Websocket:
Comparación de performance e implementación de aplicaciones web

Matias Damian Banchoff Tzancoff
matiasb@cespi.unlp.edu.ar

Director: Francisco Javier Diaz
Asesor Profesional: Andrés Barbieri

6 de diciembre de 2011

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Objetivos y motivaciones para el presente trabajo

1.1. Introducción

HTML5 es la nueva versión del lenguaje HTML. Provee nuevas tecnologías, como geolocalización, bases de datos locales al cliente, web workers, y tags de video y audio, entre otras. El protocolo Websocket comenzó siendo parte del estándar HTML5, pero luego continuó su evolución como un estándar separado.

Todas esas tecnologías en conjunto plantean un nuevo paradigma de diseño y programación de aplicaciones web. Algunos de estos elementos son tan nuevos que aún, al día de la fecha (Septiembre de 2011), no están implementados en los principales productos del mercado.

El protocolo Websocket permite mantener una **conexión full-duplex** y **con estado** entre cliente y servidor web. Aunque pensado para su utilización con Javascript dentro de un navegador web, puede ser extendido para soportar subprotocolos binarios y usarlo desde aplicaciones que se ejecuten fuera de un cliente web.

1.2. Objetivo

El objetivo de este trabajo de grado es realizar una comparación entre los desarrollos web anteriores a Websocket y los cambios introducidos por esta tecnología, las nuevas posibilidades que ofrece y las dificultades que plantea, de modo que se puedan responder preguntas como:

- ¿Cambiará la manera en que los servidores manejan sus recursos?
- ¿Habrá alguna mejora en el desempeño de las aplicaciones que usan Websocket con respecto a las que utilizan Comet?

- ¿Cuál será el costo de migrar a Websocket, tanto del lado del cliente como del servidor?
- ¿Qué repercusiones tendrá el uso de Websocket en la planificación y administración de una red destinada al web hosting?
- ¿Cuáles son las nuevas oportunidades de integración que trae Websocket?

1.3. Motivaciones

El protocolo Websocket será una tecnología central en las futuras aplicaciones web, similar a lo que es Ajax hoy en día; por esta razón es importante investigar sus posibilidades y aplicaciones.

Traerá, sin dudas, un cambio en el paradigma de desarrollo de aplicaciones web, así como también nuevos desafíos para quienes implementan y administran servidores web.

Una de las premisas del trabajo es utilizar Software Libre. Todo lo producido será una forma de contribuir a la comunidad de Software Libre, ya sea código o documentación.

Capítulo 2

Introducción a las tecnologías web

En la última década, las aplicaciones web tuvieron un crecimiento casi exponencial en cuanto a la cantidad de usuarios que atienden y recursos que consumen. Basta con comparar los sitios web de los años noventas con los actuales para darse cuenta que una de las principales diferencias -más allá de lo gráfico-, es el *estado* de las aplicaciones: hoy en día, cualquier sistema web mantiene una sesión de usuario -incluso aquellas que no parece, como los buscadores-, lo cual complica su diseño, desarrollo e implementación.

En este capítulo se verán los conceptos y tecnologías asociados a la web. También se explicará el paradigma de cliente-servidor, la forma en que funciona la web. En lo referente al paradigma, se dejará claro que siempre es el cliente quien inicia la conexión y nunca el servidor. En otras palabras, el servidor nunca puede iniciar una conversación con un cliente, tan sólo atender sus peticiones.

En esta sección se introducen conceptos básicos utilizados en este trabajo, como ser: la nomenclatura, organismos, protocolos, lenguajes, formatos y elementos comunes. Sobre cada cuestión se hará una breve explicación y se mencionarán los detalles más relevantes y destacados para el trabajo.

Quien esté familiarizado con los conceptos aquí expuestos, bien puede saltear esta parte y continuar con las siguientes secciones de este texto.

2.1. Comités y organismos de estandarización

En esta primera sección se introducen los organismos que se encargan de crear y definir los estándares para la web. Comencemos, pues, con una introducción sobre la W3C.

La *W3C* (`http://w3c.org`), *World Wide Web Consortium*, es una organización internacional abocada a la creación y mantenimiento de estándares para la *Web*. Fue fundada por Tim Berners-Lee, el creador del lenguaje HTML cuan-

do trabajaba en el Cern en el año 1990 (`http://www.w3.org/History/1989/proposal.html`).

La W3C propone distintos niveles de conformidad para los productos y no dispone, por el momento, de un programa de certificaciones. Además, las recomendaciones se patentan con licencias libres de regalías, de modo que cualquier individuo, empresa u organismo puede implementarlas.

Adicional a la W3C se encuentra la *Internet Engineering Task Force* (`http://www.ietf.org/`) o *IETF*, que se encarga de desarrollar y promover estándares relativos a Internet. Es un organismo abierto dividido en varios *grupos de trabajo* y *grupos informales de discusión*. La IETF publica memorandos sobre usos, recomendaciones, investigaciones e innovaciones relativas a Internet, llamados RFCs o *Request for Comments* (`http://www.faqs.org/rfcs/`).

Actualmente, la IETF está abocada a varios protocolos conocidos, como IPv6, SIP, RTP y Websocket, entre otros. Sobre este último, se encuentra trabajando en dos *drafts* relacionados con el protocolo de Websocket: *Websocket Requirements and Features*[12] y *The Websocket protocol*[14] bajo el nombre de *Bidirectional or Server-Initiated HTTP (hybi)*.

La *Internet Assigned Numbers Authority* (`http://www.iana.org/`) o *IANA* es la entidad encargada de controlar la asignación de rangos de direcciones IP, asignación de puertos a servicios, reserva de números de sistemas autónomos (AS), administración de los datos en los DNS raíz y tipos MIME, entre otros. Está operada por la Internet Corporation for Assigned Names and Numbers (ICANN). La IANA es la que asignó el número de puerto para el protocolo Websocket.

Otra entidad relacionada con la formulación de estándares relacionados con Internet es *Ecma International* (`http://www.ecma-international.org/`) o *Ecma*, la cual se encarga de estándares relacionados con los sistemas de información y comunicación. EcmaScript (más conocido como lenguaje Javascript), C# y OOXML son lenguajes cuyas especificaciones están dadas por este organismo.

La *Web Hypertext Application Technology Working Group* (`http://www.whatwg.org/`) o *WHATWG* es un grupo de personas interesadas en la evolución de HTML y las tecnologías accesorias. Fue fundada por gente de Apple, Opera y Mozilla. La propuesta de HTML5 de la W3C está basada en la propuesta creada por este grupo de trabajo.

Otros organismos relevantes de la industria son la *Organisation internationale de normalisation* u *OSI* (`http://www.iso.org`), integrada por representantes de diversos países, y el *Institute of Electrical and Electronics Engineers* o *IEEE* (`http://www.ieee.org/index.html`), integrado por profesionales y estudiantes asociados.

El primero es un organismo internacional formado por organizaciones de estándar nacionales. Tiene como objetivo promulgar estándares a nivel internacional, no sólo los relacionados con Internet. Uno de los aportes de este organismo a las Ciencias Informáticas es el Modelo de capas OSI (Modelo de referencia de Interconexión de Sistemas Abiertos). Es un modelo teórico y conceptual para describir las capas de una red y las funciones de cada capa. En lo relativo a esta tesis, Websocket y HTTP son protocolos de la capa de aplicación, y casi siempre

nos vamos a mantener en esta capa.

La segunda es una organización internacional sin fines de lucro cuyo propósito es realizar avances en tecnologías relacionadas con la electricidad. En Informática, la IEEE tiene relevancia en lo referente a hardware y las capas más bajas en los modelos de redes. No se hacen referencias a estándares de esta organización y, como en el caso anterior, incluye sólo por completitud.

Introducidos los principales organismos de la industria relevantes para este trabajo, se pasará a hablar ahora de las tecnologías en sí.

2.2. Estándares web

En la sección anterior se describieron los organismos involucrados en la creación de los protocolos, lenguajes y formatos usados en la web. En esta sección se verán cuáles son estos elementos.

2.2.1. Lenguaje HTML

HTML *Hyper Text Markup Language* es un lenguaje de marcas utilizado para describir texto y especificado por la W3C (`http://www.w3.org/TR/REC-html40/`). Incluye marcas o *tags* para definir imágenes, párrafos, listas y encabezados, entre otros. Además, por medio de HTML se puede incluir cualquier tipo de archivo, como zip, exe, js o png. La mayor innovación del lenguaje fue la marca de *anchor*, que permite enlazar un documento a otro, generando así hipertextos[1]. Este enlace o referencia se hace mediante una URL.

Una URL, *Uniform Resource Locator*, sirve para referenciar unívocamente a un recurso en Internet y tiene la siguiente forma:
protocolo://**usuario**:**password**@**servidor**:**puerto**/**recurso**.
Por ejemplo, *http://www.unlp.edu.ar/index.php*, *mailto:matiasb@cespi.unlp.edu.ar* o *Javascript:unaFuncion()*.

Actualmente, la versión usada de HTML es la 4.01.

2.2.2. DOM

DOM (`http://www.w3.org/DOM/`), Document Object Model, define una manera estándar de acceder y manipular documentos XML y HTML. Presenta a los documentos con una estructura de árbol, donde cada elemento de dichos documentos son nodos del árbol. Actualmente la W3C se encarga del auspicio y DOM se encuentra en su versión 3, liberada en abril de 2004.

[1] En su libro Literary Machines (1992) Ted Nelson, quien acuñó los términos *hipermedia* e *hipertexto* dice: *By now the word "hypertext""has become generally accepted for branching and responding text, but the corresponding word "hypermedia", meaning complexes of branching and responding graphics, movies and sound - as well as text - is much less used. Instead they use the strange term "interactive multimedia": this is four syllables longer, and does not express the idea of extending hypertext.*

Existen *bindings* y librerías para casi todos los lenguajes más populares, como C, Java, PHP, Ruby o Python. Dentro de los clientes web, se puede acceder y manipular los documentos HTML con lenguajes como Javascript, utilizando la API DOM correspondiente.

2.2.3. XML

XML (`http://www.w3.org/XML/`), *eXtensible Markup Language*, es un metalenguaje basado en marcas (*tags*). Permite definir gramáticas para otros lenguajes, siendo los más conocidos: XHTML, MathML, RSS, SVG, Atom, XMMP[2]. Existen numerosas librerías para procesar archivos XML. El desarrollo del lenguaje está a cargo de la W3C.

Hay que notar que el lenguaje HTML no es compatible con XML. Para utilizar un lenguaje compatible con las normas de XML, existe XHTML. Actualmente, la versión de XHTML es la 1.0.

XML es la base de la programación de *web services* (introducidos más adelante en este mismo documento), y sirve como formato multiplataforma para el intercambio de datos entre distintos sistemas. Actualmente, y debido a la complejidad de XML, se está usando JSON para estas tareas.

2.2.4. Javascript

Es un lenguaje de programación escrito para ser embebido en las páginas web. Tiene una sintaxis muy parecida a C o Java. En un principio sólo era utilizado para verificar que los formularios estuvieran bien completados por el usuario; actualmente, y gracias a su flexibilidad, Javascript es la base de la Web 2.0.

Hoy existen un sinfín de aplicaciones, librerías y frameworks basados en esta tecnología, como *Prototype* (`http://www.prototypejs.org/`), *jQuery* (`http://jquery.com/`), *MooTools* (`http://mootools.net/`) y *Dojo Toolkit* (`http://dojotoolkit.org/`), por mencionar sólo algunos, que permiten agregar componentes dinámicas para ser procesadas por el navegador web.

Existen varias implementaciones del lenguaje, entre las que destacan: *SpiderMonkey* (`https://developer.mozilla.org/en/SpiderMonkey`) de Firefox, *Rhino* (`http://www.mozilla.org/rhino/`) de Mozilla para Java, *WebKit* (`http://www.webkit.org/`) de Apple y *V8* (`http://code.google.com/p/v8/`) usado en Google Chrome, entre otros. Actualmente, los navegadores son compatibles con la versión 1.6 del lenguaje Javascript.

2.2.5. CSS - Hojas de estilo

Las hojas de estilo, CSS (`http://www.w3.org/Style/CSS/`) - Cascade Style Sheets-, son una tecnología aparecida luego de HTML y cuyo objetivo es darle formato a las páginas web. Más precisamente, la idea es quitar el formato del

[2] gTalk y Facebook usan este protocolo para sus aplicaciones de chat

código HTML y darlo con CSS, para así lograr mayor reuso e independencia entre contenido y formato. La versión más nueva del lenguaje es CSS3, aunque implementada parcialmente por los navegadores.

CSS incluye no sólo características típicas de formatos de textos, párrafos, etc, sino que también permite manipular el posicionamiento y ubicación de los elementos dentro de las páginas web.

Esto último ha abierto la programación en el lado del cliente a un gran número de aplicaciones. La mayoría de las librerías y frameworks Javascript mencionadas anteriormente se basan en la utilización de estos estándares.

2.3. Protocolos y técnicas asociadas

Hasta aquí la introducción de las tecnologías y estándares relacionados con el lenguaje HTML. Naturalmente, se podría continuar por mucho más tiempo, pero no es necesario para los fines de este trabajo. Se pasará, entonces, al protocolo usado para transportar las páginas web: HTTP.

2.3.1. HTTP

HTTP *Hyper Text Transfer Protocol* (`http://tools.ietf.org/html/rfc2616`) es el protocolo utilizado para la transferencia de archivos HTML y los archivos referenciados desde las páginas web (archivos pdf, js, ps, zip, etc.).

En esta sección se destaca solamente el rol que cumple HTTP en la web. Las diferencias entre las versiones de HTTP, así como las diferencias entre HTTP y Websocket -que serán una componente importante de esta tesina- se verán más adelante.

Entonces, por lo pronto, alcanza con saber que HTTP es el protocolo de la *capa de aplicación* usado en la web, que existen varias versiones del protocolo -Siendo HTTP1.1 la más reciente-, y que HTTP es un protocolo *sin estado* (En el Capítulo 3, Sección 3.1.2 se verán las implicaciones de esta característica).

2.3.2. HTTPS

Se denomina HTTPS (`http://www.ietf.org/rfc/rfc2818.txt`), por *HTTP Over TLS*, al protocolo HTTP cuando usa como transporte SSL/TLS en vez de TCP. La IANA asigna el puerto 80 al tráfico HTTP y el 443 al tráfico de HTTPS. La característica más sobresaliente de HTTPS es que el contenido viaja cifrado a través de Internet.

Debe tenerse en cuenta que el protocolo en sí no cambia: continúa utilizándose HTTP como protocolo de aplicación; lo que varía es el protocolo de transporte.

2.3.3. SOAP

SOAP (`http://www.w3.org/TR/soap/`), *Simple Object Access Protocol*, es un protocolo que indica cómo dos objetos que viven en diferentes procesos se pueden comunicar entre sí por medio de mensajes XML, usando RPC, HTTP o algún otro protocolo como medio de transmisión. En otras palabras, su función es la de comunicar aplicaciones escritas en diferentes lenguajes y corriendo sobre diferentes arquitecturas. Actualmente está auspiciado por la W3C.

2.3.4. Web services

Los web services son APIs accedidas vía HTTP y ejecutadas en el sistema remoto. Utilizan XML y como medio de transporte, HTTP o HTTPS.

La W3C define el término *web service* como: *un programa diseñado para soportar una interacción máquina a máquina sobre una red. Tiene una interfaz descripta en un formato entendible por una máquina (en particular, WSDL). Los sistemas interactúan con el web service en la forma impuesta en su descripción usando mensajes SOAP, serialización XML sobre HTTP* (`http://www.w3.org/TR/ws-gloss/`).

2.3.5. Java Applets

Los Java applets son programas escritos en Java (`http://www.java.com`) y que pueden embeberse en páginas HTML utilizando los tags **APPLET** u **OBJECT**. Se ejecutan dentro de lo que la documentación llama *sandbox*, que es un entorno seguro en el navegador web, en el sentido de que un applet no puede abrir archivos, sockets, ni escribir a disco.

El tag **OBJECT** define un objeto incluido en el documento HTML. Se pueden incluir archivos de audio, video, documentos pdf, applets, películas flash y hasta otras páginas web.

Los applets fueron muy usados para efectos visuales y funcionalidad que hoy en día se logra con tecnologías como Javascript, Ajax o Flash. A partir de HTML 4.1 el tag **APPLET** está *deprecated*, y se recomienda usar el tag **OBJECT**. En HTML 5 el tag **APPLET** no está soportado.

Por estar escritos en Java, es necesario instalar el entorno de ejecución de Java (JRE, Java Runtime Environment) en el cliente para que el navegador pueda ejecutar los applets.

2.3.6. Adobe Flash

Adobe Flash (`http://www.adobe.com/devnet/flash.html`) es una tecnología utilizada para llevar audio, video e interactividad a las páginas web. Se integra con la interfaz del usuario, pudiendo utilizar tanto el mouse y teclado, como el micrófono y la webcam, pero no se integra bien con los navegadores y tiene problemas con los buscadores web.

Adobe Flash posee un lenguaje de programación orientado a objetos, ActionScript (`http://www.adobe.com/devnet/actionscript.html`). Además, por no ser una tecnología nativa de la web, es necesario que los clientes posean un *reproductor* de Flash, los cuales existen para varias plataformas, como Windows, Linux, Mac y celulares. Quizás el caso más notable sea Apple iPhone/iPod/iPad (`http://www.apple.com/hotnews/thoughts-on-flash/`), que no cuentan con soporte para flash en sus versiones de Safari.

Si bien se trata de una tecnología muy apreciada por diseñadores y presente en muchas páginas web, Flash tiene varias características negativas que de a poco le van quitando mercado en la web: no se acopla al navegador, por lo que los botones de "adelante", "atrás", "detener" y los "Favoritos" no funcionan con la película flash. Además, no es una tecnología accesible (`http://www.w3.org/standards/webdesign/accessibility`); por último, los buscadores no indizan bien las películas flash.

2.3.7. Adobe Flex

Adobe Flex (`http://www.adobe.com/products/flex/`) es un *framework* open source que tiene como fin la creación de RIAs[3] con una interfaz consistente a través de todos los browsers y sistemas operativos. Nuevamente, es necesario instalar plugins para que las aplicaciones hechas con Flex puedan funcionar en el navegador web. El *runtime* de Adobe recibe el nombre de Air (`http://www.adobe.com/devnet/air.html`), Adobe Integrated Runtime.

La mayor diferencia entre Flex y Flash es que el primero está orientado a desarrolladores, proveyendo un conjunto amplio de *widgets* y sin necesidad de un navegador web para su ejecución (Es decir que una aplicación Flex puede actuar como una aplicación de escritorio); mientras que el segundo está orientado a los diseñadores y utiliza la metáfora de la película.

2.3.8. JSON y BSON

JSON (`http://www.json.org/`), *Javascript Object Notation*, es un formato liviano para el intercambio de datos. Es fácil de leer, tanto para humanos como para computadoras. Está basado en un subconjunto de Javascript. Como ya se mencionó, JSON se posicionó como reemplazo de XML para el intercambio de datos en la web por su simplicidad y facilidad de lectura.

A continuación, un ejemplo del uso de JSON tomado de `http://www.json.org/example.html`:

```
1 {
2     "nombre": "Matias",
3     "apellido": "Banchoff",
4     "edad": 25,
5     "direccion":
6     {
7         "calle": "8",
8         "ciudad": "La Plata",
```

[3] RIA: Rich Internet Application

```
9           "provincia": "Buenos Aires",
10          "codigopostal": "1900"
11      },
12      "telefono":
13      [
14          {
15            "tipo": "hogar",
16            "numero": "123-4567"
17          },
18          {
19            "tipo": "fax",
20            "numero": "123-8910"
21          }
22      ]
23  }
```

Listing 2.1: Ejemplo de JSON - Archivo Javascript

BSON es un formato binario para el intercambio de documentos similar a JSON. De hecho, BSON significa Binary JSON.

Las características más destacadas de BSON son: Formato liviano, tratando de agregar el menor *overhead* al documento; navegabilidad rápida de los datos y rapidez en la codificación y decodificación. Más información se puede obtener en el sitio de BSON, `http://bsonspec.org`.

2.4. Conceptos relacionados

2.4.1. Web 2.0

El término *Web 2.0* hace referencia a todas aquellas aplicaciones web que poseen ciertas características, como son:

Interoperabilidad
: La habilidad que tienen las aplicaciones web para compartir información entre sí. Por ejemplo, la forma de publicar enlaces en Facebook.

Diseño centrado en el usuario
: Es un proceso de diseño que pone especial atención en las necesidades, deseos y limitaciones de los usuarios. Conceptos como **accesibilidad**, **visibilidad** y **usabilidad** son elementos propios de esta metodología de diseño.

Colaboración
: La posibilidad de trabajar en forma colaborativa sobre la web. Por ejemplo, Google Docs, donde varios usuarios pueden escribir un documento al mismo tiempo y todos obtienen el *feedback* necesario para que aprecien el trabajo colaborativo.

Comunicación de la información
: Protocolos destinados a la difusión de la información bajo demanda, como RSS y Atom. Usados, por ejemplo, en blogs y diarios on line.

La Web como plataforma
: Las aplicaciones y los datos están en la web, y no en la computadora del usuario. Es la transición de un modelo de Desktop a Webtop. Esto se ve claramente en los sistemas operativos basados en la web, como EasyPeasy (`http://www.geteasypeasy.com/`), Android (`http://source.android.com/`) o MeeGo (`https://meego.com/`).

Inteligencia colectiva
: Marca la posibilidad de crear conocimiento en forma colectiva. Quizás el ejemplo más conocido y exitoso de esto sea Wikipedia.

La web 2.0 no trata de un avance tecnológico -ni siquiera de un cambio en la tecnología-, sino de un cambio en la forma en que se utilizan las tecnologías actuales para desarrollar aplicaciones web. Es un cambio en la forma en que desarrolladores y usuarios utilizan la web.

2.4.2. Web 3: La web semántica

La Web Semántica estructurará el contenido de la web, creando un entorno donde los agentes[4] puedan ejecutar tareas sofisticadas en nombre del usuario. Para que esto ocurra, la web debe tener una estructura tal que permita al software interpretar el contenido que encuentra[1]. El problema es que los tags de HTML no son lo suficientemente descriptivos como para señalar la relación semántica que hay entre varios elementos. Por ejemplo, los tags **<SPAN>** y **</SPAN>**, y **<DIV>** y **</DIV>** no tienen ningún significado semántico, y se los utilizan como contenedores genéricos dentro del documento.

