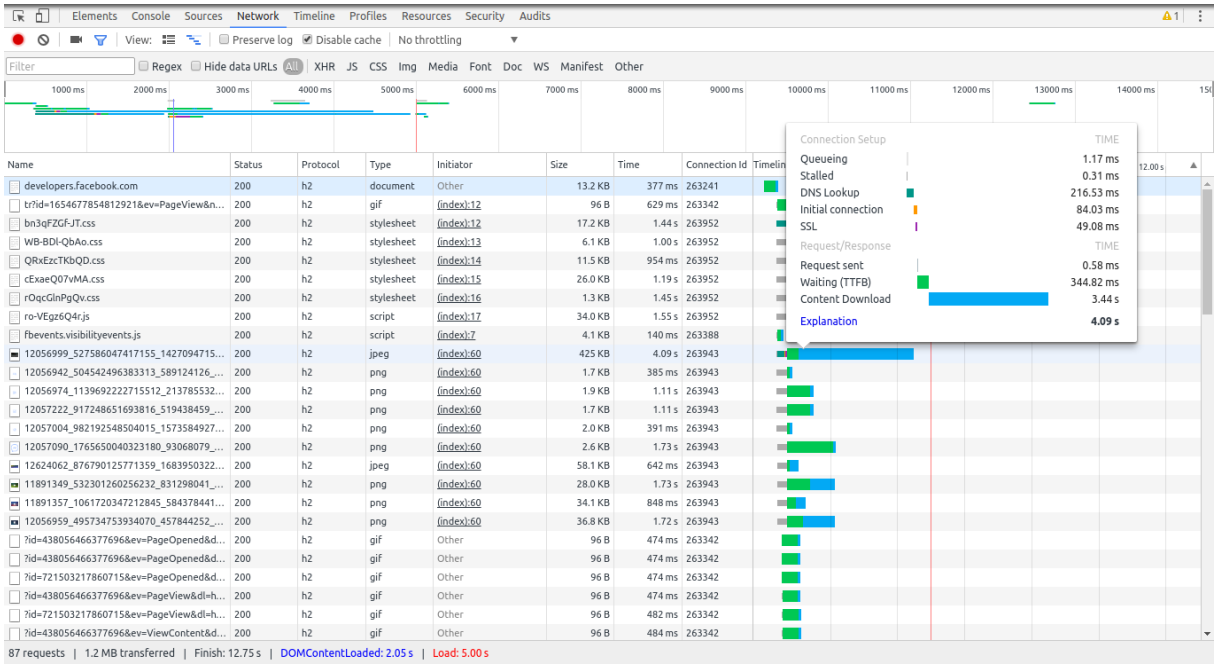


Figura 14: Queueing de un recurso HTTP/2 obtenido de https://developers.facebook.com/videos

4.6. Configurando un sitio HTTP/2 con Nginx

La instalación de Nginx es bastante sencilla, y está bien documentada[68]. Provee repositorios para una gran cantidad de distribuciones distintas de GNU/Linux, como así para otros sistemas operativos. A fines de obtener el mejor balance de características, actualizaciones de seguridad y estabilidad, se usará la release *mainline* de Nginx[69].

La configuración por defecto se encuentra en `/etc/nginx/conf.d/default.conf`:

```
server {
  listen       80;
  server_name  localhost;

  [...] # Directivas de logging, charset por defecto

  location / {
  root   /usr/share/nginx/html;
  index  index.html index.htm;
  }

  [...] # Directivas de páginas de 400, 500, etc.
}
```

Como se puede deducir, por defecto el servicio escucha en el puerto 80, no tiene ningún tipo de capa de cifrado TLS y sirve estáticamente el contenido del directorio `/usr/share/nginx/html`. Para agregar soporte de HTTP/2 en Nginx debe configurarse el sitio con TLS, para lo cual es necesario:

- Instalar el paquete `ssl-cert`. Esto hará que el sistema genere un par de certificados auto-firmados, que servirá para configurar un entorno de pruebas y análisis[70]. Éstos residirán en `/etc/ssl/certs/ssl-cert-snakeoil.pem` y `/etc/ssl/private/ssl-cert-snakeoil.key`.

[68] https://www.nginx.com/resources/wiki/start/topics/tutorials/install/
[69] http://nginx.org/en/linux_packages.html
[70] Este tipo de certificados no sirven en absoluto para un entorno productivo, ver [14]

- Hacer que el servicio escuche en el puerto 443 y definir que allí usará ssl (TLS) y HTTP/2.
- Configurar en este puerto/servicio los certificados instalados.

