Midiendo la Performance de SPDY

Pablo Maximiliano Lulic

26 de diciembre de 2013

Resumen

Actualmente, la Web se sustenta con el estándar HTTP. Este protocolo tuvo su última versión en el año 1999. Lás páginas en ese entonces eran muy diferentes a las actuales, tanto en contenido como en cantidad de recursos. Google desarrolló un protocolo en el año 2009 llamado SPDY, cuyo propósito es mejorar la performance en la recuperación de los recursos de la web. A pesar de su gran aceptación, y de sentar las bases del próximo HTTP 2.0, hay ciertas cuestiones que todavía quedan por revisar. En este paper se propone evaluar, la performance de SPDY desde dos enfoques diferentes, uno en ambientes específicos y otro en la web.

1. Introducción

El protocolo HTTP tuvo su primera versión en mayo de 1996, culminando en 1999 con el estándar actual que es el HTTP 1.1 [5]. El mismo, es un protocolo sin estado, el servidor no mantiene información acerca de las diferentes peticiones que le llegan, lo que conlleva a que se necesite realizar una petición nueva por cada recurso que se necesite de un sitio web, incluso siendo del mismo dominio. Esto genera tráfico innecesario entre petición y petición cuando el cliente necesita establecer una conexión nueva para obtener el recurso. Con SPDY esta cuestión se ha resuelto y, junto con otras características que se verán mas adelante, se mejora la performance de la recuperación de un sitio web.

Se realizarán 2 experimentos, en ambos se evaluarán los tiempos de carga de diferentes sitios (detallados más adelante), utilizando las siguientes medidas:

1. onContentLoad: Tiempo en el que el contenido del sitio se carga.

2. onLoad: Tiempo en el que es sitio se carga según el navegador.

3. ToW (Time on Wire): Tiempo en el cable, medido desde el primer paquete que se envía al servidor hasta el último.

Se busca comparar la performance de SPDY frente a HTTP y HTTPS. El primer experimento se realizará en un ambiente controlado, emulando diferentes combinaciones de Ancho de Banda y Retardo, peticionando sitios descargados previamente. El segundo experimento se realizará en un ambiente real, recuperando sitios populares.

2. La Web en la Actualidad

En comparación con lo que era la web en la época en la que se implementó el protocolo HTTP, hubo un marcado incremento en el tamaño y en la cantidad de recursos de un sitio web. Para Noviembre de 2013 [4], el tamaño promedio de un sitio era de 1614kb, en contraste con Noviembre de 2010 donde promediaba 702kb, hubo un incremento del 50 %. Estudios más recientes, indican que el crecimiento de una página promedio es del 151 % (en un lapso de 3 años) [23] (ver Figura 2).

El crecimiento se produce con velocidad [37] y hay otras cuestiones relacionadas al tiempo de carga de una página, no solo el ancho de banda [17], sino también, el RTT[1], como se puede ver en la Figura 1, por ejemplo.

3. SPDY

SPDY[38] es un protocolo de la capa de aplicación [34] que funciona sobre SSL [9], permite la transmisión de Streams[2] sobre una conexión normal de TCP, que es el Protocolo de Control de Transmisión de la Capa de Transporte [34]. A continuación se comentarán las características del Protocolo (extraídas de [38]):

[1] Tiempo que tarda un paquete de datos en ir desde el emisor al receptor y volver al emisor.

[2] Flujo de Datos.

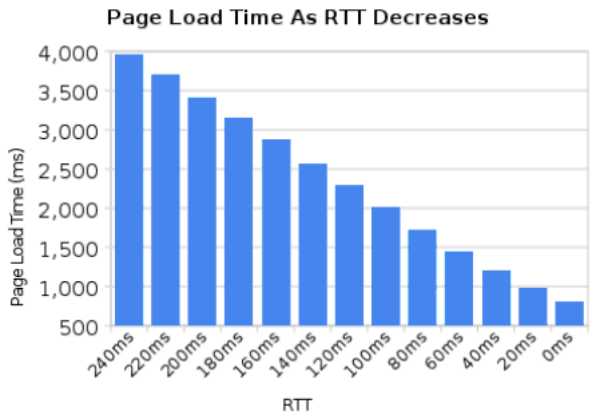

Figura 1: Tiempo de carga de una página mientras se varía el RTT, extraído de [17]