Actualmente, la forma de implementar una web semántica, representando la relación que hay entre los diferentes contenidos, es a través de *meta-datos* (similar a la lógica de predicados). Sin embargo, La Web Semántica no es el tema de este trabajo, por lo que no se profundizará más sobre este tema. Para más información sobre este tópico se recomienda The Semantic Web[1].

2.4.3. RIAs (Rich Internet Applications)

Son aplicaciones web que tienen muchas de las características de las aplicaciones de escritorio. Usadas, por ejemplo, en juegos on-line. Flash y Java applets son tecnologías comúnmente asociadas a estas aplicaciones. El problema de estas tecnologías es que necesitan un ambiente de ejecución propio, que viene en la forma de un plugin para el navegador.

En los últimos años, Javascript se posicionó como una tecnología competitiva en el campo de las RIAs, y gracias a Ajax y librerías como jQuery o Dojo, es posible hacer RIAs que antes se podían hacer solamente usando alguna tecnología no nativa, como Flash.

[4] Un *agente* es un programa que actúa *en nombre de* un usuario (persona o programa). La aparición de los agentes dio lugar al surgimiento de un nuevo paradigma de programación basado en agentes, muy relacionado con la disciplina de Inteligencia Artificial.

El mayor problema con que se encuentran las RIAs -Sin importar la tecnología con que están hechas-, es el trabajo off-line. En este aspecto, el estándar de HTML5 especifica una nueva estructura de datos del lado del cliente: las bases de datos (Web SQL Database, `http://dev.w3.org/html5/webdatabase/`), que permitirán avanzar en esta linea.

2.5. Paradigma de clientes y servidores

En la sección introductoria a HTTP se dijo que HTTP es un protocolo con una arquitectura de cliente-servidor. Un servidor es un sistema que brinda recursos, mientras que los clientes son aquellos sistemas que solicitan los recursos.

A modo de ejemplo, se tiene la Figura-2.2, donde el cliente web realiza una petición y el servidor web retorna el recurso solicitado. Notar que el servidor web, a su vez, funciona como cliente del servidor de bases de datos.

2.5.1. Cliente web

Los clientes web más comunes (en sus últimas versiones a julio de 2011) son Microsoft Internet Explorer 9, Mozilla Firefox 5.0.1, Opera 11.50, Safari 5.1, Google Chrome 12.0.742. A julio de 2011, la mitad del mercado lo posee Microsoft con sus distintas versiones de IE, seguido por Firefox y Chrome.

En el Figura-2.1 se puede apreciar el porcentaje de mercado para cada navegador. Estas cifras pueden ser imprecisas, dado que varían según la fuente. Por ejemplo, según las estadísticas de W3Schools (`http://www.w3schools.com/browsers/browsers_stats.asp`), el navegador más usado es Firefox, seguido por Internet Explorer.

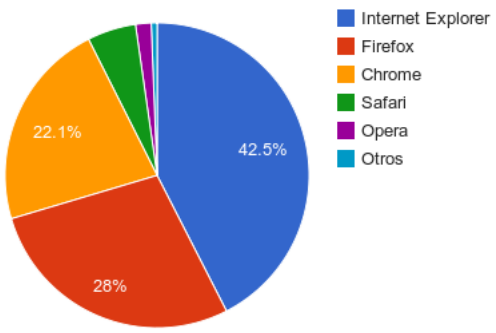

Figura 2.1: Cuota de mercado para los navegadores web más importantes (Fuente: StatCounter).

Ningún navegador implementa completamente los estándares de la W3C, aunque algunos están más cerca que otros. Esto es relevante a la hora de programar scripts y de utilizar hojas de estilo. Las librerías de Javascript, en su mayoría, funcionan con la mayoría de los navegadores y con todas sus versiones.

Otro parámetro de diferenciación es la *performance* del motor de Javascript que muestra cada navegador, lo cual pone de manifiesto la importancia que tomó el lenguaje Javascript en los últimos años.

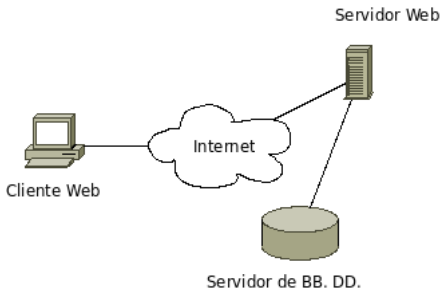

Figura 2.2: Ejemplo de un esquema de cliente-servidor.

El problema con las tecnologías no nativas de la web, como son Java o Flash, es que necesitan programas accesorios (plugins) en la computadora cliente para la ejecución de los applets o las películas flash, respectivamente.

Esto significa una complejidad más para el usuario, que debe instalar software adicional. Más aún, significa que una empresa dependerá del plugin de terceros para que su software funcione correctamente.

La mayoría de las innovaciones realizadas en la web hasta la fecha ocurren del lado del cliente, siendo Ajax la más notoria de ellas. Los servidores web generalmente no se ven afectados por estas innovaciones.

2.5.2. Servidor web

Los servidores web, por otra parte, son los que se encargan de encontrar el recurso solicitado por el cliente y retornarlo. Los servidores más usados comúnmente son Apache Httpd, Microsoft Internet Information Server (IIS), Lighttpd, NginX, Cherokee, Apache Tomcat, entre otros. A agosto de 2011, Apache posee más del cincuenta por ciento del mercado. La Figura-2.3 muestra las estadísticas de Netcraft (`http://news.netcraft.com/`) para los principales servidores web desde agosto de 1995 hasta agosto del 2011.

Cuando nació la web, el esquema de interacción era sencillo: el usuario pedía un recurso (generalmente un documento de texto) al servidor, quien respondía y terminaba la sesión. Con los años, la web evolucionó de una red para compartir documentos a una plataforma para desarrollar aplicaciones, y la interacción pasó a ser aquella típica de cualquier aplicación de escritorio.

En los últimos años esa tendencia se profundizó aún más, de modo que las aplicaciones web ahora deben reaccionar a los eventos generados por otros usuarios. Sin embargo, el protocolo subyacente, HTTP, no soporta este tipo de interacción.

HTTP es un protocolo sin estado, donde los datos de las sesiones de usuarios, por ejemplo, deben manejarse desde las distintas APIs. Surgieron, entonces,

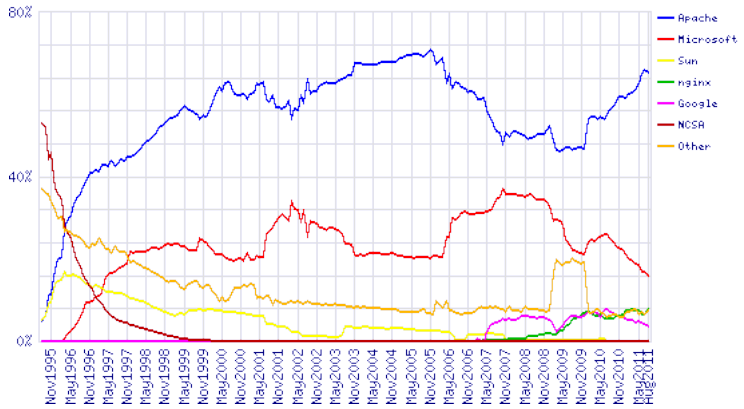

Figura 2.3: Cuota de mercado para los principales servidores web (Fuente: Netcraft)

diversas técnicas para lograr este nuevo patrón de interacción, agrupadas todas bajo el nombre de Comet. En la Sección 3.2 del Capítulo 3 de este informe se profundizará más sobre este tema.

El problema subyacente es que un servidor web no tiene forma de contactar a un cliente en particular, sino que debe esperar el requerimiento del cliente para poder responderle. Ese es el mayor impedimento con el que se encuentran actualmente los desarrolladores web, y es precisamente el problema que resuelve el protocolo Websocket, tecnología utilizada en este trabajo.

Capítulo 3

El estado de arte de las tecnologías web

En el capítulo anterior se mencionaron los aspectos básicos relacionados con las tecnologías web. El objetivo de este capítulo es comentar el estado de arte en el diseño y desarrollo web.

En el inicio de este capítulo se verán las características más importantes del protocolo HTTP, incluyendo el concepto particular que tiene de conexión. Se verá que el protocolo brinda una idea de conexión pero no de sesión, y que esto lo debe implementar el programador.

También se mostrará que el servidor no tiene forma de iniciar el envío de datos a un cliente en particular a menos que éste los solicite. Sobre este tema se analizarán las técnicas de Comet y la forma en que se complementan usando Ajax.

Por último se dará una introducción muy breve a las nuevas características de HTML 5.

3.1. Protocolo HTTP 1.1

HTTP versión 1.1 es el estándar más reciente de HTTP definido en la RFC 2616[10]. Es un protocolo con una arquitectura de cliente-servidor, donde el cliente realiza peticiones (*requests*) y el servidor responde (*responses*).

Las peticiones del cliente se hacen utilizando uno de ocho posibles métodos. Los métodos indican la acción a realizar sobre el recurso alojado en el servidor. Un listado completo de cada método, y su correspondiente explicación, se encuentra en `http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`.

A modo de ejemplo, se muestra una petición de tipo GET a `www.google.com.ar` por la página principal:

```
$ telnet www.google.com.ar 80
Trying 74.125.229.112...
```

```
Connected to www.l.google.com.
Escape character is '^]'.
GET / HTTP/1.1
```

A lo cual Google responde con el código 302 para indicar que la página se ha movido a otro servidor. Si se estuviera accediendo con un navegador web (y no por telnet), automáticamente redireccionaría la página indicada en Location.

```
HTTP/1.1 302 Found
Location: http://www.google.com.ar/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Set-Cookie: PREF=ID=632c192b9ae51318:FF=0:TM=1316116039:LM=1316116039:S=R5iDZVeCBc9t4Juk; \\
expires=Sat, 14-Sep-2013 19:47:19 GMT; path=/; domain=.google.com
Date: Thu, 15 Sep 2011 19:47:19 GMT
Server: gws
Content-Length: 222
X-XSS-Protection: 1; mode=block

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

HTTP No fue diseñado para el intercambio bidireccional de mensajes. Es decir, el servidor no tiene forma de enviar datos al cliente sin que este último los solicite.

En la siguiente captura se muestra el intercambio de mensajes del protocolo para una petición al servidor web local:

1. Se utiliza *telnet* para establecer la conexión (**telnet localhost 80**).
2. Se utiliza el método GET para solicitar al servidor un recurso. En este caso, se solicita la página sobre HTTP al servidor web local.
3. El servidor responde con el encabezado HTTP y el recurso solicitado.
4. Por último, el recurso solicitado, que en este caso es una página web escrita en HTML.

```
[1]telnet localhost 80
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[2]GET / HTTP/1.1
Host: localhost
Connection: Close

[3]HTTP/1.1 200 OK
Date: Sat, 30 Jul 2011 13:53:51 GMT
Server: Apache/2.2.14 (Ubuntu)
Last-Modified: Thu, 04 Feb 2010 22:00:48 GMT
ETag: "2e3fb-b1-47ecd77b1f788"
Accept-Ranges: bytes
Content-Length: 177
Vary: Accept-Encoding
Connection: close
Content-Type: text/html

[4]<html><body><h1>It works!</h1>
<p>This is the default web page for this server.</p>
```

```
<p>The web server software is running but no content has been added, yet.</p>
</body></html>
Connection closed by foreign host.
```

En la Figura-3.1 se muestra gráficamente el flujo de mensajes entre el cliente y servidor web para el ejemplo aquí visto. En primera instancia se establece la conexión entre el cliente y el servidor (1); luego, el cliente envía la petición utilizando el método GET (2); por último, el servidor web responde con un código (200, es decir, no hubo problemas para procesar el requerimiento) y el recurso solicitado (3). Notar que (3) y (4) son parte de la misma respuesta que envía el servidor.

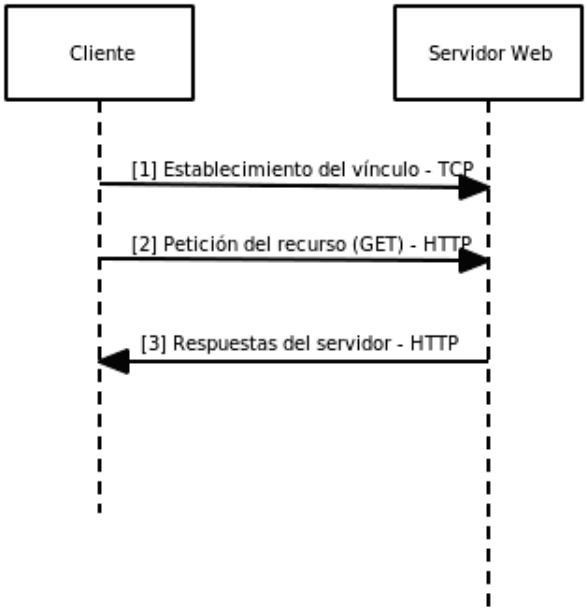

Figura 3.1: Gráfico que muestra una petición HTTP sencilla.

Las respuestas están acompañadas de un código numérico, que indica si la consulta fue exitosa, si el recurso solicitado no existía, entre otros. Una lista completa de estos códigos puede encontrarse en: `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.

3.1.1. Conexiones persistentes

El diálogo entre cliente y servidor tiene la forma "(petición, respuesta)". En las versiones previas del protocolo, la conexión se cierra cuando el servidor envía la respuesta al cliente. Es decir, la conexión se inicia, se envía la petición, se recibe la respuesta y se cierra la conexión[1].

[1] A decir verdad, este comportamiento se puede negociar en HTTP 1.0, pero no es la opción negociada por defecto y, además, algunos proxies no implementan esta funcionalidad

Ahora bien, un sitio web no está compuesto por un único recurso, sino que, además de la página web solicitada, ésta referencia otros archivos, como imágenes, hojas de estilo y archivos de Javascript. Luego, para cada recurso que conforma la página web, es necesario iniciar una conexión.

La Figura-3.2 muestra un ejemplo de cómo sería el flujo de tráfico para un ejemplo consistente de una página *index.html* que hace referencia a un archivo *unaImagen.png*. Nótese que se realizan dos peticiones diferentes, una para el archivo HTML y otra para la imagen.

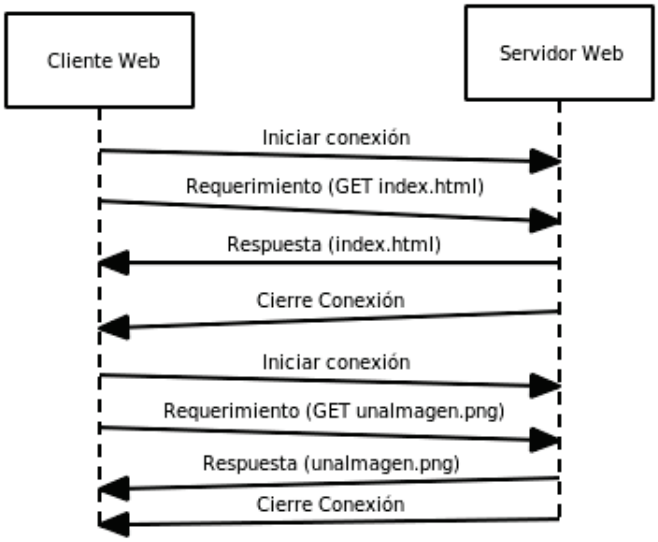

Figura 3.2: Conexión HTTP sin usar conexiones persistentes.

Uno de los cambios introducidos en HTTP 1.1 es la idea de mantener la conexión abierta por más tiempo (*keep-alive*), de forma que se pueda usar la misma conexión para atender varias peticiones. Esto ahorra recursos (tiempo, memoria RAM, CPU), pues no es necesario abrir una nueva conexión para descargar cada componente de una página web.

La Figura-3.3 muestra, para el ejemplo anterior, cómo sería el flujo de tráfico usando conexiones persistentes. Se ve claramente la reducción en la cantidad de conexiones a establecer, más aún sabiendo que una página web normalmente referencia decenas de archivos (Javascript, CSS, imágenes, películas flash, entre otros).

3.1.2. Sesiones

Otra cuestión importante es que HTTP es un protocolo *sin sesión* o *stateless*. Esto significa que el protocolo no tiene forma de asociar sesiones a un cliente. En otras palabras, para las aplicaciones web, dos peticiones de un mismo cliente aparecen como dos peticiones sin relación alguna. La ventaja de los protocolos sin estado es el menor consumo de recursos, principalmente del lado del servidor. Y esta característica hace que este protocolo sea tan sencillo de implementar.

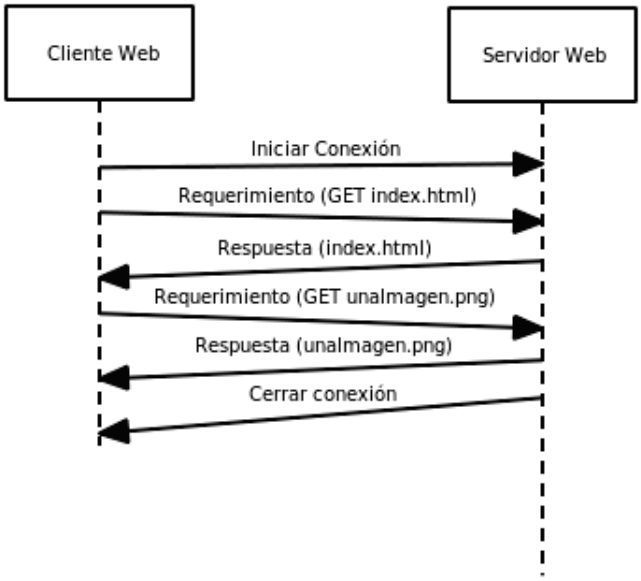

Figura 3.3: Conexión HTTP usando conexiones persistentes.

Actualmente el mantenimiento de la sesión del usuario se hace a través de *cookies*[23], que son datos intercambiados entre el cliente y el servidor. Por lo general, los lenguajes usados para programar aplicaciones web tiene la funcionalidad necesaria para abstraer al programador, de forma que no deba preocuparse por todo el trabajo que hay detrás a la hora de mantener la sesión del usuario.

Por ejemplo, en PHP se utiliza la función **session_start** para que el usuario inicie sesión en el servidor. Esta función se encarga de crear las cookies necesarias o usar un identificador de sesión en la URL en caso de que el navegador tenga deshabilitadas las cookies; además, mediante otras funciones, PHP puede destruir la sesión del usuario cuando éste lo solicite o cuando haya transcurrido un tiempo prudencial.

Hoy en día el uso de sesiones está ampliamente difundido y es una práctica común. Dado que el protocolo no soporta sesiones, es necesario implementarlas por encima, en la aplicación. Es así que cada lenguaje de programación web implementa su propia versión de mantenimiento de sesiones.

3.1.3. Método de transferencia *chunk*

La nueva versión de HTTP 1.1 introduce un mecanismo nuevo de transferencia que permite al servidor enviar datos cuyo tamaño no se conoce de antemano. A este método se lo llama *chunk*, del inglés *pedazo* o *trozo*.

Utilizando esta forma de transmisión, el servidor puede enviar contenido generado dinámicamente de a pedazos, dado que, obviamente, su tamaño se desconoce. Para lograrlo, primero responde avisando que la respuesta irá de a trozos (*chunks*), y luego devuelve los *chunks* a medida que se van generando.

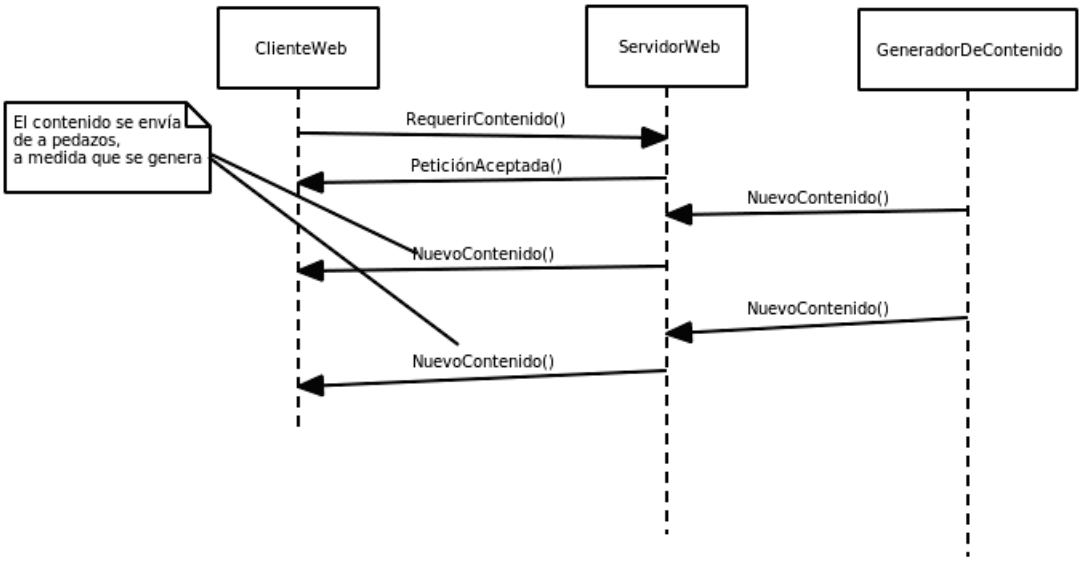

Figura 3.4: Esquema del funcionamiento del método chunk.

La Figura-3.4 esquematiza este tipo de tráfico.

3.2. Ajax y Comet

En esta sección se introducen los conceptos de AJAX y Comet. Ambas técnicas son importantes por dos motivos:

1. Son técnicas ampliamente utilizadas en la actualidad y gracias a ellas surgió y se mantiene la Web 2.0;
2. El protocolo Websocket reemplazará todas las formas de Comet, dejando a esta técnica obsoleta.

3.2.1. AJAX

AJAX (*Aynchronous Javascript and XML*), es un conjunto de técnicas usadas desde el lado del cliente para crear aplicaciones web más interactivas.

Ajax brinda una forma de tomar datos del servidor asincrónicamente, es decir, en *background* y sin interferir con el comportamiento de la página ya cargada y, por lo tanto, sin necesidad de recargar la página actual.

En otras palabras, gracias a AJAX los programas basados en la web pasan de un modelo de página a uno verdaderamente de aplicación (Basado en eventos y acciones de los usuarios)[2].

El término *Ajax* fue acuñado por Jesse James Garrett en su artículo "Ajax: A New Approach to Web Applications" (`http://www.adaptivepath.com/ideas/`

`essays/archives/000385.php`). Según dicho artículo, las aplicaciones con Ajax dependen de las siguientes tecnologías:

- HTML (o XHTML) con Hojas de estilo para la presentación.
- DOM, para la interacción dinámica con los elementos del documento.
- XML para el intercambio de datos[2], aunque en los últimos años comenzó a utilizarse JSON para este fin.
- El objeto XMLHttpRequest[8] para establecer la comunicación asincrónica entre cliente y servidor.
- Javascript, para combinar todas estas tecnologías.

Estas son tecnologías exclusivas del cliente web. Es decir, no se requiere de ningún cambio en el protocolo HTTP ni en los servidores web para escribir sitios web usando Ajax. Hay que notar, sin embargo, que estas no son las únicas herramientas disponibles. Por ejemplo, como reemplazo de Javascript se puede usar ActionScript para el caso de las películas Flash.

Sin embargo, Ajax introduce algunos retos a la hora de la entrega de datos, no manejados por la tecnología en sí.

El uso típico de Ajax implica:

1. Instanciar el objeto XMLHttpRequest.
2. Utilizar el objeto instanciado para realizar una petición al servidor, asignando funciones para los *callbacks* ofrecidos por el objeto.
3. Evaluar la respuesta del servidor en los callbacks.
4. Procesar los datos.
5. Volver al segundo punto.

Y mientras las aplicaciones Ajax utilizan HTTP como transporte, la forma en que utilizan el protocolo es muy diferente de la que se encuentra en el modelo web tradicional de páginas[2].

Seguridad en Ajax

Por cuestiones de seguridad una petición de Ajax no puede hacerse a un dominio distinto. Por ejemplo, un script cargado de **www.yahoo.com** no puede realizar peticiones a **www.google.com**.

Sin embargo, la mayoría de las aplicaciones web actuales toman información de diferentes dominios. Un ejemplo sencillo de esto son los *feed readers*, que pueden traer noticias en RSS de otros dominios. Existen varias técnicas para lograrlo, siendo los proxies la más usada. No es Ajax el tema central de este

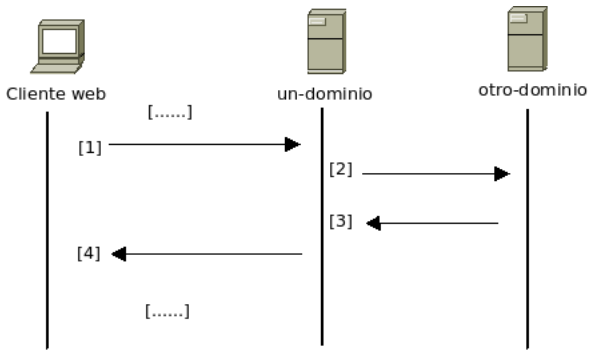

Figura 3.5: Funcionamiento de un proxy para Ajax.

trabajo, por lo que tan sólo se diagramará el funcionamiento de un proxy para Ajax.

En la Figura-3.5 se muestra la interacción entre un cliente web que cargó un script Ajax del dominio *un-dominio* y un servidor en el dominio *otro-dominio*:

1. El cliente envía una petición usando el objeto XMLHttpRequest al servidor en *un-dominio*.
2. El servidor actúa de *proxy*, reenviando la petición al servidor en *otro-dominio*.
3. El servidor en *otro-dominio* responde con los datos solicitados.
4. El cliente recibe los datos a través del proxy.

Performance con Ajax

Se mencionarán ahora las cuestiones asociadas al desempeño de las aplicaciones Ajax, tanto del lado del cliente, como del servidor.

Lo primero que hay que tener en cuenta es que ahora el cliente deja de ser un navegador web que analiza documentos y se convierte en una verdadera plataforma de ejecución de aplicaciones. Evidentemente, los clientes deberán tener más poder de cómputo y tendrán más carga de procesamiento.

El *path*, o camino, que recorren los datos entre el cliente y el servidor también se ve afectado. Actualmente la forma de intercambiar datos es a través de XML, JSON o texto plano[3]. XML en particular introduce un *overhead* importante en el tamaño de los datos, por la propia naturaleza del formato.

Del lado del servidor, los problemas son otros. Puntualmente, una acción cualquiera del usuario puede significar una petición al servidor. Un ejemplo sencillo es *Google suggests* (`http://www.google.com/support/websearch/bin/answer.py?hl=en\&answer=106230`), donde el cliente envía una petición a los

[2]También puede usarse texto plano o HTML
[3]Existen otros formatos, pero estos son los más utilizados

servidores de Google cada vez que el usuario presiona una tecla. Naturalmente, el servidor está obligado a procesar estos requerimientos.

Más adelante se examinarán las técnicas agrupadas en torno al concepto de Comet. Se verá que una de las técnicas implica mantener abierta la conexión TCP por un tiempo considerable. Esto deriva en que el servidor web podrá manejar menos consultas simultáneas, dando paso a una potencial situación de denegación de servicio (Intencional o no).

Beneficios de Ajax

El principal beneficio de Ajax es la carga asincrónica de datos sin necesidad de que el cliente web recargue la página completa. Esto implica un mayor dinamismo para la interfaz del usuario y, al mismo tiempo, le da al sitio web una impresión de aplicación de escritorio. Por lo tanto, Ajax trae aparejado un incremento en la usabilidad del sitio web.

Como se dijo, Ajax tiene un mecanismo de seguridad por el cual no es posible que un cliente web realice una petición HTTP a un dominio diferente de aquel de donde se bajó el script. Por lo tanto, para que un script en un dominio pueda cargar información de otro sitio, es necesario un proxy Ajax del lado del servidor. Teniendo esto instalado, es posible que una página web cargue datos que provienen de un dominio totalmente distinto.

La madurez de DOM y JSON hacen posible el intercambio y manipulación sencillos de los datos. Además, existen numerosas librerías para la manipulación de datos con Ajax, como son las ya mencionadas Prototype, jQuery y Dojo.

Por último, Ajax es nativo de la web, en el sentido de que utiliza tecnologías propias de la web, presentes en todos los navegadores y estandarizadas por la W3C.

Desventajas de Ajax

Sin embargo, Ajax no es la panacea. Hay que tener presentes algunas cosas a la hora de crear sitios web utilizando Ajax.

Por un lado, los motores de búsqueda no ejecutan código Javascript, por lo que cualquier texto que requiera de Ajax para ser leído no será indizado por los buscadores.

Además, Ajax, al igual que Flash, no se acopla bien al cliente web. Los botones de "adelante" y "atrás" generalmente no significan nada para Ajax (salvo que el desarrollador lo tenga en consideración). Los *bookmarks* generalmente no funcionan con Ajax, pues la URL no cambia cuando se utiliza Ajax. Además, si el cliente tiene deshabilitado Javascript, Ajax no funciona.

También se presentan problemas inherentes a la tecnología. Por un lado, las interfaces web que usan Ajax son, necesariamente, más complejas de programar, debido al dinamismo que se busca obtener al usar Ajax.

Como se observó, un script en Ajax no puede cargar datos de otro dominio, por lo que obliga a disponer de un proxy del lado del servidor para este fin. Sin

Método o propiedad	Descripción
abort()	Detiene la petición actual.
getAllResponseHeaders()	Retorna los encabezados de la respuesta como un string.
getResponseHeader("headerID")	Retorna un único encabeza de la respuesta como un string.
open("method", "URL"[,asyncFlag[, "username" [, "password"]]])	Inicializa todos los parámetros de la petición.
send(contenido)	Realiza la petición HTTP.
setRequestHeader("label", "value")	Configura en el encabezado de la petición un parámetro "label" con el valor "value".
onreadystatechange	Especifica la función que maneja los cambios de estado en la petición.
readyState	Retorna el estado de la petición: 0 = no inicializado 0 = No inicializado 1 = Loading 2 = Loaded 3 = Interactive 4 Complete
responseText	Retorna la respuesta del servidor como un string.
responseXML	Retorna la respuesta del servidor como un documento XML (Se puede usar DOM para procesarlo).
status	Retorna el código de estado de la petición.
statusText	Retorna el mensaje de estado de la petición.

Cuadro 3.1: Métodos y propiedades de XMLHttpRequest.

embargo, esto agrega una capa más de seguridad al cliente.

Por último, el patrón con que los sitios que utilizan Ajax reciben peticiones HTTP cambia. Con Ajax, los servidores reciben, típicamente, muchas más peticiones pequeñas. Por lo que el patrón de uso de la red y de los servidores cambia. En teoría (Aunque no siempre sea así), ya no son necesarios *timeouts* largos y sí es necesario que los servidores puedan manejar más peticiones en paralelo.

Ejemplo de uso

Para terminar con esta descripción de Ajax, un pequeño ejemplo[5] de su funcionamiento:

```
1 var xmlHttp = createXmlHttpRequestObject();
2
```

```
3  function createXmlHttpRequestObject()
4  {
5    var xmlHttp;
6    try
7    {
8      // Asume IE7 o mas nuevo u otro navegador moderno
9      xmlHttp = new XMLHttpRequest();
10   }
11   catch(e)
12   {
13     // Asume IE6 o anterior
14     try
15     {
16        xmlHttp = new ActiveXObject("Microsoft.XMLHttp");
17     }
18     catch(e) { }
19   }
20   if (!xmlHttp)
21     alert("Error creando el objeto XMLHttpRequest.");
22   else
23     return xmlHttp;
24 }
25
26 function process()
27 {
28   if (xmlHttp)
29   {
30     try
31     {
32       xmlHttp.open("GET", "personas.txt", true);
33       xmlHttp.onreadystatechange = handleRequestStateChange;
34       xmlHttp.send(null);
35     }
36     catch (e)
37     {
38       alert("Error al conectarse con el servidor:\n" + e.toString()
             );
39     }
40   }
41 }
42
43 function handleRequestStateChange()
44 {
45   myDiv = document.getElementById("miListado");
46   if (xmlHttp.readyState == 1)
47   {
48     myDiv.innerHTML += "Estado del requerimiento: 1 (Cargando) <br
         />";
49   }
50   else if (xmlHttp.readyState == 2)
51   {
52     myDiv.innerHTML += "Estado del requerimiento: 2 (Cargado) <br/>
         ";
53   }
54   else if (xmlHttp.readyState == 3)
55   {
56     myDiv.innerHTML += "Estado del requerimiento: 3 (Interactivo) <
         br/>";
57   }
58   // Cuando readyState es 4, leemos la respuesta del servidor
59   else if (xmlHttp.readyState == 4)
60   {
```

```
61      // Leer la respuesta solo si el estado de HTTP es "OK"
62      if (xmlHttp.status == 200)
63      {
64        try
65        {
66           // Leer el mensaje desde el servidor
67           response = xmlHttp.responseText;
68           // Mostrar el mensaje
69           myDiv.innerHTML += "Estado del requerimiento: 4 (completo)
                 . El servidor dijo: <br/>";
70           myDiv.innerHTML += response;
71        }
72        catch(e)
73        {
74           // Mostrar el mensaje de error
75           alert("Error al leer la respuesta: " + e.toString());
76        }
77      }
78      else
79      {
80        // Mostrar el mensaje de estado
81        alert("Hubo un problema al obtener los datos:\n" + xmlHttp.
              statusText);
82      }
83    }
84  }
```

Listing 3.1: Ejemplo del uso de Ajax - Archivo Javascript

El código anterior crea un objeto XMLHttpRequest. La función createXmlHttpRequestObject es un *wrapper* para contemplar todos los casos posibles; esto es necesario dado que los navegadores web no inicializan este objeto de la misma forma.

La función **handleRequestStateChange** manejará los cambios de estado del objeto y que, cuando finalice la petición, mostrará el listado de personas dentro de un DIV. Por último, la función *process* es la que realiza la petición usando el objeto XMLHttpRequest y le asocia como manejador la función **handleRequestStateChange**.

El código anterior se debe incluir en un archivo HTML, como se muestra a continuación:

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.
2  w3.org/TR/xhtml11/DTD/xhtml11.dtd">
3  <html>
4    <head>
5      <title>AJAX Foundations: Using XMLHttpRequest</title>
6      <script type="text/javascript" src="ejemplo-ajax.js"></script>
7    </head>
8    <body>
9      <p>Ejemplo tomado del libro <span style="font-weight: bolder;">
           AJAX and PHP - Building Modern Web Applications</span> - <
           em>Second Edition</em> - de Bogdan Brinzarea-Iamandi,
           Cristian Darie y Audra Hendrix</p>
10     <p><button onclick="process()">Probar</button></p>
11     <div id="miListado" />
12   </body>
13 </html>
```

Listing 3.2: Ejemplo del uso de Ajax - Archivo HTML

Por último, en el servidor debe existir el archivo *personas.txt* que se pedirá usando Ajax. En este ejemplo se utiliza un archivo de texto plano, pero podría ser cualquier tipo de recurso, como un archivo PHP o Java que consulte la información en una base de datos.

```
1 <p>Miles</p>
2 <p>Louis</p>
3 <p>Thelonious</p>
```

Listing 3.3: Ejemplo del uso de Ajax - Archivo TXT

El resultado es el que se muestra en la Figura-3.6 (Luego de hacer click en el botón):

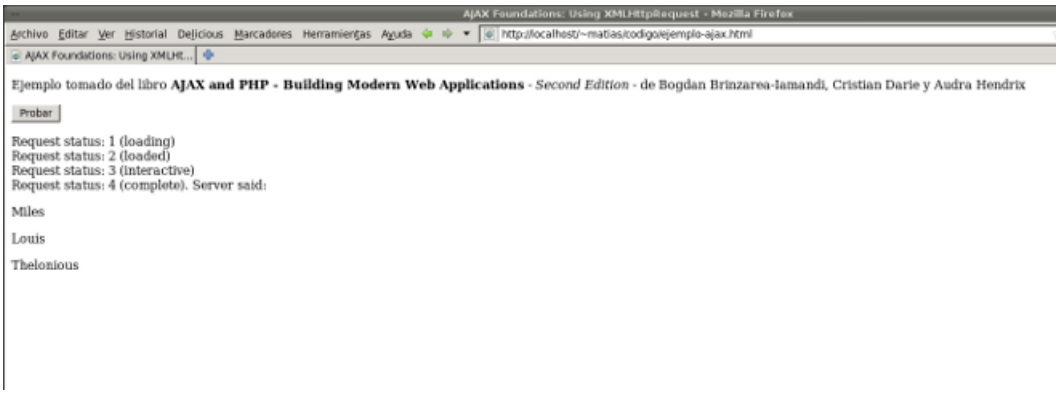

Figura 3.6: Resultado del ejemplo de Ajax.

3.2.2. Comet

Comet describe un modelo de aplicación en el cual una petición HTTP mantenida por mucho tiempo permite al servidor web enviar datos al cliente, sin que éste haya hecho la petición explícitamente.

Ejemplos comunes de esto son los chats basados en web (como los de Facebook y gMail), sistemas que muestran los valores de divisas y acciones y plataformas de trabajo colaborativo.

Hay que señalar que *comet* es una técnica, y no una tecnología específica. Más aún, utiliza herramientas nativas de la web, por lo que no es necesario instalar plugins en el cliente.

El modelo clásico de interacción web requiere que toda comunicación la inicie el cliente. Este modelo recibe el nombre de REST *(Representational State Transfer)*. No existe ninguna información de estado de la conexión en el servidor, y ninguna conexión es persistente[4].

Si bien este esquema favorece la escalabilidad del protocolo, no permite a los servidores enviar información asincrónicamente a los clientes. Sin embargo, y como veremos luego, existen varios escenarios donde esto es útil.

Hoy en día, aquellas aplicaciones que requieran *notificaciones en tiempo real* y *entrega de datos*[4] utilizan una técnica de *polling*, donde la componente cliente requiere activamente los cambios de estado usando *time-outs* del lado del cliente. Una alternativa es la técnica de *pushing*, donde el cliente se suscribe con el servidor, y el servidor publica los cambios asincrónicamente, cada vez que el contenido cambia[3].

A continuación se muestras las dos técnicas para implementar Comet.

HTTP Polling

Implica un chequeo a intervalos regulares para ver si cambió algo en el servidor. Este chequeo se hace a ciegas, haya o no ocurrido un cambio. Una alternativa es realizar el polling y dejar la conexión abierta hasta que ocurra un evento y el servidor envíe la respuesta, cerrando la conexión.

Esta técnica introduce mayor tráfico en la red y posiblemente los mensajes enviados sean innecesarios. Además, HTTP Polling tiene una repercusión en la performance de la aplicación, que debe procesar tanto la consulta como la respuesta del servidor, impactando en los tiempos de respuesta al usuario.

Si la frecuencia con que se chequea el servidor es muy baja, el cliente puede perder algunas actualizaciones[3]. Y si es muy alta, se genera tráfico innecesario y el cliente deberá manejar respuestas que no cambian su estado.

HTTP Streaming/Pushing

Las técnicas de *streaming* consisten en realizar peticiones HTTP *long-lived*, es decir, peticiones para las cuales el servidor retrasa el tiempo de cierre de la conexión. De esta forma, el cliente recibe la respuesta del servidor pero la conexión no se cierra, sino que queda abierta para que el servidor pueda transmitir más datos a medida que éstos llegan[26].

Luego del requerimiento inicial, el servidor no cierra la conexión ni da una respuesta completa. A medida que aparecen nuevos datos, el servidor los envía al cliente en modo *chunked*[10] (`http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.6.1x.`) visto en la descripción del protocolo HTTP, utilizando la misma conexión[3].

3.2.3. Implementaciones de Comet

Generalmente las implementaciones de Comet utilizan Ajax (El objeto XMLHttpRequest visto anteriormente). De ahí la importancia de Ajax a la hora de hablar de Comet.

Es interesante el artículo *Comet: Low Latency Data for the Browser*[7] que es donde se acuña el término *Comet* y se describen el surgimiento y motivación de esta técnica.

[4] data delivery

Por último, y a modo de nota, tres implementaciones Open Source de Comet son Ape, CometD y DWR. Son tres herramientas que se utilizan para dar soporte de Comet a las aplicaciones Ajax. En particular, CometD utiliza un protocolo propio llamado Bayeux. Este protocolo utiliza JSON como formato para sus mensajes.

3.3. HTML 5

HTML5 es la nueva versión del lenguaje HTML. Aún en etapa de definición, provee nuevas tecnologías, como geolocalización, bases de datos locales al cliente, web workers, Websocket y tags de video y audio, entre otras novedades más.

Cuando nos referimos a HTML5 estamos hablando del reemplazo de HTML 4.01, XHTML 1.1 y DOM2. Su principal propósito es brindar RIAs (Rich Internet Applications) sin la necesidad de instalar plugins propietarios. Todas esas tecnologías en conjunto plantean un nuevo paradigma de diseño y programación de aplicaciones web, así como proponen nuevos desafíos para los administradores de redes y servidores.

Algunos de estos elementos son tan nuevos que, a julio de 2011, no están implementados en los principales productos del mercado. Se cree que la mayoría de las características propuestas en HTML 5 estarán maduras en los clientes web dentro de varios años, aunque muchas de ellas ya están incluidas en las nuevas versiones de los navegadores.

3.3.1. Características de HTML 5

Veamos las características más sobresalientes de HTML 5.

Nuevos tags

HTML5 introduce varios tags nuevos, siendo *audio* y *video* los más interesantes. Al mismo tiempo, se deshace de tags considerados obsoletos, como *applet* o *frame*. Una lista completa se encuentra en [9].

Base de datos web

Es una base de datos en el navegador, permitiendo a los dominios web almacenar datos del lado del cliente. La base de datos se diferencia de las *cookies* en el tiempo de vida, visibilidad de la información y la forma de ser consultadas.

Se tienen dos tipos de base de datos: locales y de sesión. Los datos almacenados en la primera pueden accederse desde cualquier script del dominio que guardó la información; mientras que para la segunda, los datos se almacenan por página y por ventana, de modo que se pueden tener varias ventanas abiertas en un mismo dominio, pero sin compartir la información.

Actualmente se trata de una base de datos asociativa (Como una hash), pero la W3C está considerando agregar soporte para realizar un acceso estructurado, usando un lenguaje similar a SQL.

Audio y video

Una de las características más difundidas de HTML5 son sus tags para audio y video. Esto permite suponer un futuro en donde no sea necesario descargar plugins y codecs para visualizar videos y música.

HTML4 no provee ninguna opción para visualizar video, y los fabricantes de contenido (Microsoft, Apple, Youtube, entre otros) han utilizado sus propios formatos y codecs, que deben descargarse y agregarse a los navegadores.

En un principio la especificación de HTML5 establecía que para la reproducción de audio y video se utilizara el codec OGG[5], que es libre y gratuito. Sin embargo, en el año 2010 Apple se opuso a ello. Como resultado final, al día de hoy (Julio de 2011), la especificación de HTML5 introduce el tag video, pero no indica ningún codec a usar.

Youtube (`http://www.youtube.com/html5`) dispone de un servicio beta para reproducir video usando el tag video de HTML5 en vez de usar flash, y utiliza el codec h.264.

Por su parte, Mozilla ya incluye el codec Ogg en Firefox 3.5, así como Dailymotion[6] está brindando contenido en Ogg.

Web Workers

La idea es llevar los conceptos de *threading* a la web, dotando a Javascript de procesamiento concurrente. Ajax se utiliza para cargar y procesar contenido en background. WebWorkers sirve para ejecutar varios scripts en paralelo (no necesariamente para traer datos del servidor).

Si estos web threads se ejecutan en su propio thread nativo en el cliente, las aplicaciones web podrían aprovechar las mejoras que traen los procesadores con respecto al procesamiento multithreaded.

A modo de comentario, la mayoría de los clientes web actuales utiliza un solo motor de Javascript para todos los tabs (o pestañas) abiertos en el navegador, de modo que lo que ocurre en una pestaña afecta a las otras (En términos de performance).

Geolocalización

Es la capacidad que tendrá el navegador web de indicar la posición actual del usuario (latitud, altura, velocidad, etc.). Está pensado principalmente para los

[5] `http://tools.ietf.org/html/rfc3534`
[6] `http://www.dailymotion.com/`

dispositivos móviles y la discusión de esto se lleva en el W3CGeo, el *Geolocation Working Group de la W3C*[7].

[7] `http://www.w3.org/2008/geolocation/`

Capítulo 4

Websocket

El protocolo Websocket permite mantener una **conexión full-duplex** y **con estado** entre cliente y servidor web.

Aunque pensada para protocolos textuales, esta tecnología puede ser extendida para soportar protocolos binarios (Actualmente Javascript no soporta la transferencia de datos en modo binario). Actualmente, el W3C está definiendo la API de Javascript, la IETF está especificando el protocolo Websocket y las empresas lo están empezando a incluir en sus navegadores y servidores.