Los dos últimos ítems se definen exclusivamente en el archivo de configuración default.conf, agregando lo siguiente al final del archivo:

```
server {
  listen 443 ssl http2;

  ssl_certificate /etc/ssl/certs/ssl-cert-snakeoil.pem;
  ssl_certificate_key /etc/ssl/private/ssl-cert-snakeoil.key;

  location / {
    root   /usr/share/nginx/html;
    index  index.html index.htm;
  }
}
```

Luego, se debe reiniciar el servicio e ingresar con el navegador a `https://localhost`. Aparecerá una advertencia de seguridad del navegador, que tiene que ver con la naturaleza de los certificados, que son autofirmados. Continuamos con la carga de la página, ignorando estos mensajes, y finalmente se puede ver el index.html de ejemplo de Nginx.

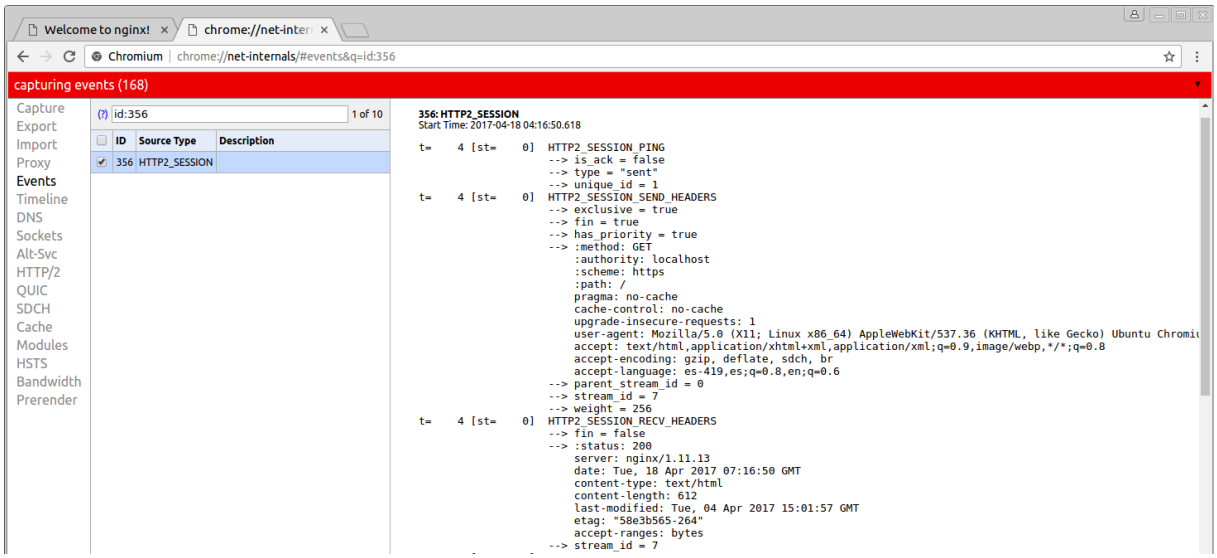


Figura 15: Vista de la sección *net-internals* de la sesión HTTP/2 contra Nginx