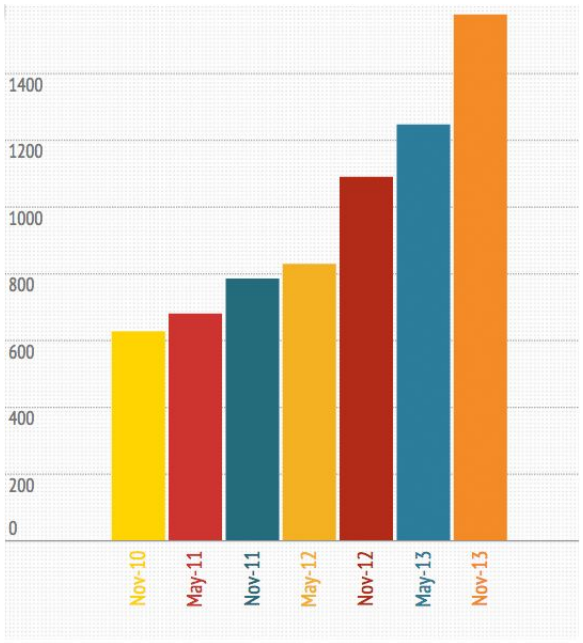

Figura 2: Crecimiento del tamaño de los sitios promedio, extraído de [23]

1. Streams Multiplexados.
2. Priorización de Peticiones.

   El Cliente puede peticionar tantos recursos como necesite del Servidor y asignarle prioridad a cada uno de ellos.
3. Compresión de los Headers HTTP [3].

   Comprime los headers de petición y respuesta HTTP.
4. Push

   Permite al Servidor enviarle recursos al Cliente sin que este se lo pida.
5. Hint

   Permite al Servidor "sugerirle" al Cliente que pida algún recurso específico.

4. Experimento 1

4.1. Preparación

Se virtualizaron 3 máquinas utilizando VirtualBox [13], disponibles para su descarga en [24]. Se diseño la topología de Red como se observa en la Figura 3.

1. SERVIDOR

   Se descargaron los sitios[3] y se configuraron los siguientes hosts virtuales en el Servidor:

   a) www.amazon.com

   b) www.bing.com

   c) login.yahoo.com

   d) www.world-flags.com [14]

   Cada sitio se brindó utilizando el servidor Apache 2.2 [25]. En HTTP plano con su configuración por defecto, en HTTPS utilizando *mod_ssl* [1] con un certificado SSL autofirmado con Ubuntu [31, Sección 4], para ofrecer SPDY, se instaló *mod_spdy* [7]. Con el propósito de poder realizar un análisis de los paquetes que viajan luego de finalizado el experimento, se realizó la siguiente modificación en la configuración del SSL, que es el archivo *"/etc/apache2/mods-enabled/ssl.conf"* [35].

   # SSL Cipher Suite:
   # List the ciphers that the client is permitted to negotiate.
   # See the mod_ssl documentation for a complete list.
   # enable only secure ciphers:
   ~~#SSLCipherSuite HIGH:MEDIUM:!ADH:!MD5~~
   SSLCipherSuite DES-CBC3-SHA

2. PROXY

   Utilizando la herramienta Dummynet [26] que viene instalada en la distribución de FreeBSD, se utilizaron diferentes comandos [27] para filtrar el tráfico en la red[4] y simular

[3] Utilizando la opción "Guardar como..." de Google Chrome, que obtiene todos los recursos externos y los almacena en una carpeta.

[4] Ancho de Banda y Retardo.