4.1. Introducción

El protocolo de Websocket comenzó siendo parte de la especificación de HTML 5, pero ahora se mantiene como un estándar separado a cargo de la IETF y de la W3C.

Websocket surge como un estándar para el reemplazo de las técnicas de Comet, especialmente para aquellas aplicaciones web que generan contenido en tiempo real, como pueden ser chats, bolsas de comercio, herramientas colaborativas o aplicaciones de subastas. Es la manera de brindar un marco estandarizado para estas aplicaciones, en vez de desarrollarlas sobre técnicas que son verdaderos parches para el protocolo.

Las metas de Websocket son[12]:

- Permitir a cada lado, cliente o servidor, transmitir información en cualquier momento.
- Utilizar una única conexión TCP para las dos direcciones.
- Reducir el *overhead* producido por los encabezados HTTP.

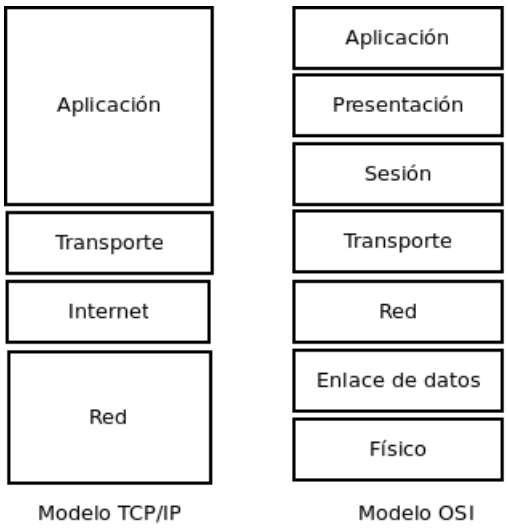

Figura 4.1: Gráfico comparativo entre los modelos OSI y TCP/IP.

4.2. Funcionamiento del protocolo

En esta sección se detallará el protocolo Websocket conforme aparece definido en la RFC The Websocket protocol[14].

4.2.1. Terminología

En esta sección se presenta la terminología que ya forma parte de la jerga de Websocket. Si bien algunos nombres se han traducido para una mejor lectura, a lo largo del trabajo se utilizarán los mismos en inglés (como habitualmente se los reconoce).

Frame (Trama)
Es la unidad básica de intercambio de información para el protocolo.

Message (Mensaje)
Es un bloque de datos interrelacionados con límites bien específicos. Un mensaje puede abarcar varios *frames* y es la unidad de datos a nivel de aplicación.

Websocket Handshake (Negociación)
Es el proceso por el cual cliente y servidor negocian la conexión a establecer. En esta etapa también se descubren las *capabilities*[1] de cada parte.

Websocket Communication Channel (Canal de comunicación)
Es el canal de comunicación bidireccional entre cliente y servidor. Se lo considera directamente encima de la capa de transporte (TCP o SSL sobre TCP).

[1] Se refiere a las capacidades de cada uno: subprotocolos soportados, restricciones de puertos, etc.

Websocket Sub-Protocol (Subprotocolo)
Es el subprotocolo negociado para utilizar en el canal de comunicación. Este protocolo especifica el *framing* a utilizar, la codificación, el timing, etc.

4.2.2. Requerimientos de Websocket

En [12] se especifican todos los requerimientos y características que deberá cumplir el protocolo Websocket.

Se verán a continuación los requerimientos más importantes para cualquier implementación del protocolo, tanto del lado del cliente como del servidor y de elementos intermedios (Como Proxies).

Requerimientos del protocolo

1. El protocolo Websocket debe correr encima de la capa de transporte (sobre la que se hizo el *handshake*). En la Figura-4.1 se muestra una comparación entre las capas de los dos modelos más conocidos: el modelo OSI (el más usado conceptualmente) y el modelo TCP/IP, que es el más implementado y usado en la actualidad.

2. El protocolo debe ser capaz de fragmentar los mensajes en *frames* de un tamaño determinado.

3. Se debe poder enviar un mensaje aún si su tamaño se desconoce o sobrepasa un tamaño preestablecido. Esto es necesario para enviar mensajes desconociendo su tamaño total (Por ejemplo, para hacer *streaming*).

4. Los protocolos en texto plano deben utilizar la codificación UTF-8.

5. El protocolo debe poder diferenciar entre protocolos binarios y basados en texto plano.

6. El protocolo debe permitir responder a peticiones tanto de Websocket como de HTTP en el mismo puerto. Esto se debe a que generalmente los puertos usados por HTTP (80 y 443) se ven favorecidos por los firewalls y proxies, en el sentido de que no se los filtran y hasta reciben una mayor prioridad a la hora del procesamiento de paquetes.

Requerimientos de los clientes

1. El cliente debe ser capaz de establecer una conexión Websocket mediante una negociación (*handshake*) bien definida.

2. El protocolo debe proveer un método para cerrar una conexión cuando el cliente lo solicite.

3. Al mismo tiempo, el protocolo debe soportar pérdidas de conexión y cierres abruptos de una conexión por parte del usuario.

4. El cliente debe poder especificar al servidor un subprotocolo específico durante el handshake.

5. Debe poder enviar y recibir tanto datos binarios como en texto plano[2].

Requerimientos para los servidores

1. El servidor que acepta peticiones de Websocket por parte de un cliente debe utilizar un *handshake* bien definido.

2. Debe poder enviar y recibir tanto datos binarios como en texto plano.

Requerimientos para los proxies

1. El protocolo debe poder operar con los proxies con la misma facilidad que lo hacen HTTP y HTTPS.

Requerimientos de seguridad

Estos requerimientos aún no han sido incluidos en el draft, pero son cuestiones que se tendrán en cuenta a futuro. En [24] se describe una vulnerabilidad en el *handshake* del protocolo y la propuesta que hicieron a los diseñadores de Websocket para solucionarla.

1. El protocolo debe utilizar el modelo de seguridad *basado en origen* usada por los navegadores web para restringir las páginas que pueden contactar al servidor de Websocket cuando se utiliza el protocolo desde una página web.

2. Cuando se lo utiliza directamente (no desde una página web), el protocolo debe utilizar un modelo de seguridad equivalente al utilizado por HTTP o HTTPS cuando se los usan directamente.

3. El protocolo debe ser robusto frente a ataques de *cross-protocol*[27] y *cross-site*[28].

4.2.3. Descripción del protocolo

Ya vimos que la unidad para el envío de datos es el *mensaje*. Sin embargo, el envío y recepción de mensajes se produce al nivel de la aplicación.

También mencionamos que en capas más bajas el PDU[3] se llama *frame*. Los frames tienen un *tipo* asociado y el protocolo define estos tipos de frames:

[2] Actualmente Javascript no tiene un tipo de dato binario, por lo que no es posible manipular datos binarios en Javascript y, por ende, la API de Javascript para Websocket no soporta subprotocolos binarios. Sin embargo, si se utiliza Websocket por fuera del navegador web, es posible la utilización de un protocolo binario

[3] Protocol Data Unit, la unidad de transmisión de datos para el protocolo

- Datos en texto plano con formato UTF-8. En este caso, la interpretación de los datos se delega a la aplicación.
- Frames de Control, usados para la señalización del protocolo. Por ejemplo, para mantener la conexión o para cerrar la sesión.

A pesar de los distintos tipos de *frames*, el protocolo de Websocket está diseñado para que haya la menor cantidad de datos de framing y así ahorrar espacio.

Conceptualmente, Websocket es una capa justo por encima de TCP que agrega:

- Un modelo de seguridad basado en origen para los navegadores web.
- Un mecanismo para soportar varios servicios en un mismo puerto y varios nombres de host en una misma dirección IP.
- Una capa encima de TCP para trabajar con paquetes -a la manera de IP-, pero sin un límite en el tamaño.

La idea es darle al desarrollador algo lo más parecido a un socket[4] TCP. La única relación entre Websocket y HTTP es que los servidores web interpretan el *handshake* y hacen el *upgrade* al protocolo Websocket.

Además, por recomendación de la IANA, Websocket usa por *default* los puertos TCP/80 y TCP/443 para las conexiones sin cifrar y cifradas, respectivamente.

Establecimiento de la conexión

La forma más sencilla de establecer la conexión es a través del puerto 80, pero se corre el riesgo de que un proxy intermedio intercepte los mensajes y los descarte. La forma más segura es establecer una conexión cifrada con SSL/TLS al puerto 443.

Cuando un cliente solicita una conexión, el servidor recibe una petición GET con la oferta de Upgrade al protocolo Websocket. Para servicios con demasiada carga se pueden balancear varios servidores de Websocket.

La URL para el protocolo es *WS://Servidor/Recurso*. Para las conexiones cifradas, el protocolo usados es WSS.

Subprotocolos para Websocket

Un cliente puede especificar un subprotocolo en particular cuando establece la conexión con el servidor. El campo Sec-Websocket-Protocol se utiliza para especificar el subprotocolo.

Los nombres de los subprotocolos no necesitan estar registrados en ningún organismo, aunque se recomienda utilizar el nombre de dominio de quien crea el

[4] Un socket es un extremo de una comunicación bidireccional entre procesos que tiene lugar sobre un protocolo de red, como IP

subprotocolo. Por ejemplo: **unprotocolo.yahoo.com** y **unprotocolo.google.com** serían dos protocolos diferentes que, de no llevar en su nombre el dominio de quien lo creó, podrían solaparse en algunas instalaciones.

Además, el *draft* de la especificación sugiere versionar los subprotocolos, de modo que **v1.unprotocolo.google.com** sería distinto que **v2.unprotocolo.google.com**.

Framing

En esta sección se presenta la forma que tiene un *frame* de Websocket. Sin entrar en demasiados detalles sobre cada campo que conforma el frame, pues no es la finalidad de este trabajo analizar con detenimiento el protocolo, se explicarán aquellas partes importantes para este documento.

La Figura-4.2 detalla un frame de Websocket típico. Hay que aclarar que el programador trabaja tan sólo a partir de los datos de la aplicación. Es decir, no debe considerar cuestiones tales como la longitud del frame o el tipo de frame, pues esto lo resuelve la API intermedia. Por último, se debe tener en cuenta que la unidad de trabajo es el *mensaje*, y que varios *frames* componen dicho mensaje.

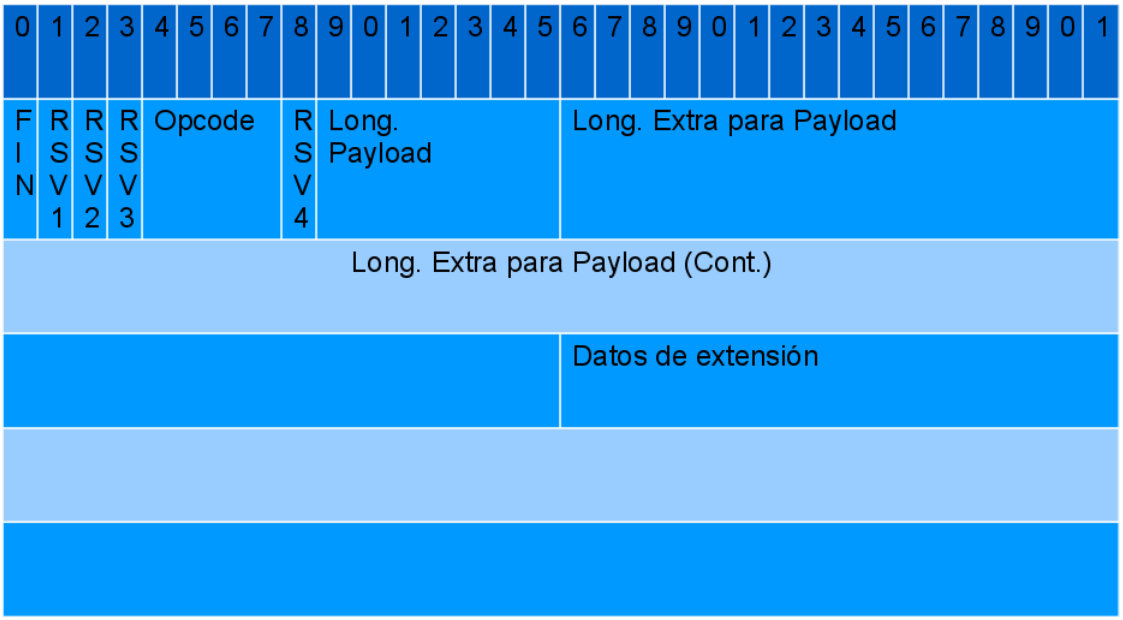

Figura 4.2: Formato de un *frame* Websocket

Veamos los principales campos de un fragmento. Para una descripción más detallada, el lector puede consultar [14].

FIN

Es un campo de 1 bit que indica si es el fragmento final del mensaje. Para el caso de mensaje pequeños, el primero fragmento también puede ser el último.

RSVx
Cada campo reservado tiene una longitud de 1 bit y su valor debe ser 0, a menos que se negocie una *extensión* que defina un significado para esos campos.

Opcode
Indica la forma en que debe interpretarse el *payload*. Más adelante se mencionan los tipos básicos de mensajes.

Longitud del payload
Indica la longitud del payload y tiene 7 bits (0 a 127)[5]. La longitud marcada es la longitud de la extensión de datos más la longitud de los datos de la aplicación.

Extensión de datos
En principio, puede ocupar N bytes, y es 0 a menos que haya presente un bit o un opcode reservados. En este caso el protocolo interpreta que se negoció una extensión.

Datos de la aplicación
Son los datos de la aplicación, con una longitud arbitraria. Luego de la extensión se considera que el resto son bytes de datos de la aplicación.

Frames de control

Son frames que se usan para comunicar el estado del Websocket. Todos los frames de control deben tener un tamaño de 125 bytes o menos y no deben fragmentarse.

Close - Opcode 0x01
Las aplicaciones no deben enviar más mensajes luego de haber enviado un frame de cierre.

Si el cuerpo del mensaje de cierre coincide con el cuerpo de un mensaje de cierre enviado previamente, se lo considera un ACK. De otra forma, se lo considera una solicitud para cerrar el enlace.

Cuando uno de los extremos recibe un mensaje de cierre, debe enviar el ACK tan pronto como pueda.

Se considera que el Websocket está cerrado cuando un extremo recibe un ACK de cerrado o envía un ACK a un mensaje de cierre.

Ping - Opcode 0x02
Cuando un extremo recibe un *ping*, debe enviar un *pong* en respuesta, tan pronto como le sea posible. Los cuerpos del *ping* y el *pong* deben coincidir.

Pong - Opcode 0x03
Ídem Ping.

[5]El campo indica la longitud si está entre 0 y 125. Si el valor es 126, la longitud del payload son los próximos dos bytes, interpretado como un entero de 16 bits sin signo. Si el valor es 127, los siguientes 8 bytes, interpretados como un entero de 64 bits sin signo, indican la longitud del payload.

Frames de datos

Cualquier otro tipo de *frame* que no aparece listado en la sección anterior son *frames* de datos. es decir, datos de la aplicación. Los frames de datos se dividen en dos:

Texto plano
El *payload* es texto en formato UTF-8.

Binarios
El *payload* es información en formato binario cuya interpretación depende de la aplicación.

Extensiones

El protocolo de Websocket está diseñado para ser extensible. Cualquier extensión se negocia durante el *handshake* entre las partes. El draft de la RFC que define el protocolo Websocket[14] reserva para extensiones los *opcodes* 0x6 a 0xF, el campo de datos *extensión* y los bits RSV1, RSV2, RSV3 y RSV4.

Manejo de errores

Al día de la fecha (Julio 2011) el draft no menciona nada respecto a la forma en que se manejan los errores y las excepciones.