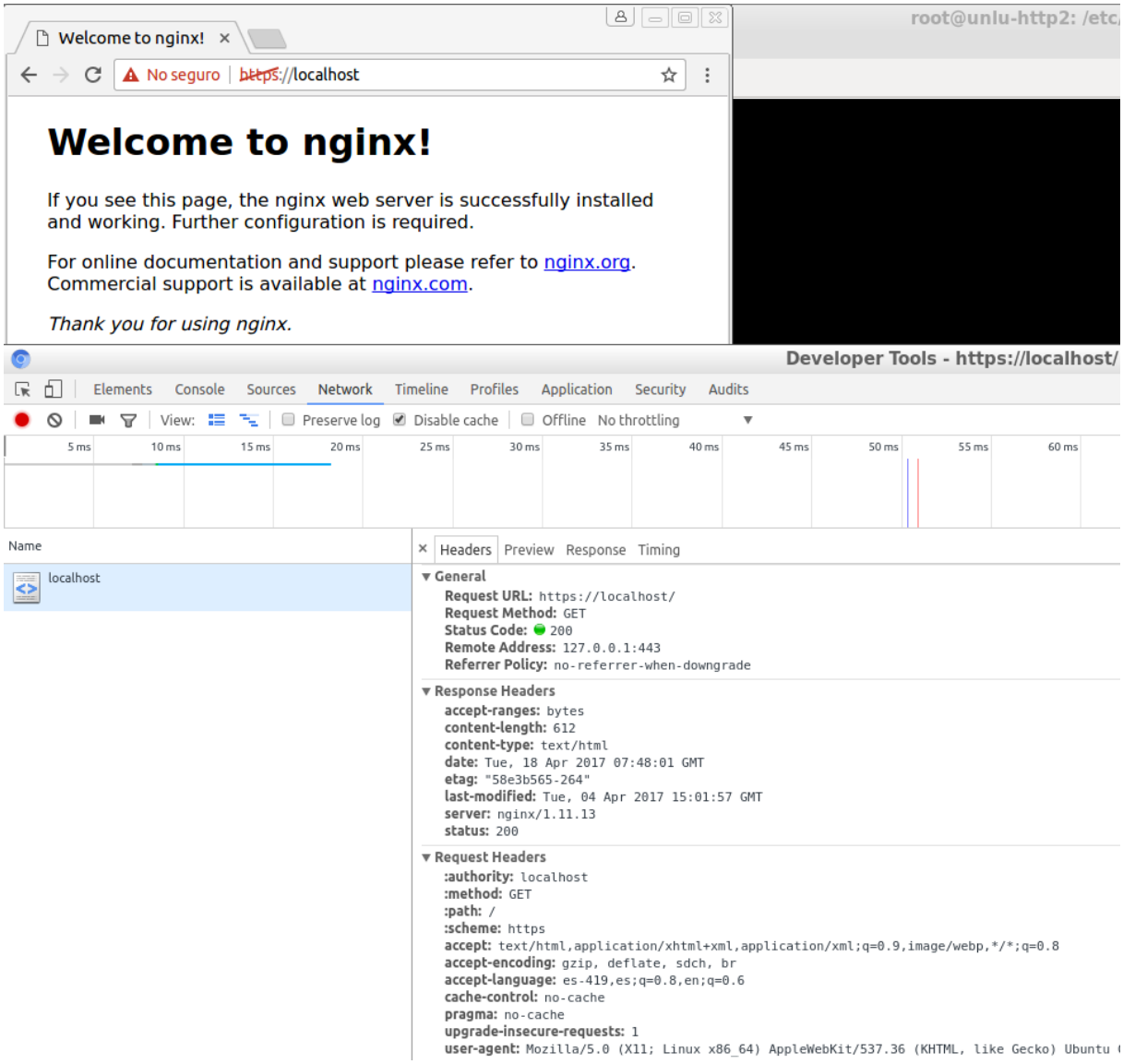


Figura 16: Bienvenida de Nginx con las Developer Tools abiertas, mostrando el uso de HTTP/2

El módulo de HTTP/2 posee opciones de configuración, que permiten hacer ajustes finos del protocolo[71]. Además, intencionalmente no se profundizó en la configuración en detalle de TLS, dado lo extenso que resultaría, aunque se pueden consultar estas referencias[72] [73]. Otra opción muy utilizada que se ha dejado de lado intencionalmente es configurar al servidor en el puerto 80 para que redireccione automáticamente mediante un HTTP 301/302 a la versión sobre TLS del sitio en el puerto 443.

[71] https://nginx.org/en/docs/http/ngx_http_v2_module.html
[72] https://wiki.mozilla.org/Security/Server_Side_TLS
[73] https://mozilla.github.io/server-side-tls/ssl-config-generator/