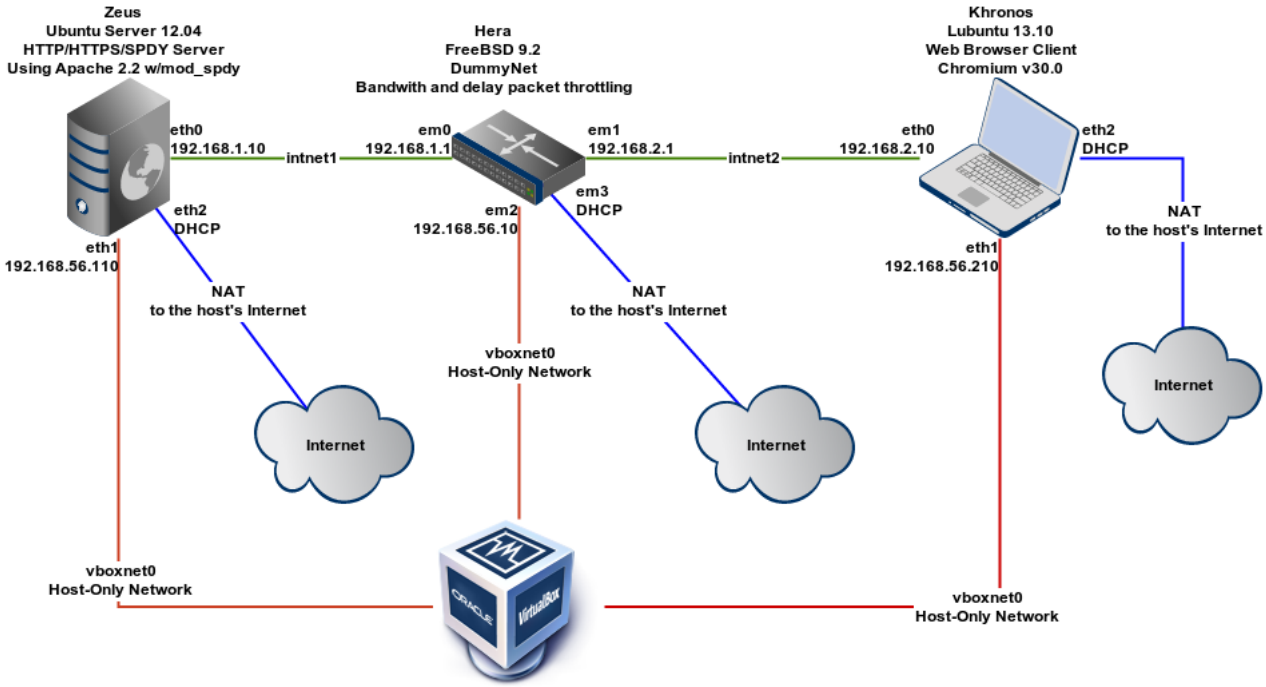

Figura 3: Diagrama de Red del Entorno Virtual del Experimento 1

diferentes entornos. Se habilitó la Dummynet modificando el archivo *"/etc/rc.conf"* con las siguientes líneas

```
firewall_enable="YES"
firewall_type="OPEN"
gateway_enable="YES"
```

Se configuró el siguiente flag:

```
sysctl net.inet.ip.forwarding=1
```

Y por último, para que la Dummynet inicie junto con el Kernel del FreeBSD se agregó la siguiente línea en el archivo *"/bot/loader.conf"*

```
dummynet_load="YES"
```

3. CLIENTE

   Se instaló *NodeJS* [8] y se descargó el software *chrome-har-capturer* [18] de GitHub [28]. Se utilizó el navegador *Chromium* Versión 30 que, a travéz de su API de depuración remota [21], permite al *chrome-har-capturer* interactuar con dicho navegador para poder navegar un sitio en particular y obtener un archivo *.har*[5], que contiene los resultados de la interacción del navegador con el sitio. Por otro lado, fue utilizada la herramienta *TShark*[12] para poder capturar los paquetes que viajan en la red durante el experimento.

4.2. Metodología

Con la idea de comparar los métodos HTTP, HTTPS y SPDY en diferentes ambientes simulados, se definieron los siguientes valores:

Ancho de Banda (BW)	Retraso (RTT)
100 Kbps	10 ms
256 Kbps	50 ms
512 Kbps	100 ms
1024 Kbps	200 ms
2048 Kbps	250 ms
5120 Kbps	500 ms
10240 Kbps	

En el Proxy, se crea una *tubería*[6] de la siguiente manera.

```
ipfw add 1000 pipe 1 ip from any to any;
```

Y los valores se van configurando automáticamente mediante el siguiente comando:

```
ipfw pipe 1 config bw 1000Kbp/s delay
100ms;
```

[5] Archivo con notación JSON [6] que contiene la traza del Navegador Web con el sitio

[6] Objeto intermediario donde se simulan diferentes entornos

Se combinaron todos los Anchos de Banda con todos los Retrasos para cada uno de los métodos por página[7].

En el Servidor se activa *mod_spdy* según es requerido, utilizando:

a2enmod spdy

O desactivar:

a2dismod spdy

Cuando el test no es para el método SPDY, en el cliente se desactiva también la utilización desde el Browser con el flag[8]:

–use-spdy=off

El algoritmo[9] fué el siguiente:

Algorithm 4.1: EXPERIMENTO1()

```
for ancho de banda ∈ anchos de bandas
  do
  { for retardo ∈ retardos
      do
    { configurar valores en el proxy
      for metodo ∈ (http, https, spdy)
        do
      { if metodo == spdy
          then activar spdy en el servidor
          else desactivar spdy en el servidor
        for sitio ∈ sitios
          do
        { iniciar chrome
          iniciar captura con tshark
          ejecutar chrome − har − capturer
          cerrar chrome
          cerrar tshark
```

Se repitió el experimento 7 veces para poder promediar los resultados. La información obtenida (archivos .har y .pcap) se procesó para obtener los tiempos basados en el *.har* [33], OnLoad y OnContentLoad y el basado en la captura de *Tshark*, ToW. Estos resultados se almacenaron en una base de datos *SQLite* [11] para su posterior análisis.

[7]Por ejemplo: 100Kbps de Ancho de Banda con 100ms de Retraso accediendo al host virtual de Amazon por HTTPS (https://www.amazon.com).

[8]http://peter.sh/experiments/chromium-command-line-switches/

[9]Disponible en [30, exp1.sh]

4.3. Resultados

En cuanto a las medidas seleccionadas para el experimento, se observó que, el valor del *onContentLoad* presenta una diferencia notable con respecto a las otras 2 medidas (*onLoad* y *ToW*) cuando el sitio tiene mucho contenido (por ejemplo en los sitios de amazon y worldflags) (Ver Figura 4). En cambio cuando el sitio tiene escaso contenido (el caso de yahoo y bing), las medidas se encuentran más cercanas entre sí (Ver Figura 5).