Handshake para el inicio y cierre de la conexión

El handshake inicial está diseñado para ser compatible con los servidores HTTP y demás software intermedio (Proxies, por ejemplo), de forma que se pueda utilizar tanto Websocket como HTTP en el mismo puerto. Por esta razón, el *handshake* de apertura es un mensaje HTTP solicitando pasar a Websocket:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: Websocket
Connection: Upgrade
Sec-Websocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-Websocket-Origin: http://example.com
Sec-Websocket-Protocol: chat, superchat
Sec-Websocket-Version: 6
```

El orden de los encabezados no tiene importancia. Es por medio de ellos que un cliente especifica subprotocolos, cookies, etc.

Para cerrar la conexión, cualquiera de los dos extremos puede enviar un mensaje con un *opcode* de cierre de conexión. Cuando el otro extremo lo recibe, envía un ACK de cierre.

Dado que un análisis exhaustivo del protocolo no forma parte de las motivaciones de este trabajo no se mencionarán aquí los detalles del establecimiento y finalización de una conexión Websocket. Puede encontrar una explicación detallada en el draf de la RFC [14].

Es de notar que luego de ser implementado en Chrome, Firefox y Opera, Websocket fue deshabilitado en las siguientes versiones de esos navegadores hasta el presente (Mediados de 2011) debido a una vulnerabilidad en el mecanismo de *handshaking*.

La vulnerabilidad se describe en [24]. Dicho trabajo también describe vulnerabilidades similares para Java y Flash.

4.3. Websocket con HTML5: Un nuevo paradigma de desarrollo web

Anteriormente se presentaron los conceptos básicos relacionados con la Web y se habló de las nuevas tecnologías que introduce HTML5, como son los web workers, las bases de datos del lado del cliente y, por supuesto, el protocolo Websocket.

En esta sección se tratará de dar una visión futura de las aplicaciones web teniendo en cuenta estas tecnologías en conjunto.

4.3.1. La web pre-AJAX

La web anterior a Ajax se caracteriza por páginas HTML con una baja interacción por parte del usuario.

El patrón de tráfico entre el cliente y el servidor consiste de requerimientos hechos por el navegador que traen recursos relativamente grandes (textos extensos e imágenes); así mismo, el tiempo entre requerimientos es considerable.

Las páginas web tienen una extensión importante y el cliente descarga todo el contenido de una vez, tratando de optimizar el uso de su cache.

Es necesario mencionar que las conexiones a Internet anteriores al año 2005 tenían un ancho de banda inferior al actual y la comunicación vía modems era la norma.

4.3.2. La web basada en AJAX y Comet

La masificación de la banda ancha y la conexión a Internet de tiempo completo acelera el surgimiento de la web como una plataforma para la ejecución de aplicaciones.

Este período favorece la utilización de Javascript y CSS de forma casi masiva. El lenguaje de la Ecma pasa a ser el centro neurálgico de toda aplicación web.

Sin embargo, la tecnología disponible es insuficiente para los requerimientos

de las nuevas aplicaciones. Esto deriva en la aparición de técnicas que suplen las falencias que, por diseño o implementación, tiene la web.

Por ejemplo, con la masificación de la transmisión de video por Internet se utiliza Flash, dado que la W3C no muestra ninguna alternativa viable.

Lo mismo ocurre con las comunicaciones bidireccionales: varios productos en paralelo y al mismo tiempo comienzan a usar técnicas para establecer una comunicación bidireccional entre el cliente y el servidor. Estas técnicas luego se agrupan bajo el nombre de Comet.

Queda de manifiesto, entonces, que los organismos que rigen la web deben dar un nuevo marco de desarrollo que contemple el nuevo uso que se está haciendo de la web.

4.3.3. La web con HTML5

El nuevo estándar de la W3C contempla la web como una plataforma de desarrollo de aplicaciones, y no como un repositorio de documentos HTML. Entiende que ahora Javascript es esencial para toda aplicación web, de ahí que establezca pautas para la utilización de *threads* en Javascript, conocidos bajo el nombre de WebWorkers.

La movilidad es otra característica creciente de la web actual. Sin embargo, los medios de comunicación subyacentes no proveen una disponibilidad total de Internet a los dispositivos móviles. Es así que sería deseable que las aplicaciones web pudieran almacenar datos de forma local, tanto para su utilización posterior de forma *off-line*, como para su modificación y posterior carga a Internet una vez el dispositivo haya obtenido una nueva conexión. Las bases de datos locales sirven a este propósito.

Gracias a las bases de datos locales, en el caso más extremo el cliente puede descargar la aplicación web, trabajar en modo *off-line* y luego cargar la información a Internet. Para el caso general, se puede almacenar información de sesión sin necesidad de recurrir al servidor web.

Otra tecnología íntimamente ligada a los dispositivos móviles es la geolocalización. El estándar HTML5 provee mecanismos para obtener las coordenadas de un dispositivo -móvil o no-. Actualmente esta información se utiliza ampliamente en varios portales sociales.

La web, además de transformarse en una plataforma para ejecutar aplicaciones, se torna en un medio **social**. Esto quiere decir que un usuario debe enterarse de las acciones ejecutadas por los otros usuarios en su círculo social.

Además, se espera que la plataforma avise inmediatamente de cualquier cambio. Es decir, el usuario no debe preguntar si algo cambió; la plataforma debe avisarle.

Claramente se revierte el paradigma de cliente servidor, siendo el servidor quien contacta al cliente para informarle que algo cambió. Y son muchas las cosas que pueden cambiar: un nuevo resultado en una búsqueda recurrente del usuario, un comentario nuevo para un recurso subido por el cliente, el arribo de

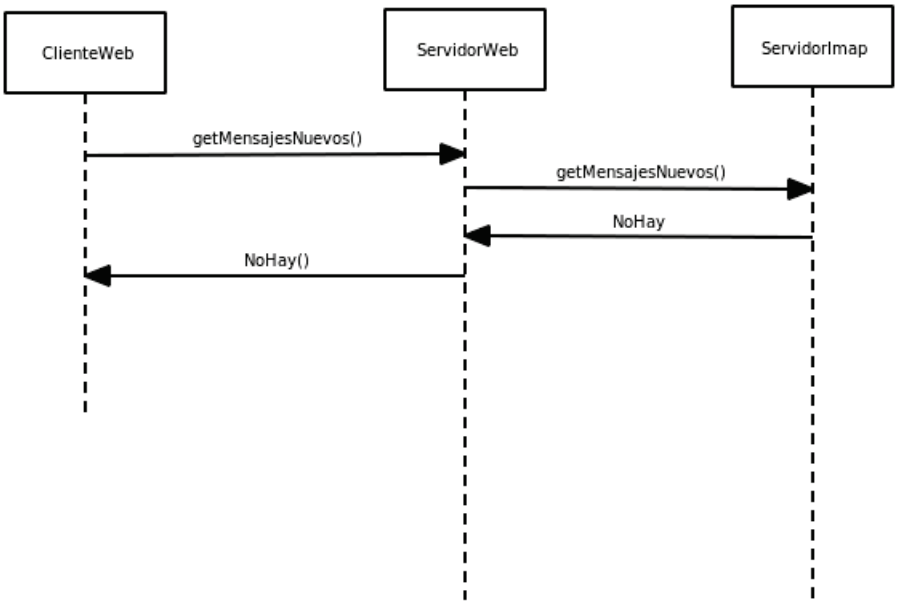

Figura 4.3: Uso de Imap desde un servidor web

un email en una plataforma de *webmail*, por mencionar algunos ejemplos.

Para resolver esta situación la W3C propone el nuevo protocolo Websocket que permite implementar este tipo de comunicación entre cliente y servidor web.

Dado que Websocket se puede integrar con otros protocolos, como IMAP[17] o XMPP[16], se puede desacoplar al servidor web de todo ese tráfico y procesamiento. La Figura-4.3 muestra la forma en que se acoplan estos servicios actualmente.

Por otro lado, la Figura-4.4 describe la arquitectura aquí planteada, en donde el cliente web dialoga directamente con el servidor de Websocket. Más aún, notar que el cliente no tiene que preguntar constantemente si hay o no mensajes nuevos; es el servidor de Websocket quien le avisa cuando tiene un mensaje nuevo.

4.3.4. La nueva web

Las nuevas tecnologías que conforman HTML5 perfilan la web hacia un ambiente más social e interactivo, con aplicaciones escritas en HTML, CSS y Javascript, utilizando JSON o XML sobre Websocket para el intercambio de datos.

AJAX y WebWorkers se encargan del manejo de eventos, mientras que Comet tiende a desaparecer gradualmente. Sin embargo, la desaparición de Comet no implica el fin para los productos que lo implementan. Por ejemplo, Aped (el servidor de Comet usado en este trabajo) ya implementa Websocket como mecanismo de comunicación.

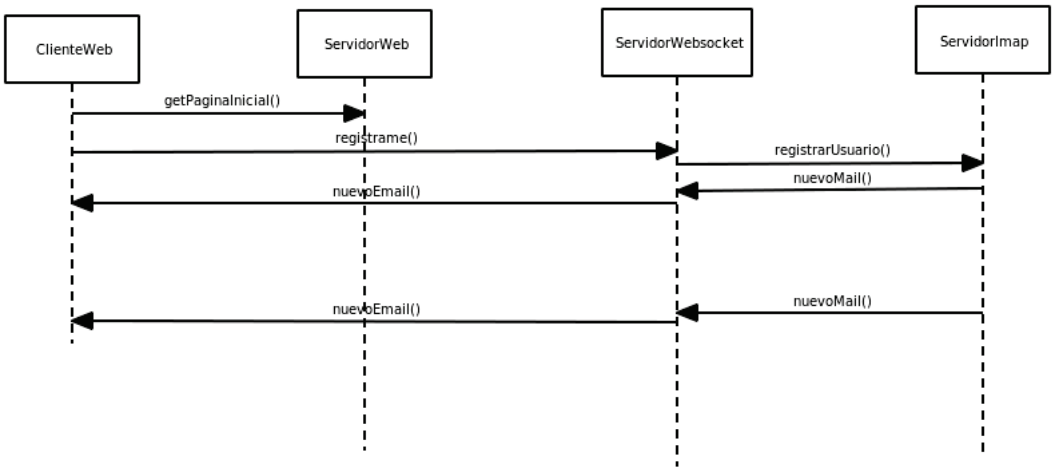

Figura 4.4: Servicio de Imap usando Websocket

De esta forma, aunque las técnicas de Comet dejen de utilizarse, los productos que las implementan pueden sobrevivir si comienzan a implementar Websocket. De esta forma, y si el cambio de Comet a Websocket se realiza de forma transparente a la aplicación, se reduce el costo en tiempo y dinero de migrar a Websocket.

4.4. Posibles usos

Como se mencionó en secciones anteriores, el principal objetivo del protocolo son las RIAs, **Rich Internet Applications**. En particular, aquellas cuyo estado se deba conocer en tiempo real. Un ejemplo típico de esta clase de aplicaciones es un chat basado en web, donde los usuarios deben ver las actualizaciones de los mensajes instantáneamente. Otro ejemplo son las aplicaciones colaborativas, como paquetes ofimáticos on-line.

Otra ventaja importante que presenta Websocket es su integración directa con otras tecnologías. De esta forma se libera al servidor web de esta carga. Un ejemplo de estos son los webmails. Estos sistemas consisten de tres capas: El cliente web, el servidor web con la aplicación web y el servidor de email. Usando Websocket es posible implementar una aplicación web que interactúe con el servidor de mail directamente, sin pasar a través del servidor web. Esto, claramente, libera recursos del lado del servidor (Aunque lleva esa complejidad al servidor Websocket y al cliente web, de ahí que sean tan importantes las tecnologías de Web workers y bases de datos del lado del cliente).

A continuación se enumeran los campos donde el protocolo Websocket es de utilidad:

- Trabajo colaborativo basado en la web, donde, por ejemplo, los escritores vean en tiempo real los cambios hechos por los demás.

- Plataformas de subastas y bolsas de comercio.
- Como transporte para otros protocolos binarios o textuales, como Jabber[6]
- Chats basados en web.
- Sistemas de conferencias web y meetings online, similar a webex.
- Sistemas para administrar computadoras de forma remota.
- Integración de HTTP con otros protocolos con estado mediante proxies Websocket.

Debe notarse que Websocket no está diseñado para el *streaming* de audio y video, pues se ejecuta encima de TCP, que no es un protocolo de transporte conveniente para hacer *streaming* en tiempo real. Además, para esto HTML5 tiene los tags de audio y video.

Por último, no hay que olvidar que Websocket está inserto en un contexto más amplio, en particular, HTML5. Son las características de HTML5 junto con Websocket los que posibilitan la creación de nuevas RIAs eficientes y basadas en protocolos abiertos.

[6]De hecho ya existe un draft de la IETF para soportar Jabber sobre Websocket[15].

Capítulo 5

Principios para una comparación entre Websocket y Comet

En este capítulo se compararán exhaustivamente Comet y Websocket. Para esto se crearán algunas aplicaciones web sencillas que simulen el comportamiento de aplicaciones web que requieren el tipo de conexión propia de Comet/Websocket.

5.1. Elección de las herramientas para la comparación

Para realizar las pruebas que se describen más abajo se acudió a herramientas escritas por terceros. Esta sección describe las herramientas usadas y el criterio según el cual fueron elegidas.

5.1.1. Servidores de Comet

Los requerimientos para los escenarios implementados en Comet eran bastante simples. Requería un *framework* que implementara la técnica de Comet (Ya sea Ajax polling, long polling o streaming). Además, su instalación y configuración debían ser sencillas. En otras palabras, debía correr hasta en las plataformas más básicas.

En Comet Daily hay una tabla comparativa con algunas implementaciones de Comet y su madurez[21].

Sin embargo, la implementación elegida no figura en dicha tabla. Para este trabajo se eligió Ape (`http://www.ape-project.org/`), Ajax Push Engine. Se trata de una plataforma Open Source muy recomendada, integrable con

Mootools, jQuery, Dojo[1] y cuyo funcionamiento es basante sencillo, lo cual es beneficioso a la hora de hacer las pruebas y comparaciones de este capítulo.

En el Anexo-C se explica Ape con más detalle.

5.1.2. Servidores de Websocket

La elección del servidor Websocket estuvo un poco más restringida, dado que no hay muchos. La primer opción utilizada fue PhpWebsocket (`http://code.google.com/p/phpWebsocket/`) que es una implementación del protocolo en PHP. El problema de esta implementación es la falta de mantenimiento por parte del creador -aparte de unos cuantos *bugs*-.

Otra opción, esta vez en Ruby, es Websocket Ruby [2]. Se trata de un proyecto personal para implementar un cliente y servidor de Websocket. El problema con este proyecto es el código, que resulta bastante caótico, sobre todo para quienes no están habituados a programar con Ruby.

Además, se consideró utilizar Node.js. Sin embargo, la mayoría son proyectos personales que a la larga dejan de estar mantenidos. Por esta razón, cuando se intentó utilizar Node Websocket (`https://github.com/guille/node.Websocket.js`) la librería era incompatible con la versión más actual de Node.js.

Por último, se encontraron dos implementaciones más en Ruby. La primera, Sunshowers (`http://rainbows.rubyforge.org/sunshowers/index.html`), se encuentra momentáneamente detenida "...hasta que el protocolo quede estable y se implemente en la mayoría de los navegadores web..." (*This project is currently unmaintained. We will pick it up again when the Websockets protocol is stable, finalized and implemented on major browsers. Otherwise don't waste your time*). Por cierto que no dan mucha seguridad.

Es así que se terminó usando una librería de Websocket para EventMachine: EM Websocket (`https://github.com/igrigorik/em-Websocket`). EventMachine (`http://rubyeventmachine.com/`) es una implementación en Ruby del patrón de diseño Reactor (`http://en.wikipedia.org/wiki/Reactor_pattern`).

EM Websocket está muy bien documentado, mantenido y se pueden encontrar muchos ejemplos de uso. Las herramientas accesorias, como EventMachine y Ruby en general, también están muy bien documentadas. Además, tienen una comunidad de usuarios bastante amplia.

5.2. Criterios para las pruebas

Antes de plantear una batería de pruebas es necesario saber *qué* se quiere probar. A la hora de hacer este planteo surgen, para este trabajo en particular, varias preguntas:

- ¿Qué parámetros se desean monitorear y cuáles no? En otras palabras,

[1] Estos son tres *frameworks* de Javascript.
[2] `https://github.com/gimite/web-socket-ruby`

¿Qué es necesario medir y qué no? O, siendo más explícito, ¿Qué se pretende medir para este trabajo?

- ¿De qué manera influye en el resultado final el tipo de aplicación usada para las pruebas?
- ¿Cómo simular un entorno realista para las pruebas?
- ¿En qué se diferencian estas pruebas de las hechas para aplicaciones web en general (Como los *benchmarks* para correr *stress tests* con AB[3] y otras plataformas de testing)?

5.3. Parámetros a monitorear

La primera cuestión es saber qué se quiere monitorear y medir. Dado que se desea establecer el desempeño de un protocolo, es necesario conocer el *overhead* que impone sobre la comunicación; y el costo de implementar y migrar una aplicación para que utilice este protocolo. Este trabajo se encarga de lo primero, siendo lo segundo parte del campo de la Ingeniería del Software, escenario en el cual no está planteado este trabajo.

Establecido el parámetro a monitorear, debe averiguarse qué elementos en la comunicación son los que inciden sobre su desempeño. Por ejemplo, el modelo particular de CPU no incide en el *overhead* del protocolo, así como tampoco no influye el tamaño de memoria RAM o la velocidad del disco rígido.

Por lo visto en el párrafo anterior, el desempeño del protocolo no depende del *hardware* instalado en las máquinas cliente y servidor. ¿Influyen la latencia y tipo de red en el desempeño del protocolo? La respuesta en "no", aunque sí influye en las mediciones, porque no es lo mismo un enlace ethernet que uno PPP, de modo que las mediciones tomadas con Wireshark (Programa utilizado para las pruebas y descripto más abajo) varían de un caso a otro. De todas formas, la infraestructura es la misma para los dos escenarios, de modo que para ambos las condiciones son las mismas.

Dado que Websocket se apoya en TCP, todas las características de la red quedan encapsuladas en las capas inferiores de la pila de red.

Por lo tanto, el *hardware* de las computadoras, el tipo de red, la latencia, ni el hardware intermedio influyen en el desempeño del protocolo. Y por lo tanto no se consideran en las pruebas.

Es así que las pruebas se centran exclusivamente en los *overheads* de los protocolos y la forma en que manejan o manipulan sus respectivos *payloads*.

5.4. Simulación de un entorno realista

Las aplicaciones web se someten a pruebas de *stress* para obtener una dimensión de su comportamiento bajo diferentes escenarios de carga.

[3] Apache Benchmark.

Básicamente, esta metodología de *testing* involucra una aplicación web a la cual se le hacen cierta cantidad N de requerimientos, siendo C la cantidad de requerimientos en concurrente. Por ejemplo, se puede probar una aplicación corriendo N=1000 requerimientos, con C=50 requerimientos en concurrente.

Por supuesto, no alcanza con eso. Para identificar los posibles inconvenientes que puedan afectar el desempeño de una aplicación web se deben monitorear los parámetros propios del servidor web donde se ejecuta. Para esto, se deben monitorear consumo de CPU, memoria RAM, cantidad de conexiones con la base de datos, errores de las interfaces de red, ancho de banda utilizado, y varios otros parámetros más.

Pero, ¿Cuál es la diferencia entre la forma de probar aplicaciones web y el desempeño del protocolo Websocket? En principio, las pruebas de stress se hacen de cara a la experiencia del usuario. Los resultados de las pruebas pueden variar incluso dependiendo del momento en que se corren.

Por esta razón, minimizando el problema, una prueba de stress trata de medir el tiempo de respuesta de una aplicación web. Sin embargo, esto no es lo que se pretende medir aquí. Debido a esto, no es posible utilizar las mismas herramientas para estas pruebas que para las de estrés. Más aún, no es importante el tiempo de respuesta de la aplicación. De hecho, para las aplicaciones normales la idea es medir el tiempo entre que inicia y se satisface el requerimiento; mientras que un enlace de Websocket no se da por finalizado, pues mantiene la conexión abierta.

5.5. ¿Cómo realizar las pruebas?

La forma de hacer los análisis es monitorear el tráfico para las pruebas. Los datos no se pueden analizar "a mano" dado su volumen y, además, por ser propenso a errores.

Por esta razón se van a utilizar herramientas propias del análisis de tráfico de web, armando filtros específicos para el tráfico que nos interesa aquí. Estas herramientas brindan datos como cantidad de bits por segundo para un protocolo en particular, así como un gráfico de protocolo.

La captura de tráfico se hace con tcpdump (`http://www.tcpdump.org/`), que es una herramienta Open Source para la captura de tráfico. tcpdump graba los datos del tráfico en formato libpcap[22].

Para el análisis de estos datos se utiliza Wireshark (`http://www.wireshark.org/`), que es una herramienta OpenSource que puede abrir archivos en formato libpcap y permite analizar todo el tráfico que se alla registrado allí. Dispone, además, de estadísticas por protocolo y una interface que permite filtrar el tráfico por protocolo y otras características más.

5.6. Tipos de aplicaciones o interacción entre cliente y servidor

En la sección anterior se vieron cuáles son las variables a tener en cuenta durante las pruebas (y cuales se pueden descartar con la seguridad de que no afectarán los resultados de las mediciones).

En esta sección se plantean dos escenarios donde se realizarán las mediciones. El mayor problema que se enfrenta aquí es la creación de un escenario **realista**, para que las mediciones lo sean.

Es muy tentador aislar el protocolo y el escenario en un entorno de condiciones óptimas, pero el resultado final no se correspondería con el uso real que se dará al protocolo, y por lo tanto, este trabajo tendría validez tan solo en un entorno aislado y óptimo.

Capítulo 6

Escenarios planteados para una comparación entre Websocket y Comet

En este capítulo se describen los escenarios planteados para medir y comparar el desempeño de Websocket y Comet.

6.1. Escenario 1: Sistema básico de logs

Se trata de una consola web donde se muestran los logs más recientes de un servidor. En el caso general, el volumen de tráfico será bastante bajo, y la comunicación fluirá desde el servidor Websocket hacia los clientes. Esto últimos no envían ningún comando al servidor, sino que reciben todo lo que transmite el servidor y lo muestran en la pantalla sin previo procesamiento.

Es cierto que los clientes podrían implementar diversas herramientas con los logs recibidos, como filtrarlos, enviar emails o graficar variables. Sin embargo, ningún procesamiento que realice el cliente sobre los datos tendrá influencia en la performance del protocolo, y en el peor de los casos el usuario notará una demora en la respuesta de la aplicación.

La Figura-6.1 muestra los pedidos reiterados que hace el cliente. Esta implementación recibe el nombre de Polling.

En la Figura-6.2 se muestra el otro tipo de implementación de Comet, en el cual el cliente realiza el primer pedido y el servidor, en vez de responder y cerrar la conexión, envía respuestas en *chunks*. Esta alternativa recibe el nombre de Streaming.

Dado que el procesamiento de esos datos no influye en el desempeño del protocolo -ni en su análisis-, cualquier modificación en el cliente para hacerlo más agradable o útil al usuario es superflua a los fines de este trabajo. Sin embargo, y aunque no se implemente aquí, esta es una buena oportunidad para

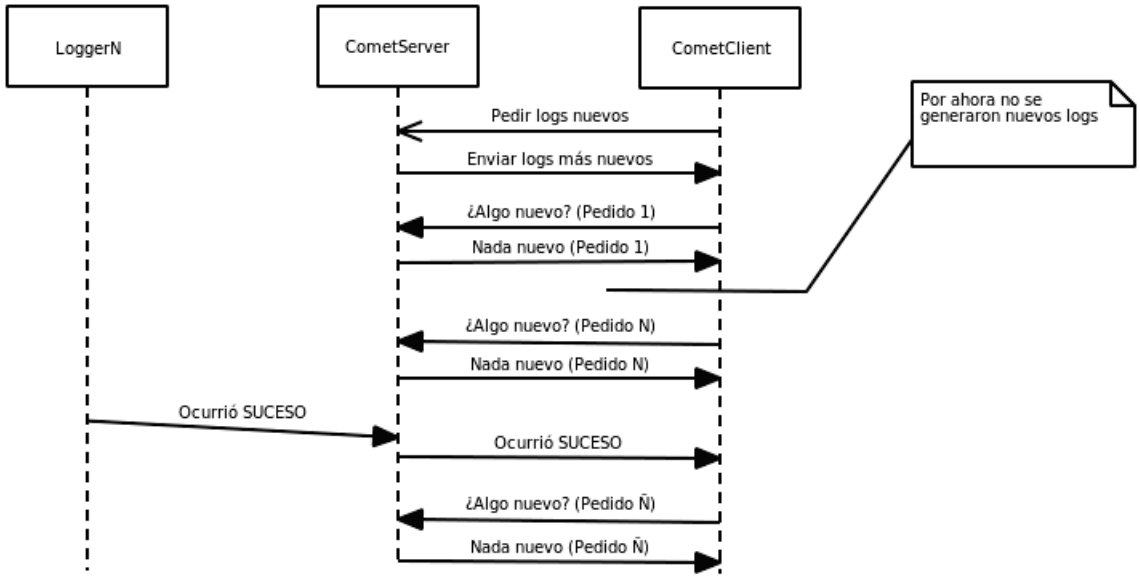

Figura 6.1: Flujo de mensajes para el escenario 1.

mencionar posibles opciones y modificaciones al cliente de logs aquí mencionado. Algunas modificaciones podrían ser:

- Colorear la sintaxis del texto correspondiente a los *logs*
- Filtrar los *logs* según las preferencias del usuario. Por ejemplo, mostrar todo sobre Apache2 y Postfix proveniente del servidor A.B.C.D, y filtrar el resto.
- Generar alertas por email cuando ocurra un suceso determinado. Para esto el cliente debería tener la funcionalidad para hacer *pattern matching* usando expresiones regulares.
- También sería posible que el cliente correlacione eventos, de modo que el usuario pueda escribir reglas que se apliquen al texto enviado por el servidor.
- El servidor podría modificar los *logs* antes de enviarlos para que se ajusten a algún formato estándar de *logging*.

Como se muestra, existen muchas aplicaciones posibles para un sistema como el utilizado en este escenario de prueba.

En la Figura-6.3 vemos diagramada la solución usando Websocket. Comparada con los otros dos diagramas, es notoria la baja en el tráfico entre cliente y servidor y, de hecho, no existe tráfico superfluo.

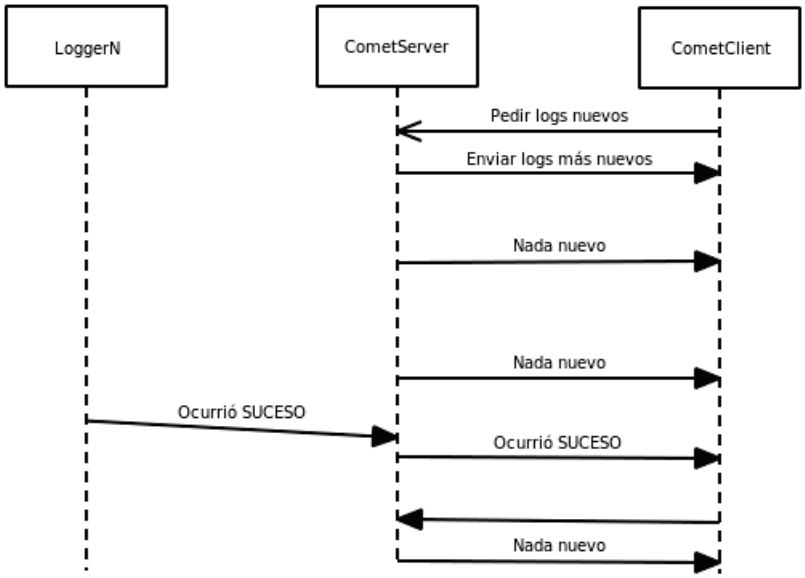

Figura 6.2: Alternativa de implementación para el escenario 1.

Sistema de logs

El log es la forma que tiene una aplicación de comunicar al administrador lo que está ocurriendo internamente. Por lo general, una aplicación puede tener varios niveles de logging, de acuerdo a la cantidad de información que brinda.

Para abstraer a la aplicación, existen algunos estándares de facto para generar logs. El más común de todos es Syslog. Cuando una aplicación utiliza Syslog, delega la escritura y forma del log en Syslog.

Una característica interesante de Syslog es que puede correr como cliente de otro Syslog. Esto permite tener un servidor de logs Syslog centralizado, mientras varios Syslogs son clientes de él y le envían los logs de las aplicaciones que corren localmente.

Para este trabajo se eligió Syslog por ser el más utilizado, aunque existen otras aplicaciones para hacer logging, así como otros formatos de logs. Por ejemplo, syslog-ng (`http://www.balabit.com/network-security/syslog-ng`) es una aplicación desarrollada por la empresa Balabit que de a poco se perfila como el reemplazante de Syslog, manteniendo la compatibilidad con este.

Por otro lado, las arquitecturas Windows utilizan un tipo de log diferente: EventLog. Incompatible con Syslog, pero de todas formas guarda la misma clase de información.

6.1.1. Implementación del escenario

Para que las implementaciones del escenario sean similares, tanto en Comet como Websocket, se diseñó una arquitectura lo más estándar posible. De esta

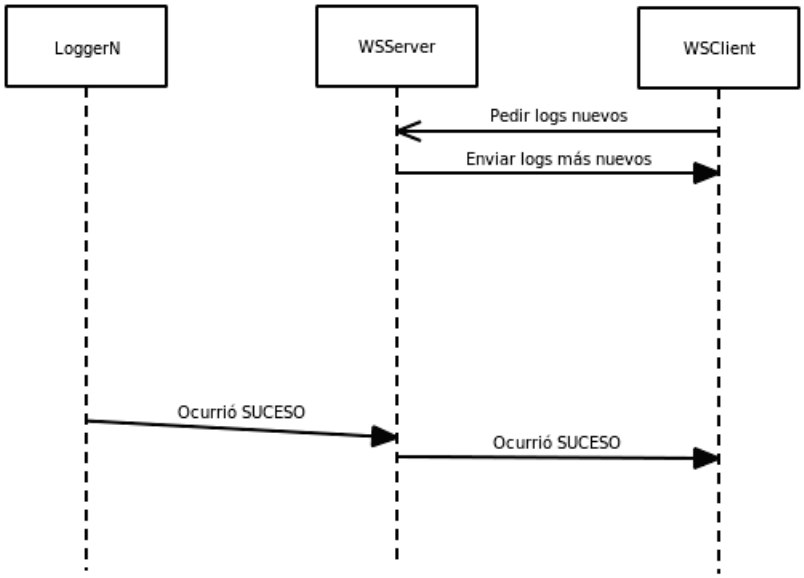

Figura 6.3: Implementación del escenario utilizando Websocket.

forma, la lógica de las aplicaciones es similar. Además, como veremos en un momento, es una arquitectura que escala muy bien a decenas de servidores.

Cada aplicación genera sus *logs*, que los envía al sistema de logging Syslog. El escenario toma de Syslog todos los logs y los envía al cliente web, en un caso utilizando Websocket, y en el otro, Comet.