A. Ejercicios Propuestos

A.1. Recuperación y análisis de páginas web vía HTTP/2 vs. HTTP/1.x

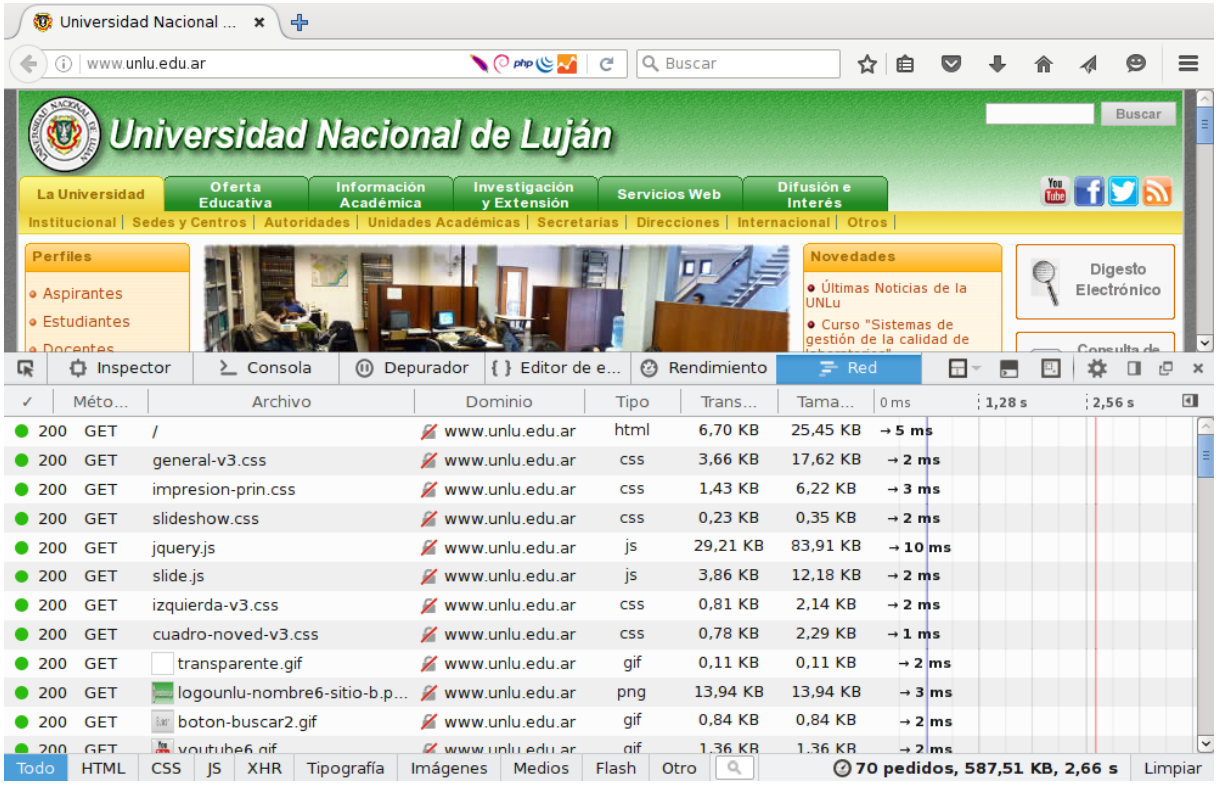
Utilizando un navegador web moderno (Google Chrome[74] o Mozilla Firefox[75]) con soporte para protocolo HTTP/2 activado, en modo incógnito (ej: `google-chrome --incognito` o `firefox --private-window`) con las herramientas de desarrollo activas y la pestaña "Red" activa, acceda al siguiente sitio:

- https://developers.facebook.com/videos/

A.1.1. Indique:

- Sitio web y dirección URL solicitada.
- Protocolo utilizado para la solicitud (HTTP/1.0, HTTP/1.1 o HTTP/2)
- Cantidad de recursos obtenidos.
- Tamaño total de los datos transferidos.
- Cantidad de streams y frames HTTP/2 utilizados.
- Tiempo transcurrido desde que el navegador recibió la orden de navegar dicha URL.
- Si sucede *Head-of-Line Blocking*, y en tal caso, en qué instante de tiempo.

Incluya en la resolución del trabajo una captura de pantalla de la cascada de peticiones, donde se muestre la petición a la URL original y los primeros recursos solicitados, similar a la siguiente:



[74] https://www.google.com/chrome/
[75] https://www.mozilla.org/es-AR/firefox/

A.1.2. Repita la operación anterior, deshabilitando el soporte para HTTP/2, y compare los gráficos de cascada.

Si utiliza Google Chrome, ejecútelo con:

```
google-chrome --disable-http2 --incognito
```

Si utiliza Mozilla Firefox, inicie el navegador en forma habitual, acceda a la dirección `about:config`, confirme el acceso a la configuración, busque las opciones `network.http.spdy.enabled.http2` y `security.ssl.enable\_alpn`, establezca ambas en `false`. Luego reinicie el navegador Firefox y acceda a los sitios solicitados.

A.1.3. Repita las operaciones efectuadas en los dos puntos anteriores con las direcciones siguientes:

- https://www.google.com.ar/webhp
- https://www.denizmotorum.com/

Incorpore en una tabla las métricas obtenidas para las tres direcciones (cantidad de recursos, tamaño de los datos transferidos, cantidad de frames, tiempo requerido, etc.). Compare entre sí las mismas y esboce una explicación acerca de los resultados obtenidos.