La medida *onLoad*, es la que se encuentra más cerca del tiempo que percibe el usuario cuando el navegador que está utilizando, termina de renderizar el sitio en cuestión.

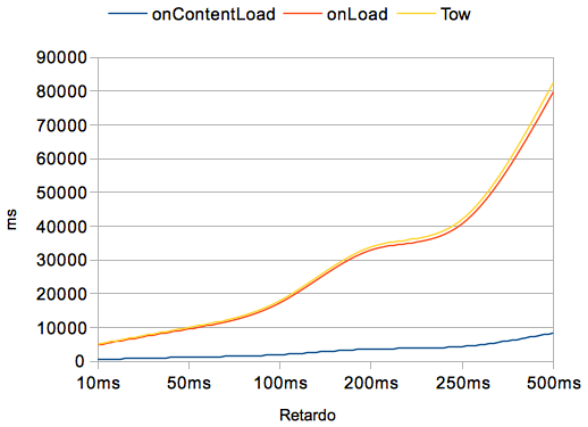

Figura 4: www.worldflags.com - HTTPS - 1024Kbps

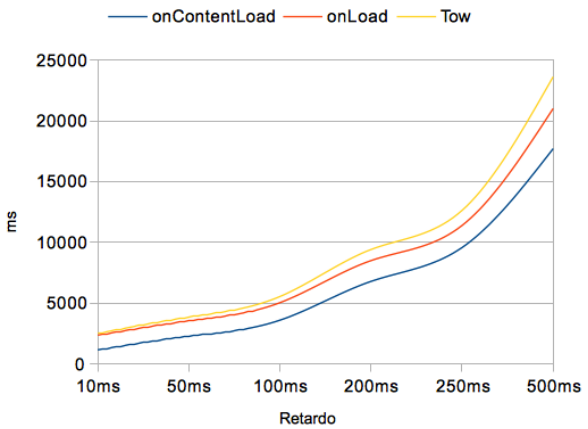

Figura 5: www.bing.com - HTTPS - 1024Kbps

Hay un caso en particular donde, la utilización de SPDY mejora notablemente los tiempos de carga del sitios, lo que se observa con el caso de *www.worldflags.com*, como se puede ver en la

Figura 6 y 7.

Este sitio está tomado de [32] y se puede ver online en [14]. Es el caso emblemático para el protocolo, ya que reune las características para que el funcionamiento de SPDY sea el óptimo. Es un sitio sencillo con 196 recursos (imágenes) todas almacenadas en el mismo lugar, lo que aprovecha al máximo la utilización de los flujos de datos del protocolo; en contraposición con la cantidad de conexiones que se deben realizar (utilizando HTTP o textschttps) con el servidor para traer tantos recursos.

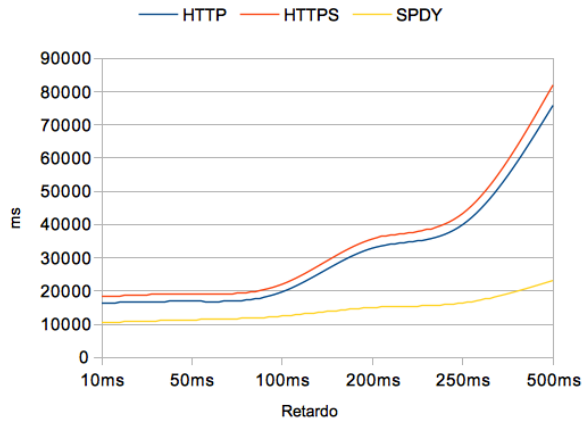

Figura 6: www.worldflags.com - onLoad - 256Kbps

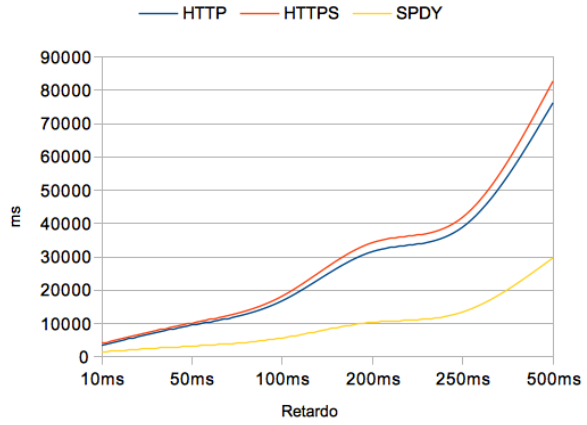

Figura 7: www.worldflags.com - Tow - 10240Kbps

Se observó el mismo comportamiento que plantea *Mike Belshe* en [17]. Esto se puede ver en la Figura 8, en donde al incrementar el Ancho de Banda manteniendo el mismo retardo de 500ms, el tiempo de carga del sitio va formando una constante.