```
1  // Comando de Comet que atiende las peticiones
2  // de los navegadores web.
3  Ape.registerCmd("foocmd", false, function(params, infos) {
4    if ($defined(params.channel) && $defined(params.data) && $defined
       (params.raw)) {
5      var chan = Ape.getChannelByName(params.channel);
6      if (!$defined(chan)) return ["401", "UNKNOWN_CHANNEL"];
7
8      // Abre una conexion local y recibe datos por el puerto privado
         8001
9      var socket = new Ape.sockClient(8001, "127.0.0.1", {flushlf:
         true});
10
11     // Los datos que llegan por el puerto 8001 los reenvia a los
         clientes
12     socket.onRead = function(data) {
13       Ape.log("Llego data: "+data+"\n");
14       chan.pipe.sendRaw("postmsg", {"message": data});
15     }
16
17     // Avisa a todos que hay un cliente nuevo
18     socket.onConnect = function(){
19       Ape.log("Conectados!!!");
20       this.write("hola!!!!\n");
21     }
22
23     socket.onDisconnect = function(){
```

```
24        Ape.log("Desconectados!!!!");
25      }
26
27    }
28    else
29    {
30      return 0;
31    }
32  })
```

Listing 6.1: Comando de Aped para los escenarios

Estructuralmente las implementaciones del escenario basado en Comet y en Websocket son similares:

- Las aplicaciones del servidor generan logs[1].
- Mediante un pipe (|), se envían los logs de Syslog a una instancia de NetCat (nc).
- Los servidores de Websocket y Comet abren una conexión como *clientes* al proceso NetCat, de modo que puedan recibir todos los logs por esa conexión. Además, abren un puerto para escuchar por conexiones Websocket y Comet entrantes.
- Cuando se inician los clientes de Websocket y Comet, cada uno inicia una conexión con su respectivo servidor, y dicha conexión queda establecida hasta que el usuario la corte[2].

```
1  #Cliente de WebSocket en ruby:
2  require 'rubygems'
3  require 'em-http-request'
4
5  # Arranca el cliente y hace una peticion a
6  # la maquina 192.168.0.118 puerto 1234.
7  EventMachine.run {
8    http = EventMachine::HttpRequest.new("ws://192.168.0.118:1234").
         get :timeout => 0
9
10   http.errback { puts "Hubo un error" }
11
12   http.callback {
13     puts "Conectado"
14     http.send("Hola")
15   }
16
17   http.stream { |msg|
18     puts "Recibido: #{msg}"
19   }
20 }
```

Listing 6.2: Cliente de Websocket en Ruby

[1] Gracias a que Syslog puede recibir logs de otros servidores, es posible que en realidad los logs enviados al cliente sean los del servidor más los de N servidores adicionales, cuyos logs recibe el Syslog instalado localmente, pero esto es transparente al servidor de Comet y Websocket.

[2] A decir verdad, sólo la conexión Websocket queda establecida hasta que el usuario la corte. La conexión por Comet se regenera cada determinado tiempo, según lo indique el *timeout* en la configuración del servidor

La razón por la cual los servidores reciben los datos a partir de NetCat y no los leen directamente del FileSystem es *abstracción*. De modo que esta misma arquitectura, sin cambios, servirá para los dos escenarios.

6.2. Escenario 2: Sistema de alertas basado en web

Una variante del sistema anterior es un programa donde se envíen datos (alertas) sólo bajo ciertas condiciones. En este caso, el volumen de tráfico es muy bajo. Sin embargo, este tipo de interacción entre cliente y servidor es el que pone de manifiesto las diferencias en desempeño entre Websocket y Comet.

La idea de este escenario es utilizar algún mecanismo que simule alertas provenientes de un IDS o de un sistema de monitoreo con un período de generación de alertas aleatorio.

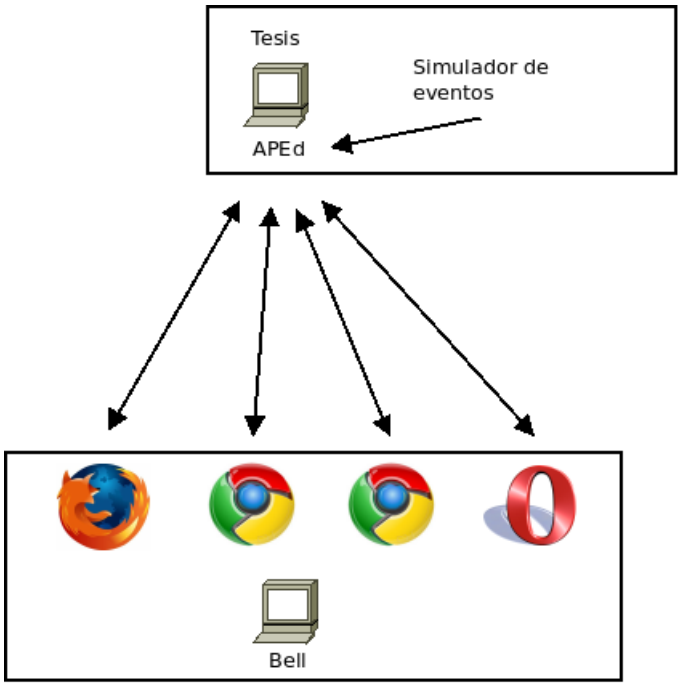

Figura 6.4: Diagrama para el escenario 2 usando Comet.

Dado que el tráfico es muy bajo, puede ocurrir -si no se generan muchas alertas- que el tráfico de control de Comet sea mayor que el tráfico propio para los datos. Sobre todo si se da el caso de tener varios clientes.

```
1 #Servidor de WS en Ruby:
2 require 'rubygems'
3 require 'em-websocket'
4
5 # Arranco EventMachine. El script escucha por
6 # conexiones en el puerto publico 1234
```

```
7  # y recibe datos por el puerto local 8000.
8  # La clase Echo entuba lo que recibe por el puerto 8000
9  # y lo saca por el 1234.
10 EventMachine.run {
11   # Canal que permite la comunicacion uno a muchos desde
12   # el servidor a todos los clientes.
13   $channel = EM::Channel.new
14
15   class Echo < EventMachine::Connection
16     def post_init
17       send_data 'Hola'
18     end
19
20     # Reenvia a los clientes conectados los datos recibidos.
21     # Este es el envio efectivo.
22     def receive_data(data)
23       $channel.push "\nDatos: #{data}"
24     end
25   end
26
27   EventMachine::WebSocket.start(:host => "0.0.0.0", :port => 1234,
         :debug => false) do |ws|
28
29     ws.onopen {
30       ws.send "Hola"
31       sid = $channel.subscribe { |msg|
32         ws.send "\nSuscripto: #{msg}"
33       }
34       # Avisa a todos los clientes que llego un nuevo cliente.
35       $channel.push "#{sid} conectado"
36     }
37     # Ejemplo iniciado en el servidor hacia todos los clientes.
38     # No se usa.
39     ws.onmessage { |msg|
40       ws.send "\nPing: #{msg}"
41       $channel.push "\nPing al canal entero: #{msg}"
42     }
43     ws.onclose {
44       puts "WebSocket cerrado"
45       $channel.unsubscribe(sid)
46     }
47     ws.onerror {
48       |e| puts "Error: #{e.message}"
49     }
50   end
51
52   EventMachine.connect '127.0.0.1', 8000, Echo
53
54 }
```

Listing 6.3: Servidor de Websocket en Ruby

La estructura de este escenario es similar a la del primer caso, aunque con algunas variaciones importantes. Por un lado, la cantidad de tráfico por segundo está calculada sobre datos reales. Además, la simulación incluirá varios clientes, tanto para Comet como para Websocket. Por último, el tamaño medio de cada mensaje también está calculado en base a datos reales.

6.2.1. Obtención de datos reales

Para simular un sistema de monitoreo se tomó como punto de partida las estadísticas de la red de la Universidad Nacional de La Plata. En base a esos números se armarán los datos y se definirá la cantidad de bytes a enviar por hora.

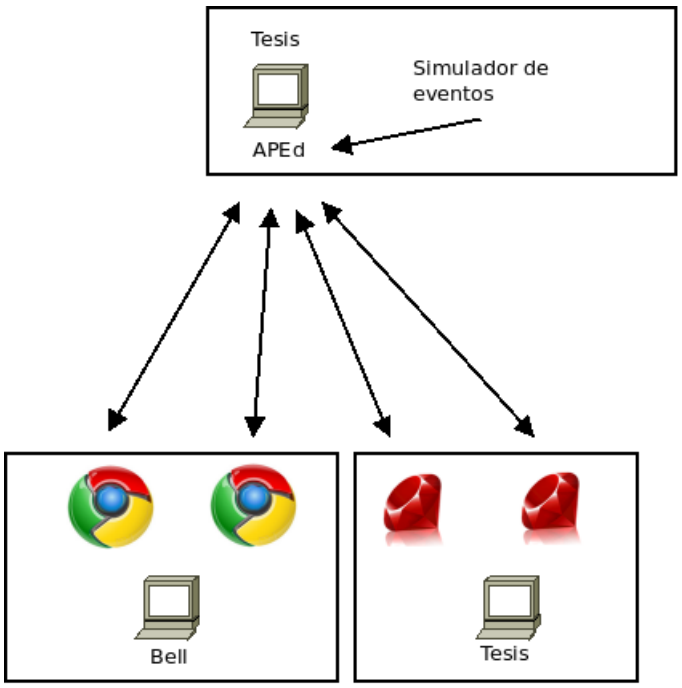

Figura 6.5: Diagrama para el escenario 2 usando Websocket.

Datos de la red

La red universitaria cuenta con más de 20 nodos pertenecientes a facultades, colegios, laboratorios y otras dependencias de la universidad (Así como algunas que dependen del Conicet). La red que administra Soporte Técnico consiste de más de 20 routers, una decena de switches, enlaces de varias tecnologías (inalámbrico, fibra óptica, cables de cobre) y casi una centena de servidores[3].

Para monitorear esta infraestructura se utiliza Nagios (`http://www.nagios.org/`), que es un sistema Open Source de monitoreo de redes y servidores. Este software envía una alerta cada vez que ocurre un evento notable en la red.

A modo de ejemplo, a continuación se muestra un mensaje real enviado por Nagios para alertar que la carga en el servidor *servercespi* con IP 163.10.0.84 volvió a un estado *normal*. En el mensaje se puede ver el nombre del servidor, su dirección IP, el tipo de notificación, y datos específicos de la alerta (En este caso, la carga del CPU).

[3]Datos solicitados al CeSPI.

```
***** Nagios *****

Notification Type: RECOVERY

Service: Promedio de carga
Host: servercespi
Address: 163.10.0.84
State: OK

Date/Time: Thu Jul 14 14:54:57 ART 2011

Additional Info:

Load : 0.56 0.77 0.97 : OK
```

Estadísticas de la red

Ahora que la infraestructura es conocida, es tiempo de encontrar algunas estadísticas reales útiles a los fines de este trabajo. La Figura-6.6 muestra la cantidad de alertas producidas para la semana del 13 al 20 de julio de 2011.

Overall Totals

Host Alerts

State	Soft Alerts	Hard Alerts	Total Alerts
UP	570	64	634
DOWN	2401	47	2448
UNREACHABLE	1688	27	1715
All States	4659	138	**4797**

Service Alerts

State	Soft Alerts	Hard Alerts	Total Alerts
OK	1074	1034	2108
WARNING	1344	242	1586
UNKNOWN	54	156	210
CRITICAL	696	813	1509
All States	3168	2245	**5413**

Figura 6.6: Alertas de Nagios para la semana del 13 al 20 de julio de 2011.

Como se muestra en la Figura-6.6, durante esa semana se tuvieron un total de 4797 eventos relacionados con *hosts* y 5413 relacionados con servicios. Cabe aclarar que mucha de esta información podría ser repetida, dado que si un host falla, todos los servicios que corre también fallarán. Sin embargo, para este trabajo se considerará el total de eventos.

Datos para las pruebas

Por último, resta obtener un marco de prueba real para este escenario. En base a los datos de la sección anterior, se puede asumir que:

- El total de eventos por semana es de $4797 + 5413 = 10210$.

- Es decir, por día son: 10210,0/7,0 = 1458,57. O sea que se tienen 1458,57/12,0 = 121,54 **eventos por hora**[4].
- Además, hasta 4 personas encargadas del monitoreo. O sea, la simulación deberá tener al menos 4 clientes.
- Queda saber el tamaño medio de los mensajes. Se tomará como media el tamaño del mensaje de arriba, **213 bytes**.

Resumiendo, la simulación deberá enviar 213 bytes a al menos 4 clientes unas 122 (121,54 redondeado) veces en el marco de una hora. Eso equivale a enviar 25986 bytes (26KB) a 4 clientes en una hora. La Figura-6.4 muestra la configuración usada para las pruebas de este escenario utilizando Comet, mientras que la Figura-6.5 muestra la configuración para las mismas pruebas, pero usando Websocket.

[4] Asumiendo un servicio de monitoreo de 12x7

Capítulo 7

Análisis de los resultados obtenidos

En este capítulo se analizan los diferentes resultados obtenidos para los escenarios planteados.

Las pruebas se llevaron a cabo utilizando dos computadoras diferentes. La primera -llamada **Tesis**- corría los servidores de Websocket y Comet, mientras que la segunda -llamada **Bell**-, hacía de cliente.

En Bell se utilizó un *sniffer*[1], llamado **tcpdump**, para capturar todo el tráfico y luego se filtró ese tráfico usando Wireshark.

Dado que se guardaron todas las capturas en formato PCAP, es posible volver a abrirlas usando, por ejemplo, Wireshark, y así corroborar los datos y afirmaciones hechas en el presente trabajo.

Por último, y para comenzar ya con la descripción de cada escenario, se recordarán algunos aspectos técnicos de importancia: La máquina **Tesis** corre un Ubuntu Maverick con Apache/2.2.16, AJAX Push Engine Server 1.00 (Aped, servidor de Comet), Ruby 1.8.7, EventMachine-0.12.10 y la librería em-Websocket-0.2.1 (La librería que implementa Websocket con EventMachine).

Además, la máquina **Bell** corre un Ubuntu Lucid. Los navegadores usados fueron: Mozilla Firefox 3.6.13 para los escenarios con Comet y Google Chrome 10.0.648.204 para Websocket. Del lado del cliente no es necesaria ninguna otra aclaración.

Estos son los datos técnicos para todos los escenarios (Donde haya algún cambio será notado como corresponda).

[1] Un sniffer es un programa para monitorear y analizar el tráfico en una red de computadoras.

7.1. Escenario 1: Sistema básico de logs

El escenario 1 simula una aplicación web para el estudio y análisis de logs. Dado que a los efectos de este trabajo sólo interesa estudiar y analizar el tráfico entre cliente y servidor y no el procesamiento de los logs por parte del cliente, la lógica del lado del cliente se limita a imprimir lo que recibe.

Un sistema real debería hacer alguna clase de procesamiento con los datos que recibe, pero esa funcionalidad extra no repercute en estas pruebas, y por lo tanto no se la implementa.

Tanto para este escenario como para los demás, todos los datos enviados, así como las capturas de tráfico y las aplicaciones desarrolladas, se encuentran disponibles en el CD que acompaña a este trabajo.

Se corrieron cinco pruebas para este escenario. Los datos usados fueron *strings* de diferentes tamaños, todos representando mensajes cortos del log. La variación de una prueba a otra se puede medir como la proporción *bytes enviados/segundos*, que a partir de ahora se llamará **densidad**.

El análisis de las pruebas se divide en dos partes (Este esquema se mantendrá también para el escenario 2). Primero se analiza el desempeño de Comet y Websocket midiendo el *overhead* en bytes de cada uno; luego, en la segunda sección, se analiza la *performance* de uno y otro en función de la densidad de tráfico.

Los datos de las pruebas realizadas se pueden encontrar en el Anexo-A.1. Allí se pueden ver los datos obtenidos junto con una pequeña descripción de la prueba.

7.1.1. Descripción de las pruebas realizadas

Esta sección describe brevemente las cinco pruebas realizadas para el escenario 1. Para más información puede recurrir al Anexo-A.1.

La prueba 1 duró 5 minutos y mostró una densidad de 2,68 bytes/seg. La prueba 2 también duró 5 minutos, pero su densidad fue mayor: 4,74 bytes/seg.

La prueba 3 fue la más larga, con una duración de 12 minutos y una densidad de 4,8 bytes/seg. A su vez, la prueba 4 fue la más corta, con una duración de tan sólo 1 minuto; sin embargo, ésta fue la prueba que mostró mayor densidad de tráfico, con 123,13 bytes/seg.

Por último, la prueba 5 duró 5 minutos y tuvo una densidad de 0,05 bytes/seg., siendo la prueba con densidad más baja.

7.1.2. Comparación del volumen de datos y el *overhead* en bytes

Si se compara el volumen de bytes que utiliza uno y otro, la diferencia es grande: Cuando la densidad no es ínfima, Websocket tiene un *overhead* de entre 2 y 5 veces la cantidad original de datos. En estas condiciones, Comet requiere

entre 70 y 100 veces esa cantidad[2].

Prueba	X veces más tráfico con Comet que con Websocket
1	17.93
2	17.64
3	17.01
4	10.90
5	36.03

Cuadro 7.1: Cuadro que muestra que Comet requiere entre 10 y 36 veces más tráfico que Websocket.

La Figura-7.1 muestra una comparación entre la cantidad original de bytes (los datos), los bytes usados por Websocket para encapsular esos datos y enviar por la red y los bytes que se necesitaron para hacer lo mismo con Comet. El gráfico muestra claramente la eficiencia de Websocket por sobre Comet.

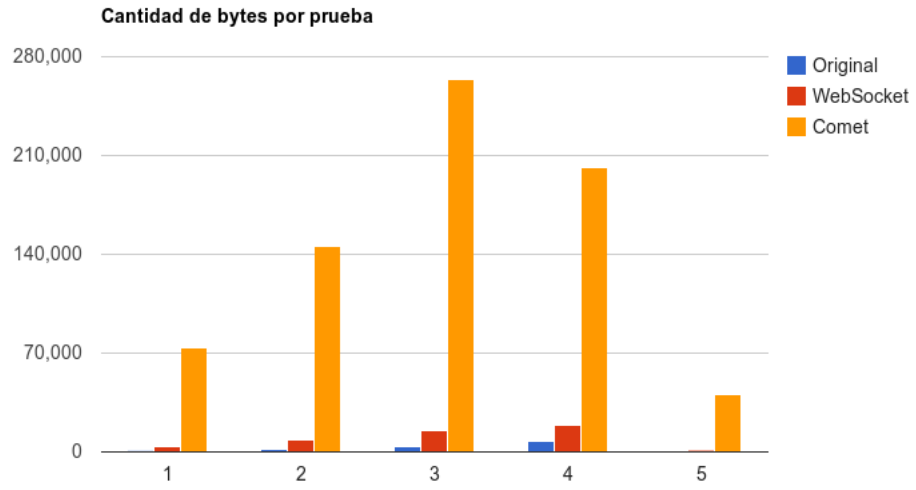

Figura 7.1: Cantidad total de Bytes.

La Figura-7.2 compara los *overheads* en bytes requeridos para el envío de los datos en cada caso (El overhead es 0 para los datos originales). Asimismo, el Cuadro-7.1 muestra cómo la menor densidad perjudica bastante a Comet y requiere 36 veces más tráfico que Websocket. Claramente Websocket muestra un desempeño muchísimo mejor que Comet.

Por último, la Figura-7.3 muestra gráficamente los datos tabulados en el Cuadro-7.1.

[2]Con la excepción de 27 veces para la prueba 4 de alta densidad

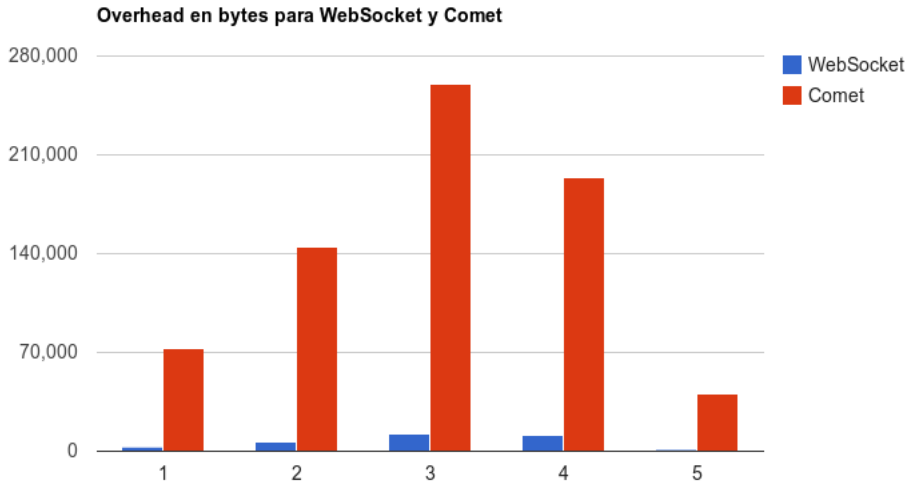

Figura 7.2: Overhead en Bytes.

7.1.3. Comparación de desempeño en función de la densidad de tráfico

Como se dijo anteriormente, la densidad es la proporción *bytesenviados/segundos*. Las pruebas van de densidades muy bajas (0,05, Prueba 5) a densidades más bien altas (123,3, Prueba 4).

El Cuadro-7.2 tabula el *overhead* mostrado por Websocket y Comet. Se ve que para densidades normales, el overhead de Websocket es bajo e, incluso, predecible; mientras que el de Comet es muy variable, ya se trate de una densidad alta o baja. Lo único que se puede suponer del tráfico de Comet es que será de al menos 30 veces más que el volumen de bytes original. Por ejemplo, para el segundo caso se enviaron 5.0944 veces más bytes con Websocket, y 91.344 con Comet.

Las pruebas llevadas a cabo muestran que, a menor densidad, ambos, Comet y Websocket, tienen un *overhead* bastante grande. Aunque hay que aclarar que el desempeño de Comet es 36 $(2724,5333/75,6)$ veces peor que el de Websocket (Figura-7.4).

Densidad	Overhead WS	Overhead CM
0.05	75.6	2724.53
2.68	5.09	91.34
4.74	5.80	102.45
4.88888	4.40	74.97
123.133	2.50	27.28

Cuadro 7.2: Resultados obtenidos para distintas densidades de tráfico.

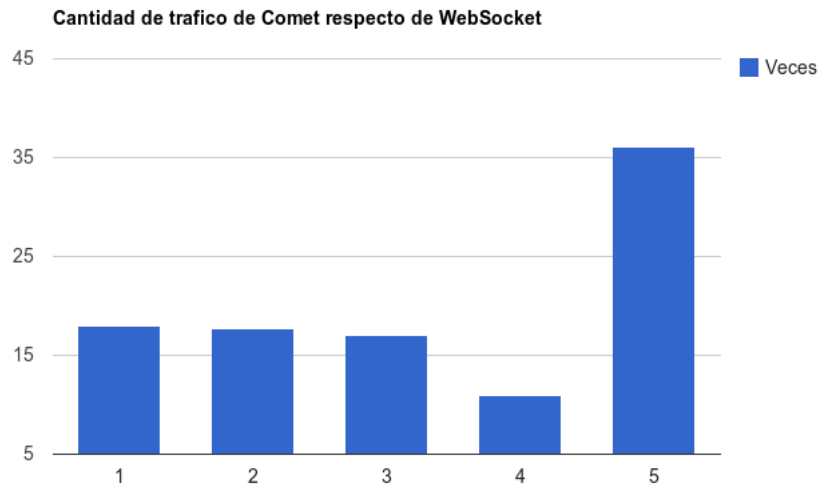

Figura 7.3: Gráfico con cantidad de veces más de tráfico que utiliza Comet con respecto a Websocket.

La última prueba realizada requiere una mención aparte. Para este caso, Websocket requiere 75 veces la cantidad de bytes original, pero Comet requiere aún más: 2724 veces.

Por otro lado, a mayor densidad, la proporción tiende a disminuir, mostrando Websocket una mejora con respecto a Comet de *sólo* 10 veces (Ver Cuadro-7.1 y Figura-7.3).

Es así que, hasta el momento, los datos demuestran que Websocket es un protocolo altamente superior a Comet (Al menos al implementado por Aped). Tanto si vemos la relación de desempeño como el volumen de bytes enviados, Websocket aventaja, por mucho, a Comet.

7.1.4. Comentarios sobre el desempeño en este escenario

Se vio que a mayor densidad, tanto Comet como Websocket tienden a hacer un mejor uso del ancho de banda disponible, debido a que la proporción entre el *payload* (los datos en sí) y los encabezados de los protocolos (Websocket y HTTP para Comet) es mayor. En otras palabras, el *overhead* se ve reducido cuando cada dato enviado es de mayor tamaño (Captura 4). Esto ocurre porque lo que se envía "encaja" mejor. Es decir, la proporción $DATO/Paquete$ es mejor cuando el dato es grande que cuando el dato es pequeño.

El tráfico en Internet es por ráfagas. Este escenario simula esta clase de tráfico, con pequeñas ráfagas de datos enviados desde el servidor al cliente. Estos datos, que en el presente escenario son strings pseudoaleatorios, podrían ser en una transmisión real, comentarios de una foto, un mensaje en un webchat, el nuevo precio para un producto o la nueva cotización de una moneda. Con este tipo de tráfico Websocket se destaca por sobre Comet.

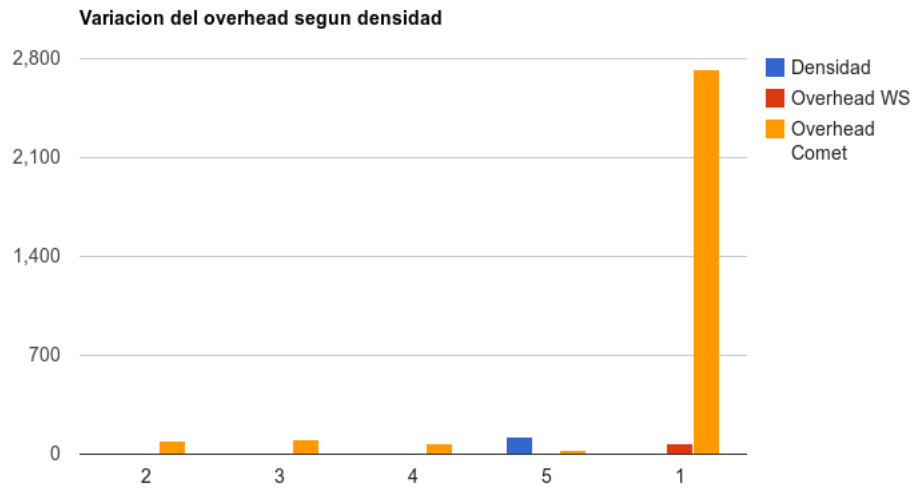

Figura 7.4: Comparación del desempeño conforme la densidad varía.

Otra resultado es que a mayor densidad, los protocolos tienden a algo similar, por lo que Websocket es *sólo* 10 veces mejor que Comet; pero cuanto menor sea la densidad, más eficiente será Websocket con respecto a Comet.

Por último, y para concluir con este escenario, se ve que el tráfico de Comet es entre 10 y 36 veces mayor que el tráfico de Websocket.

7.1.5. Más allá del escenario 1

Como se dijo más arriba, cualquier funcionalidad accesoria que se implemente para este escenario es superflua, dado que no influye en las mediciones planteadas. Sin embargo, y tan sólo a modo informativo, se enumerarán algunas ideas para agregar a este escenario:

- Graficar los logs. Por ejemplo, la cantidad de usuario que se autentican por minuto, los emails recibidos en una hora, y demás información estadística que se obtiene a partir de los logs.
- Filtrar los logs por algún criterio definido por el usuario.
- Ejecutar alguna regla para correlacionar eventos recibidos.

Se dijo que el procesamiento de los logs por parte del cliente es algo que no influye en las mediciones aquí planteadas. De todas formas, cabe preguntarse qué influencia tendría ese procesamiento en la percepción que tiene el usuario acerca del desempeño general de la aplicación. Más aún si se tiene en cuenta que al usar Comet, el cliente necesariamente debe entrar en una iteración para consultar el estado al servidor.

Por las características de la aplicación[3], el servidor retornará un mensaje advirtiendo que no hubo ningún cambio -mensaje que de todas formas deberá procesar el cliente-. De esta forma, a la carga propia de la aplicación se le debe agregar la carga de procesar los mensajes Comet. Por supuesto, si se utiliza Websocket en vez de Comet, esta carga no existe.

7.2. Escenario 2: Sistema de alertas basado en web

Para este escenario se tomaron datos y estadísticas reales de la red de la U.N.L.P. Estos datos fueron proporcionados por el Ce.S.P.I., y se corresponden con las eventualidades diarias durante una semana cualquiera.

La idea de este escenario es dar una perspectiva más realista a la hora de comparar Websocket y Comet.

7.2.1. Descripción de las pruebas realizadas

Para este escenario se realizaron 2 pruebas, pues desde el punto de vista de la densidad existen solamente dos opciones: Eventos agrupados y eventos dispersos.

La prueba 1 (Figura-7.5) simula un entorno en donde los eventos tienen lugar durante los primeros 10 minutos, para luego cesar su actividad (Aunque las conexiones entre cliente y servidor continúan establecidas).

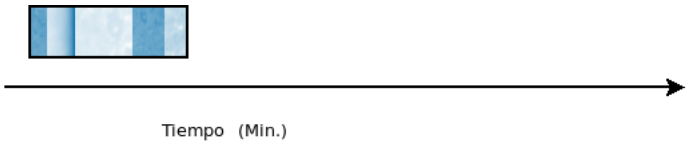

Figura 7.5: Distribución de los eventos a lo largo de la prueba 1 (Escenario 1).

La prueba 2, por el contrario, simula un ambiente en donde los eventos surgen con cierta regularidad, lo que da como resultado una distribución más pareja de los sucesos a lo largo del tiempo de prueba. Esta distribución regular de los eventos se aprecia en la Figura-7.6

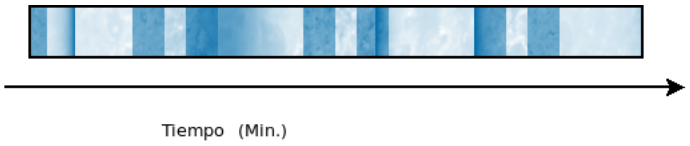

Figura 7.6: Distribución de los eventos durante la prueba 2 (Escenario 2).

Para más información sobre las pruebas, puede recurrir al Anexo-A.2

[3]La relación *logsgenerados/cantidaddeconsultas* es, por lo general, muy baja. Esto significa que para muchas consultas del cliente, el estado en el servidor no habrá cambiado.

7.2.2. Comparación del volumen de datos y el *overhead* en bytes

La Figura-7.7 grafica la cantidad de bytes original y los bytes efectivamente enviados por la red, tanto para Websocket como para Comet. Los valores tabulados para la Figura-7.7 se pueden encontrar en el Cuadro-7.3.

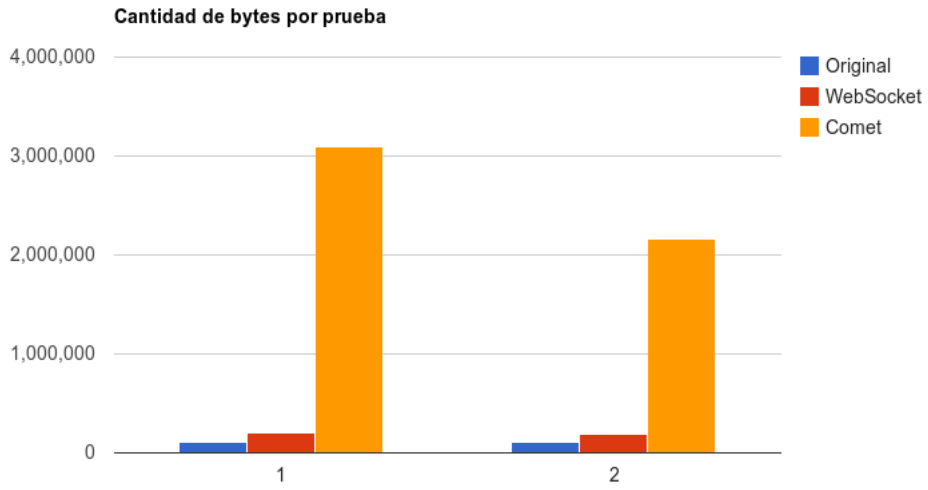

Figura 7.7: Cantidad de bytes

Original	Websocket	Comet
103944	201994	3091819
103944	185564	2157695

Cuadro 7.3: Cantidad de bytes original, enviados con Websocket y con Comet

La Figura-7.8 grafica el *overhead* en bytes para las dos pruebas, tanto para Websocket como para Comet. Nuevamente, los valores tabulados se encuentran en el Cuadro-7.4

Bytes	Overhead Websocket	Overhead Comet
103944	98050	2987875
103944	81620	2053751

Cuadro 7.4: Overhead en bytes para Websocket y Comet

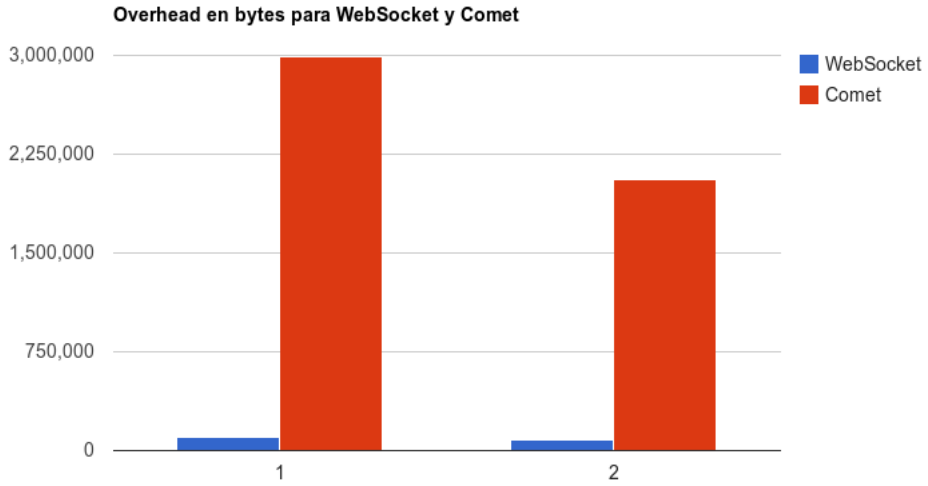

Figura 7.8: Overhead en bytes

7.2.3. Comparación de desempeño en función de la densidad de tráfico

Este es uno de los puntos más interesantes con respecto a Websocket. Como se ve en el Cuadro-7.5, el desempeño de Websocket permite tener un overhead menor al doble de la carga original. Es decir, si la carga original es de 103944 bytes, con Websocket se requiere menos del doble (1,94 y 1,78, respectivamente para cada prueba) de bytes para todo funcione.

Por otra parte, Comet muestra un desempeño totalmente opuesto. Por un lado, no es tan constante como Websocket, y por el otro, necesita más de 20 veces la cantidad de bytes original para su funcionamiento.

Bytes	Overhead Websocket	Overhead Comet
103944	1,94	29,74
103944	1,78	20,75

Cuadro 7.5: Cantidad de veces que se envía el tráfico original

Es interesante notar que, aunque la densidad sea la misma para las dos pruebas, el *overhead* es notoriamente variable para Comet. Esto se debe al tráfico superfluo, es decir, necesario para mantener la conexión, pero que no transporta datos. En el primer ejemplo, el envío de datos ocurre al principio; pasado esa ráfaga, Comet queda en reposo enviando tan solo los paquetes necesarios para mantener la conexión abierta. Y aunque estos paquetes sean pequeños, sumados dan un *overhead* importante.

Para la segunda prueba, el tráfico fue más disperso a lo largo del tiempo (Ver Figura-7.6), de modo que hubo menos tráfico superfluo de Comet.

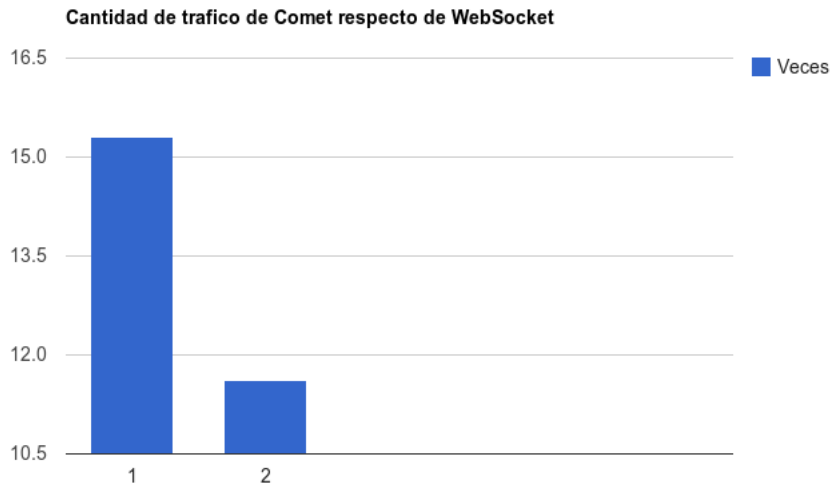

Figura 7.9: Cantidad de veces que se envía el tráfico original

Este análisis pone de manifiesto dos cosas importantes:

1. El protocolo Websocket es al menos 10 veces más eficiente que las técnicas de Comet.
2. La eficiencia de las técnicas de Comet depende tanto de la densidad del tráfico como del tipo de tráfico (si es continuo o en ráfagas), mientras que **la eficiencia de Websocket tiende a ser constante y predecible**.

En cuanto a la efectividad relativa, es decir, cuan eficiente es Websocket con respecto a Comet, se ve en el Cuadro-7.6 que ésta varía entre 11 y 15 veces. Es importante notar que esta variación viene dada por la oscilación que sufre Comet en función del tipo de tráfico, ya que, como se dijo anteriormente, la efectividad de Websocket es casi constante. La Figura7.10 muestra gráficamente los datos tabulados en el Cuadro-7.6.

Densidad (bytes/seg)	Efectividad relativa
28	15.3
28	11.62

Cuadro 7.6: Efectividad relativa en función de la densidad

7.2.4. Comentarios sobre el desempeño en este escenario

Para la ejecución de este escenario se utilizó un modelo basado en datos reales, de modo que la densidad del tráfico tiende a ser realista. Aparte de la densidad, se experimentó con dos tipos diferentes de tráfico: por ráfagas y

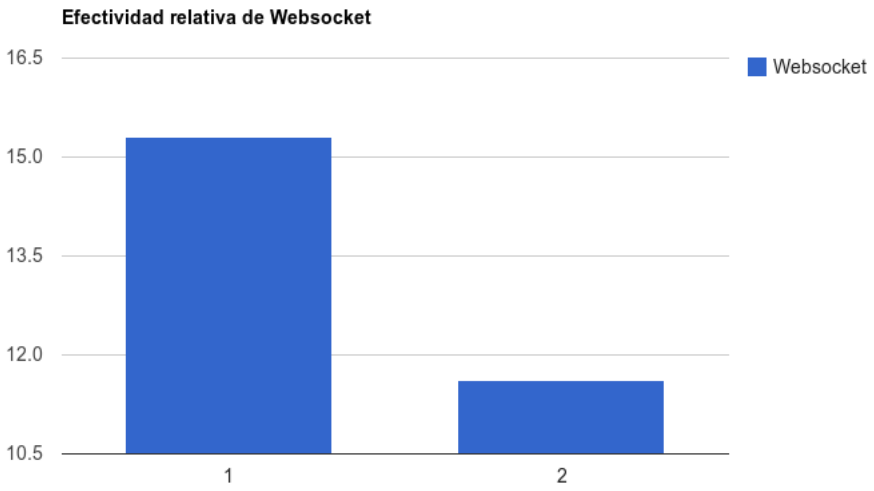

Figura 7.10: Efectividad relativa en función de la densidad

constante. Se vio que en ambos casos Websocket se desempeña mucho mejor que Comet.

Más aún, en ambos escenarios quedó comprobado que el *overhead* presentado por Websocket tiende a ser constante (y por ende, predecible), mientras que el de Comet depende tanto de la técnica implementada (polling o streaming) y del tipo de tráfico (en ráfaga o constante).

Capítulo 8

Más allá de Websocket

En este capítulo se analizan los requerimientos en la infraestructura de los servidores web para la utilización de Websocket, así como también las ventajas que conlleva su utilización. Se tratará en lo posible de dar argumentos tanto desde el punto de vista informático como económico.

8.1. Nuevas exigencias

Tratándose de un protocolo nuevo, existen nuevas exigencias, tanto para los programadores como para los administradores de los servidores. También de ambos depende la masificación -o no- de Websocket.

Los programadores tienen por delante la tarea de reescribir los programas de forma que utilicen el nuevo protocolo, para lo cual deberán recibir capacitación en Websocket y HTML5. La mayoría de los navegadores modernos ya implementan Websocket (Firefox, Chrome, Opera), por lo que ahora es el turno de las implementar el protocolo del lado de la aplicación.

Muchos servidores de Comet, como es el caso de Aped, ya implementan Websocket, de modo que Aped quedaría como una capa intermedia entre la aplicación y el protocolo Websocket. Esto sin duda facilitaría la migración al nuevo estándar.

La sección Manejo de errores en el *draft* de la IETF, así como las posibles vulnerabilidades que puede tener el protocolo, no están desarrolladas aún, por lo que este es un punto que sin duda deberán mejorar en el corto plazo.

La interacción con Websocket por parte de los desarrolladores es a través del envío y recepción de mensajes, por lo que la curva de aprendizaje no debería ser muy pronunciada. Esta característica del protocolo y su interfaz hacia los desarrolladores sin duda motivará su rápida aceptación entre la comunidad de programadores.

8.2. Mejora de performance

Existen muchas aplicaciones web que tienen un modelo de interacción basada en eventos. Más aún, estos eventos no los genera el usuario, sino que por el contrario, el cliente web queda a la espera de recibir nuevos sucesos.

Ejemplos de esta clase de aplicaciones son: interfaces web para sistemas de IDS e IPS (como Snort); interfaces web para sistemas de logging (Como Syslog-ng); sistemas web para el análisis de tráfico de email, HTTP o de cualquier protocolo en general; servicios que podrían catalogarse como *web services*, como APIs de distintos fabricantes (Twitter, Facebook, FourSquare, Delicious, etc).

En la actualidad, como se vio al inicio de este trabajo, esa funcionalidad se implementa mediante alguna de las técnicas de Comet. Es el objetivo del protocolo Websocket proveer un mecanismo más natural para este patrón de comunicaciones.

La mejora en la implementación de este patrón de comunicación -donde el cliente queda a la espera por tiempo indefinido a que el servidor le envíe información- trae buenas noticias tanto para la infraestructura donde se aloja el servidor como para el cliente.

Por un lado, los clientes sólo deberán procesar eventos con datos significativos, en vez de procesar eventos que indican que nada cambió del lado del servidor (Como sucede frecuentemente con cualquier técnica de polling); y por el otro lado, el menor consumo de ancho de banda se traduce en un tiempo de vida más largo para la infraestructura de los *datacentres*; además, dado que al utilizar Websocket el servidor queda exento de *pollings*, su carga de CPU también se reduce, alargando también su vida útil.

8.3. Integración de sistemas usando Websocket

El protocolo Websocket provee una conexión estable entre cliente y servidor. Es posible, entonces, diseñar proxies para integrar clientes web con servidores de otros protocolos, como pueden ser servidores de Jabber, de MSN, de IMAP, de impresión, de email y otros.

8.4. Balance de carga y HA con Websocket

Cualquier sitio web cuyo tráfico sea significativo debería estar balanceado e implementar algún mecanismo de *alta disponibilidad* (HA). Es obvio que Websocket tiene que poder encajar en ese contexto también, de lo contrario se convertiría en un cuello de botella para el sistema.

Como se menciona en [20], existen muchas maneras de balancear aplicaciones web. N. Qveflander[20] propone utilizar balance por origen y confiar en que, ante una desconexión no duradera, la información de sesión permanecerá en el servidor web lo suficiente como para que el cliente la retome.

Por otra parte, la alta disponibilidad (HA) se puede garantizar utilizando alguna implementación del protocolo VRRP (Virtual Router Redundancy Protocol), como Keepalived. Esta funcionalidad tiene lugar capas más abajo y por esa razón ocurre de forma transparente para Websocket.

8.5. Gateway de Websocket e integración con otros protocolos

La motivación detrás de un proxy Websocket es la de crear un *adaptador* de un protocolo a otro. Por ejemplo, y continuando con el tema propuesto en [15], es posible diseñar un subprotocolo Websocket para integrar XMPP (El protocolo de Jabber) con la web.

Es posible diseñar subprotocolos de Websocket para fines muy distintos, como pueden ser el envío y recepción de email o la administración de equipos de forma remota.

Hay que recordar que, si bien por el momento el uso más común de Websocket se hace a través de los navegadores web, nada impide que se lo utilice desde una aplicación cualquiera.

Téngase en cuenta que Websocket está pensado para transmitir en dos modos, texto plano o binario. Sin embargo, por el momento Javascript no puede trabajar con datos en modo binario y, por lo tanto, sólo las aplicaciones de escritorio pueden utilizar el modo binario en el protocolo Websocket.

Otra aplicación posible podría ser el *streaming* de audio y video utilizando un subprotocolo de Websocket. Esto que en principio parece una buena idea debido a la conexión orientada a sesión propia de Websocket, podría ser contraproducente, pues Websocket es un protocolo que se apoya en TCP, y por lo general las soluciones de *straming* utilizan UDP[1]. Además, la W3C está trabajando en los *tags* de audio y video para HTML5.

Moving the Entire Service Logic to the Network to Facilitate IMS-Based Services Introduction[25] es un ejemplo más de cómo las empresas están empezando a migrar a Websocket. Además, en *An XMPP sub-protocol for Websocket*[15] se propone un *binding* para XMPP sobre Websocket. Estos son dos ejemplos de entidades que ya se encuentran planeando una posible migración de su infraestructura actual a una basada en Websocket.

Al momento de escribir este trabajo, la empresa Kaazing ha desarrollado un *gateway* que permite a tecnologías basadas en la web comunicarse con otros protocolos (`http://www.kaazing.com/products/kaazing-Websocket-gateway.html`).

[1]Todos los mecanismos de que dispone TCP para hacer una entrega garantizada de los datos tienden a ser contraproducentes en el envío de audio y video por Internet, por esa razón se prefiere utilizar UDP.

8.6. Aplicaciones remotas sobre Websocket usando X Window

Bajo el nombre de X Window[18] (O simplemente X) se conoce al protocolo y varios sistemas anexos que tienen como objetivo la creación de una capa de abstracción entre los elementos de una GUI[2] y los programas que lo utilizan remotamente.

X tiene una arquitectura de cliente-servidor. Bajo esta arquitectura, las aplicaciones pueden ejecutarse en una computadora central con gran poder de cómputo y solicitar al servidor X que dibuje la interfaz para la salida de la aplicación en computadoras remotas con hardware más modesto. Este protocolo permitió el uso de programas remotos con GUIs locales.

Es posible implementar un subprotocolo de Websocket para encapsular el protocolo de X Window, permitiendo usar las aplicaciones remotas desde un navegador.

En una dirección similar avanza el proyecto noVNC (`http://kanaka.github.com/noVNC/`). Se trata de un cliente de VNC[3] basado en Websocket y Canvas, de modo que se puede acceder remotamente a una máquina usando un navegador web.

[2] Graphical User Interface

[3] Virtual Network Computing, es un sistema que permite compartir escritorios remotos, mapeando el teclado y mouse locales en la computadora remota

Capítulo 9

Conclusiones finales

9.1. Desarrolladores

Desde el punto de vista del desarrollador, el protocolo Websocket soluciona un problema de *impedance mismatch*[1]: las tecnologías utilizadas actualmente en la web no soportan todas los usos deseados. Por esta razón, los desarrolladores deben no sólo implementar las aplicaciones, sino que además deben encontrar formas de resolver los problemas del medio que usan (HTTP).

Continuando con el punto anterior, hasta hace un tiempo, para escribir una aplicación que utilice una comunicación bidireccional entre cliente y servidor, era necesario implementar tanto la aplicación como la técnica que permitiera llevar a cabo esa comunicación.

Actualmente las técnicas están implementadas (Aped, por ejemplo), permitiendo a los desarrolladores avocarse exclusivamente a la aplicación. Esto, sin embargo, con un costo alto en el desempeño de dicha comunicación.

9.2. Diseñadores de Websocket y usuarios finales

Pueden distinguirse dos períodos importantes en el desarrollo e implantación de Websocket en el mercado.

Por un lado, un primer período de puesta a punto del protocolo, donde se hagan los ajustes necesarios desde el punto de vista de la seguridad y el manejo de errores.

El segundo período será la aceptación por parte de los desarrolladores (Y la aceptación implícita del protocolo por parte de los usuarios al utilizar navegadores actualizados).

Actualmente, Websocket se encuentra en la primera etapa. Los *drafts* de las RFCs se actualizan regularmente conforme surgen inconvenientes o sugerencias[24].

[1] Se refiere a los problemas técnicos y conceptuales encontrados al implementar una aplicación utilizando un lenguaje que no reconoce dicho paradigma.

Siempre transcurre cierto tiempo desde la aparición de una tecnología nueva hasta su aceptación de forma masiva. En el caso de Websocket, esa aceptación masiva implica que el parque de navegadores se actualice a versiones recientes y que los desarrolladores comiencen a implementar sus servicios utilizando el protocolo. Actualmente, HTML5 y Websocket no tienen una fecha definida para su lanzamiento oficial.

9.3. Desde la infraestructura

Desde el punto de vista de la infraestructura y servidores, la mayor ventaja es el ahorro en el ancho de banda, dado por la ausencia de envíos y respuestas innecesarios. Además, el tráfico es más "limpio", pues ya no se tiene una marea de pequeñas peticiones y respuestas HTTP que nublan la visión general de la red.

Por otro lado, los servidores y proxies deben soportar el nuevo protocolo. Para esto pueden implementarlo internamente, mediante módulos especializados o bien delegando la tarea a servidores de Websocket.

9.4. Un nuevo paradigma necesita nuevas herramientas

Es evidente que HTML5 avanza hacia la generación de aplicaciones web con características de aplicaciones de escritorio. Websocket es fundamental, dado que permite una comunicación de eventos casi óptima. Sin embargo, no es suficiente.

Es necesario un nuevo lenguaje que permita la creación de interfaces gráficas similares a las de una aplicación de escritorio. Actualmente existen muchas bibliotecas de Javascript que permiten crear *widgets* con características similares a sus pares de escritorio. Por ejemplo, es posible crear ventanas, botones, menús, diálogos y el manejo de todos sus eventos utilizando jQuery. Pero, esto no deja de ser un parche, dado que HTML no dispone de *tags* para caracterizar estos elementos.

La solución a este inconveniente es un nuevo lenguaje promovido por la W3C, quizás similar a XUL (`https://developer.mozilla.org/en/XUL_Reference`)[2], que caracterice todos los elementos de una interfaz de usuario. Para el manejo de eventos y comunicación entre cliente y servidor se podrá utilizar Websocket.

9.5. Considerando los objetivos propuestos

Al inicio de este trabajo se plantearon algunas preguntas relativas a Websocket. Estas preguntas consideran tres aspectos:

[2]XUL es un lenguaje basado en XML usado por las aplicaciones basadas en Mozilla para la generación de sus GUIs.

- Costo de migrar a Websocket.
- Desempeño del protocolo.
- Cambios y mejoras para las aplicaciones web.

Para el primer punto es necesario un análisis más detallado utilizando técnicas de la Ingeniería de Software. Sin embargo, este trabajo se enfoca en el desempeño de Websocket y no pretende ser un análisis de costos. Por esta razón, no se puede concluir nada sobre el costo de migración.

En cuanto al desempeño del protocolo, se vio en las pruebas que Websocket presenta una mejora de al menos 10 veces con respecto a las técnicas actuales (Comet). Además, el ancho de banda consumido por Websocket tiende a ser constante (apenas debajo del doble de tráfico original), mientras que el desempeño de Comet es variable en función de si el tráfico viene en ráfagas o es constante.

Por último, en cuanto a los cambios y mejoras relativos a las aplicaciones, Websocket permite integrar protocolos y servicios (como jabber o email) a la web con un costo muy bajo. Del lado del cliente se experimenta una mejora en el desempeño, ya que el navegador no necesita consultar regularmente por actualizaciones. Esto, por último, deriva en una mejora en el uso de la aplicación web.

Apéndice A

Datos de las pruebas

En este apéndice se muestran los datos obtenidos para las pruebas.

A continuación se explica la estructura general de las muestras para facilitar su lectura:

```
Tiempo de la prueba: N MINUTOS.
Tráfico WS: TRÁFICO DE WS ÚNICAMENTE bytes (CANTIDAD DE PAQUETES WS) overhead: (TRÁFICO WS - CANTIDAD DE DATOS ENVIADOS)
                                            bytes ((TRÁFICO WS / CANT. DE DATOS ENVIADOS) veces)
Tráfico APEd/Comet: TRÁFICO DE COMET bytes (CANTIDAD DE PAQUETES COMET) overhead: (TRÁFICO COMET - CANTIDAD DE DATOS ENVIADOS)
                                            bytes ((TRÁFICO COMET / CANT. DE DATOS ENVIADOS) veces)

Cantidad de datos enviados: TAMAÑO DE INFORMACIÓN ENVIADA en bytes
(TRÁFICO COMET / TRÁFICO WS) Veces más de tráfico con Comet que con WS.
Densidad (bytes enviados / tiempo): (CANTIDAD DE BYTES ENVIADOS/TIEMPO DE LA PRUEBA) bytes/seg
```

A.1. Escenario 1: Sistema básico de logs

A.1.1. Prueba 1

```
Tiempo de la prueba: 5 minutos.
Tráfico WS: 4101 bytes (44 paquetes) overhead: 3296 bytes (5.09 veces)
Tráfico APEd/Comet: 73532 bytes (308 paquetes) overhead: 72727 bytes (91.34 veces)

Cantidad de datos enviados: 805 bytes
17.93 Veces más de tráfico con Comet que con WS.
Densidad (bytes enviados / tiempo): 2.68 bytes/seg
```

A.1.2. Prueba 2

```
Tiempo de la prueba: 5 minutos.
Tráfico WS: 8259 bytes (85 paquetes) Overhead: 6837 (5.8 veces)
Tráfico APEd/Comet: 145689 bytes (607 paquetes) Overhead: 144267 (102.45 veces)

Cantidad de datos enviados: 1422 bytes
17.64 Veces más de tráfico con Comet que con WS.
Densidad (bytes enviados / tiempo): 4.74 bytes/seg
```

A.1.3. Prueba 3

```
Tiempo de la prueba: 12 minutos.
Tráfico WS: 15508 bytes (160 paquetes) Overhead: 11988 (4.4 veces)
Tráfico APEd/Comet: 263918 bytes (1086 paquetes) Overhead: 260398 bytes (74.97 veces)

Cantidad de datos enviados: 3520 bytes
17.01 Veces más de tráfico con Comet que con WS.
Densidad (bytes enviados / tiempo): 4.88 bytes/seg
```

A.1.4. Prueba 4

```
Tiempo de la prueba: 1 minuto.
Tráfico WS: 18488 bytes (144 paquetes) Overhead: 11100.0 (2.5)
Tráfico APEd/Comet: 201600 bytes (804 paquetes) Overhead: 194212 (27.28)

Cantidad de datos enviados: 7388 bytes
10.9 Veces más de tráfico con Comet que con WS.
Densidad (bytes enviados / tiempo): 123.13 bytes/seg
```

A.1.5. Prueba 5

Aquí la densidad disminuye mucho, para mostrar un caso con baja densidad.

```
Tiempo de la prueba: 5 minutos.
Tráfico WS: 1134 bytes (16 paquetes) Overhead: 1119 bytes (75.6)
Tráfico APEd/Comet: 40868 bytes (174 paquetes) Overhead: 40853 bytes (2724.53)

Cantidad de datos enviados: 15 bytes
36.03 Veces más de tráfico con Comet que con WS.
Densidad (bytes enviados / tiempo): 0.05 bytes/seg
```

A.2. Escenario 2: Sistema de alertas basado en web

A.2.1. Prueba 1

```
Tiempo de la prueba: 60 minutos.
Tráfico WS:  201994 bytes (1292 paquetes) Overhead: 98050 (1.94 veces)
Tráfico APEd/Comet:  3091819 bytes (13678 paquetes) Overhead: 2987875 (29.74 veces)

Cantidad de datos enviados: 103944 bytes
15.3 Veces más de tráfico con Comet que con WS.
Densidad (bytes enviados / tiempo): 28 bytes/seg
```

A.2.2. Prueba 2

```
.
tiempo de la prueba: 60 minutos.
Tráfico WS:  185564 bytes (1074 paquetes) Overhead: 81620 ( 1.78 veces)
Tráfico APEd/Comet: 2157695 bytes (9205 paquetes) Overhead: 2053751 (20.75 veces)

Cantidad de datos enviados: 103944 bytes
11.62 Veces más de tráfico con Comet que con WS.
Densidad (bytes enviados / tiempo): 28 bytes/seg
```

Apéndice B

Bibliotecas de Javascript

A lo largo de este trabajo se mencionaron varias bibliotecas de Javascript. En este anexo describen brevemente dichas bibliotecas.

B.1. Prototype

`http://www.prototypejs.org/`

Prototype es una librería Javascript que facilita el desarrollo de aplicaciones web dinámicas. Brinda facilidades para trabajar con AJAX, JSON y DOM.

A modo de ejemplo, el siguiente código sirve para obtener el elemento con ID **comments**, le agrega una clase HTML llamada **active** y luego lo hace visible (Asumiendo que antes estaba oculto):

```
\$('comments').addClassName('active').show()
```

B.2. MooTools

`http://mootools.net/`

MooTools es otra librería o biblioteca de Javascript usada para los mismos fines que Prototype. Sin embargo, a diferencia de la anterior, MooTools también incluye la funcionalidad necesaria para el manejo de eventos, *drag and drop*, disponibilidad de plugins y efectos visuales.

B.3. jQuery

`http://jquery.com/`

jQuery es la librería más utilizada en la actualidad. Presenta la misma funcionalidad que MooTools y Dojo, con una comunidad de usuarios en constante crecimiento.

Además del desarrollo principal de jQuery, el proyecto alberga tres subproyectos más:

- jQuery Plugins, un repositorio con miles de plugins que implementan diversa funcionalidad.
- jQuery UI, una biblioteca que implementa los elementos típicos de una interfaz gráfica de usuario, como ventanas, menús, manejo de eventos, tablas y temas gráficos (themes).
- jQuery Mobile, es la versión de jQuery para dispositivos móviles.

Por ejemplo, para agregar la clase **prueba** a todos los enlaces:

```
$("a").addClass("prueba");
```

Notar que Prototype y jQuery definen la función **$**, por lo cual son incompatibles.

B.4. Dojo

`http://dojotoolkit.org/`

Dojo es la última librería que se mencionará en este trabajo. Si bien dispone de la misma funcionalidad que las anteriores, su fuerte son las interfaces gráficas.

Más aún, Dojo es de las primeras en su tipo, y de las primeras en implementar una solución encapsulada para Comet.

Como las anteriores, Dojo es multiplataforma y multinavegador, incluyendo plataformas móviles.

Por último, la Fundación Dojo también trabaja en una implementación de Comet, llamada Cometd (`http://cometd.org/`), y en la especificación del protocolo Bayeux, sobre el que se basa Cometd (`http://svn.cometd.com/trunk/bayeux/bayeux.html`).

Apéndice C

Funcionamiento de APE

APE es una solución open source escrita en C que implementa la técnica de *streaming*[26]. Además del servidor de Comet, APE tiene un *framework* de Javascript que encapsula la interacción entre el desarrollador y el servidor. APEd también soporta nueva funcionalidad por medio de módulos implementados por el usuario.

APE también cuenta con su propio protocolo, basado en comandos y que utiliza los métodos POST y GET. Una explicación de este protocolo se encuentra en `http://www.ape-project.org/wiki/index.php/Protocol_Basics`.

APE soporta varios métodos de transporte (Mencionados en `http://www.ape-project.org/wiki/index.php/Tutorial:Use_different_transport_method_%28JSONP%2C_XHRStreaming%29#Use_different_transport_methods`). Si bien en el listado aparece Websocket, el soporte para éste se agregó recientemente, cuando ya se había elegido y armado la solución para los escenarios de Websocket. De lo contrario, se hubiera usado APE para los escenarios de Websocket y Comet.

La interacción entre usuarios se lleva a cabo mediante canales y mensajes. Un usuario puede enviar mensajes directamente a otro usuario o a varios por medio de un canal. Más información sobre este tema se encuentra en: `http://www.ape-project.org/wiki/index.php/Ape_server`

Por último, cabe destacar que se utiliza Javascript para implementar los comandos del lado del servidor.

Apéndice D

Funcionamiento de EM-Websocket

EventMachine es una implementación del patrón *Reactor*, descrito en: `http://en.wikipedia.org/wiki/Reactor_pattern`. Existen muchas implementaciones de acuerdo al lenguaje de programación elegido.

EM-Websocket es una implementación en Ruby del protocolo Websocket para EventMachine. La aplicación se diseña para actuar o reaccionar ante determinados eventos, como puede ser el arribo de un mensaje o la aparición de un error (**ws.onmessage** y **ws.onerror**, respectivamente en el código del servidor de Websocket implementado para este trabajo).

Bibliografía

[1] Tim Berners-Lee, The Semantic Web

[2] Lori MacVitti, The Impact of AJAX on the Network

[3] E. Bozdag, A. Mesbah y A. van Deursen, Performance Testing of Data Delivery Techniques for AJAX Applications

[4] Roy T. Fielding y Richard N. Taylor, Principled Design of the Modern Web Architecture

[5] AJAX and PHP Building Modern Web Applications - *Second Edition* Bogdan Brinzarea-Iamandi, Cristian Darie, Audra Hendrix (Packt Publishing, 2009)

[6] Google Inc. SPDY Protocol
`http://dev.chromium.org/spdy`

[7] Alex Russell - Comet: Low Latency Data for the Browser
`http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/`

[8] W3C - XMLHttpRequest W3C Candidate Recommendation
`http://www.w3.org/TR/XMLHttpRequest/`

[9] W3School - HTML5 Tag Reference
`http://www.w3schools.com/html5/html5_reference.asp`

[10] R. Fielding y otros - Hypertext Transfer Protocol - HTTP/1.1
`http://www.ietf.org/rfc/rfc2616.txt`

[11] G. Wilkins, P. Hintjens - Bidirectional Web Transfer Protocol BWTP/1.0
`http://tools.ietf.org/html/draft-wilkins-hybi-bwtp-00` *Expiró*

[12] G. Montenegro, Ed (Microsoft Corporation) - Websocket Requirements and Features
`http://tools.ietf.org/html/draft-ietf-hybi-Websocket-requirements-02`

[13] T. Yoshino (Google Inc.) - Websocket Per-frame DEFLATE Extension
`http://tools.ietf.org/html/draft-tyoshino-hybi-Websocket-perframe-deflate-00`

[14] I. Fette (Google Inc.) - The Websocket protocol
`http://tools.ietf.org/html/draft-ietf-hybi-theWebsocketprotocol-09`

[15] E. Cestari (ProcessOne) - An XMPP Sub-protocol for Websocket (Expiró)
http://tools.ietf.org/html/draft-moffitt-xmpp-over-Websocket-00

[16] P. Saint-Andre (Jabber Software Foundation) - Extensible Messaging and Presence Protocol (XMPP): Core
http://datatracker.ietf.org/doc/rfc3920/

[17] M. Crispin (University of Washington) - Internet Message Access Protocol (Version 4)
http://datatracker.ietf.org/doc/rfc1730/

[18] Robert W. Scheifler (MIT) - X Window System Protocol, Versión 11
http://tools.ietf.org/html/rfc1013

[19] I. Hickson (Google, Inc.) - The Websocket API
http://dev.w3.org/html5/Websockets/

[20] Nikolai Qveflander (Universidad de Umeå, Suecia) - Pushing real time data using HTML5 Websocket

[21] Comet Daily
http://cometdaily.com/maturity.html

[22] Documentación de WireShark - Libpcap File Format
http://wiki.wireshark.org/Development/LibpcapFileFormat

[23] A. Barth - HTTP State Management Mechanism
http://tools.ietf.org/html/rfc6265

[24] Lin-Shung Huang, Eric Y. Chen, Adam Barth, Eric Rescorla y Collin Jackson - Talking to yourself for fun and profit

[25] Hai Lin, Y. Chadli, B. Fourestié, M. Faye (Orange Labs) - Moving the Entire Service Logic to the Network to Facilitate IMS-Based Services Introduction

[26] Ajax Patterns
http://ajaxpatterns.org/HTTP_Streaming

[27] Cross-site Scripting (XSS)
https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29

[28] Steven Cook (SANS Institute) - A Web Developer's Guide to Cross-Site Scripting