A.1.4. Utilizando el navegador web Google Chrome, navegue a las siguientes direcciones en dos nuevas pestañas:

- chrome://net-internals/#http2
- chrome://net-internals/#events&q=type:HTTP2_SESSION%20is:active

Luego, abra una nueva pestaña y navegue a la dirección https://wordpress.com/. Busque en la pestaña *net-internals* la sesión HTTP2_SESSION correspondiente al dominio `wordpress.com:443` y haga clic en ella. Analice la traza informada en la ventana (en el panel derecho), indique la cantidad de streams y de frames, y represente en un diagrama de secuencia de mensajes el diálogo intercambiado entre el agente de usuario (*User-Agent*) y el servidor web (*Web Server*).

A.2. Obtención de una página web sobre HTTP/2 mediante curl

Ejecute una terminal de comandos y verifique que la utilidad curl contenida (un cliente de línea de comandos que permite hacer peticiones y transferir archivos y datos mediante varios protocolos) implementa el protocolo HTTP/2. Por ejemplo:

```
$ curl --version
curl 7.53.1 (x86_64-pc-linux-gnu) libcurl/7.53.1 OpenSSL/1.0.2g zlib/1.2.8 nghttp2/1.21.1
Protocols: dict file ftp ftps gopher http https imap imaps [...] smb smbs smtp smtps telnet tftp
Features: IPv6 Largefile NTLM NTLM_WB SSL libz TLS-SRP HTTP2 UnixSockets HTTPS-proxy
```

A continuación, efectúe la descarga en modo "verbose" de una página web desde un servidor web que (sabe que) soporta protocolo HTTP/2 sin TLS. Por ejemplo:

```
$ curl --http2 -v http://nghttp2.org
Connected to nghttp2.org (...)
...
```

La salida de este comando es una traza de las acciones efectuadas para obtener el recurso.

Analice la traza, indique la cantidad de streams y de frames, y represente en un diagrama de secuencia de mensajes (MSC Chart) el diálogo intercambiado entre el agente de usuario (*User-Agent*) y el servidor web (*Web-Server*).

A.3. Recuperar una página de ejemplo y analizar captura

Recuperar, capturando todo el tráfico y las claves TLS para su posterior descifrado y análisis, la siguiente página con HTTP/2: https://www.google.com.ar/intl/es-419/about/.

Luego, responder:

- Con qué método (soporte conocido de antemano, HTTP/1.1 UPGRADE, TLS) se estableció la sesión HTTP/2.
- En el caso de que la sesión haya sido negociada mediante TLS, ¿qué protocolo se usó, NPN o ALPN? ¿En qué capas y tramas puntuales se negocia?
- ¿Qué tipos distintos de Frame identifica en la captura? Describa brevemente para qué se utiliza cada uno en dicho contexto.
- Realizando una captura de la misma página con Firefox y con Chrome/Chromium: ¿puede ver alguna diferencia a simple vista al momento de establecer la sesión HTTP/2 en cuanto a los Frames enviados hacia el servidor?

Referencias

[1] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015.

[2] Michael Belshe. More bandwidth doesn't matter (much). Technical report, Google Inc., 2010.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[4] V. Dukhovni. Opportunistic Security: Some Protection Most of the Time. RFC 7435 (Informational), December 2014.

[5] R. Fielding, Y. Lafon, and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Range Requests. RFC 7233 (Proposed Standard), June 2014.

[6] R. Fielding, M. Nottingham, and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching. RFC 7234 (Proposed Standard), June 2014.

[7] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Authentication. RFC 7235 (Proposed Standard), June 2014.

[8] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. RFC 7232 (Proposed Standard), June 2014.

[9] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard), June 2014.

[10] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231 (Proposed Standard), June 2014.

[11] S. Friedl, A. Popov, A. Langley, and E. Stephan. Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension. RFC 7301 (Proposed Standard), July 2014.

[12] Ilya Grigorik. Making the web faster with http 2.0. *Commun. ACM*, 56(12):42–49, December 2013.

[13] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608 (Proposed Standard), June 1999. Updated by RFC 3224.

[14] Russ Housley and Tim Polk. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2001.

[15] Google Inc. Spdy: An experimental protocol for a faster web. Technical report, Google Inc., 2010.

[16] Robert L. R. Mattson and Somnath Ghosh. Http-mplex: An enhanced hypertext transfer protocol and its performance evaluation. *J. Netw. Comput. Appl.*, 32(4):925–939, July 2009.

[17] R. Peon and H. Ruellan. HPACK: Header Compression for HTTP/2. RFC 7541 (Proposed Standard), May 2015.

[18] Steve Sounders. *High Performance Web Sites*. O'Reilly Media., 2007.

[19] Daniel Stenberg. Http2 explained. *SIGCOMM Comput. Commun. Rev.*, 44(3):120–128, July 2014.