El rendimiento de los protocolos cuando la velocidad es alta es similar, en cambio, cuando son

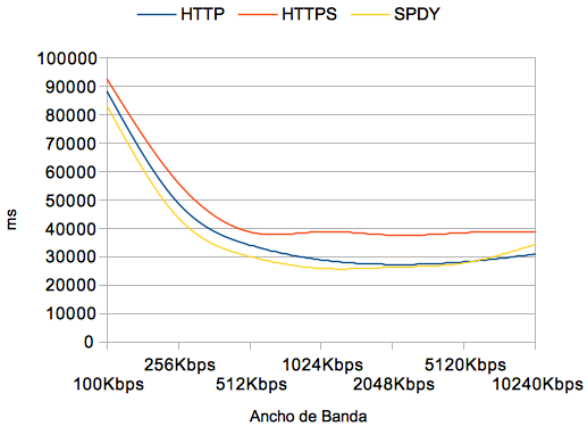

Figura 8: login.yahoo.com - onLoad - 500ms

sometidos a velocidades bajas, SPDY mejora levemente su performance frente a los demás, tal como se ve en la Figura 9.

5. Experimento 2

5.1. Preparación

Se configuró un Cliente con las mismas características que las del Experimento 1 (ver sección 4). Se seleccionaron los siguientes sitios, basados en el artículo [22], además se agregaron 2 sitios con contenido diferente al de la página principal de 2 de los sitios testeados (blogspot-blogger y wordpress):

1. www.facebook.com
2. www.google.com
3. www.youtube.com
4. www.blogger.com
5. www.twitter.com
6. www.wordpress.com
7. www.imgur.com
8. www.youm7.com
9. consigueregalos.blogspot.com
10. oprojetopedal.wordpress.com

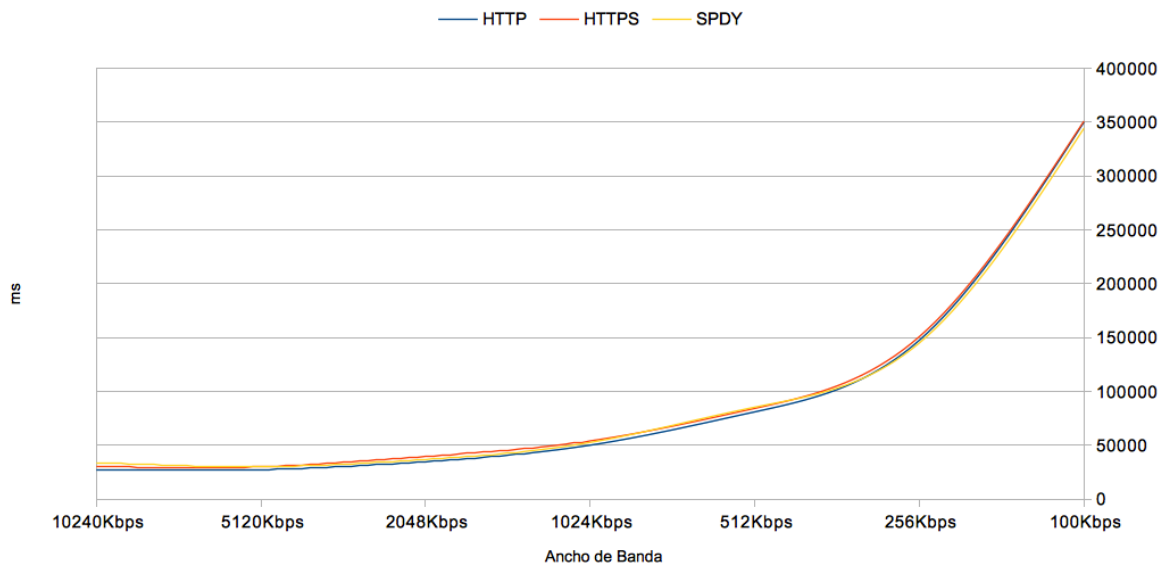

Figura 9: www.amazon - onLoad - 250ms

5.2. Metodología

A continuación se observa el Algoritmo en el cual se ha eliminado todo lo relacionado a configuraciones de Ancho de Banda y Retardo del algoritmo utilizado en el experimento anterior:

Algorithm 5.1: EXPERIMENTO2()

for $metodo \in (http, https, spdy)$
do
  for $sitio \in sitios$
  do
    $iniciar\ chrome$
    $iniciar\ captura\ con\ tshark$
    $ejecutar\ chrome - har - capturer$
    $cerrar\ chrome$
    $cerrar\ tshark$

Luego, se dejó el experimento corriendo durante 7 días, ejecutándose el mismo en diferentes horarios:

1. 00:00
2. 04:00
3. 08:00
4. 12:00
5. 16:00
6. 20:00

Finalizada la semana, se recolectaron los datos de la misma manera que en el Experimento 1 (ver 4.2).

5.3. Resultados

Se recolectaron entre 44 y 47 capturas de cada página por método, se promediaron los resultados, los cuales se observan en los siguientes cuadros.

consigueregalos.blogspot.com

	onContentLoad	onLoad	Tow
HTTP	3585	5909	8419
HTTPS	3533	5383	7831
SPDY	3249	4995	8046

oprojetopedal.wordpress.com

	onContentLoad	onLoad	Tow
HTTP	4175	6293	8332
HTTPS	5234	6668	8596
SPDY	4603	7457	9375

www.blogger.com

	onContentLoad	onLoad	Tow
HTTP	3801	3911	5510
HTTPS	3782	3841	5439
SPDY	3374	3473	4832

www.facebook.com

	onContentLoad	onLoad	Tow
HTTP	4638	6433	8072
HTTPS	3508	4462	6150
SPDY	2880	3915	5827

www.google.com

	onContentLoad	onLoad	Tow
HTTP	1750	2645	4348
HTTPS	1641	2552	3819
SPDY	1525	2270	3713

www.imgur.com

	onContentLoad	onLoad	Tow
HTTP	5299	12122	14569
HTTPS	5174	10305	13115
SPDY	5506	12690	15145

www.twitter.com

	onContentLoad	onLoad	Tow
HTTP	4864	6008	7553
HTTPS	4381	5454	6863
SPDY	3793	4853	6867

www.wordpress.com

	onContentLoad	onLoad	Tow
HTTP	2522	2648	4363
HTTPS	2654	2794	4430
SPDY	2788	2907	4189

www.youm7.com

	onContentLoad	onLoad	Tow
HTTP	5909	37539	40364
HTTPS	4990	14957	17641
SPDY	4534	17363	20064

www.youtube.com

	onContentLoad	onLoad	Tow
HTTP	1455	1577	3200
HTTPS	1657	1712	3440
SPDY	1730	1730	3445

En el 50 % de los casos SPDY se encuentra debajo de los tiempos de respuesta con respecto a HTTPS y a HTTP. Lo cual ha resultado favorable (frente a HTTP), dado que el 93 % de los sitios en Internet usan HTTP como se observa en el gráfico de la Figura 11 [4, run 15/11/2013].

Por otra parte, este porcentaje frente a HTTPS, también es favorable en vistas de un posible HTTP 2.0 [15] que necesite ser encriptado obligatoriamente [16] [19] [20]. Ver Figuras 10, 12 y 13.

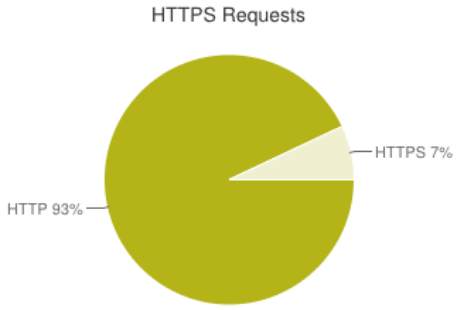

Figura 11: Porcentajes de utilización de los protocolos HTTP y HTTPS.

SPDY presenta resultados desfavorables en sitios con grandes contenidos, aquellos que necesitan de mucho tiempo para completar las peticiones, son proclives a tener pérdidas de paquetes que necesiten de retransmitir el mensaje. Este tiempo que toma retransmitir todo el mensaje completo, hace que disminuya su rendimiento en la situación mencionada anteriormente.

6. Conclusiones

SPDY propone las bases para el futuro HTTP 2.0, ofrece nuevas características que pueden ser aprovechadas y mejora la performance en ciertos casos. El protocolo es fuerte en condiciones en las que se presentan enlaces con Velocidades Bajas y con cierto Retardo, en condiciones favorables, el protocolo puede presentar inconvenientes por cuestiones de pérdida de paquetes y retransmisión. Por otro lado, cabe destacar que, todas las técnicas utilizadas para incrementar la performance[10] no son necesarias para el buen funcionamiento del protocolo. El mejor escenario para SPDY es el sitio con las características de *world-flags*, es decir, con todos o la mayoría de los recursos en el mismo dominio, lo que aprovecha al máximo la multiplexación de streams de datos.

7. Trabajos Futuros

A futuro una de las pruebas a realizar sería, utilizando la suite de prueba del Experimento 1, ejecutar los tests pero con otros servidores que soporten SPDY, tales como *Jetty Web Server*[11], *Python implementation of a SPDY server*[12], *Ruby SPDY*[13] o *node.js SPDY*[14]. Para poder medir la performance de SPDY en diferentes implementaciones del protocolo.

Otra herramienta apropiada para realizar mediciones es Selenium [10], con ella, se pueden automatizar diferentes interacciones con los sitios utilizando un navegador web. Si bien estos tests concluyen que SPDY mejora la performance de la carga de un sitio, las características del protocolo se desempeñan al máximo cuando el usuario se encuentra navegando el sitio. Con esta herramienta se podría medir como se comporta SPDY en una utilización cuasi real.

[10] Css sprites, domain sharding, combinaciones de archivos de javascript, combinación de archivos css, recursos asíncronos, etc.
[11] http://wiki.eclipse.org/Jetty/Feature/SPDY
[12] http://github.com/mnot/nbhttp/tree/spdy
[13] https://github.com/igrigorik/spdy
[14] https://github.com/indutny/node-spdy

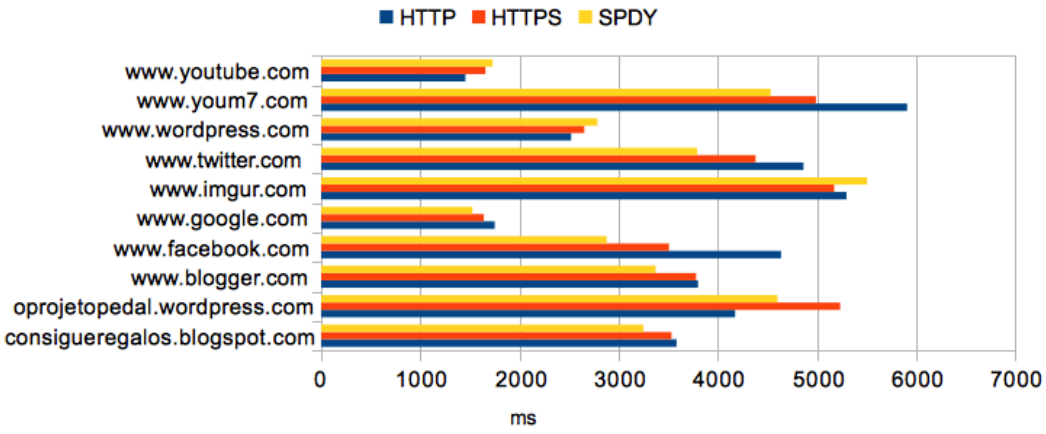

Figura 10: onContentLoad

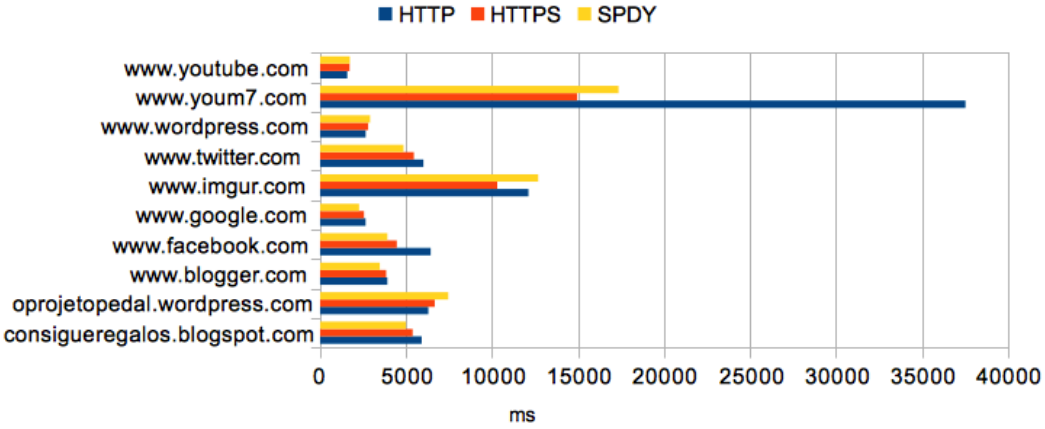

Figura 12: onLoad

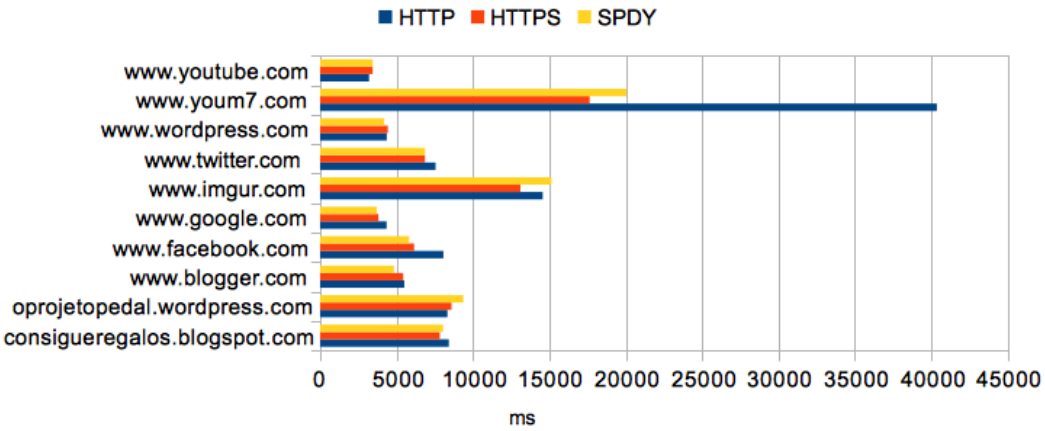

Figura 13: ToW

Referencias

[1] Apache module mod_ssl. http://httpd.apache.org/docs/2.2/mod/mod_ssl.html.

[2] Freebsd. http://www.freebsd.org/es/.

[3] Header field definitions. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html. part of Hypertext Transfer Protocol – HTTP/1.1 - RFC 2616.

[4] Http archive. http://httparchive.org.

[5] Hypertext transfer protocol – http/1.1. http://www.ietf.org/rfc/rfc2616.txt.

[6] Introducing json. http://www.json.org/.

[7] mod-spdy - apache spdy module. http://code.google.com/p/mod-spdy/.

[8] nodejs. http://nodejs.org/.

[9] The secure sockets layer (ssl) protocol version 3.0. http://tools.ietf.org/html/rfc6101.

[10] selenium - browser automation framework. https://code.google.com/p/selenium/.

[11] Sqlite. http://www.sqlite.org/.

[12] Tshark. http://www.wireshark.org/docs/man-pages/tshark.html.

[13] Virtual box. https://www.virtualbox.org/.

[14] World flags mod_spdy demo. https://www.modspdy.com/world-flags/.

[15] Internet architects propose encrypting all the world?s Web traffic. Hypertext transfer protocol version 2.0. http://tools.ietf.org/search/draft-ietf-httpbis-http2-09. 04/12/2013.

[16] M. Belshe, Twist, R. Peon, Inc Google, Ed. M. Thomson, Microsoft, Ed. A. Melnikov, and Isode Ltd. Internet architects propose encrypting all the world's web traffic. http://arstechnica.com/security/2013/11/encrypt-all-the-worlds-web-traffic-internet-architects-propose/. 14/11/2013.

[17] Mike Belshe. More bandwidth doesn't matter (much). 4 de Agosto 2010.

[18] Andrea Cardaci. Capture har files from a remote chrome instance. https://github.com/cyrus-and/chrome-har-capturer.

[19] Brad Chacos. Next-gen http 2.0 protocol will require https encryption (most of the time). http://www.pcworld.com/article/2061189/next-gen-http-2-0-protocol-will-require-https-encryption-most-of-the-time-.html. 13/11/2013.

[20] Richard Chirgwin. Mandatory http 2.0 encryption proposal sparks hot debate. http://www.theregister.co.uk/2013/11/14/http_20_encryption_proposal_sparks_hot_debate/. 14/11/2013.

[21] Google Developers. Remote debugging protocol. https://developers.google.com/chrome-developer-tools/docs/debugger-protocol.

[22] Yehia Elkhatib, Gareth Tyson, and Michael Welzl. The effect of network and infrastructural variables on spdy's performance. 29 de Julio 2013.

[23] Tammy Everts. The average web page has grown 151 % in just three years. http://www.webperformancetoday.com/2013/11/26/web-page-growth-151-percent/. 26 de Noviembre 2013.

[24] Marcelo Fernandez. http://www.marcelofernandez.info/files/.

[25] The Apache Software Foundation. Apache. http://www.apache.org.

[26] FreeBSD. Dummynet. http://www.freebsd.org/cgi/man.cgi?query=dummynet.

[27] FreeBSD. Ipfw. http://www.freebsd.org/cgi/man.cgi?query=ipfw&sektion=8&apropos=0&manpath=FreeBSD+9.2-RELEASE.

[28] GitHub. Github build software better, together. https://github.com/.

[29] Lubuntu. lubuntu, lightweight, fast, easier. http://www.lubuntu.net/.

[30] Pablo Maximiliano Lulic. spdy-tests. https://github.com/maxisoad/spdy-tests.

[31] mdsteele@google.com. Getting started with mod_spdy. https://code.google.com/p/mod-spdy/wiki/GettingStarted.

[32] mod_spdy. mod_spdy - news and updates for the apache spdy module. https://www.modspdy.com/.

[33] Jan Odvarko odvarko@gmail.com. Har 1.2 spec. http://www.softwareishard.com/blog/har-12-spec/.

[34] W. Richard Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley Professional, 1994. Section 1.2 Layering.

[35] Chris Strom. Ssl that can be sniffed by wireshark. http://japhr.blogspot.com.ar/2011/05/ssl-that-can-be-sniffed-by-wireshark.html.

[36] Ubuntu. Ubuntu server - for scale-out computing. http://www.ubuntu.com/server.

[37] website optimization. Average web page size triples since 2008. http://www.websiteoptimization.com/speed/tweak/average-web-page/.

[38] website optimization. Spdy: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.